FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Using malware detection techniques for dependency detection of R programs
Student:	Bc. Petr Adámek
Supervisor:	doc. Ing. Filip Křikava, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2024/2025

Instructions

The R programming language is a popular choice for data science. It features a large ecosystem of over 19k user-contributed packages hosted at CRAN (The Comprehensive R Archive Network). CRAN is a curated repository; each submitted package goes through a series of checks, including the execution of its runnable code, i.e., R programs extracted from package examples, tests, and vignettes. One way to reduce the risk of malicious or misbehaving packages entering CRAN is to audit the use of the package dependencies. This thesis aims to design and implement a tool that will track dependencies of R programs on Linux and collect them into sandboxed environments.

Analyze methods for tracking dependencies of computer programs used in malware detection and various sources of non-determinism for sandboxed environments.
Get familiar with the R programming language and study how these methods could be used for tracing dependencies in R programs.
Design and implement a tool to trace dependencies of R programs.
Design and implement a tool that can create the program sandbox using the recorded traces.

The resulting tools should become part of the R4R grant; therefore, they should be cleanly written, reasonably documented, and tested.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 16 February 2024 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Using malware detection techniques for dependency detection of R programs

Bc. Petr Adámek

Department of Information Security

Supervisor: doc. Ing. Filip Křikava, Ph.D.

May 9, 2024

Acknowledgements

I would like to thank my supervisor, Filip Křikava, who brought this fascinating topic up to me. For keeping me going with it and supporting me along any hitches, pitfalls, and boulders on the path to creating this work. My family for convincing me it is worth going forward with my studies, for all the food and letting me focus on the work when I needed it the most. And all my friends who kept my spirits up outside of school.

Declaration

Citation of this thesis

Adámek, Petr. *Using malware detection techniques for dependency detection of R programs.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Abstract

R is a programming language commonly used in data science. This is possible due to the large number of publicly accessible packages. To reduce the number of misbehaving packages, some checks are made. These include automatically running code examples but still leave a lot of room for manual checking.

A tool that would automatically gather the dependencies of any given R program could significantly reduce the manual overhead required. It would also find use in the context of creating an environment in which a program's result could be consistently reproduced.

This thesis imitates some of the work done in tools used to analyse or sandbox potentially harmful programs.

Using the system call interposition mechanisms in the Linux kernel, I have created a tool which can track dependencies of a given program. These dependencies can then be used to generate a report for auditing or create an environment in which the program execution can be reproduced.

Even though the tool cannot be directly used for any security-critical purposes due to the used mechanism and its associated race conditions, the tool is useful in reproducing shell scripts, execution of R programs, and others.

The thesis also mentions many of the variables and potential attack vectors a complete solution would have to consider, which are often glossed over. Moreover, other potential manners in which a tracing mechanism such as this could be implemented are explored.

Keywords sandboxing, reproducibility, R programming language, dynamic analysis, program dependencies tracking

Abstract

R je programovací jazyk, který se běžně používá v datové vědě. Toto je možné díky velkému množství veřejně dostupných balíčků. Aby se snížil počet nesprávně se chovajících balíčků, provádí se několik kontrol. Ty zahrnují automaticky spouštěné příklady kódu, ale i přesto stále zůstává velký prostor pro nutné ruční kontroly.

Nástroj, který by automaticky shromažďoval závislosti libovolného programu napsaného v R, by mohl výrazně snížit nutnou manuální práci. Využití by našel také v souvislosti s vytvářením prostředí, v němž by bylo možné konzistentně reprodukovat výsledek programu.

Tato diplomová práce se inspiruje stávajícími řešeními analýzy a sandboxingu potenciálně škodlivých programů.

Použitím mechanismu pro odchycení systémových volání z Linuxového jádra jsem vytvořil nástroj, který umí sledovat základní závislosti daného programu. Znalost těchto závislostí lze následně využít pro vytvoření zprávy pro audit nebo vytvoření prostředí, ve kterém lze program reprodukovat.

Nástroj nelze přímo použít pro analýzu škodlivých programů. Zvolený přístup kontroly systmových volání je příliš zranitelný útoky, které využívají nejasností v pořadí výkonu jednotlivých operací. Přesto je nástroj užitečný při reprodukci shellových skriptů, spouštění programů R a dalších.

V práci se také zmiňuje celý aparát, proč je možné tuto analýzu provést a co vše by muselo být vyřešeno pro úplné řešení. Pro tyto účely jsou zmíněny i jiné metody sledování operací a závislostí, které by mělo být možné využít pro robustnější řešení.

Klíčová slova sandboxing, reprodukovatelnost, programovací jazyk R, dynamická analýza, sledování závislostí programu

Contents

List of Figures

Introduction

R is a programming language commonly used in data science. But currently no tools exist for automated analysis of dependencies of R programs.

Package repositories for R such as CRAN[1] contain thousands[1] of different libraries. Checking such large count of packages by only hand would be a very long process. As such, there are automated methods for getting at least the most basic checks done. CRAN in particular uses *R CMD check* which does an analysis of specific language constructs and then attempts to run code tests and vignettes - essentially documentation combined with runnable code examples.[2, Appendix A , 22]

While this can serve as a good starting point to check that everything seems to work, there is still quite a bit of user involvement. For example the CRAN repository guidelines explicitly state that '*Packages should not write in the user's home filespace ...*'[3]. But no automated check for this seems to be provided along with the aforementioned checks.

Potential uses. By analysing exactly how the R program interacts with the computer it is running on, a sandbox could be constructed such that the R program would behave the exact same inside the sandbox and outside of it.

This approach could be taken even further. These little sandboxed environments could then be serialised in some manner and transferred to another device. Such a sandbox would then solve a part of the R4R grant[4] which aims to create reproducible environments out of real-world R not just packages which have to adhere to specific standards.

Portability. While the base principles described in this paper can be applied to any operating system, the specific described are working with the Linux kernel and are not trivially transferable. An operating system using a different kernel — especially if it were not using a unix-based one — would require significant changes, as the entire tracing and logging mechanism would have to be reimplemented.

[1]https://cran.r-project.org/

Security. Is should be noted, that I am not attempting to create a *secure* sandbox. That is I will not preemptively attempt to stop the R program from causing harm to the system it is being run on or otherwise affecting it. I will merely be tracing what actions are taken. Nor will I attempt to explicitly create an environment which would try to contain the workings of the sandboxed program to just that environment, as that might require sandboxing vectors I have not considered. The created tool is not meant to handle every possible case, just the usual one.

The main distinction between what this work covers compared to traditional anti-viruses is that I only ever need to track the behaviour of a program, but I will not further create signatures of the files in an attempt to classify it. Any detection avoiding or sandbox-detecting of a malware sample could be used by the code I will be working with and needs to be accounted for. This program never needs to restrict a certain action which would not be restricted otherwise. I need to ensure all the actions reaching outside of the sandbox are accounted for and logged.

Sandboxing. The definition of a sandbox I will be working with is '*We use the term **sandboxing** to describe the concept of confining a helper application to a restricted environment, within which it has free reign.*' [5] The restricted environment may include functionality which could be used to lift some of the sandbox restrictions. Or it may in and of itself be insecure as the program we have attempted to sandbox is malicious and we will not restrict the program from say spreading to another system over the network.

Sandboxes are generally created with two modules. One is the restricting engine which ensures that all relevant actions taken will be subject to restrictions. The other is the policy engine which decides whether or not to allow individual actions. This can be observed in many different sandboxing approaches[6]. But the policy being enforced will mimic that of the original operating system — a sandbox itself where each process can be considered a helper application.

What I try to create is the most constraining set of rules, which will not actually restrict the runtime of the program in any way. These are rules which *have* to be met for the program to run unimpeded and any outside it are to be restricted by the sandbox environment.

This issue could also be framed as getting list of all resources the program depends on and necessary permissions to those resources.

I do not attempt to solve the issue of ensuring resources are recreated and transferred even if they are on a contained on a different system or get modified between the start of the program and them getting stored for reuse. They are merely captured at one point. I do this as I assume the program is neither self-destructive nor self-modifying. That two runs of a program will inherently want to produce the same result. This is described in section 5.

Reproducibility. It is also worth mentioning why reproducibility is so important. Just having the correct files might be enough, to rerun a program and get the same output is good, but it does not necessarily convince us of the validity of the output. For that re-running the program with a different input or changing the algorithm in some way may be intended. But if we were to share only a portion of a library, any form of modification could break the program in unexpected ways. So, a method which detects the dependencies on a higher level may be interesting.

But any given program may depend on not just files. But perhaps a specific external device. Or a specific configuration of his system. Reproducing such programs is hard, if not outright impossible. The tool is meant to assist a nontechnical user in creating an environment which can work as a baseline for gathering all the possible dependencies.

The tool. A prototype is not really sufficient for many of the things mentioned here. While it may be possible to implement only a portion of the necessary functionality, in reality, the portion has to be self-contained and quite exhaustive. Otherwise. even a slightly more complicated program will break. But since these are functions used in a very limited number of cases, it is a lot of work which is not seen by the user at all. This thesis is not attempting to create a prototype which does a lot of things which sort of work but one thing which it does well and can be further extended.

A tool that can reliably report all of the dependencies could also have other uses. The tool could alleviate some of the burden of analysing unknown libraries in CRAN. Not only would it give a good overview of what programs the library depends on. It could also point out differences between two versions of a library. To my knowledge, no fully-automated process exists for checking CRAN packages, and much of the analysis has to be done by hand. If an R library suddenly gains a new dependency, this tool could raise a red flag and inform the auditor that a more thorough check might be needed.

Chapter 1

Malware analysis

To make an analysis of malware, we first need to say what malware even is. I will be using the following definition: '*Malicious code (or malware) is defined as software that fulfills the deliberately harmful intent of an attacker*' [7, p. 1]

The decision on whether or not any given software is to be considered malware commonly follows the following procedure described in [7, p. 1] [8, 88:2]

1. A signature of the file is calculated. The signature differs depending on which anti-virus checks the given file. Getting a relevant signature is a large part of malware analysis. This signature may be calculated both on the basis of what the software does on static analysis of the file.

2. If a given signature matches any known software, it is considered to be that type of software — malware or safe.

3. If the signature is not matched, it is then further analysed by a person with the assistance of various tools.

Most of the mechanisms used to get a good signature are not important for this thesis. Nor is the signature matching to other known software a relevant issue.

The third part of the analysis is the only truly relevant one. I intend to make use of part of the mechanisms used to gather information about the executable rather than decision making. Mechanisms used in manual analysis of a file when automated matching fails can be used for this analysis as well.

The relevant bits can be tools which gather information about a program in order to restrict further executions of it or a dedicated analyser.[9, 10, 11] This does not particularly matter.

1.1 Determinism

In general, all programs are by definition deterministic. With the same set of input values, the output shall remain the same. The set of input variables may be very hard, if not for all practical purposes impossible, to reproduce.

All the different inputs can be considered dependencies. Some are more or less reliably reproducible than others. The operating system or modules loaded into it can be considered to be dependencies. So can anything accessible on the filesystem or network. The goal is to reduce these dependencies to a set of values which can be then be restricted to a sandbox or transferred to another system.

Some dependencies may have to be mapped to other resources. For example, the fact that 'file descriptor X cannot be written to' is not important in and of itself unless I know what resource the descriptor is referring to. What is important is the fact that the handle is to a file somewhere on the disk and that when it is open, it should only allow these privileges to the given user.

But some things simply cannot be restricted nor reproduced easily. Different hardware setups can be considered a dependency. Some software may only work on specific processor types. Or require having a specific graphics card. Unless I made a list of all the possible quirks and features of hardware components, it is impossible to determine if a given piece of hardware is necessary for the functionality of a program.

Many malware may decide not to employ their payload based on fragments found on the operating system.[8, 88:2][12] The values upon which this change of behaviour is based are the exact values that have to be tracked. The dependencies of a program. I have to ensure they are the same in the sandboxed environment as they are in the live environment.

1.2 Program analysis methods

Programs can be analysed using two main approaches — *static* and *dynamic* analysis.[8, 88:2] Both have some flaws that make them more or less suited for this work.

Two key terms for any sort of analysis are *soundness* and *completeness.* An analysis is sound when it will always mark all occurrences of thing it is meant to detect — it may return false positives. An analysis is complete when it will never produce false negatives, missing things it is meant to detect.

In an ideal world, a complete and sound analysis would be required. This proves to be impossible as will be explained in the following sections.

A trivial sound analysis would say that the program requires the exact way the filesystem is set up to be the same, or even that the entire computer has to be setup in the same way, including the PIDs of processes. Such an analysis would surely be sound as there are no files which it could miss. If I were to run the program with the same filesystem state, it would produce — assuming no other dependencies are present — the same output. The analysis is very likely not complete as the odds of having a program depending on all the files on a filesystem are very low.

It is not impossible for that to be the case. A person could launch

```
ls -lR /
```

and list information about all files on the system.

1.2.1 Static Analysis

Static analysis works by searching the binary code of the program itself. It attempts to get information about the program without ever having to run the program.

This analysis is particularly vulnerable to many code obfuscation techniques which may cause the analysis to quickly lose precision and relevance. [11, 8, 13]

Even if I were able to guarantee that the input program I encounter will not be obfuscated or even that I have access to the source code of it, this will not be sufficient. There are ways in which the values of variables can be encoded such that resolving these encoded constants to actual useful values end up as an NP-hard problem.[11]

Moreover, it will not solve the issues where nondeterminism is introduced into the program at runtime through some other manner.

I could try to enforce that the variables input by the user are constant for this subset of the issue. The naive approach would only suggest that closing the input stream - ensuring EOF is always reached - and a pre-set program environment variables could be sufficient. This would not to consider other avenues in which user-dependent input can be reached in runtime. This is further discussed in 3.

This does not mean that, in a very specific environment, static analysis could not be used for sandboxing. Taking a very specific programming language with constraints such that the invariants which I need to enforce are met could be done. And before loading a program, I would have to prove that the program does not do anything that I would consider malicious. This would, however, still allow for attacking the sandbox itself — as described in section 2.2 the most high level attack vectors attacks either via exploiting the analysis, compiler, or code execution environment. The existence of a general language which meets such criteria would directly contradict Rice's theorem.

Methods for ensuring this can still be done exist. For example the Linux kernel eBPF mechanism [14] does this by not being a true Turing complete language. It restricts the code length, disallows loops and such. Furthermore, it ensures that all interactions with the outside world, the provided API, are done with valid arguments, not that some generic correctness is reached.

1.2.2 Dynamic Analysis

Dynamic analysis consists of actually running the program and observing its behaviour. This will intrinsically only ever cover a very specific code path.

A much more sophisticated approach, which attempts to resolve all possible code paths does exist. This is referred to as symbolic execution. It involves solving a series of constraints on all code branching points and resolving the branched paths with a new set of constraints.

It is possible to use this in very specific circumstances running in a very specific sandbox to extend dynamic analysis in some ways to attempt to detect code paths executing based on sources of nondeterminism which may be used by malware to avoid detection. [7, p. 1]

This is, however, very technically and computationally expensive and is unsuitable for our needs as it needs to be run in a machine which can deterministically execute code even in a multithreaded context and can revert the machine state.

Using just plain dynamic analysis can be done in two ways. Either the program can be run under a debugger for the sake of examining behaviour of specific parts of the code. Or the program can be examined for its interactions with the outside world. [8, 88:2] Since I assume to work with code which is effectively opaque to me, only the latter approach is viable.

There are security issues bound to using dynamic analysis as it involves running the code in an environment which may be modified or otherwise used for communication by the running code. [8, 88:2]

Due to the nature of only executing a singular code path, dynamic analysis results will differ wildly depending on the input presented. It is very dependent on any sources of non-determinism. [7, p. 1]

My dynamic analysis can never be sound and complete for all possible program inputs. It can potentially satisfy these requirements for a very specific set of input values.

1.2.3 Manual program analysis

Manual analysis can either be done by slowly observing the behaviour of a program being run under a debugger. Alternatively, some form of sandbox is employed to observe the behaviour of the program being run in a known state.[15][16][7, pp. 1-2]

These tools rely on the entire environment being well known. They also require all programs to be transferred to the target. Instead, my tool is mean to run on the target machine and observe the changes made there. Because of this, the tools cannot be taken and directly reused.

1.3 Evading detection

Malware evasion detection can take many forms depending on which type of analysis it is meant to avoid.

In the case of attempting to avoid signature-based detection, changing just a couple of bytes can result in the signature being marked different. And as such is fairly easy to avoid by for example encrypting the program data. [7, pp. 6-7] This approach may involve both static and dynamic analysis but is still susceptible to change even if not behavioural change occurs.

The following detection avoiding mechanisms all have to do with modifying how the program behaves in runtime or otherwise In general when a program detects it is running under analysis it may decide to not deploy its payload and instead only act as a normal piece of software.[8, 88:14] If I were to detect all possible sources of nondeterminism and successfully recreated them in a new environment, this could be entirely sidestepped as then the program would always act in the same manner.

Another approach to avoiding detection to execute code at a higher protection level.[8, 88:13] Essentially making some privileged, untraced portions of the sandbox execute code on behalf of the malware itself. Although this may seem like an issue in the design of the sandbox itself and could be avoided if engineered better, it may be an intrinsic part of how the software operates. When getting new hardware, a driver for it may have to be loaded into the kernel or be unusable otherwise. However, since this path exists, it can be potentially abused.

If I am talking about sandboxing purely in user-space, any way in which the sandboxed program gets root privileges will have the potential for the root privileges getting abused to avoid detection.

When a sandbox gets constructed it intrinsically has to consider a portion of the code to be immutable and for the codeto behave in a certain well-known way. This separation will be further discussed in the chapter 2.

Chapter 2

Sandbox construction

As was hinted at in the previous chapter, I will be using some form of dynamic analysis. For static analysis to be usable, I'd have to constrains on the running program which may not be possible for an arbitrary input which is not tailored for a specific execution environment.

The analysis will consist of running the unknown user code in some known environment which will allow me to keep track of what it uses. I will be calling this environment a sandbox.

To ensure a consistent definition of a sandboxing is used, I will be using the following definition: '*We use the term **sandboxing** to describe the concept of confining a helper application to a restricted environment, within which it has free reign.*' [5]

Sandbox can be considered as a blanket term for a restricted environment - a distinction does not have to be made whether the sandbox is a VM, an isolated computer, or just some user-space mechanism.

Innately, a sandbox has no concept of security. A sandbox is only an environment which can then, in some manner, communicate with the outside world. If no such mechanism were present, the sandbox would be useless. The only communication that could happen would effectively be done by side channels not meant for communication — such as observing the power draw of the processor. The defined way to communicate with the outside world will be referred to as the API of the sandbox.

Using the sandbox for security purposes is the same as enforcing some sort of policy on the provided API. This can be a firewall, user access rights, seccomp, or even an antivirus program. All of these approaches take your request to the provided API and decide if the request should be allowed to happen and in what scope.

For a sandbox to be *secure* I need to to prevent the part, where malware 'fulfills the deliberately harmful intent of an attacker.' But even defining what harmful intent is is hard and outside of the scope of a particular sandboxing mechanism.

Realistically, a sandbox is just the mechanism behind the policy enforcement. Not policy enforcement itself. Anti-viruses are policy enforcers that use some form of sandbox. Malware analysis tools are programs which do not enforce a policy, but instead log relevant communication going through the API.

It should be noted that the restricted environment may not be as small as a quick glance would suggest. The numerous ways of escaping a chroot environment on Linux if a user has sufficient permissions [17] should be sufficient indicator of this. But proving that a certain invariant is not accessible in any way in a sandbox may end up as a variation of the halting problem in a system not designed for such a usecase.

There are many differences between different types of sandboxes, but only two are important for this work.

The first distinction is the scope of the sandbox. Is just one process being sandboxed? Or is it the entire system? This is covered in the section 2.3.

The second is the actual policy enforcement mechanism used. How am I enforcing what I want on a provided API It may be based on debugging facilities, antivirus mechanisms or other logging features. The reason this is covered explicitly is because an existing sandbox may be used. One which I cannot modify due to whatever reason. This is further explored in the section 4.

2.1 Dependency tracking

To ensure that all the possible sources are captured or at least accounted for, thesis makes use of existing sandboxing mechanisms. Traditionally, sandboxing is used by operating systems to separate individual processes or even whole systems using virtualisation. Each individual sandbox has its own rights to system and other resources. [18]

To gather all the dependencies, I wish to create the smallest possible sandbox such that the behaviour of the run program will not be affected and then analyse that sandbox. The construction of a sandbox is important as it should guarantee nothing outside of the restricted environment will be accessed or otherwise affected and thus the program behaviour should be retained.

The sandbox I am constructing here need not necessarily be secure. The resulting program is not meant to be a security-critical thing. Running the code may still cause a computer to be infected if it, for example, directly demonstrates an attack on the host system or infects other computers on the network. Preventing such things is beyond the scope of this thesis. I merely keep track of a subset of the behaviour of the program.

There are many layers at which a sandbox could be constructed. Each layer describes some sort of boundary which is being communicated across.

Each layer separates the running environment into two parts. The trusted code base part and the unknown untrusted part, which may do anything. By having good knowledge of the API and the trusted code all dependency accesses can be kept track of.

Depending on the layer a sandbox is implemented at, different methods for detecting a presence of an analysis framework may be present. These along with the resources present in the sandbox are considered as the possible sources of nondeterminism. Although logically, they should be considered dependencies anyway and are mostly used as sanity checks.

A minimal sandbox would be a sandbox which does not contain any dependencies necessarily needed for the program to run unimpeded.

Each constructed sandbox has to have a distinct API through which all communication will happen. If the API could be curcumvented,

This thesis does not attempt to ensure a sandbox is isolated even in the presence of possible side-channel attacks or otherwise communications. These will only be mentioned in the list of possible dependencies.

2.2 Sandbox structure

To summarize what construes a sandbox. A sandbox requires a distinct API which will encapsulate all possible communication between a trusted, privileged part of an application and untrusted, sandboxed code. Some sort of policy enforcement mechanism may be added upon this API. This entire concept hinges on the fact that there is some method of ensuring that the API cannot be bypassed.

Thus, each sandbox will have at least three man directions of attack from the inside of the sandbox.

1. The API enforcement
2. The API implementation
3. The sandbox policy

A simple example would be in a Unix system. The kernel-space/user-space processor ring separation is the underlying mechanism for API enforcement. The API are system calls. Or more precisely, due to the enforcing mechanism, all faults which end up executing code in kernel mode. Practically, these are system calls though other interrupts can be considered a part of the API. The policy is, for example, that by default processes cannot access each other's address space. A case of attacks would be:

1. The processor incorrectly handles a specific instruction leading to memory being leaked.[2]

[2] https://downfall.page/

2. A buffer overflow in the implementation[3]

3. The policy enforcer incorrectly handles an API commonly used for another case and allows for access of memory from an arbitrary memory location as a side-effect.

2.3 Sandboxing boundaries

A sandbox could be implemented at a couple of distinct layers. Each layer comes with a different kind of API through which communication is done and different resources which may be accessed.

The sections are ordered from the lowest-level approach to the highest-level approach. All higher level approaches have to somehow consider all the resources the lower level approaches handle. For example, when examining at the hardware level, only blocks of data coming from the hard drive would be observed, while at the kernel level, I would consider contents of files or directories.

The higher level API give more semantic information to what is happening, so it will always be a trade-off. Some layers may even greatly simplify the handling of certain specifics of lower layers.

Any particular separation layer has to be unavoidable. The API it also has to be granular enough and contain enough information for the needed policy to be enforced properly.

Each of these layers has to somehow be enforced. That is, something has to ensure the API has to be used for communication.

Another thing to consider is that as more nuances are added with each layer, the policy will inevitably have to grow more complicated to account for all the possible variations of all the lower layers.

This enforcement is typically implemented using some combination of hardware and software. Even the separation of virtual and physical memory depends on a software handler for assignment of new pages in the case of allocation.

It is also worth mentioning that for the sake of dependencies a sandbox does not need to be reproduced in its entirety. Only the functionality inside the sandbox has to remain the same. For example if a hypervisor made network devices inaccessible, then there is no need to reproduce a hypervisor and the inaccessible devices — just the lack of network devices.

The separation proposed here is similar to the separations mentioned in other papers such as[19]. And the layer are shown in the image 2.1.

[3]https://nvd.nist.gov/vuln/detail/CVE-2023-6238

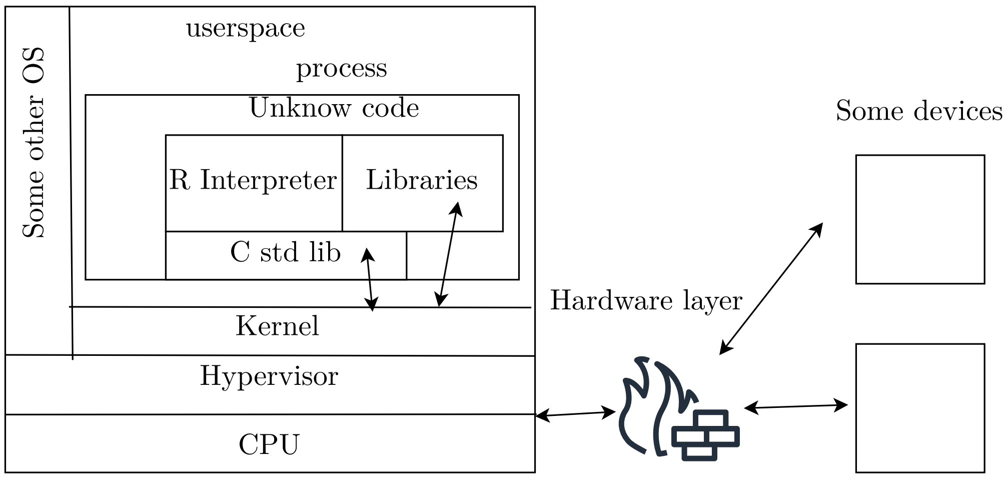


Figure 2.1: *A diagram showing all the possible communication layers which could be sandboxed*

2.3.1 Hardware layer

The first layer a sandbox could be implemented at is the hardware level. The communication happening in between the processor and other peripheries. Or from another point of view, the communication between the kernel — or possibly hypervisor if it is present — and peripheries.

It does not provide much semantic information as if I wanted to use a sandbox on this layer, I would require a complex understanding of the protocols used for communication. As such, the semantic information would need to be somehow side-loaded in the creation of the sandbox.

A working example of an implementation of this is a network firewall. It could be implemented using a distinct device on the network, which decides on what communication is allowed to pass through between other computers. It also requires specific knowledge of the trusted side of the network — such as which ports devices are actually meant to be listening on and so on.

It points to a first possible source of differences. Different hardware can lead to different behaviour of the program.

The relevant portions for this thesis are:

- RAM
- GPU
- Hard disks and other persistent storage
- Other devices on the network

To explain the reasoning for the inclusion of some of these components. Each program will have a lower bound on the amount of RAM required to function properly. A program demonstrating a specific exploit [4] might require a very specific setup of the memories.

What GPU is present may very well affect the functionality of a program. If utilising CUDA for any calculations, an absence of a conforming GPU will make the program stop working. R also has some libraries that use CUDA, such as Rpud[5].

The persistent storage specifics are less relevant when operation in the scale of files, folders, and such. But if I were to try and sandbox such operations on this layer, some operations may even be cached by the kernel itself and I would only know which areas of the storage devices had been used. But with no semantics.

If a computer is connected to any network, I would require complex knowledge of each device the computer can communicate with. This may prove to be impossible if I connect the computer to the internet. No matter the enforced policy, a method for subverting it will probably exist unless communication is restricted to a select few well-known devices.

Many other devices could get connected to the computer and affect the way it works. This includes USB disks, keyboards and mouse, display devices, audio devices, and others. While this may seem overkill, there may be reasons for restricting access to such parts of the system. If I did not consider video devices at all and allowed anything to be passed through this channel, it could be used for data exfiltration from the sandboxed system. The project [20] uses the EM side-channel to reproduce video passed over wires. But even just the fact that the program is communicating with another device is problematic, as it should be reproduced.

2.3.2 Hypervisor layer

These two next layers are not separated by a physical barrier. Instead, privilege ring levels of a processor are used. This means that from now on all code actually depends on CPU architecture compatibility.

One of the privilege levels is the hypervisor, which allows multiple different kernels to operate in the context of one processor.

On its own, it works mostly as an isolation layer. Policy enforcement at this layer consists mostly of granting access to different resources for different virtual machines. This can be seen in hypervisor configurations shown in [21].

This layer still does not provide a sufficiently fine-grained access to resources as the most distinct point are operating systems while using something in the context of processes is necessary.

[4]https://en.wikipedia.org/wiki/Row_hammer
[5]https://github.com/cran/rpud

2.3.3 Kernel layer

This is the layer most programs will operate on. It separates the privileged kernel-space code from unprivileged user-space code. While having root privileges does effectively subvert the sandbox, as it allows loading new arbitrary code into the kernel via kernel modules, in the case of unprivileged programs many assumptions can be made. This layer is enforced by the protection rings of the CPU.

Any API function of this layer will be referred to as *syscall*(system call).

A limited number of ways in which context switches between user mode and kernel mode can be made exist. All of these are hardware mechanisms.

A subset of these will be used for communication. Although, in theory, a page fault could be intentionally invoked from an arbitrary address and thus be used as means for communication. I consider this to be unimportant for this particular usecase and will assume that the syscalls are the only API used for actual communication.

In this thesis I will only be concerned with the Linux kernel. And even that brings enough complexity.

Utilising a sandbox at this layer would allow for the usage of the pre-existing OS process-based sandbox.

The kernel can cause wildly different results for operations on different files depending on which filesystem they are on and if they are a file or a special device. While the API is the same, the semantics of writing to a file on an EXT4 filesystem and writing to a file in the virtual procfs filesystem are completely different operations. One eventually results in the data being written to an underlying physical device while the other will alter information about a process in the kernel — such as the name[6].

Because kernel modules can create custom files, providing new API to the sandbox and they can affect the kernel, they have to be known and accounted by the sandbox policy engine as otherwise a module could provide a custom device interaction with which would result in unknown behaviour and possibly alter the state the sandbox assumes the process is in.

It is also worth mentioning that the kernel contains other isolation mechanisms which will be covered in the chapter 4.

2.3.4 Language layer

The last layer mentioned here is the language layer. By utilising specific language features or subsets of a language some form of isolation can also be attained. This method is completely orthogonal to the previous ones mentioned and is independent of the underlying implementation. The same behaviour

[6]Writing to /proc/self/comm - see https://www.man7.org/linux/man-pages/man5/proc.5.html

could be enforced whether the code is running in an interpreter, a virtual machine or on bare metal.

Depending on which layer exactly the language is used — and the API provided by it — all of the issues mentioned in the aforementioned layers have to be considered.

One of the benefits of this approach is the potential for improved semantics of a given call. For example, the discontinued Java security mechanism gives the entire call stack of context.[22] In R this could give the policy engine information on which library is being used for this specific call or which library calls are actually being utilised.

Aside from using language features for enforcing a sandbox cannot be subverted a technique called *software fault isolation* can be utilised for ensuring a given piece of code adheres to a specific restriction which had been decided upon.[23] This is very hard to get right and will not be discussed further as I do not considered this approach viable to the issue at hand.

Library shimming

One way to implement this is to enforce all code will communicate with the outside world using only libraries which have the policy enforcement embedded inside them. This could be done by utilising LD_PRELOAD[7]. This cause a specified path to be searched for libraries before any other.

Overall the library replacement path is not viable. While some projects[24] have successful utilised it, they suffer issues with programs not compiled under C. There is a lack of an API enforcement. Attempting to shim all possible libraries is an enormous task, and there is no guarantee that no library will explicitly make a call to the underlying API by itself. Almost any program could do it. A more robust solution[25] of a similar problem ends up utilising kernel features to ensure functionality.

This mostly covers the options for sandboxing code compiled to native code. There are many nuances not covered here. If I had access to the compiler used to create all of the binaries, I could enforce — with a variation of Software fault isolation — that only the one library is always invoked for any underlying API usages. The same could be said for the underlying sandbox API. This is however not by any means trivial to enforce and prove to be correct.

Interpreter isolation

If I have a language running in an interpreter — or a VM, the difference is only in the provided API for this case — a unique opportunity arises.

If no native language language libraries are loaded for the language code to call, only the default packages have to be shimmed for some sort of policy enforcement mechanism. This would sidestep the issues of any underlying

[7]https://www.man7.org/linux/man-pages/man8/ld.so.8.html

```
local sharedTable = {};
function closure(key){
    return sharedTable[key];
}
keyTable = {};

    sharedTable[keyTable] = 123;
    assert(closure(keyTable) == 123);
```

Figure 2.2: *A Lua code example. The resulting state which would be practically impossible to transfer to another virtual machine.*

compiled language implementation specifics and the policy engine could be built into the standard library of the language.

This, however, requires that all loaded libraries consist of language code. Not direct native code.

Debugging facilities of a language could be used to a similar effect. When a native code invocation happens, a policy engine would then be invoked. But such a policy engine could equally as well be built into the libraries themselves.

Language features

The last possible sandbox can be construed inside the language itself.

This also allows for more fine-grained isolation of data while allowing for sharing of values which would have changed semantics if they were to be serialised transferred between two interpreters and if they were used in two sandboxes inside the same interpreter.

Take the Lua code on the figure 2.3.4where tables are considered first-class variables: Tables are first-class values. But what if the table identity does not get preserved when transferring values. Transferring the sharedTable and keyTable variable separately by serialising them would result in the keyTable variable containing a different value than the sharedTable does. Also, closures are practically unsharable as the context they are bound to is basically impossible to reproduce.

The kernel-space/user-space like separation could be simulated by these means instead of the syscall-based API. I just need the language to not be able to invoke an arbitrary native function.

Another example usage of this would be wrapping of all provided native functions in policy enforcing wrappers. Not all languages can however safely hide values

Utilisation for R

Unfortunately, R is not suitable for this sort of sandboxing. While R does run in an interpreter and as such the interpreter could be modified to allow for sandboxing, it is common for R libraries to consist of native code. There is no guarantee that the native code libraries will not invoke syscalls manually nor that the native code will depend on all syscalls being indirectly invoked by a specific library that I have decided to shim. For many cases, tracing the C standard library may end up being sufficient - this is, however, not guaranteed and both [24] and [25] have shown it to not be ideal.

Another consideration is the sheer size of some of the standard libraries. On my system, the c++ standard library exports 6169 functions[8] and the c standard library exports 3023[9]. Compared to the less than 400 system calls of the kernel[10], this is a much larger attack surface.

Moreover, this would not resolve the issue where if a new helper process is spawned, there are no guarantees for the process to be an R program or any other program I have knowledge of.

[8]nm -D /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.30 | wc -l

[9]nm -D /usr/lib/x86_64-linux-gnu/libc.so.6 | wc -l

[10]https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl

Chapter 3

Sources of nondeterminism

As has previously been outlined, I will need to track all possible sources of nondeterminism. I am interested in what can cause the program to have a different output at the same execution point. By definition, programs are deterministic. Any and all nondeterminism cannot come from the program code but from the hardware it is running on or the environment it is running in. If I ensured all sources of nondeterminism return the same value, the program would have to produce the same result.

But having a sandbox which would perfectly reproduce the dependencies for one exact program is also meaningless. Recording all the sources of nondeterminism and reproducing them identically is not a good solution.

Not only would this be extremely difficult to pull off in the face of a multithreaded program. If I decided to change anything about the sandboxed program's behaviour — in-line with other access rights already required — the program would not behave as it should since the reproduced API may no longer be semantically correct. After all, it is just a recording with no logic behind it.

I need to recreate the initial conditions so that the program will enact the same behaviour. For potential sources independent of the initial state, the user should at least warned.

In general, I don't consider side channel attacks. No matter what sort of sandbox I construct, side channel attacks are something no sandbox can really help with. Sandboxing is about constricting a specific API. Side channels are creating a new API effectively attacking the policy enforcement. These attacks may exploit various hardware bugs which I cannot easily consider as a part of the provided API.

3.1 Hardware

The lack of some devices may cause incompatibility, such as a missing GPU.

On the other hand, a missing audio device might not affect our program at all, as the rest of the operating system will substitute it with another one. But this cannot really be decided as a program could depend on a specific feature of that device. As such, any mismatch in the underlying hardware may be a relevant issue, unless we can decide which exact device we are working with.

Another issue is the internal state of any connected device. I we are communicating with another PC over the network, there is no way to reproduce the state of the other PC. In general this cannot really be solved, and such a sandbox cannot be automatically constructed.

A significant change in the CPU architecture may cause the entire program to be inoperable. This does not have to be any complex feature; even the most basic CPUID[11] instruction used to identify a processor and its features can make the actual instructions executed differ.

Realistically, the only way to ensure compatibility in these regards is to make a list of the original hardware specifics and then check them against specifics at sandbox startup.

3.2 Kernel

The kernel also has some state which may be unobservable from user-space. At the very least, most kernels should be started relocated to a random address via Kernel Address Space Layout Randomization[26]. Other variables may also apply. Any loaded modules or kprobes(see section 4.2.2), even the exact state of all processes in the OS are saved in the kernel.

But even discounting communication with running processes, previously ran processes could have created shared memory sections which could be accessed by a pre-defined key.[27]

All sorts of configuration are technically stored on the filesystem but realistically have affected the Kernel when loading. The simplest example is the fstab file[28] which causes a filesystem to be mounted.

When the sandbox runs, there will be no knowledge of what configurations are used and from where they are used. A prime example are mounts. Mountpoints are not inherently stored on a filesystem — rebooting with a mount will unmount it — and yet they act as directories in many manners.

The semantics of different handled syscalls will be talked about in the tool implementation chapter 5.4.3 as they are tied to my solution for handling them.

The semantics behind users and their home directories or even passwords are entirely a user-space mechanism. The kernel merely switches an internal ID associated with a process.[29]

As user-related values which are relevant for the sandbox should be accessed directly by some process in the sandbox and thus be detectable. This

[11]https://www.felixcloutier.com/x86/cpuid

may, however, rely on executing an suid binary and the kernel not interrupting such a tracing mechanism.

3.3 Filesystems

Filesystems are a troublesome terminology. There are three different types of filesystems which need to be named separately.

Firstly, physical filesystems which are persistent filesystems sort on some medium. Secondly, there are virtual filesystems which are used by Linux to provide an API for interaction with parts of the kernel — such as procfs or devfs. Lastly there is the view of these filesystems provided by the Kernel which the users actually interact with. When referring to the filesystem, I am referring to the view of the filesystems provided by the kernel.

All of these systems work like directed acyclic graphs with some nuances. Generally, the structure consists of files and directories. But not all files have to referencible from a filesystem. It is possible to unlink a file and yet until a file descriptor to that file exists, it will not be deleted. Physical filesystems may also include a mechanism for unique identification of different entities — e.g. inodes. These can be used, for example, to detect if two files are actually the same file.

Any file that is interacted with can be considered a dependency. But a user does not interact with a single filesystem. In theory, interacting with files from two different physical filesystems may carry its own implications. The trivial implication is that a hard link cannot be created between them. But this may also carry implications for path resolution. The openat syscall[30] has a flag RESOLVE_NO_XDEV which restricts path resolution to the same physical filesystem.

3.4 Inherent

Some sources of nondeterminism that cannot be properly handled are going to appear due to the nature of the hardware we use nowadays.

Processors contain multiple cache levels. Depending on the state of the caches the processor time necessary to execute a given instruction may vary wildly. As such, any interactions with processor time is bound not to be deterministic. This is used in many timing-based cache attacks.[31, 32]

Speaking of time, the real current time is also an issue. If a program interacts with the outside world having a valid current time is important, otherwise some authentication algorithms will fail as they give a certain timeframe when a token is valid. Even certificates have a validity interval. So interactions with the current time will inherently lead to nondeterminism which cannot be easily reproduced.

When multiple programs or threads work with the same data or dependencies, the output can end up reordered in the best case. In a worse case the programs have introduced some worse form of a race condition. Depending on the number of cores, the way the scheduler decides to schedule threads, the way caches are set up, the speed of the RAM which contains the data of my program, or even hardware interrupts happening invisible to the sandbox can affect the order or result of operations. There is no reasonable way to create a reproducible environment in the face of these issues unless we forcibly run the program in an environment which can enforce determinism.

Reading uninitialised memory may be more or less of a issue depending on how it is viewed. The Kernel will zero out any memory which it initially provides to prevent data from being leaked to another process. This can even be an issue on other devices connected to the computer, such as a GPU as demonstrated by [33]. Reading such memory is not inherently an issue. But semantically this means that a piece of memory was allocated and not initialised before being used. Is one of the ways in which programs may begin to act unpredictably. There is no real way for the sandbox to detect this. In theory, this should be reproducible, but any form of nondeterminism in the memory allocator used by the program will lead to unreproducible behaviour.

3.5 Reproducing the process sandbox

At this layer, we are essentially tracking a process — or more specifically a tree of processes stemming from one spawned process. There are ways in which this tree may interact with other process trees. This can be done via modifying a file used by both groups — such behaviour would be very hard to detect as it is a mostly user-space-based communication. From kernel space we only ever see accesses to the same files, not knowing whether intentional — or unintentional — communication is happening. The methods for tracking accesses to a filesystem outlined in the previous chapter could be used for handling these changes.

A process has multiple variables associated with itself in the kernel. Most of these can be seen as the potential flags of the clone[34] call.

- memory space
- thread group
- System V semaphor undo list
- signal handlers
- thread local storage
- the parent PID

- namespaces
- debugger attachment

Some parts of this list are more easily reproduced than others. Assuming we control the process which spawns the sandboxed program, we can ensure that all but the memory space will be the same. The memory space is impossible to reproduce due to the most basic exploitation protection technique ASLR(address space layout randomisation).

Special care has to be taken when creating a new process as it theoretically could retain some information from the parent. For example, the alarm[12] mechanism is retained across execution of a new program but not clone.

There are also the capabilities of a program that greatly influence the potential operations a program can execute.[13] Closely tied to this is the no_new_privs attribute from prctl[14].

Another value which cannot be easily managed is the PID of the parent process or even the process itself. This depends on what other programs are running on the system and what state the Kernel is in. This can be mitigated using namespaces(see chapter 4.1.3). Other groups a process is a part of which will be covered later.

3.6 Privilege escalatuion

One of the potential malware evasion techniques is privilege escalation. When a sandbox executes any sort of code which will affect the behaviour of the kernel in unexpected ways, the sandbox cannot guarantee to be complete. The scope of the escalation does not matter at that point. I could affect any layer above the userspace and be undetected due to the layer I have chosen to implement the sandbox at.

3.7 Analysis framework detection

Some tools used to detect that a given executable is running exist. For example the pafish tool [15] for windows checks the following information which could be used to detect the presence of a framework.

- registry
- cpuid instruction
- timing of various instructions

[12]https://www.man7.org/linux/man-pages/man2/alarm.2.html
[13]https://www.man7.org/linux/man-pages/man7/credentials.7.html
[14]https://www.man7.org/linux/man-pages/man2/prctl.2.html
[15]https://github.com/a0rtega/pafish

- sandbox specific operating system configuration, such as network rules
- presence of a debugger
- known usernames
- drive names, sizes
- memory size
- uptime
- sleep duration consistency
- presence of user-space hooks — essentially detecting if a DLL injection-like mechanism is happening
- mouse presence and interaction
- know sandbox-specific DLLs

A similar list of techniques can be found especially for windows [35] but categorically they are the same as has been listed so far. Other tools for automated detection, such as [16], exist, but they are focused on the Windows platform.

[16]https://github.com/LordNoteworthy/al-khaser

Chapter 4

Tracing implementation

The policy enforcement part of a sandbox can equally be utilised for tracing purposes. Instead of deciding if a certain call should be allowed to proceed, the call can be logged in some way.

I need to trace programs on a process-based granularity. Any layer lower than the kernel layer is not usable for this application.

If I were to sandbox purely at the hypervisor layer, the entire OS startup process would have to be logged as well. The tool would be quite cumbersome to use and require getting information back from the kernel. It would require knowing how the kernel keeps track of processes. But that goes against the principle that a sandbox should consider the code inside it as insecure.

Since this use case is not really suitable for any sort of language based isolation, all that is left is using the kernel layer.

This does not mean that the implementation cannot make use of the hypervisor for sandboxing. Cooperation between the kernel and a hypervisor is possible, as has been demonstrated by[13].

The kernel provides many facilities for potential sandboxing of user-space programs. I will explore them here with evaluation of suitability for keeping track of dependencies.

4.1 User-space based approaches

The Linux syscall API provides many ways in which one program may affect another. Some of them could be used to track dependencies.

An issue permeating all of the user-space solutions that would have to be resolved with this approach is keeping a track of relevant PIDs. Linux does not provide any way to obtain the PID of the creator of a given process. So an API which would only attempt to operate on PIDs is insufficient on its own.

4.1.1 Inode watchers

In a world where the only dependencies which I will try to reproduce are files on the system, just seeing what occurs on the filesystem should be enough.

The Linux kernel has a dedicated inotify[36] API for this purpose. The reason why it cannot be used is that the API cannot distinguish which process had made an interaction. To quote the documentation '*The inotify API provides no information about the user or process that triggered the inotify event.*' [36]

Moreover, the watcher gets attached to a specific inode and does not recurse further in the directory tree. A watcher would have to be created for each directory on the entire filesystem. And any newly mounted filesystems would be missed.

4.1.2 Filesystem watcher

Another approach is the newer fanotify[37] API for watching filesystem events.

The events provided by this API contain the PID of the invoker, so processes interactions could now be tracked. This approach would involve marking all relevant mounts, filesystems, and notifications as tracked. By then logging each notification, a complete list of all filesystem dependencies should be created.

It is to be noted that this API also provides a method for denying access to a given operation. This could be a method in which our sandboxed program could communicate with an unsandboxed program and would make all programs which are watching relevant directories with permission checks dependencies of the sandboxed program.

4.1.3 Isolation mechanisms

In Linux, namespaces[38] are means of isolation of resources. While these could be used for improving reproducibility somewhat as they can ensure that the sandboxed program cannot access some dependent resources in the first place. But they do not provide any means with which to trace used files. Furthermore, their usage is essentially an attempt in creating a well-know, reproducible environment in the host system — rather than analysing the existing one.

Control groups[39] are another way to restrict access to resources for a group of processes. These are more along the lines of hardware resources, open file handles and such. They are an effective sandboxing utility for preventing DOS-like attacks, not for fine grained access control.

Other control mechanisms include user IDs, Group IDs, or Session IDs [40]. These are relevant for reproduction purposes and access control. Not so much for process isolation or tracing.

4.1.4 System call interposition

Another major way to trace events happening is to directly interpose ourselves between the kernel and a user-space application. This has been shown to work with many issues in the past. [41, 42]

This approach is inherently prone to race conditions since system calls may be executed in parallel. By simply seeing the calls, it is not quite possible to even safely check the parameters of the call with the guarantee that they will not be modified [42].

Seccomp

Seccomp[43] is a feature relating to sandboxing of programs. One of the things it offers is to install a custom syscall filter. This is used by some programs [44] to voluntarily restrict themselves.

The filter is written using cBPF a subset of eBPF(extended Barkley packet filter). It is not eBPF, which is commonly used in other parts of the kernel. The filter is inherently stateless. Moreover, there is no way to dereference user-space pointers for data inspection. But while some attempts have been made to extend the kernel to support eBPF it is not yet a part of the mainline kernel. [45]

Even if all of this were added to the kernel, there is no simple way of getting arbitrary-length strings out of eBPF. [46]

So, the filter cannot be used directly to do the logging. However, there are methods for utilising seccomp to invoke other facilities, which could be used.

For one, the filter can say that a given syscall should be audited — SECCOMP_RET_LOG. This means that the action will be logged in the audit log. The audit log is a systemd feature[47] and would make systemd a hard dependency[48]. Without it, the seccomp filter will not actually log anything. Furthermore, what data is logged cannot be configured and thus it is unknown if the output would even be consistent across kernel versions.

Another potential result of a filter is that a ptrace (see section 4.1.4) tracer should be notified - SECCOMP_RET_TRACE. This comes with other security implications noted down in the manual. Then the behaviour is essentially the same as with a ptrace-based tracer.

The last potential usage is via SECCOMP_RET_USER_NOTIF which would instead devolve into another process having to execute the operation on behalf of the sandboxed program. This is not really useful for our usecase, unless some form of emulation of syscalls is necessary.

Ptrace

Ptrace[49] is the Linux syscall for all debugger-related purposes. Just about any debugger-related action is routed through this system call. A program(the tracer) traces another program (the tracee) and can essentially cause arbitrary modifications of the tracee. The tracer can trace any action the tracee performs. A tracee is any individual thread rather than a process, meaning the tracer has to ensure it is attached to all threads.

One limitation of a ptrace-based approach is that it will significantly affect the traced application. Meaning detection of a debugger is possible in many ways. And not only detection. Even the behaviour of some syscalls changes, such as blocking reads, when made by a process that is being debugged. Furthermore, tracing across an suid boundary requires special permissions.

4.2 Kernel-space approaches

If I wanted to use some form of tracing from the kernel itself, there is no need to directly modify the kernel code for my needs. The kernel already has some built-in mechanisms for tracing events.

These may have completely different semantics as, for example, path resolution operates on the level of nodes on the filesystem, whereas the user-space API have to actively seek these values to interact with them in this manner.

Any usage of these parameters depends on understanding the specific part of the kernel that is being traced, including all possible ways to bypass it. Incorrect usage could be just as bad, or even worse, than using just plain system call interposition mechanisms.

4.2.1 Linux security modules

Linux security module(LSM) is a framework for a hooking into functionality provided by the kernel at points which may require access rights. They need to be compiled as part of the kernel and cannot be loaded as individual modules at runtime. [50]

Compared to a simple system call interposition mechanism, these calls are made in the internals of the kernel when all relevant values have already been transferred into kernel-space and no race conditions should occur.

Instead of writing a full-blown kernel module, it is possible to write eBPF filters for these modules and load them at runtime for modern kernels.[51, 52]

4.2.2 Kernel tracing

Another avenue is the one used by the tool bpftrace(see section 4.3.1). This involves the usage of some kernel built-in mechanisms for debugging or performance accounting.

The first approach worth mentioning are kprobes. These are hooks that allow adding a tracing routine to most parts of the kernel. This essentially works as if breakpoints with dedicated handler functions were used with a debugger.[53]

The other are kernel tracepoints. These are pre-declared places in the kernel which may be traced. Compared to kprobes, these points do not patch the kernel when it is executing, but at compile time. [54].

4.3 State of the art tracers

Some tools which trace program interactions with the kernel and or filesystem already exist. This section contains description of the most significant ones.

4.3.1 bpftrace

Bpftrace[17] is a tool which aggregates a bunch of different tracing mechanisms and adds a unified API for interacting with them.

As the name suggests, the programs for this tool are written in eBPF. But they may be inserted into various parts of the kernel, essentially allowing for debugging of the kernel. The tracing mechanisms are not limited to just kernel tracing, but to user-space programs and libraries as well.

I have not found a simple way in which to keep a track of processes which should be traced from inside this tool. And tracing the entire kernel is undesirable as it may produce many false-positives.

4.3.2 strace

Strace[18] is a utility which internally utilises ptrace to track syscalls made by a tracee.

This tool creates a reasonable textual dump of the performed system calls. That is all it really does. Theoretically, a parser could be added on top of strace to then filter out relevant bits of the information it had parsed. But for the purposes of this thesis not all syscalls or all their arguments will be relevant.

It also shows off that seccomp can be used instead of ptrace to, I presume, ensure the tracer is not notified of all syscalls but only the ones which the BPF program decides should be traced.

4.3.3 CARE

CARE[19] is a tool built to create an exact reproduction of a program. It creates an archive of all accessed files along with a script which recreates the initial set of environment variables and reruns the command.

[17]https://github.com/bpftrace/bpftrace
[18]https://github.com/strace/strace
[19]https://proot-me.github.io/care/

This seems to be doing exactly what I am envisioning for this thesis. But the exact approach is very different. To ensure more compatibility, the tool is built over the tool PRoot[20].

The tool PRoot seems to emulate many necessary system calls to allow unprivileged users to perform privileged operations such as mount or chroot. Essentially translating operations on the filesystem through itself to provide a consistent filesystem view. Glancing at the source code shows that the tool also attempts to keep the API stable in translating ptrace syscalls and trying to allow for nested tracing.

Conceptually, it does what I will be doing with tracking dependencies. I will attempt to avoid any system call emulation if possible.

4.3.4 fakechroot

Fakechroot[24] is a tool that modifies the c standard library and, in doing so, modifies relevant calls to convince the sandboxed process that the current user is in fact UID 0. The tool does no do tracing in and of itself, but the same mechanism could be used to instead log the actions rather than modify the return values.

The ways in which the tool can be bypassed have already been mentioned in the chapter 2.3.4.

[20]https://proot-me.github.io/

Chapter 5

R4R Tool

I need a program that can track the dependencies of R programs. But, as was outlined previously, keeping track of all dependencies is not possible. So a subset has to be chosen and expanded as realistically needed.

This will be done based on a couple of assumptions and needs.

- Programs we are tracing are not actively malicious and will not actively try to introduce race conditions into the running code.
- This includes not modifying the values of strings and other structures pointed to by pointers passed into the kernel while a syscall is executing.
- The programs should not use ptrace as it is after all mainly a debugging mechanism, not an inter-process communication tool.
- The programs will not be actively destructive and if a program runs multiple times it will depend on the same files and will not attempt to destroy its own dependencies.
- The program will not attempt to subvert the detection by any pre-established communication side channel on the host, e.g. a filesystem watcher.
- The tool, due to the intention of usage within the R4R grant, should not require the users to install potentially problematic kernel modules nor require unnecessary privileges.

If the assumption on variable dependencies were not true, the tool would have to persist the dependency in some manner before the program is allowed to interact with it.

The tracing will be implemented using ptrace. But as this is knowingly not a secure mechanism undetectable by the running code, the tool will be designed such that replacing the ptrace tracer should be simple enough.

The result of the tool should not depend on the tracer used and how it was interpreted. It should only require a list of dependencies and be able to create the sandbox or a report based on this.

First, I assume that tracing interactions with the view of the filesystem should be sufficient for a rudimentary dependency analysis. This is supported by the fact that the CARE program, which solves a very similar problem, works by tracing operations on the filesystem.

Ideally, the tool should at least warn the user if access to unreproducible dependencies is detected.

Interactions with any potential graphical user interface or other management engine such as systemd have not been explored and are assumed to use the standard communication avenues the kernel provides.

The program is also currently not designed to work with user programs that require the user to explicitly terminate them. The program could be designed to attempt to create the report upon receiving the first kill signal along with passing the signal to the child. A prime example where this could be used is the *ping* utility on Linux.

The system calls which are actually handled are system calls which have been encountered while testing the tool and I have deemed to be the most important to be handled. This means the amount of handled system calls is far from exhaustive.

5.1 R code execution

R research code is usually created in R notebooks[21]. They are very similar to Jupyter notebooks[22] from python. The notebooks are just specially formatted files with inline executable code.

The code inside the notebooks can be executed in batchs from the commandline using *rmarkdown::render()*. Documentation for R libraries is also often created using this method. Otherwise R code is just source code with no special surprises and is executed using the R interpreter.

The specifics of the language are unimportant for this tool as due to the nature of libraries, oftentimes native code is executed and that needs to be traced as well.

The way library-based dependencies are handled are referred to in the section 5.6.1.

5.2 The tool API

Since all R programs can be executed using just the commandline, we can create our tool such that it will be passed arguments as if the program were

[21]https://bookdown.org/yihui/rmarkdown/notebook.html
[22]https://jupyter.org/

to be executed normally by pre-pending the tool executable to the usual way to launch the program. And the tool can be, as far as executing, agnostic to the used program.

```
./tracer [executable] <arguments to executable>
```

Care has to be taken not to accidentally pass only a part of a script to the executable.

```
./tracer cat /etc/passwd | cut -d: -f1
```

Will only trace the cat call. Not the cut call as well. This can be worked around by wrapping them in an explicit execution in a shell. But this can be assumed to be common knowledge for a user.

```
./tracer bash -c 'cat /etc/passwd | cut -d: -f1'
```

The tool will not restrict the execution of the program in any way. It only traces what the tool is doing for further analysis.

The stdout an stderr of the traced file will be redirected to a file so it is separated from output of the tool.

5.3 Tool architecture

There are three main parts of the tool.

The frontend, the tracing implementation in and of itself. This layer consists of the ptrace specifics and handling of traced system calls.

The middleend. This layer will essentially emulate information the kernel keeps about a process and duplicate what a kernel is doing, regardless of what mechanism will be used to catch different syscalls. I does not emulate the actual execution itself but it does keep track of information such as the file descriptor table. It mostly translates operations on kernel identificators to actual dependencies.

The backend which takes the dependencies in a pre-define form. It will not have to care about what tracer was used to gather the information. It can either produce some sort of a report or a way for a sandbox to be set up.

The architecture is also shown on the figure 5.1.

The entire tool is written in C++ due to the close relation between the system call API and C. Any of the layers could be written in another language, though data transfers would have to be handled in some way and would add unnecessary complexity for a prototype.

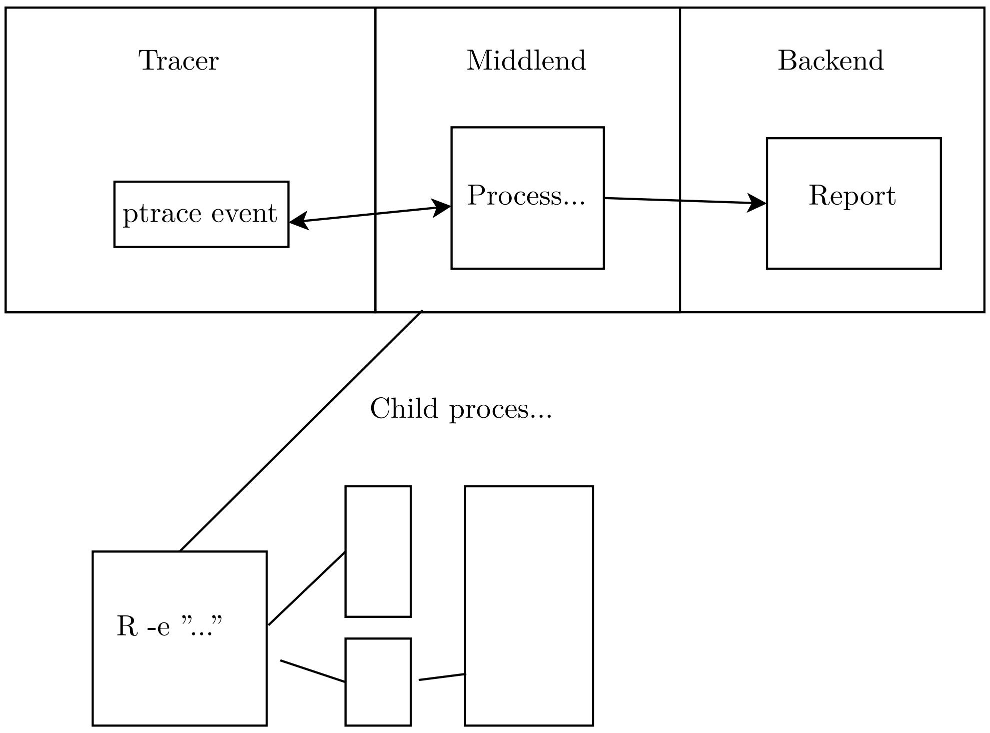


Figure 5.1: *A diagram showing the communication between different portions of the tool and the traced program*

5.4 Tracer - FrontEnd

The only tracer implemented is a ptrace-based tracer. Alternatively, this could be replaced by piggybacking on strace and parsing its output or some other alternative mentioned in the last chapter.

If we wanted to use an LD_PRELOAD mechanism, or some form of in-R tracing the middleend would have to be replaced as well. This is due to the fact that it would operate on different layers. The API and resources passed around by the C standard library or even R are wildly different from what the kernel provides. And the middleend would have to either provide a compatibility layer for changing FILE*s to handles or even have to be expanded to account for objects which cannot be easily represented by kernel resources.

If the tool were to ever be expanded with some form of in-kernel module it is prudent for the middleend to be replaced as well. While a proof-of-concept tracing mechanism can be created that uses the existing middleend, it fails to actually utilise many of the benefits of using the kernel-space. A trivial implementation could potentially eliminate the inherent race condition in using a ptrace-based mechanism, while letting the middleend recreate much of what the kernel already does.

Using ptrace will also mean that the tracer cannot trace itself, as a program cannot have more than one tracer. CARE attempts to resolve this by emulating the behaviour of ptrace.

5.4.1 Process tracing

The root of the entire sandboxed process tree is the tracer itself. It spawns a child with its stdout and stderr redirected to predefined files. This redirection may cause differences in output as a program can detect if it is outputting to a file or an attached terminal. After the child spawns but before it executes the new image, it marks itself to be traced via PTRACE_TRACEME.

When tracing syscalls the tracer is notified twice. Once at syscall entry and once at syscall exit. While no official method of deciding whether a syscall was just entered or exited, in my testing the return value from the syscall is the error code *-ENOSYS*[23]. This can be used alongside the heuristic of whether or not a syscall has been entered so far to detect which part of the syscall the tracee is in.

But these are not the only values which have to be caught. If a process is stopped via a signal, that has to be handled as well. *Strace* has a very nice document on which cases should be handled for a complex enough program[24]. Many of these quirks are not handled correctly by my program. I assume that no process will suddenly die due to a signal, nor that a thread group will terminate without first terminating individual children. While the system may handle such cases it has not been tested.

The very first issue arises from the waiting for child processes. As in my testing, any non-trivial program ends up spawning at least one another program.

As even the most trivial invocation of R is actually an interpreted bash script which sets up the R environment, it is necessary to correctly handle at least the process forking properly. This means that multiple processes have to be handled.

5.4.2 Multi process handling

While one of the variations of the wait function[25,26] can be easily used to wait for any child process, this results in an issue.

Child processes can be marked as traced right away with the correct ptrace flags. But the wait function cannot wait for just one TID, as this would break the tracer in case of blocking system calls. This means that the state of each

[23]https://man7.org/linux/man-pages/man3/errno.3.html
[24]https://github.com/strace/strace/blob/master/doc/README-linux-ptrace
[25]https://www.man7.org/linux/man-pages/man2/wait4.2.html
[26]https://www.man7.org/linux/man-pages/man2/wait.2.html

thread has to be handled separately — including possible syscall arguments and such.

Currently, there is no way to tell which process spawned another process.

One option is that forking syscalls such as clone[27] or fork return the new pid on exit. But the clone call may be slower than the child spawning process. The child process will notify the tracer that it had been spawned before the parent ever returns. Or even worse, if the *CLONE_VFORK* flag is used, the parent call will not return until the child has terminated.

The creating process has to be known for middleend purposes. As of now the system keeps track of all processes which are currently in some form of a clone call and have not been assigned a child thread and makes a guess at which process is the real parent.

There is no need to nonblocking wait for any other events to try and seek which process may have spawned the thread — such a thread could not have entered the syscall yet.

But how should syscalls themselves be handled?

5.4.3 Syscall number mapping

Before I get to the actual details of what needs to be handled about a syscall, a different question has to be answered. How are syscalls mapped to actual functions in the kernel? The specifics of the API and ABI may differ wildly depending on the operating system used. Basically any trap which causes execution of code in kernelspace could be used. Apparently, Windows used invalid instruction errors at one point in time.[55] The syscall man page[28] mentions a lot of different options for executing a trap.

They all have one thing in common: a syscall number is passed into the kernel in some way. On the x64 it gets passed in the RAX register and at least for the purposes of ptrace is stored into the ORIG_RAX value. This number has to be mapped to a handler. The handler is cannot be directly bound to a number directly as the numbers may vary between architectures.

Due to the possible ABI differences, the ABI-specific functionality is hidden away in a single file that could be quite easily replaced.

To generate the mappings, kernel headers are used and all references are done using *SYS_<syscall name>* macros. To get a complete list which could potentially be handled, a dump was created using the *ausyscall*[29] tool. When compiling on another system, the names of syscall may get mismatched but the kernel headers therein should be correct and the correct handler should be called.

[27]https://www.man7.org/linux/man-pages/man2/clone.2.html
[28]https://www.man7.org/linux/man-pages/man2/syscall.2.html
[29]https://linux.die.net/man/8/ausyscall

5.4.4 Syscall handling

When faced with handling of syscalls a design choice had to be made. How shall the tracers syscall handlers be described?

While functionally the same, there are multiple approaches which can be taken, but they all end up in the same manner. I need an entry handler, an exit handler, and a way to pass a variable structure between the exit and entry. As a bonus, I also added a logging handler for mostly development and debugging purposes.

While an argument could be made that there is no need for an explicit entry handler — we could just store the register values at function entry and restore the arguments at exit. This is not quite applicable as some syscalls will block in the syscall or will never return.

Most syscalls are only parsed and the relevant parts are sent to the middleend for further handling.

This essentially means that we need to either solve this using virtual dispatch or using a very large switch-case. The virtual dispatch turns out to contain a switch-case in and of itself in the form of actually creating relevant objects.

My solution consists of a generic handler used for dynamic dispatch and a template which is specialised for each relevant syscall. The template itself and the virtual interface don't actually enforce the relation, it is merely convention.

To not have to specify a huge switch-case manually the object creation is handled using a c++ std::unordered_map based solution. Unless inserted into the map, the mapper will create some form of a default handler. This is shown on the figure 5.2

This solution creates a unique_ptr. If I wanted to avoid allocating memory, the benefits of using a template-based solution appear. While C++ does not contain any reflection as of now, the necessary parts can be emulated using the correct API. Since all syscall handlers are templates with a numeric key and the key is mostly continuous with only a couple hundred options, this can be looped through by detecting that a specialisation exists. And for all existing template specialisations a handler can be inserted into the map. This does, however, require all the relevant headers to be included and not error will be printed if a new handler is added but not registered for this map. The relevant portion of teh code is shown in the figure 5.4.4.

Since we can loop through all known handlers, an in-language code generator which would convert this solution to a large switch-case can be later created. The virtual dispatch could be removed entirely, but performance testing is needed.

The loop could also be used to remove the dynamic allocation. Since all the classes are known, along with their needed space and alignment, a constant buffer could be created in the process state, which would serve for this purpose.

```
struct SyscallHandler{
        virtual ~SyscallHandler() = default;
        ...the actual API
}

template<size_t syscallNr>
class TSyscallHandler;


//Handler for the Open syscall
template<>
struct TSyscallHandler<SYS_open> : public OpenHandler{};

inline std::unique_ptr<SyscallHandler> createNullOpt() {
  return std::make_unique<ErrorHandler>();
}

struct Mapper {
  std::unordered_map<int, decltype(createNullOpt)*> map;
  std::unique_ptr<SyscallHandler>
   get(decltype(map)::key_type syscallNr) const {
    auto it = map.find(syscallNr);
    if (it != map.end()) {
      return (it->second)();
    }
    else {
      return createNullOpt();
    }
  }
};
```

Figure 5.2: *How are the system call handlers initialised*

```
template<int nr = MaxSyscallNr>
void filler_desc(decltype(Mapper::map)& map) {
  if constexpr (IS_COMPLETE(TSyscallHandler<nr>)) {
    map.emplace(nr, [](){
    return std::make_unique<TSyscallHandler<nr>>();
    });
  }
  if constexpr (nr < 0) {
    return;
  }
  else {
    filler_desc<nr - 1>(map);
  }
};
```

Figure 5.3: *Utilising template overloads for registration of system call handlers.*

This would not remove all memory allocations in the handlers, as they often allocate in and of themselves, but would remove the most obvious one.

It is also interesting that many system calls end up being handled by the very same handler with different flags. For example, fork/vfork and clone are different system calls which all translate into a clone call.

5.4.5 Syscall logging

Due to the nature of this program, we keep a track of all operations on handles of any kind. This means that, unlike strace, this program can output known, translated values. This is especially interesting for writes reads and others. But even open is really internally calling openat which provides an optional working directory other than the current working directory. This allows me to provide more human-readable output of the actual semantics of a given program. A portion of sample output is listed in the figure 5.4.5

5.4.6 Shared libraries

One of the main reasons for choosing ptrace as the prototype was that it should be able to trace any used library. But will it actually detect that a shared library is being loaded?

As it turns out, yes, it will. For executing a program, the kernel maps the executed image into memory along with vDSO[30] objects (one of those is the dynamic linker[31]). And the dynamic linker then proceeds to use the

[30] https://www.man7.org/linux/man-pages/man7/vdso.7.html
[31] https://www.man7.org/linux/man-pages/man8/ld.so.8.html

```
newfstatat(/home/adamep/R/.../Meta/features.rds,"",-167648) = 0
read(/home/adamep/R/.../Meta/features.rds) = 123
read(/home/adamep/R/.../Meta/features.rds) = 0
lseek(/home/adamep/R/.../Meta/features.rds) = 123
read(/home/adamep/R/.../Meta/features.rds) = 0
close(/home/adamep/R/.../Meta/features.rds) = 0
newfstatat(AT_FDCWD,"/home/adamep/R/.../libs",-156096) = -2
readlink(AT_FDCWD,"/home") = -22
```

Figure 5.4: *A small portion of logged system calls when the tool is created with logging enabled.*

standard syscall API to load further shared libraries into the process space. This means that the dynamic linker needs to be detected and listed as a dependency explicitly in some other manner.

5.5 Middleend

The middleend is essentially a kernel emulator combined with a logger.

The initial intention was for it to only serve as a logger but the emulation functionality was iteratively added as it was needed for individual syscall handling. A large part of the emulation could be instead replaced by reading the information off of the procfs.

5.5.1 The emulator

The emulating portion of the tool. For a finished product as much of this functionality should be deferred to the operating system itself to prevent bugs which come up from incorrectly mirroring the kernel. But due to the separation of logic, the changes should only be necessary in the middleend. If anything, such changes would result in some syscalls no longer needing a handler.

The fact that I am mirroring the OS goes against what the literature [41] recommends as the correct course of action. Changing the query to what the working directory and other status information are should be trivial, but there are portions of the emulator which cannot be easily removed while using the syscall API.

File creation

The open syscall [30] can either open an existing file or create a new one. But unless a flag is specified that a file should only be created, there is no way to know what happened from the outside world. This means that each open call

— whether it eventually fails or not — has to check if a file with the given name exists.

The issue arises from path resolution. I have to resolve the opened path in the same manner the kernel would. This will inherently be a race condition, as another thread could change a directory in the path to a symlink in the meantime.

Another issue are the semantics. Different access rights are considered when creating a new file and opening an existing one. By creating it the file can even be opened with different access rights than are actually granted to the created file.

File descriptors

The path resolution may be started from a file descriptor instead of the working directory. This leads to a need to keep track of all open files. and which values they are pointing to. But these open file descriptors may be manipulated in other ways, duplicated or otherwise overwritten.

By emulating this behaviour instead of falling back to asking the kernel about a particular file descriptor, many unrelated and unnecessary syscalls have to be handled.

The behaviour is not even properly cloned right now as I do not respect the close on exec flag which can be set on file descriptors. This means that I may resolve a file descriptor which has already been closed and will only know so because the associated traced system call fails.

Sockets can also be used to share file descriptors between two processes. This would require complex knowledge of both processes using a unix socket and emulating what the kernel does to share the file descriptor. This is currently not resolved.

Path resolution

When opening any relative path, the path must be appended to the base and resolved. This may happen in just about any function operating on the filesystem. For simplicity I use algorithms provided by the c++ filesystem library for all my path resolution needs.

This means that the kernel behaviour may not be perfectly replicated.

Long-term, an algorithm of this sort will still *need* to remain in the middleend due to logging needs, which are covered later.

Process information

Currently I also keep track of the current working directory of all processes. This should be extended to include information about the user and group IDs which the process may use for access rights.

Process execution

Another vexing issue are the actual semantics of file execution resolution.

Execution of a program has a couple of issues. The relevant syscall — *execve*/*execveat* — simply provides a path to the executable in some manner. But the executed program may do different things depending on the format of the actual executable. I only ever see that the program intends to execute a file from a specified path. But the executable itself could either be an actual binary or it could be an interpreted script instead. This means that the interpreter would not be detected at all. The only way to truly know which way a program will be run is to manually open the files themselves and check if they begin with a shebang[32] or not.

This means that I have to emulate the behaviour of the kernel with this call and if relative paths are used, the path resolution algorithm needs to be implemented as well.

The actual executed loader could differ due to recognised custom binary format[33], though this is currently not handled in any way.

Furthermore, somewhere in the process initialisation process every program seems to call the *getrandom* syscall. I am unsure if this is to initialise some internal state or a part of the ASLR but it means that no two processes will ever be the same unless this is spoofed.

5.5.2 The logger

The actual portion of the middleend which logs all file accesses and other dependencies. This is the portion of the library which will very likely see little to no change in the logic behind it if the frontend were ever replaced. But the API semantics of data tracking are built around the syscall API.

FS unrelated information

Aside from the filesystem dependencies, the environment within which the process is actually run should also be retained. The various IDs the process is associated with, the environment variables, the arguments with which the program is invoked, and the relevant devices the program works with.

So far, the environment variables which would be passed to the *execve* call are stored by the tracer along with the arguments. The rest are unhandled.

File interactions

For each file I track, separately from individual file descriptors, the following information:

[32]https://en.wikipedia.org/wiki/Shebang_%28Unix%29
[33]https://docs.kernel.org/admin-guide/binfmt-misc.html

- Is the file currently on the disk
- Was the file initially on the disk
- Was the file ever deleted
- Was the file ever created
- The type of the file
- For directories — Are the contents necessary

The files are identified by their absolute path on the disk. Floating file descriptors for files which are no longer on the disk are not considered so far. I have not yet made any effort to merge this tracked information, as that would require decoupling the logic behind files and locations on the filesystem.

To ensure that no symlinks are missed, I log all accesses to a given file alongside their paths and required permissions.

A proper solution would require creating a proper graph of the filesystem and emulating the path resolution algorithm on the local graph. But the graph would require two sets, one which mirrors the initial state, which needs to be reconstructed, and another that mirrors the current state.

Note that due to the iterative implementation of this tool, this will not correctly handle cases where rename[34] was called on a pair of directories with one being a working directory or other edge cases as I have yet to encounter this syscall actually utilised.

As of now, no mechanism for enforcing a file will be stored for reproduction before it is modified exists. This could either be done upon the entry to any syscall which could potentially change its contents or a syscall which would result in opening a file descriptor which would allow for modifications of the file. Or a combination of both for calls such as unlink.

5.6 Backend

The goal of the backend is to create a report on all the dependencies a program has. It can also be used to generate a reproducible environment in which the tool can be rerun. It only ever depends on the information provided to it by the middleend, it will never try to query the frontend directly.

The distribution-specific package managers have been chosen because that is the distribution I have on hand, not for any other reason.

[34]https://linux.die.net/man/2/rename

5.6.1 Package dependencies

Many files with which a program interacts may not actually be files created by the user. They may be well-know executables downloaded from the Internet. If we were able to figure out how to download a file, the dependency could be moved from only files to packages. A public package is guaranteed to not contain user sensitive data. Moreover, a dependency will probably be on a package rather than a single file from a package. This means that a larger part of unexplored code paths would be retained.

If this is chosen as the way forward, a method which checks that the hash of all file dependencies matches would be desirable, to prevent accidental file mismatch due to incorrect versioning.

I could miss some relevant configuration file that was modified by the user. But the modification is not detected in any way. Only the original version is downloaded. Such issues would be pointed out by the hash checks.

Debian package manager(dpkg)

For finding files in package repositories, dpkg provides the -S flag.[35]

Unfortunately, finding the corresponding package is never as simple as making a query on the absolute path of a file. Sometimes, the file is actually symlinked from elsewhere. Or the file is a hardlink to another location. A typical usecase for this is with libraries. For example libtinfo.so.6 is a simlink to libtinfo.so.6.3. This occurs with just about any library. Since programs rely on the shorter name (libtinfo.so.6), a solution was created.

The developed solution is slow making this one of the longest running parts of the backend. While using native libraries to speed this up may be possible, the documentation[36] is atrocious, and I was unable to find the functionality I needed.

Finding the potential package owner is done in three queries. They support some internal wildcards and are done in the form

- <fullpath>
- */<filename>
- */<filename>.*

To ensure the correct file was found, I match the inode numbers of the files made with the query with the inode number of the file I wanted to find. This will resolve both symbolic and hard links. Though if the files were on different filesystems, the match could result in a false-positive.

This is sufficient to give me a package which I depend on. It does not give me a way to install the package.

[35]https://www.man7.org/linux/man-pages/man1/dpkg-query.1.html

[36]https://www.dpkg.org/doc/libdpkg/

For package installation I depend on apt.

Advanced Package Tool(apt)

Just having a package version from *dpkg* is not sufficient to be able to install a package. The correct remote repository is also needed. In short, *apt-cache policy <packageName>* is invoked and parsed to see which remote contains the specified package.

Sometimes a package has no specified remote. In such a case I fall back to a version which does and warn the user. I have yet to encounter a package which was installed directly and has no remote. Such a package would require the user to provide a way to install the package directly or should fall back to simply copying the required files.

But just having the list of remotes is not sufficient. The remotes listed from this command are in a different format than the configuration needed to install a repository. I sidestep this issue by matching lines in the local configuration with the required remote.

If a package repository requires specific gpg keys they also need to be transferred. As of now, I only hardwired the R repository key and left the rest for the future. They need to be somehow matched from the local store to the repository and then transferred.

R packages

R packages are mostly independent of the system packages. Some packages are installed by default along with the executable, but detecting the version is still necessary.

Resolving just the packages the program was directly dependent on is insufficient. For a package to be installed, the dependencies required may be significantly different from the runtime dependencies, since I hit a code path which does not require them.

To decide if a file belongs to an R package, I replicated the logic R uses to validate a library before it is loaded. Specifically, R checks for a presence of specific file in a predefined subfolder.

Invoking an R program, which dumps relevant information about the package, including the buildtime and runtime dependencies, is done.

Then I search all directories containing R packages, which I have detected during the program runtime, for missing packages.

Finally, when I have found all packages and their versions, I do a topological sort of them, to determine the order in which they should be installed as I need to guarantee the versions will match. Finally, using the *remotes* R package, I install the necessary version.

Furthermore, since R packages are often compiled directly on the target machine, any hash-based verification will likely fail for the compiled code even if the source is the same. This will have to be taken into account when adding such a check.

5.6.2 Sandboxes

The sandboxes serve two purposes. For one, it will allow the program output to be reproduced on another system. It will also serve as a guarantee that the report contains all the necessary dependencies.

Chroot

While chroot is not a good security mechanism, it still has its uses. Namely, when attempting to make a clean build of a software package or testing out upgrades of packages without affecting the system itself. [56]

By copying or linking all the relevant files to a new directory structure, we can test that the given program will still run properly when chrooted into the directory. This approach is harder to scale when attempting to transfer files to a completely unrelated device.

Docker Image

Another technology that could be used to create an environment is Docker[37]. This is the method I ended up actually properly implementing.

This technology is based on having a base OS image, which is incrementally modified by different commands. The current OS image this is built upon is *ubuntu:22.04* as that mirrors my development environment.

This provides a baseline system which can then be modified in more flexible ways than only by overwriting files. It also gives the-end user more control and semantics over how and why individual files were transferred.

[37]https://www.docker.com/

Chapter 6

Result assessment

There are two important factors which need to be tested for the tool.

The actual correctness of the tool. How well is the tool implemented, what sorts of programs was it tested on, how confident can we be that the tool actually detects dependencies? And if the tool can be create a reproducible environment at all.

The other is the time overhead. How much extra time does the tool take to allow a program to run.

And lastly a short consideration is made of the usefulness of the evaluated results.

6.1 Correctness

Most programs I have tested the tool were reproduced in the docker environment with some caveats. Mostly that dependencies need to be installed using the package manager as some nuances were missed.

6.1.1 Known issues

When properly testing the tool with various inputs a lot of issues were encountered. Follows a list of the currently unresolved issues.

Directories that have only been listed do not retain their contents. That means a simple command like *ls /tmp* will not reproduce the contents of the *tmp* directory. It will only ensure the */tmp* directory exists. While specific system calls exist for reading the contents of directories, they may not be used. Opening a file descriptor of a directory and reading from it will proceed to list the contents of a given directory.

Passing of some signals is not correctly handled. Even a simple *sudo whoami* results in the program hanging forever as sudo is not properly notified of the child termination. The *whoami* program is left in a zombie state while *sudo* is blocked in a poll call. *Strace* does not suffer from this issue.

The *ioctl* system call is not always handled correctly. Since the call changes its beaviour based on the type of a passed file descriptor, some things are missed for example the variant which operates on terminals[38] can create a file descriptor which will currently not be known at all.

Suid flags are not respected. Due to the interaction with ptrace, a call to an suid program will result in the suid flags being discarded. Sudo will result in the error *'sudo: effective uid is not 0, is /usr/bin/sudo on a file system with the 'nosuid' option set or an NFS file system without root privileges?'*. This works differently if the tracer itself is run with root privileges a call such as *su adamep -c whoami* will execute correctly.

The docker build script depends on actually having permissions to all the relevant files and the files not being device files. It simply copies them over. A call to *su* requires access to */etc/shadow* which should not be shared for obvious reasons. But a lack of correct user configuration means that programs may fail to reproduce properly.

Symbolic and hard links are not reproduced at all. This would break a lot of programs and indeed it does. Thankfully, these are mostly used in package configurations. When package files are installed as packages, this has not yet resulted in any issues.

No effort has been made to retain modification times and the exact access rights on the transferred files.

Few package-related issues have been found. For one, if a package has no remote repository, it will be listed with the source as */var/lib/dpkg/status* and most likely need to be installed. This prevents the docker image from being built.

6.1.2 What works

As it turns out, links are very often used and so is depending on checking of file or folder existence through directories.

Even R libraries are detected based on the existence of specific files and only partially based on contents of actual files. When trying to transfer packages based on accessed files, the packages will miss a couple of files the contents of which are not actually important and have issues loading.

This means that R programs are not currently reproducible unless the packages are actually installed rather than just the opened files copied over.

Many simple unix utilities, programs and shells work fine. The process to build a working version of this thesis has been successfully reproduced. Not the tool, however.

I tested various Rmarkdown programs taken from the documentation of various R libraries. I also tested a machine learning-based project from Kag-

[38] https://www.man7.org/linux/man-pages/man2/ioctl_tty.2.html

type	user	system	real
traced	2:05	33:10	27:01
plain	0:26	0:11	0:25
overhead	380%	17990%	6384%

Figure 6.1: *Average runtime in seconds of a sample program taken from Kaggle when traced and untraced.*

gle[39] which analyses data of passengers of Titanic. With the package-based detection of dependencies, these could all be reproduced.

6.2 Time overhead

Currently, the tool is very unoptimised. The time utility was used to determine the run-time of a program which was compiled not to do any post-analysis or reporting. This should eliminate any overhead independent of the program execution. The actual command used was *time tracer [program] <args>*.

But as it turns out, the overhead is very input dependent. And the dependency analysis takes much longer. The Kaggle project took on average of three runs for the clean untraced portion of it. Unfortunately, when debugged the process sees a lot of variance in how long it run. I have had mixed results running while debugged with one finishing in 23 minutes and one taking over 30 minutes. This major slowdown is most certainly due to the tracer not being able to handle parallelised programs very well. Using *htop* most of the child processes are constantly either suspended or trapped — waiting for the tracer to handle them — and a single core is fully utilised. This is shown in the table 6.1.

The overhead for analysing the dependencies is even worse. The Keggle project depended on about 1400 files including temporary files which were deleted. Gathering the information about which packages they may belong to took over 20 minutes. Most of this time is spent querying dpkg if a particular file is perhaps a part of some package. The docker setup took over an hour.

The very simple toy program mentioned in the section 6.3.1 had much more reasonable results as shown in figure6.2.

6.3 Provided information

When testing the project taken from Kaggle, an alarming observation was made. And even the basic execution is troublesome, as the project directly states its dependencies, but they are incomplete. When running the project,

[39]https://www.kaggle.com/code/erikbruin/titanic-2nd-degree-families-and-majority-voting

type	user	system	real
traced	0:2.6	0:3.7	0:9.2
plain	0:2.3	0:1.0	0:3.9
overhead	13%	270%	135%

Figure 6.2: *Average runtime in seconds of a simple program for creating a histogram in pdf.*

it will error out unless at least 2 other packages are installed. This highlights just how difficult manual tracking of dependencies is.

The tested project directly used 171 different system packages and executed 11 different executables. When installing in the docker, this results in having to install 489 dependencies with some libraries already present.

And these are just the dependencies of this one run, not even of the used libraries. The R dependencies are just as large. The program directly required 106 packages — and to install 131 different packages need to be installed.

And if that weren't enough, this entire setup depends on 324 other files — 398 including temporaries. Odds are that at least one of those configuration files will differ on my system from the file on the maker's system.

Keeping a track of this manually, in a world where one dependency change can break the entire program is extremely hard.

6.3.1 A simple histogram

So a question comes up. Is this a common, normal occurrence? Will even a seemingly simple program which creates output depend on this many files and different dependencies?

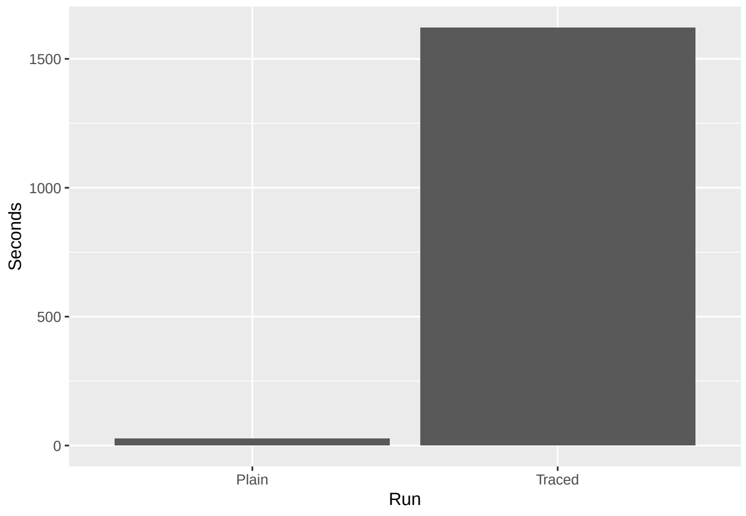
A program 6.3.1 which just prints a very simple histogram and is rendered using *rmarkdown::render(”./sampleHist.Rmd”, ”pdf_document”)* gives us more interesting statistics. The seemingly simple program renders internally using *pandoc* and *latex* to eventually end up as a pdf. This involves using 666 files, 122 system packages and 42 R packages and depends on 16 different executables including *perl*, *sed*, and *pdflatex*.

Rendering to HTML changes a large part of the dependencies, turning these modest numbers to 845 files, 158 system packages and 46 R packages.

The resulting histogram can be seen in 6.3

```
library(knitr)
library(ggplot2)
df <- data.frame(
  Run = factor( c("Plain","Traced") ),
  Seconds=c(27, 60*27+1));
ggplot(df, aes(Run, Seconds) ) +
  geom_bar(stat = "identity")
```

```
ggplot(df, aes(Run, Seconds) ) +
  geom_bar(stat = "identity")
```



1

Figure 6.3: *A simple histogram showing the difference in traced run and an untraced run of the Kaggle project.*

Chapter 7

Current limitations and possible extensions

As of now the tool tracks a basic set of filesystem dependencies and provides a framework for adding more dependencies. But it still has many limitations on what it can and cannot do. I will cover some of the potential pitfalls the tool could encounter and extensions which could be utilised.

7.1 Potentially legal issues

If the resulting environment of the tool were just simply taken and transferred to another system, a question of legality appears. Especially if it were shared openly on the Internet rather than as the user's own backup.

The sandboxed program could depend on libraries which have to be shared in a specific way depending on their licence, such as ffmpeg[40]. There is no way to guarantee that they will be transferred including all the components using just the dependency detector. The package resolver mitigates much of this issue, but will never resolve files which were not actively touched.

A way for the user to decide if files are safe to share and if any files need to be bundled along with the environment needs to be implemented.

7.2 Files which should not be shared

This raises another issue. I assume that files are safe to transfer over, but this may not always be right.

Would anyone willingly share their .ssh folder with the Internet? On the other hand, some keys will be necessary if the program depends on them.

[40]https://ffmpeg.org/legal.html

But there is no simple way to determine which files are unsafe to share. While access rights allowing only the user himself access may be a good indicator, they are not a definitive answer.

Some form of substitution for the dependencies should be added.

7.3 Device recording

In the face of communication with external devices, any form of determinism cannot be guaranteed.

This could be potentially resolved if the communication with the device were recorded and replayed in some manner. Tools such as umockdev[41] exist for this purpose. It might be possible to utilise this to allow user input into the sandboxed program.

7.4 Improvements in networking

Currently, network traffic is not handled at all. This could be extended with actual understanding of the some well-know underlying protocols.

For example, DNS resolution could allow for limiting access to just some websites. And it could determine which remote devices the program actually depends on.

Alternatively, even basic understanding of which processes are communicating over a socket is necessary for the analysis. It is required to ensure that only traced processes are communicating among themselves on the system.

7.5 Privilege handling

The tool currently does not check which user and privileges it is running with.

The tool could see more usage if it had a way to gain privileges for just the tracing portion of it, so that suid programs could be correctly traced. Currently, the sandboxed programs must manually drop their privileges rather than this being done in the tracer.

7.6 Multithreading

For simplicity's sake, the initial implementation contains no multithreading. The entire tool is purely single threaded.

This means that any traced program gets its system calls serialised and until one system call has been handled no other can begin or exit. This could be improved if a way to spread out the tracees among multiple tracers was devised.

[41]https://github.com/martinpitt/umockdev

Though testing would be needed to actually measure any performance changes.

7.7 Supporting other architectures

Currently, only the ABI for x64 is implemented. For another architecture to be supported, two things have to be considered.

If the system call API is not the same, the individual handlers may need to be modified.

The system call ABI will differ, so the way system call arguments are resolved needs to be changed. Moreover, the current way that reading strings from user tracees is implemented may break if the endianness changes as the code has not been tested.

7.8 Supporting other distributions

Right now the tool assumes Ubuntu with dpkg and apt is used. This could be extended to more distributions and package managers.

Support for R packages from sources other than CRAN could be implemented.

This is an extension touching only the backend of the tool. The other parts of the tool do not care for the specific distribution as long as the API remains the same.

Chapter 8

Conclusion

This thesis had four main goals all of which were reached.

In the chapters 1, 2, and 3 I explored the ways in which a sandbox could be implemented and summarised the potential limitations and issues. R turned out not to be possible to sandbox on a language level due to the widespread usage of native libraries and requires a more general approach.

Chapter 4 considers all the other possible methods for program tracing of which I chose to use *ptrace* to trace system calls made by all processes of a process tree. I have shown this approach to work for R programs, and many unix shell scripts and utilities, though it has many limitations as of yet as shown in chapter 6.

Because of the chosen technology, the dependency tracker is not secure and could be bypassed using a clever enough program, especially by abusing race conditions. Moreover, some parts of the system call API are not handled yet. For these reasons, the tool should not be directly used in any security-related setting.

The tool which traces the required dependencies along with the creation of a sandboxed environment is covered the chapter 5. While it is possible to create a sandbox which constrains a particular programs behaviour, tracing all dependencies is hard. As such a subset of the program was created, which traces many, but not all, of the dependencies on the filesystem.

But since dependencies often take the form of packages rather than individual files, much work was put into resolving file dependencies to package dependencies instead. Because of this, a potential sandbox or analyst has much more semantic information about the actual dependencies.

The sandbox is implemented using docker images for ease of transfer, though the actual construction is problematic.

The tracer has significant performance overhead even discounting the time it takes to analyse dependencies and create the reproducible environments, but no effort has been made in this regard and there are options for potential performance gains. Many of the possible improvements are covered in the chapter 7.

Bibliography

1. TEAM, CRAN. *Contributed Packages* [online] [visited on 2024-05-02]. Available from: `https://cran.r-project.org/web/packages/index.html`.
2. WICKHAM, Hadley; BRYAN, Jenny. *R packages* [online]. 2nd ed. Sebastopol, CA: O'Reilly Media, 2023 [visited on 2024-05-02]. Available in html format from `https://r-pkgs.org/`.
3. TEAM, CRAN. *CRAN Repository Policy* [online] [visited on 2024-05-02]. Available from: `https://cran.r-project.org/web/packages/policies.html`.
4. *R4R: Reproducible Data Analyses for All* [online]. Publications Office of the European Union, 2024-01 [visited on 2024-05-02]. Available from: `https://cordis.europa.eu/project/id/101081989`.
5. GOLDBERG, Ian; WAGNER, David; THOMAS, Randi; BREWER, Eric. A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker). In: *Proceedings of the Sixth USENIX UNIX Security Symposium*. 1996. Available also from: `https://www.usenix.org/legacy/publications/library/proceedings/sec96/full_papers/goldberg/goldberg.pdf`.
6. MAASS, Michael; SALES, Adam; CHUNG, Benjamin; SUNSHINE, Joshua. A systematic analysis of the science of sandboxing. *PeerJ Computer Science*. 2016, vol. 2, e43. Available from DOI: `10.7717/peerj-cs.43`.
7. MOSER, Andreas; KRUEGEL, Christopher; KIRDA, Engin. Exploring Multiple Execution Paths for Malware Analysis. In: *2007 IEEE Symposium on Security and Privacy (SP '07)*. 2007, pp. 231–245. Available from DOI: `10.1109/SP.2007.17`.

8. OR-MEIR, Ori; NISSIM, Nir; ELOVICI, Yuval; ROKACH, Lior. Dynamic Malware Analysis in the Modern Era—A State of the Art Survey. *ACM Comput. Surv.* 2019, vol. 52, no. 5. ISSN 0360-0300. Available from DOI: `10.1145/3329786`.
9. GOONASEKERA, Nuwan; CAELLI, William; FIDGE, Colin. LibVM: An architecture for shared library sandboxing. *Software: Practice and Experience.* 2014, vol. 45. Available from DOI: `10.1002/spe.2294`.
10. FLEURY, Nicolas; DUBRUNQUEZ, Theo; ALOUANI, Ihsen. *PDF-Malware: An Overview on Threats, Detection and Evasion Attacks* [online]. 2021 [visited on 2024-05-02]. Available from arXiv: `2107.12873 [cs.CR]`.
11. MOSER, Andreas; KRUEGEL, Christopher; KIRDA, Engin. Limits of Static Analysis for Malware Detection. In: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007).* 2007, pp. 421–430. Available from DOI: `10.1109/ACSAC.2007.21`.
12. A0RTEGA. *GitHub.* Pafish [online]. b497899 on Nov 9, 2021 [visited on 2024-05-03]. Available from: `https://github.com/a0rtega/pafish`.
13. LEON, Roee S.; KIPERBERG, Michael; LEON ZABAG, Anat Anatey; ZAIDENBERG, Nezer Jacob. Hypervisor-assisted dynamic malware analysis. *Cybersecurity.* 2021, vol. 4, no. 1, p. 19. ISSN 2523-3246. Available from DOI: `10.1186/s42400-021-00083-9`.
14. SCHULIST, Jay; BORKMANN, Daniel; STAROVOITOV, Alexei. *Linux Socket Filtering aka Berkeley Packet Filter (BPF)* [online] [visited on 2024-05-02]. Available from: `https://www.kernel.org/doc/html/v5.12/networking/filter.html`.
15. BAYER, Ulrich; KRUEGEL, Christopher; KIRDA, Engin. TTAnalyze: A Tool for Analyzing Malware. In: 2006. Available also from: `https://api.semanticscholar.org/CorpusID:11273952`.
16. KIRAT, Dhilung; VIGNA, Giovanni; KRUEGEL, Christopher. BareBox: efficient malware analysis on bare-metal. In: *Proceedings of the 27th Annual Computer Security Applications Conference.* Orlando, Florida, USA: Association for Computing Machinery, 2011, pp. 403–412. ACSAC '11. ISBN 9781450306720. Available from DOI: `10.1145/2076732.2076790`.
17. EARTHQUAKE. *GitHub.* chw00t: chroot escape tool [online]. 1fd1016 on Jun 13, 2019 [visited on 2024-05-03]. Available from: `https://github.com/earthquake/chw00t/`.
18. SILBERSCHATZ, Abraham; GALVIN, Peter; GAGNE, Greg. Chapter 14 Protection [online]. In: *Operating System Concepts, 9th edition.* John Wiley & Sons, 2013. ISBN 978-1-118-06333-0. Available also from: `https://archive.org/details/operating-system-concepts-9th-edition/`.

19. GOONASEKERA, Nuwan A.; CAELLI, William J.; SAHAMA, Tony. 50 Years of Isolation. In: *2009 Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing*. 2009, pp. 54–60. Available from DOI: 10.1109/UIC-ATC.2009.86.

20. MARTINMARINOV. *GitHub*. TempestSDR [online]. ff37de8 on Jul 4, 2023 [visited on 2024-05-03]. Available from: https://github.com/martinmarinov/TempestSDR.

21. MARTINS, José; PINTO, Sandro. Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems. In: *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2023, pp. 40–53. ISSN 2642-7346. Available from DOI: 10.1109/RTAS58335.2023.00011.

22. COKER, Zack; MAASS, Michael; DING, Tianyuan; LE GOUES, Claire; SUNSHINE, Joshua. Evaluating the Flexibility of the Java Sandbox. In: *Proceedings of the 31st Annual Computer Security Applications Conference*. Los Angeles, CA, USA: Association for Computing Machinery, 2015, pp. 1–10. ACSAC 2015. ISBN 9781450336826. Available from DOI: 10.1145/2818000.2818003.

23. TAN, Gang. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends® in Privacy and Security*. 2017, vol. 1, no. 3, pp. 137–198. Available from DOI: 10.1561/3300000013.

24. DEX4ER. *GitHub*. fakechroot [online]. b42d1fb on Feb 11, 2020 [visited on 2024-05-03]. Available from: https://github.com/dex4er/fakechroot.

25. LURE-SH. *GitHub*. fakeroot [online]. b2da39c on Oct 24, 2023 [visited on 2024-05-03]. Available from: https://github.com/lure-sh/fakeroot.

26. THE KERNEL DEVELOPMENT COMMUNITY. *Kernel Self-Protection* [online] [visited on 2024-05-04]. Available from: https://www.kernel.org/doc/html/v5.4/security/self-protection.html?highlight=kaslr.

27. KERRISK, Michael. *Linux/UNIX system programming training*. shmget(2) — Linux manual page [online]. 2023-12-22 [visited on 2024-05-03]. Available from: https://www.man7.org/linux/man-pages/man2/shmget.2.html.

28. KERRISK, Michael. *Linux/UNIX system programming training*. fstab(5) — Linux manual page [online]. 2023-12-22 [visited on 2024-05-03]. Available from: https://www.man7.org/linux/man-pages/man5/fstab.5.html.

29. KERRISK, Michael. *Linux/UNIX system programming training.* setuid(2) — Linux manual page [online]. 2023-12-22 [visited on 2024-05-03]. Available from: `https://man7.org/linux/man-pages/man2/setuid.2.html`.

30. KERRISK, Michael. *Linux/UNIX system programming training.* openat2(2) — Linux manual page [online]. 2023-12-22 [visited on 2024-05-03]. Available from: `https://www.man7.org/linux/man-pages/man2/openat2.2.html`.

31. LIPP, Moritz; SCHWARZ, Michael; GRUSS, Daniel; PRESCHER, Thomas; HAAS, Werner; FOGH, Anders; HORN, Jann; MANGARD, Stefan; KOCHER, Paul; GENKIN, Daniel; YAROM, Yuval; HAMBURG, Mike. Meltdown: Reading Kernel Memory from User Space. In: *27th USENIX Security Symposium (USENIX Security 18).* 2018.

32. KOCHER, Paul; HORN, Jann; FOGH, Anders; GENKIN, Daniel; GRUSS, Daniel; HAAS, Werner; HAMBURG, Mike; LIPP, Moritz; MANGARD, Stefan; PRESCHER, Thomas; SCHWARZ, Michael; YAROM, Yuval. Spectre Attacks: Exploiting Speculative Execution. In: *40th IEEE Symposium on Security and Privacy (S&P'19).* 2019.

33. SORENSEN, Tyler; KHLAAF, Heidy. *LeftoverLocals: Listening to LLM Responses Through Leaked GPU Local Memory.* 2024. Available from arXiv: `2401.16603 [cs.CR]`.

34. KERRISK, Michael. *Linux/UNIX system programming training.* clone(2) — Linux manual page [online]. 2023-12-22 [visited on 2024-05-03]. Available from: `https://www.man7.org/linux/man-pages/man2/openat2.2.html`.

35. LTD, Check Point Software Technologies. *Evasion techniques* [online]. © 1994-2024 [visited on 2024-05-04]. Available from: `https://evasions.checkpoint.com/`.

36. KERRISK, Michael. *Linux/UNIX system programming training.* inotify(7) — Linux manual page [online]. 2023-12-22 [visited on 2024-05-03]. Available from: `https://man7.org/linux/man-pages/man7/inotify.7.html`.

37. KERRISK, Michael. *Linux/UNIX system programming training.* fanotify(7) — Linux manual page [online]. 2023-12-22 [visited on 2024-05-04]. Available from: `https://www.man7.org/linux/man-pages/man7/fanotify.7.html`.

38. KERRISK, Michael. *Linux/UNIX system programming training.* namespaces(7) — Linux manual page [online]. 2023-12-22 [visited on 2024-05-04]. Available from: `https://www.man7.org/linux/man-pages/man7/namespaces.7.html`.

39. KERRISK, Michael. *Linux/UNIX system programming training*. cgroups - Linux control groups [online]. 2023-12-22 [visited on 2024-05-04]. Available from: `https://www.man7.org/linux/man-pages/man7/cgroups.7.html`.

40. KERRISK, Michael. *Linux/UNIX system programming training*. credentials(7) — Linux manual page [online]. 2023-12-22 [visited on 2024-05-04]. Available from: `https://man7.org/linux/man-pages/man7/credentials.7.html`.

41. GARFINKEL, Tal. Traps and pitfalls: Practical problems in system call interposition based security tools. In: *In Proc. Network and Distributed Systems Security Symposium*. 2003. Available also from: `https://cs155.stanford.edu/papers/traps.pdf`.

42. WATSON, Robert N. M. Exploiting concurrency vulnerabilities in system call wrappers. In: *Proceedings of the First USENIX Workshop on Offensive Technologies*. Boston, MA: USENIX Association, 2007. WOOT ’07.

43. THE KERNEL DEVELOPMENT COMMUNITY. *Seccomp BPF* [online] [visited on 2022-02-27]. Available from: `https://docs.kernel.org/userspace-api/seccomp_filter.html`.

44. PROJECTS, The Chromium. *Chromium Docs*. Linux Sandboxing [online] [visited on 2024-05-04]. Available from: `https://chromium.googlesource.com/chromium/src/+/HEAD/docs/linux/sandboxing.md`.

45. JIA, Jinghao; ZHU, YiFei; WILLIAMS, Dan; ARCANGELI, Andrea; CANELLA, Claudio; FRANKE, Hubertus; FELDMAN-FITZTHUM, Tobin; SKARLATOS, Dimitrios; GRUSS, Daniel; XU, Tianyin. *Programmable System Call Security with eBPF*. 2023. Available from arXiv: `2302.10366 [cs.OS]`.

46. BPFTRACE. *GitHub*. bpftrace [online]. Jan 5 2024 [visited on 2024-05-04]. Available from: `https://github.com/bpftrace/bpftrace/issues/305`.

47. LIASSICA. *archlinux*. Audit framework [online]. 12 September 2023, at 02:14 [visited on 2024-05-04]. Available from: `https://wiki.archlinux.org/index.php?title=Audit_framework&oldid=787488`.

48. LINUX-AUDIT. *Github*. audit-userspace [online]. 4939b85 on May 3, 2024 [visited on 2024-05-04]. Available from: `https://github.com/linux-audit/audit-userspace`.

49. KERRISK, Michael. *Linux/UNIX system programming training*. ptrace(2) — Linux manual page [online]. 2023-12-22 [visited on 2024-05-04]. Available from: `https://www.man7.org/linux/man-pages/man2/ptrace.2.html`.

50. WRIGHT, Chris; COWAN, Crispin; SMALLEY, Stephen S. Linux Security Module Framework. In: 2002. Available also from: `https://api.semanticscholar.org/CorpusID:7635257`.

51. LAWLER, Frederick. *Live-patching security vulnerabilities inside the Linux kernel with eBPF Linux Security Module* [online]. 06/29/2022 [visited on 2024-05-04]. Available from: `https://blog.cloudflare.com/live-patch-security-vulnerabilities-with-ebpf-lsm/`.

52. THE KERNEL DEVELOPMENT COMMUNITY. *LSM BPF Programs* [online]. (C) 2020 Google LLC [visited on 2024-05-04]. Available from: `https://docs.kernel.org/bpf/prog_lsm.html`.

53. KENISTON, Jim; PANCHAMUKHI, Prasanna S; HIRAMATSU, Masami. *Kernel Probes (Kprobes)* [online] [visited on 2024-05-04]. Available from: `https://www.kernel.org/doc/html/latest/trace/kprobes.html`.

54. DESNOYERS, Mathieu. *Using the Linux Kernel Tracepoints* [online] [visited on 2024-05-04]. Available from: `https://www.kernel.org/doc/html/latest/trace/tracepoints.html`.

55. CHEN, Raymond. *The hunt for a faster syscall trap* [online]. December 15th, 2004 [visited on 2024-05-02]. Available from: `https://devblogs.microsoft.com/oldnewthing/20041215-00/?p=37003`.

56. LAHWAACZ.BOT. *chroot* [online]. 1 February 2024, at 18:03 [visited on 2024-05-04]. Available from: `https://wiki.archlinux.org/index.php?title=Chroot&oldid=799152`.

Appendix A

Acronyms

API	Application programming interface
API	Application binary interface
TCB	Trusted code base
FS	Filesystem
PID	Process ID
TID	Thread ID
VM	Virtual machine
eBPF	extended Barkley packet filter

Appendix **B**

Contents of the attatchments

```
 readme.md ............................ the file with contents description
 Implementation...... the directory with all practical parts of this thesis
 Text .................. the thesis text and LaTeX source codes directory
   DP_Adamek_Petr_2024.pdf........... the thesis text in PDF format
```