Securing Process Execution by Verifying the Inner Process State Through Recording and Replaying on Different Platforms

Master's Thesis of

Maik Schäfer

Institute of Theoretical Informatics
Competence Center for Applied Security Technology (KASTEL)
Department of Informatics
Karlsruhe Institute of Technology

Reviewer:	Prof. Dr. Jörn Müller-Quade
	Prof. Dr. Thorsten Strufe
Advisor:	M.Sc. Felix Dörre

21.06.2021 – 21.12.2021

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text and that I have observed the statutes of the KIT to ensure good scientific practice in the respective valid version.

Karlsruhe, 14.12.2021

. . . .
Maik Schäfer

Contents

Acronyms

μarch microarchitecture.

API application programming interface.

ASLR Address Space Layout Randomization.

CEE corrupt execution errors.

CPU central processing unit.

ISA instruction set architecture.

libc C standard library.

manpage manual page.

MPC Multi Party Computation.

opcode operation code.

OS operating system.

PID process identifier.

PIE position independent executable.

protobuf protocol buffer.

rop return oriented programming.

SPERRIPS Securing Process Execution by Recording and Replaying the Inner Process State.

syscall system call.

VDSO Virtual Dynamic Shared Object.

Abstract

While computer systems are ubiquitous and prevalent in our daily lives, they are not free from bugs and misbehavior. Those can either be existent in hard- or software components and may thus influence the application and data we use on the systems. Among other, causes for bugs and misbehavior are increasing design complexity, smaller hardware fabrication sizes, or expanding software complexity. Furthermore, intentionally inserted backdoors are a conceivable scenario, too. Eventually, it requires trusting the vendors that their hard- and software components operate as expected and that they are free from bugs and backdoors.

This work introduces a novel approach for verifying the correctness of an application execution without being dependent on trusting the vendors. The approach named *Securing Process Execution by Recording and Replaying the Inner Process State (SPERRIPS)* verifies the correctness of application execution across two different systems on the abstraction level of system calls (syscalls). Therefore, the application is executed and traced on two different systems to detect possible deviations in their executions. An execution is correct and verified if it runs identical on both systems and if there are only acceptable differences in the execution. In the case of unacceptable differences, the application execution will be aborted. This work introduces acceptable and unacceptable differences on the example of the syscalls of the cat application while respecting, among others, different system environments, activated Address Space Layout Randomization, and nested structures. Potentially detected unacceptable differences indicate misbehavior, rooted in a component below the considered abstraction level of syscalls in one of the two systems. In particular, this affects either hardware components or internal operating system kernel procedures.

This thesis proposes both a conception and an implementation of SPERRIPS. The implementation has been evaluated with four different applications, namely echo, hostname, cat, and ping. It demonstrates the feasibility of the approach to successfully verify application executions' correctness and detect differences in their executions. All side-effects on application execution through intentionally inserted malicious Linux kernel modifications have been detected.

Zusammenfassung

Computersysteme, die wir alltäglich verwenden und von denen wir gewissermaßen von abhängen, sind nicht frei von Fehlern. Diese können hardware- oder softwareseitig ursächlich sein und als Konsequenz Einfluss auf die Programme und Daten haben, die wir auf den Systemen verwenden. Gründe für Fehler sind steigende Designkomplexität, kleinere Fertigungsbreiten von Hardware oder komplexer werdende Softwaremodule. Darüber hinaus können auch bewusst eingebrachte Hintertüren dafür sorgen, dass ein System sich anders verhält als erwartet. Letztendlich bleibt nur übrig den Herstellern zu vertrauen, dass ihre Soft- und Hardwareprodukte wie versprochen funktionieren und frei von Fehlern und Hintertüren sind.

Diese Arbeit stellt einen Ansatz vor, um die Korrektheit einer Anwendungsausführung zu überprüfen, ohne darauf angewiesen zu sein den Herstellern zu vertrauen. Der Ansatz namens *Securing Process Execution by Recording and Replaying the Inner Process State (SPERRIPS)* verifiziert die Korrektheit einer Anwendungsausführung über zwei Systeme hinweg auf dem Abstraktionslevel von Systemaufrufen (engl. system calls). Dazu wird die zu verifizierende Anwendung auf zwei unterschiedlichen Systemen ausgeführt und auftretende Unterschiede in den Programmausführungen inspiziert. Eine Ausführung gilt als korrekt und verifiziert, wenn sie auf beiden Systemen gleich stattfindet, indem höchstens akzeptierbare Unterschiede auftreten. Falls unakzeptierbare Unterschiede auftreten, wird die Programmausführung abgebrochen. Unter Berücksichtigung von unter anderem unterschiedlichen Systemumgebungen, Addressraumverwürfelung und verschachtelten Datenstrukturen werden in dieser Arbeit akzeptierbare und unakzeptierbare Unterschiede am Beispiel der Systemaufrufe der cat Anwendung definiert. Etwaige unakzeptierbare Unterschiede deuten auf Fehlverhalten einer der beiden Systeme hin, welches in den Komponenten unterhalb des betrachteten Abstraktionslevel von Systemaufrufen begründet ist. Dies betrifft entweder Hardwarekomponenten oder interne Abläufe im Betriebssystemkern.

Diese Arbeit liefert eine Konzeption und eine Implementierung für SPERRIPS. Die Implementierung wurde anhand der vier Anwendungen echo, hostname, cat und ping evaluiert. Dies demonstriert die Fähigkeit des Ansatzes und der Implementierung die Korrektheit von Anwendungen zu überprüfen bzw. Abweichungen in ihren Ausführungen festzustellen. In einen Linuxkern bewusst eingebautes Fehlverhalten, das zur Laufzeit zu Abweichungen in den Programmausführungen führte, wurde erfolgreich detektiert.

1. Introduction

Nowadays, computer systems are ubiquitous in every part of our lives. Either directly, by, e.g., using smartphones or desktop systems, or indirectly through web services or systems for maintaining our infrastructures. Our dependency on them grants great power to the devices based on their massive impact on our lives. However, due to steadily enhancing development, both hardware and software design's complexity increase. Consequently, one cannot easily validate the correct functioning of each involved component. Instead, it requires relying on the vendors and the integrity of their supply chains that each component works and behaves as promised. However, even major hardware components like the central processing unit (CPU) contain bugs, which may lead to unexpected or faulty behavior [1], [2]. Furthermore, a security inspection of fabricated motherboards uncovered that certain manufacturing subcontractors were infiltrated by attackers, such that the manufactured motherboards are compromised with an additional malicious chip [3]. Analyzing the chip revealed its spying and manipulating mechanisms. Even though the story's legitimacy is highly debated, it shows the potential threat of compromised supply chains and their security implications [4]. Furthermore, crucial software components like the operating system (OS) kernel might also include bugs, intentionally incorporated or not.
In short, there is no guarantee that the systems we rely on daily operate exactly as we expect or if an attacker perhaps compromised them. Our usage relies on trust.

This work researches the feasibility of a new generic approach for verifying the correctness of two application executions on potentially untrustworthy systems. To do so, an application is executed on two different systems under identical conditions on both. Our approach verifies whether both executions behave the same. Deviations in the executions could indicate a fault in either the hard or software components in one of the systems. We presume that a fault that might exist in a component of one system does not exist in the other one. For example, a bug in the CPU of System A is not present in System B, as it uses the CPU of a different series or vendor.

Different levels of abstraction can be evaluated for detecting deviations in two independent application executions [5]. Possible levels range from very low-level ones up to high-level ones. A fine-grained low-level abstraction could be comparing the order and results of

executed processor instructions by observing execution within an emulator like QEMU. On the other side, a high-level approach could implement Secure Multi Party Computation (MPC), which calculates a shared result between mutually untrustworthy parties [6]. In the context of this work, the mutual result could be the application's output. Using MPC, however, must be adapted and implemented to the application's logic. Thus, it requires domain-specific knowledge and work and hence does not express a generic approach. Another possible abstraction level, between the above mentioned, could be analyzing the instructions of a byte-code interpreted language, like Java or JavaScript [7].
The proposed approach in this work operates on the abstraction level of system calls (syscalls). Syscalls are the interface between a user-land application and the operating system (OS) kernel. We assume that syscalls are significant to indicate whether two program executions behave the same if and only if their order, input arguments, and return values[1] are equal. It originates from the observation that *a deterministic algorithm with the same input leads to the same output.* We consider syscalls to be the only possible source for non-determinism for an application. Hence, two exact copies of an application must behave the same on two different systems if the order and execution of all syscalls are the same.

The work is structured as follows. Chapter 2 explains the technical fundamentals on which it is based. It explains the relevant details of syscalls and Linux-specific OS details. Further, it points out cases of bugs in the Linux kernel and on the microarchitectural level. Chapter 3 defines the goal and scope of this work. The succeeding Chapter 4 introduces our approach on a conceptual level and elaborates on solutions for non-trivial practical challenges in more depth. Chapter 5 presents and discusses selected aspects of the actual implementation. In Chapter 6, the proposed approach is evaluated in realistic scenarios. It is tested against a maliciously manipulated Linux kernel to prove the approach's feasibility. Related work is presented in Chapter 7. Finally, Chapter 8 summarizes the results and suggests aspects for future work.

[1]Arguments, which are modified by the syscall as the result of an operation, are considered to be a return value in this context.

2. Background

This chapter introduces the fundamentals of an operating system kernel, which are involved in executing applications on modern computers and are relevant to this work. Furthermore, it gives an overview of discovered flaws in the Linux kernel and processors and points out their relevance in a security context.

2.1 Operating System

The central component of an operating system is its kernel. This software component manages all relevant operations required for a functional system [8]. It provides an abstraction to the applications for, e.g., hardware access, memory management, thread scheduling, and access control. The kernel code is executed on the CPU with higher privileges than regular applications. This divides code execution in either running in kernel-mode with high privileges or in user-mode with lower privileges [8]. Hence, to enable a user application to perform privileged operations or functions, it must interact with the kernel through a dedicated interface called "system calls".

2.1.1 System Calls

System calls are the interface between user-mode applications and the kernel for initiating privileged operations from user-space like file access or network connections [8]. They abstract implementation details such as file system implementations, hardware access, and access control. As of now, the Linux kernel version 5.15.1 has more than 350 different syscalls implemented[1] . On x86 64bit architectures (x86-64), a system call is invoked through the special processor instruction `syscall`[2]. It calls the appropriate syscall handler of the kernel directly with high privileges on the CPU [9]. Compared to previous syscall invocation methods, this is significantly faster as it does not require "expensive interrupts" [11].
A unique operation code (opcode) identifies each syscall[1]. On Linux x86-64 systems, it is passed on invocation of the `syscall` instruction via the `RAX` register and thereby tells

[1]https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/x86/entry/syscalls/syscall_64.tbl?h=v5.15.1

[2]The instruction set architecture (ISA) of Intel and AMD slightly differs in this regard [9] [10]. However, in 64-bit mode, both support the `syscall` instruction.

the kernel, which system call operation is requested by the application [12]. Then, the kernel operates with its kernel privileges to execute the syscall's functionality, e.g., reading content from a file. According to the system's calling convention, the application can pass up to six arguments to a syscall.
Syscall arguments are of different types, depending on their purpose. The `read` syscall[3], for example, expects three arguments: A file descriptor (integer type), an address to a buffer, and the number of bytes, to read[4]. In this case, the buffer pointed to by the given address can be interpreted as an indirect return value, as the buffer gets filled by the kernel as a result of the read operation. However, despite indirect return values, each syscall returns a programmatically return value via the `RAX` register. In the case of read, it is the number of actual read bytes.
Linux syscalls are well documented on their manpage entries, which are available through the terminal application man[5] via `man 2 <syscall-name>` or online via the online manpages [13]. A manual page (manpage) entry includes the syscall's signature, purpose, return value, and error codes. If not stated otherwise, all syscall signatures given in this work are referred from the manpages.

Frequent syscall invocations affect the system's performance, as they cause frequent context switches and consequently slow down the system's performance. Linux mitigates this by providing the Virtual Dynamic Shared Object (VDSO). It is a shared library mapped by the kernel into a process' address space [14]. Through the VDSO, certain syscalls can be executed directly, instead of invoking an actual syscall since the required code is mapped as read-only into user-space. In particular, on x86-64 systems the syscalls `clock_gettime`, `getcpu`, `gettimeofday` and `time` are provided via the VDSO.

2.1.2 Calling Conventions

A calling convention defines how arguments are passed from the caller to the callee. The term caller refers to a piece of code that calls a function, while the function that has been called is denoted as the callee. The calling convention applies both to function calls in user-mode and to syscalls. Hence, compilers have to ensure that the generated code meets the requirements of the target system.
This work focuses on the System-V calling convention, used by the Linux kernel on x86-64 systems [12]. With System-V, the first six arguments of a called function are passed via processor registers in the defined order of `RDI, RSI, RDX, RCX, R8, R9`. Any more arguments must be stored on the stack, where the callee reads them from. Finally, the callee writes the return value into the `RAX` register. However, the System-V specification restricts the number of arguments for syscalls to at most six arguments.

The convention of passing the syscall's opcode and its return value via the `RAX` register on a `syscall` instruction is also a definition by the System-V standard. Furthermore, the convention for calling the kernel interface and thus affecting the `syscall` instructions differs from the previous register order as instead of `RCX` the `R10` register is used.

2.1.3 Strace and Ptrace

While debugging and analyzing an application, it can be helpful to track and evaluate the syscalls which an application called. The command-line utility strace traces and prints all

[3]`ssize_t read(int fd, void *buf, size_t count)`
[4]The `size_t` type is in fact an integer
[5]If `man-db` is installed.

system calls of a given process or program. It uses the Linux kernel's interface named *ptrace*, which enables tracing and manipulating other processes and is implemented as a syscall itself. In this context, the observing process is referred to as "tracer" and the observed process as "tracee". The ptrace mechanism equips the tracer with primitives for intercepting process execution on signals or syscalls and reading and writing into the tracee's virtual address space or processor registers. Hence, ptrace not only enables passively observing the tracee's execution but also actively manipulating data and altering the control flow.
With the aid of ptrace, strace traces each issued syscall of a given application or process and displays its name, including the arguments and the resulted return value in a human-readable fashion.

```
# strace echo Hello world!
execve("/usr/bin/echo", ["echo", "Hello", "world!"], 0x7fffffffe500) = 0
brk(NULL) = 0x55555555f000
[..]
mmap(0x7ffff7df2000, 1540096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|
    MAP_DENYWRITE, 3, 0x25000) = 0x7ffff7df2000
mmap(0x7ffff7f6a000, 303104, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
     0x19d000) = 0x7ffff7f6a000
[..]
write(1, "Hello world!\n", 13Hello world!) = 13
close(1) = 0
[..]
```

Listing 2.1: Examplary output of strace.

Listing 2.1 gives a shortened output of `strace echo Hello world!` where the names of some system calls and their arguments and return values are printed. If possible, it prints the names of argument flags rather than their numerical values for easing understandability, as in the case of `PROT_READ|PROT_EXEC`.

2.1.4 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) is one of the various techniques that modern OSs implement to prevent common exploiting strategies. The purpose of Address Space Layout Randomization (ASLR) is to randomize the address offsets of memory areas of shared libraries, file mappings, and necessary data structures[6] within an application's address space. By doing so, it is harder for an attacker to know or guess the exact position of specific code or data areas, which he tries to leverage for attacks. ASLR mitigates return oriented programming (rop) attacks, which make use of the already existing application or library code within executable memory regions. To successfully run a rop attack, an attacker chains pieces of an application or library code (called gadgets) such that their chained execution implements the exploit logic. However, the attacker needs to know the exact memory addresses to chain the gadgets. Randomizing the address space layout makes it harder to know or guess these addresses.
With activated ASLR, the Linux kernel randomizes the mapping addresses of the memory areas mentioned above on each application execution with fresh randomness. ASLR is enabled by default in the Linux kernel[7] and was introduced with Linux 2.6.12 in

[6]In particular, the global offset table (GOT).

[7]https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/admin-guide/kernel-parameters.txt?h=v5.15.1

2005 [15]. However, it can be disabled by providing the `norandmaps` kernel boot parameter or by writing `0` into `/proc/sys/kernel/randomize_va_space` with appropriate permissions.
ASLR does not randomize the base address of an executable's code section, however. If this is desired to increase the exploit mitigation level further, executables must explicitly be compiled to become so-called position independent executables (PIEs).

2.1.5 Kernel Bugs

Like every piece of software, even the most essential part of a system, its kernel, is not free from bugs. By nature, kernel bugs can have severe consequences, ranging from privilege escalation to data loss.
Concerning syscall bugs, Bagherzadeh et al. conducted a detailed analysis of the Linux kernel syscall interface changes by analyzing the code changes [11]. They broke down the changes into different classifications, grouped by syscall type. Among others, these include bug fixes for architecture compatibility, concurrency fixes, and, most important for this work, falsely returned error code fixes and semantic bug fixes. With the term "semantic fixes", the authors refer to fixes that correct falsely implemented syscall behavior. Interestingly, among all syscalls, semantic bug fixes are the most common type, with 58% of all fixes. Unfortunately, the authors do not break down the affected syscalls per bug type. However, they reveal that the syscalls `ptrace`, `signal`, `ioctl`, `futex`, `ipc`, `mmap`, `perf_event_open`, `readdir` and `splice` retrieved the most bug fixes. The author reason the frequent changes to the ptrace syscall with its inherent implementation complexity because of its extensive tracing capabilities.
This work shows that even the crucial syscall interface is not free from bugs. Further, semantical bug fixes show that syscalls might even contain semantical bugs.

2.2 C Programming Language

Since the Linux kernel is written in the C programming language [16], this section introduces the necessary fundamentals of the language and its relations to the Linux kernel. It covers the relation between syscalls and C-wrapper functions as well as the principle of customizable data structures, called structs.

2.2.1 C Standard Library (libc)

The C standard library (libc) is the reference implementation of the C programming language specification for Linux-based OSs [17]. Its application programming interface (API) provides essential functions and libraries as a base for programming an application in C. Therefore, it is commonly dynamically loaded for applications running on Linux systems. The libc library provides wrapper functions for commonly used syscalls, such as reading and writings files or connecting to network sockets. Syscalls for which no wrapper is provided can be manually invoked with the special function `syscall`[8]. It expects the syscall's opcode as the first parameter and its arguments subsequently.
The function signature of a libc wrapper is not always identical with its underlying syscall. The wrapper function may abstract specific details, for example, in the case of the `brk` wrapper function. However, for the sake of better readability, in this work, we mainly use the syntax of the C-language wrapper function for a syscall's signature, unless noted otherwise.

[8] `syscall(long SYS_number, arguments...)`

2.2.2 C Structs

The C language supports defining custom data structure types via the `struct` language feature. A structure (struct) stores its content in a contiguous memory area, thus enabling storing and accessing related data through a single data structure. Therefore, some Linux syscalls also use pointers to structs as their arguments.

The definition of a struct consists of its name (e.g., `example`) and its embedded members and their types. Listing 2.2 shows an exemplary definition of a struct with two members. Instead of passing both the `value` and `name` values as two separate arguments to a fictive syscall, it is sufficient to give a reference to the initialized struct. Hence, the syscall is able to access the two members through the given reference (`ref->value` and `ref->name`).

```c
struct example {
    int value;
    char[8] name;
}
```

Listing 2.2: Sample definition of a struct called `example` with two members `value` and `name`.

Struct members are stored in memory in the same order as they appear in the struct's definition [18]. However, depending on the member's type size, the compiler may add additional padding for aligning the data in memory. Therefore, the size of a struct is at least the size of its member's types, plus possible additional padding. Furthermore, the layout of a struct depends on the compiler and its target platform.

2.3 Microarchitecture

The term microarchitecture (μarch) refers to the hardware design of a CPU, which implements a specific instruction set architecture (ISA). For one ISA multiple microarchitectures can exist. With the x86 platform, this is the case for different generations of a particular CPU model or CPU products by different vendors (e.g., Intel vs. AMD). To run an OS or an application on a CPU, it must be compiled for the target ISA and be executed on a corresponding CPU. Since the μarch of modern processors is a highly complex design, it is error-prone. This section introduces different kinds of errors that originate from this overwhelming complexity.

2.3.1 Bugs

The vendor Intel composes a list of known CPU bugs in their products [1]. The document uses the term "errata" which is described as "[e]rrata are design defects or errors. These may cause the processor behavior to deviate from published specifications." [1]. The most recent entry on the list is from May 2020. However, the authors state that entries are removed from the list once the product line has been updated: "Errata remain in the specification update throughout the product's lifecycle, or until a particular stepping is no longer commercially available. Under these circumstances, errata removed from the specification update are archived and available upon request."
A very popular, although old and historic bug, is the *FDIV* bug in Intel processors from the Pentium model series. In some cases, a fault in the floating-point unit caused wrong division with high-precision numbers involved. After the existence of this bug gained popularity,

Intel eventually recalled the affected processors. However, this example demonstrates that even rarely occurring bugs may exist. Depending on the context, the consequences range from simply wrong calculations to severe problems, e.g., when performing security or safety-critical operations.

Hochschild et al. published a paper concerning *corrupt execution errors (CEE)* which they discovered on some CPU cores through an internal investigation of Google's misbehaving production "massive-scale data-analysis pipeline" [2]. CEE are a result of malfunctioning CPU cores either due to errors in manufacturing processes of the highly complex and dense architecture designs or because of influences like the operating temperature, voltage, etc. However, the authors do not fully understand the root cause of CEE yet since not all cores of a processor model are equally affected. Some might yield incorrect results, while others work correctly. Indisputably, CEE are a severe problem. The authors give real-world examples where miscalculations lead to application and kernel crashes and different causes of data corruption and loss. They aim to create awareness of CEE with their paper. Furthermore, they call for more research to detect such errors early and to find solutions for dealing with current hardware affected by CEE.

With SiliFuzz, researchers from Google published an approach to fuzz test[9] x86-64 bit CPUs for electrical defects [19]. Their given title *proxy fuzzing* originates from their approach to first fuzz processor simulators to collect interesting test cases. Then, they continue by running these test cases on real hardware. Other than logical bugs, electrical bugs might or might not be present in a CPU core. Further, they might appear over time due to hardware effects like circuit aging. Even though the authors themselves assess their approach to be still in an early stage, they could find different classes of bugs on multiple CPUs on Google's huge hardware fleet. They do not give an exact number of how many bugs they found, but they state that 45% of the detected bugs are exclusively detected by SiliFuzz and were not detected by other approaches before. Their research adverts to the danger of Silent Data Corruption (SDC) and its consequences. Since it results from a faulty CPU computation, it is hard to detect soon, as it happens without notice to the processor and application. Furthermore, the authors predict more research in this field in the following years, uncovering multiple new CPU bugs.

2.3.2 Side-channel Attacks

Side-channel attacks neither exploit logical nor electrical bugs in the CPU, but they leverage microarchitectural states within the CPU to perform an attack. In the past years, multiple side-channel attacks have been part of security research. Side-channel attacks make use of the CPU's intended features, but in a way that triggers side-effects, e.g., for gaining unauthorized information. Meltdown is one of the most prevalent findings or publications [20]. It had a huge impact, as it affected almost every CPU, independent of the running operating system.

Meltdown

The Meltdown attack exploits the CPU feature *speculative execution* in combination with a cache timing attack to access memory content protected from unauthorized access [20].

[9]In short, fuzz testing (or fuzzing) feeds sophisticatedly crafted and randomly mutated data into an application or interface to observe its reaction. It aims to find faults by triggering buggy code paths through the unusual input. For security research and reliability testing, fuzzing has become an industry-standard in recent years.

Modern CPUs use speculative execution to speed up code execution by predicting which instructions will be executed next and hence execute these speculatively in advance. If the prediction is wrong or the instruction throws an exception (e.g., cause it is unauthorized), it will be discarded. Even though the results are discarded and the registers are cleared, the execution may leave other traces in the microarchitectural state. In the case of Meltdown, the CPU is trapped to fetch a byte from a protected memory location (e.g., a kernel address) and use this as an index for a probe array[10]. Due to speculative execution, this array access happens before the exception handler notices that this memory access was illegal. After the exception has been identified, the speculative execution steps are discarded. However, the array access and hence cache access already happened. The Meltdown implementation now performs a cache timing attack. It measures the access times to each entry of the probe array to find indices with fast access times. Fast access times reveal that the corresponding cache line has been accessed recently. Thereby, the indices of fast accessibly array entries are the peeked by values, as they have been used as the probe array index during speculative execution. Repeatedly flushing the cache and accessing memory sequences allows reading entire memory pages. Meltdown is independent of specific operating systems or software implementations and works on all affected CPUs.

2.3.3 Backdoors

Until today, no CPU backdoors induced by design are known to be present in actual hardware products. Therefore, research in this field primarily covers the potential feasibility of hardware backdoor conceptions and implementations. For example, Duflot presents a possible hardware backdoor implementation and studies its implications on the system's security [21]. However, this discussion is based on a self-designed hypothetical backdoor in an x86 processor.

Closest to a discovered backdoor is the discovery of an undocumented debugging feature in the C3 x86 processor model of the vendor VIA [22]. In his work, Domas describes how he found a mechanism to escalate privileges from ring 3 to ring 0, which offers the highest permissions. With the help of fuzzing the ISA, he was able to identify an undocumented x86 instruction. Domas leveraged this instruction to gain access to a particular core, which enabled to alter kernel memory and gain root permissions on an unprivileged process on a Debian 6 system. A successful exploit requires having the *god mode bit* set, which is commonly the default case. The author claims that this is the first discovered processor backdoor, as it "constitute[s] what is commonly understood as a backdoor" [22]. However, he believes that the discovered mechanisms are not maliciously included backdoors but forgotten leftovers of debugging features.

Related to CPU backdoors is the discovery of malicious chips on server motherboards [3]. Robterson and Riley describe the discovery of malicious chips found on server motherboards. An investigation revealed that the rice corn-sized chips had been placed onto the motherboards during fabrication by infiltrating the supply chain through compromised Chinese manufacturing subcontractors. These chips are attached to the board's management interface. Hence, they have broad access to the running host system and can thus alter CPU or memory data. Further, they contain memory and network modules

[10]The probe array has 256 entries, one for each possible byte value. Further, it is properly aligned to the CPU cache line length by multiplying 256 with the cache line length. Hence, each access of the form $i * cache\_line\_length$ (where i is a byte value) resolves to a different cache line.

for sending and receiving commands to neighboring spy modules or servers. At least 30 companies from the United States are estimated to be affected by thereby backdoored servers, including Amazon and Apple. However, Amazon, Apple, and Supermicro later declined the legitimacy of the story and its validity is highly debated [4]. Nonetheless, it demonstrates the potential threat of introduced hardware backdoors through supply-chain compromising.

2.4 Recording/Replaying Systems

Recording/replaying systems refer to approaches that aim to reproduce an application's execution deterministically by eliminating non-determinism. This is, e.g., desired in the field of software testing, debugging, and malware analysis as it eliminates differences in executions and thus allows recreating the same application state for analysis [23].

A particular recording/replaying system collects ("records") runtime-dependent data from the application of a specific abstraction level, commonly those that introduce non-determinism [23]. Then, the recorded data is used to alter a consecutive execution by feeding this data back into the application, denoted as "replaying". The two steps happen in two phases, namely the recording and replaying phase, in which replaying eliminates potential non-determinism from the re-executed application. However, as Rittinghaus states, replaying only guarantees to eliminate non-determinism above the chosen abstraction level [23]. Hence, depending on the replaying objectives, choosing the right abstraction level is essential for correctly eliminating non-determinism.

Research covers various recording/replaying systems on different abstraction levels, including levels on syscalls [24] [25] [26], Java Virtual Machine to system interaction [7], and CPU instructions [23]. Differences between this work and the mentioned recording/replaying systems are denoted in greater detail in Section 7.

3. Analysis

This chapter defines the goal of this work and highlights its contributions. Furthermore, it introduces the underlying trust model that we assume for our approach and the work's scope.

3.1 Goal and Contributions

This work aims to research the feasibility of a new approach for verifying the correctness of application execution. The correctness is determined by executing the same application on two different systems under identical conditions and comparing both executions. If and only if no differences or only acceptable differences during the executions occurred, we consider both executions as correct.

The program code of an application itself executes deterministic on a computer [23]. However, different influences introduce non-determinism to the execution. Cornelis et al. and Ronsse et al. identified four different sources for non-determinism of an application's execution, presented in Figure 3.1 [24], [27].
In this work, we primarily focus on non-determinism induced by system calls as they are essential for application execution. The other three aspects (network connections to shared memory) are thus beyond the scope of this work.
Therefore, we assume that two executions of the same application must be equal if and only if the order and semantical behavior of the involved syscalls are equal.

Consequently, our approach operates on the level of syscalls as illustrated in Figure 3.2 on Page 13. Each syscall invocation and return is compared to the corresponding invocation and return on the other system. If they are equal, the execution continues; otherwise, it will be aborted. Differences will be saved for later analysis.

Then, potentially detected differences in both application executions originate from components below the syscall level (see Section 2.4). Hence, this indicates either a hardware or software fault in one of the systems. Consequently, an underlying goal of this work is finding bugs or backdoors in involved components below the level of syscalls by identifying deviating results.

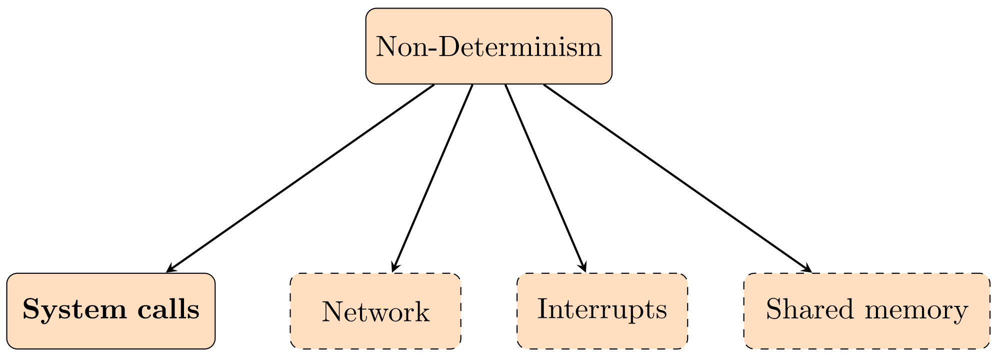

Figure 3.1 Sources of non-determinism in application execution [24] [27].

Thus, our contributions with this work are:

1. Proposing a conception for the goal of verifying application execution across two different systems on the abstraction level of system calls
2. Identifying practical issues of a conception's implementation and presenting solutions
3. Proposing an implementation of the conception
4. Evaluating the conception's feasibility

3.2 Trust Model and Scope

This section describes our trust model and the assumptions which are made within this work.

We consider two different systems on which we want to verify correct application execution. However, we expect one of both systems to behave incorrectly, as it might contain unknown bugs or backdoors in either software or hardware components. Nonetheless, we assume that a bug that might exist on one system does not exist on the other. Therefore, if our approach identified a difference between both executions, we are not able to determine on which of both systems the application was executed incorrectly. Instead, we can only identify the presence of differences in one of the components.

The scope of this work lies in the contribution to successfully applying our concept to the cat application. Therefore, supporting the syscalls of the cat application and its underlying practical challenges are primarily in focus. However, regarding the time frame, additional aspects for supporting syscalls and challenges beyond the ones of cat are also discussed.

We focus on Linux-based OSs on an x86 ISA with a 64bit architecture.

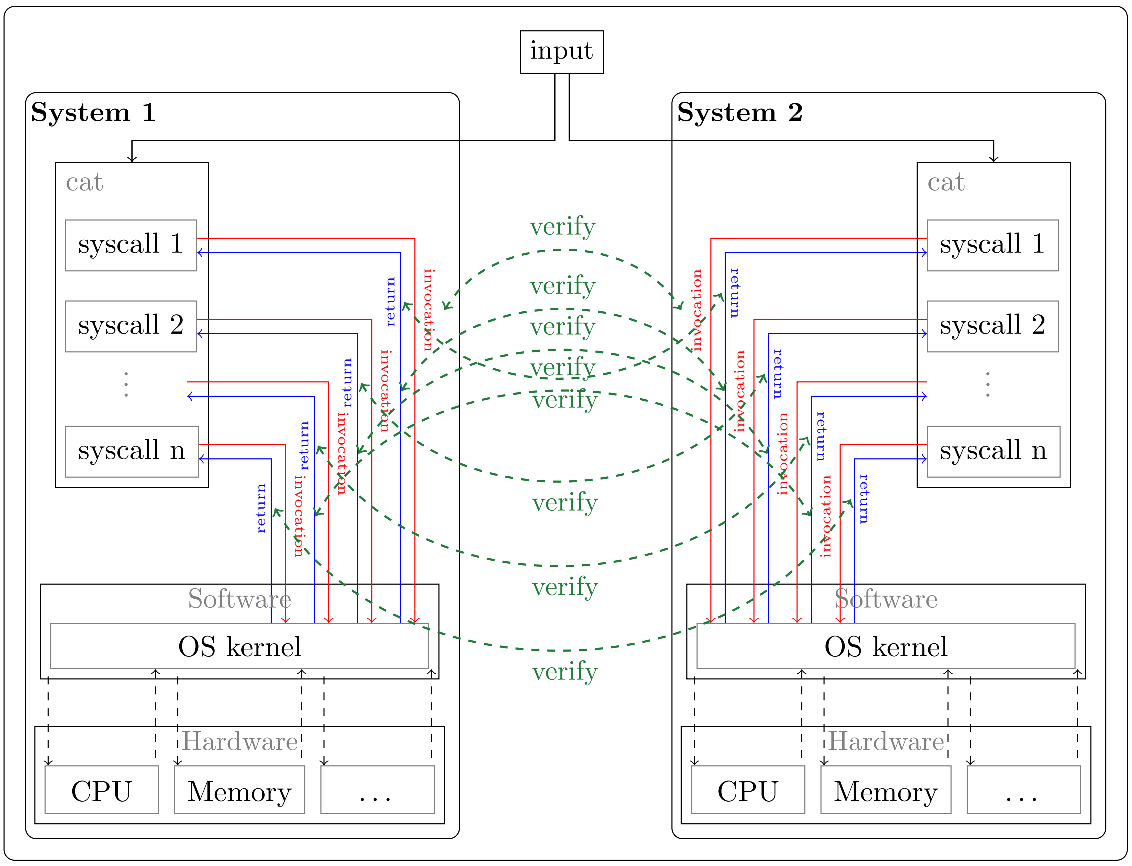

Figure 3.2 Overview of the idea of verifying application execution on the level of system calls.

4. SPERRIPS

This chapter introduces our concept named "Securing Process Execution by Recording and Replaying the Inner Process State (SPERRIPS)". Section 4.1 gives an overview of the general principle and a definition of "inner process state" as considered in this work. Then, a case study of the involved syscalls of the cat application is followed by a motivating example. The case study is used for deriving requirements for certain challenging aspects regarding the conception and implementation. Section 4.3 discusses these recognized issues in greater detail and presents solutions. However, some of the contemplated aspects in Section 4.3 go beyond cat's requirements as they also discuss additional aspects.

4.1 Concept and Overview

The SPERRIPS approach uses the abstraction level of syscalls for the verification of application executions across two different systems. It is divided into two phases, similar to common recording/replaying systems (see Section 2.4). The phases are also referred to as *recording* and *replaying* phases.
The recording phase takes place on the first system, while the replaying phase is performed on the other system. Figure 4.1 on Page 16 illustrates the recording phase on System 1 and Figure 4.2 on Page 17 the replaying phase on System 2. In both phases, an executor, which we refer to as "tracer", executes the target application, e.g., cat. The executed application, which we refer to as "tracee", is observed during its execution. In the recording phase, the tracer records the order of all invoked syscalls, together with their calling and returning argument values, and saves the data to a file (*recording.bin*). In the replaying phase, the tracer executes the tracee a second time and again traces the invoked syscalls. In contrast to the recording phase, it now uses the recorded argument values from the *recording.bin* file to compare them with those appearing in the replaying phase.

Therefore, it requires each system call to be explicitly executed on both systems rather than simply replaying recorded values. This is a fundamental difference to recording/replaying systems on the level of syscalls, which replay recorded values instead of re-executing intercepted operations. Only executing the syscalls on both systems can reveal potential differences in their execution.

In the case of an encountered difference, the differing value might be replaced with the recorded one. We refer to this operation as “replaying”. Replaying forces the tracee to operate on the recorded data and thus continue execution deterministically and equally as in the recording phase. However, only *acceptable differences* are overcome by replaying the recorded value. In case *unacceptable differences* occur, the tracer will abort the tracee’s execution, and all unacceptable differences are saved to a file (*differences.bin*). Therefore, replaying or aborting depends on the definitions of acceptable differences. We give definitions of acceptable and unacceptable differences for the execution of cat in Section 4.2.

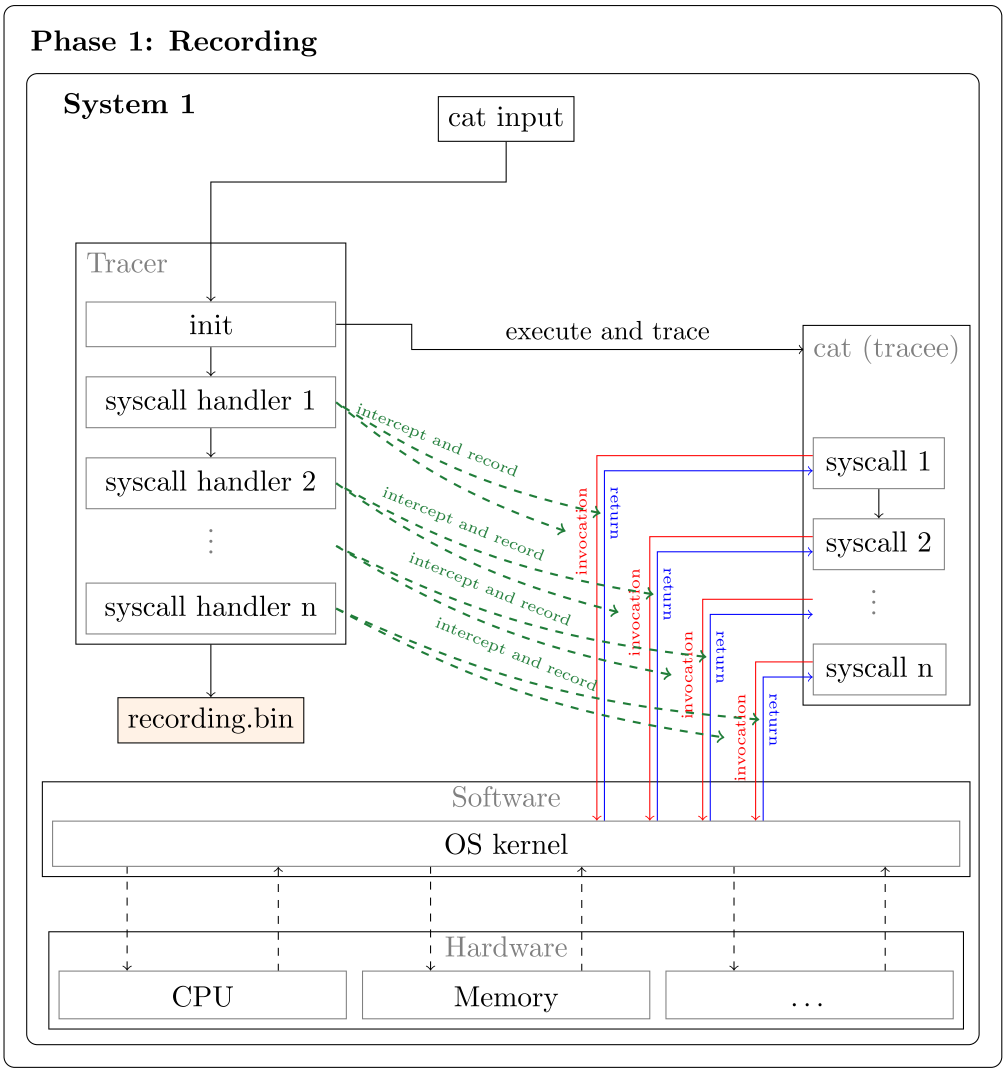

Figure 4.1 A conceptual overview of SPERRIPS recording phase.

The Inner Process State

This section introduces the term *inner process state* as used in this work and clarifies which data are obtained from the tracee in both phases.

The difference between an application and a process is that a process is the running instance of an application [8]. The OS kernel manages its address space, including the

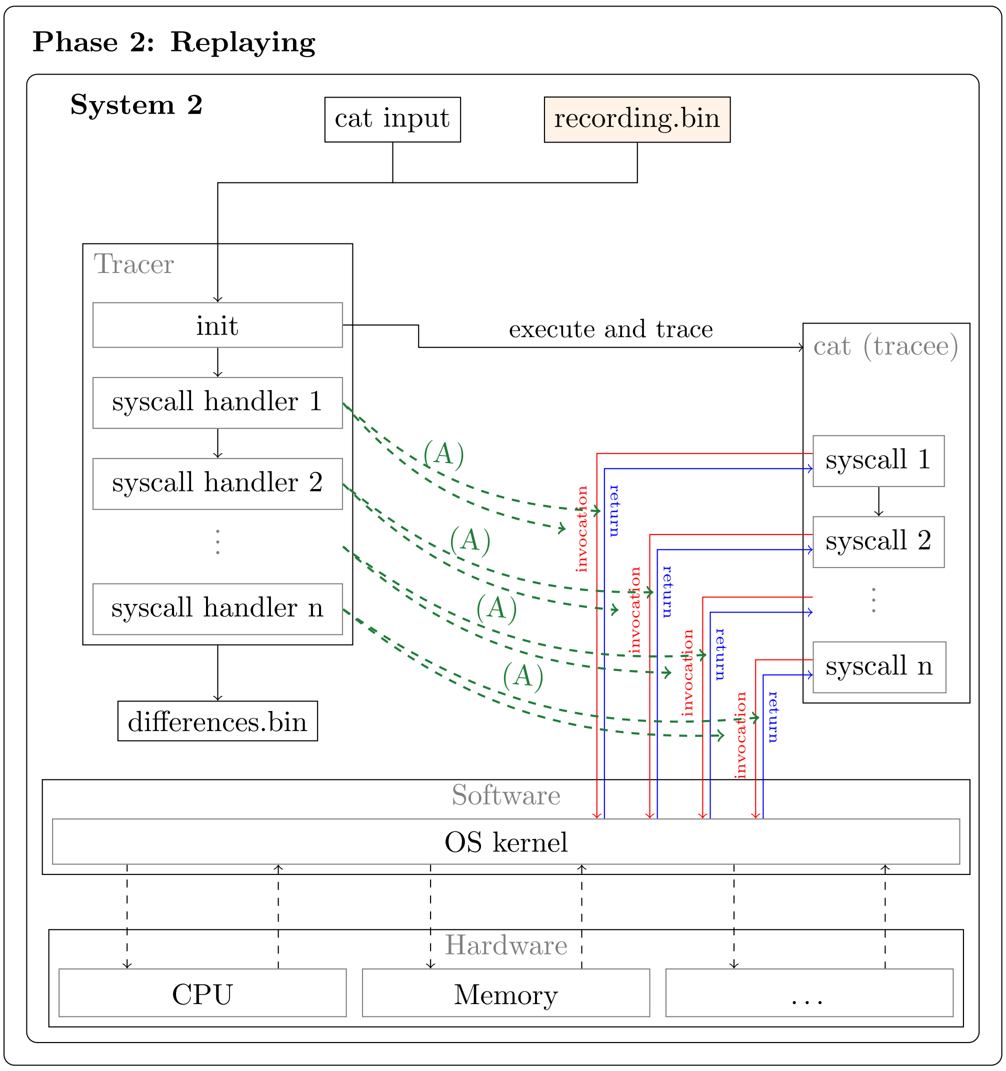

Figure 4.2 A conceptual overview of SPERRIPS replaying phase. On (A), the retrieved arguments are compared to the recorded ones.

application's code and data sections, loaded libraries and mapped files, the application's heap, stack, and process and thread-related controlling data structures. An exemplary simplified virtual address space of a cat process is illustrated in Figure 4.3 as obtained via `cat /proc/self/maps`.

When speaking of saving and restoring the inner process state in this work, the SPERRIPS approach collects all relevant data on the chosen abstraction level of syscalls which are necessary for comparing the syscall's invocation and effect. This includes all data that are either an argument value or return value of a syscall, including possible return arguments. Therefore, the data which is to be obtained depends on the syscall's signature and purpose. All these data are available from user-space. However, the invocation of a syscall produces side-effects in internal kernel structures, as pointed out by Cornelis et al. [24]. For example, internal management data for opened files, mapped memory pages, etc. [8]. We purposely do not trace and record/replay these internal data stored in kernel-space. By re-executing each syscall on both systems, rather than replaying recorded values, we demand both systems to produce the same side-effects. Consequently, potentially detected differences originate from differing behavior within the kernel or hardware components and are thus exactly what our approach aims to detect.

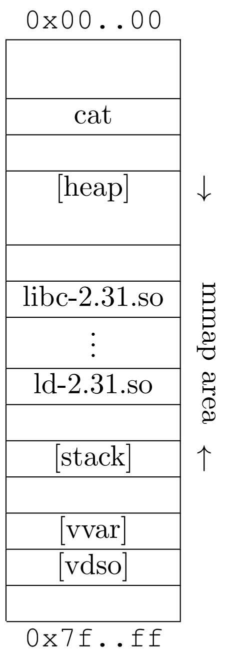

Figure 4.3 Simplified virtual address space layout of cat with activated ASLR and the mapped Virtual Dynamic Shared Object. The arrows indicate the address-wise growth directions.

4.2 Case Study: cat

As defined in Chapter 3, the overall contribution of this work is the verification of the execution of the cat application across two different systems. Therefore, in this section, we investigate the involved syscalls of a cat process to determine requirements for the SPERRIPS conception. Additionally, the two phases of SPERRIPS are illustrated by a motivating example of the read syscall. Then, in Section 4.2.2, we provide our definitions for a verified correct execution of cat on two different systems. These definitions later guide the strived implementation.

4.2.1 Involved System Calls and Motivating Example

The cat command-line tool[1] expects one or more arguments as file paths and writes the contents of these files to `stdout`. Therefore, cat necessarily leverages syscalls for opening and reading files and writing content to `stdout` or a file descriptor. However, even more syscalls are involved. The following shows a complete list of all involved syscalls determined by strace of an execution of `cat /etc/hostname`[2] on a Debian 11 system with version 2.31 of libc and disabled VDSO feature. The system calls are listed in the order of their

[1]An application from Linux *coreutils*

[2]cat (GNU coreutils) 9.0.12-f60a3 built from source code, from git tag v9.0

occurrence but with removed duplicate entries. We refer to its manual page entry for a more detailed description of a syscall's purpose and behavior. The syscalls `arch_prctl` and `exit_group` have no libc wrapper equivalent. Therefore, they are listed as an invocation via the generic `system` function.

1. brk: `int brk(void *addr);`[3]
2. uname: `int uname(struct utsname *buf);`
3. access: `int access(const char *pathname, int mode);`
4. openat: `int openat(int dirfd, const char *pathname, int flags);`
5. fstat: `int fstat(int fd, struct stat *buf);`
6. mmap: `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
7. close: `int close(int fd);`
8. read: `ssize_t read(int fd, void *buf, size_t count);`
9. mprotect: `int mprotect(void *addr, size_t len, int prot);`
10. arch_prctl: `int syscall(SYS_arch_prctl, int code, unsigned long addr);`
11. munmap: `int munmap(void *addr, size_t length);`
12. fadvise64: `int posix_fadvise(int fd, off_t offset, off_t len, int advice);`
13. write: `ssize_t write(int fd, const void *buf, size_t count);`
14. exit_group: `noreturn void syscall(SYS_exit_group, int status);`

Table 4.1 summarizes all argument types by their programmatic types. Additionally, we classified them into two different classes, namely *pointers* and *numerical values*. Technically, all syscall arguments are passed as numerical values via the processor registers. The `access` syscall for instance retrieves a `const char *pathname` pointer as an argument. But in terms of verifying the inner process state, it is not significant that the buffers' addresses are equal, but their content. In fact, with activated ASLR, the buffers' addresses are different in two separate executions. Precise conventions for argument comparison, depending on their different types and the effect of ASLR, are dealt with in Sections 4.2.2 and 4.3.3.

In the following, we illustrate the procedure of both the recording and the replaying phase of SPERRIPS for verifying an intercepted `read` syscall while tracing `cat /home/example.txt`. In this example, the file `/home/example.txt` exists on both systems and has the following content.

```
Hello world!
```

Listing 4.1: Content of the example file located at `/home/example.txt`

[3]The wrapper function returns an integer value to indicate an error. However, the original brk syscall returns a `void*` pointer as the new program break.

Data type	Classification
`void*`	pointers
`const char*`	
`struct utsname*`	
`struct stat*`	
`const void*`	
`int`	numerical values
`off_t`	
`size_t`	
`ssize_t`	
`unsigned long`	

Table 4.1 Overview and classification of the involved argument types of all syscalls of `cat /etc/hostname`.

Recording Phase

We assume SPERRIPS encounters the `read` syscall, executed by cat. The `read` syscall takes three arguments and returns a `ssize_t` value (see syscall No. 8) in the above enumeration. According to the System-V calling convention (see Section 2.1.2), these argument values are passed via the processor registers in the order of `RDI`, `RSI` and `RDX`. When intercepted before the syscall is executed, they possess the following exemplary values: `RDI` = `0x3`: The file descriptor number for the file to read, `RSI` = `0x7f8b203dd000`: A pointer to the buffer, where the kernel stores the read bytes, `RDX` = `131072`: The number of bytes to fill into the allocated buffer.
By definition, `read` returns the actual number of read bytes as its return value. According to the considered calling convention System-V, this happens to be in the `RAX` register.

In terms of preserving the inner process state, it is insufficient to save and restore the pointer value from `RSI` since not the pointer value is of relevance, but the buffer's content. Therefore, SPERRIPS need to record the content of the buffer, pointed to by the pointer in `RSI`, since this holds the actual read data. Hence, the memory content must be retrieved from the tracee's virtual address space and then saved together with the argument values.

For this example, we assume that the memory at address `0x7f8b203dd000` is allocated for `139264` bytes and holds "Hello world!" continued by null-bytes after the syscall's execution.

Replaying Phase

Now, in the replaying phase, SPERRIPS again intercepts the `read` syscall. Due to activated ASLR, the address of the buffer is different on this execution. The value of `RSI` is now `0x7f6538e36000`. After the read syscall has been executed, SPERRIPS resolves the buffer content again. But instead of comparing the raw pointer values, it compares the actual buffer content with the recorded one. We assume both systems operated identical, and thus, the buffer from the replaying phase also contains "Hello world!" followed by null-bytes.

This toy example demonstrates the verification procedure for the buffer of the `read` syscall. In contrast to recording/replaying systems, the syscalls are executed on both systems to compare their results. Differences are only overcome by replaying if they are acceptable.

4.2.2 Verified Correct Execution of cat

In this section, we define how two executions of cat are considered to be equal and thus correct. It explicitly stretches which differences are *acceptable differences* and which are *unacceptable differences*.

First, we state the three possible scenarios when intercepting a syscall in the replaying phase.

1. Syscall type is equal to the recorded type, and all argument values (including the return value) are equal, too.
2. Syscall type is equal to the recorded type, but not all argument values (including the return value) are equal.
3. Syscall type is different from the recorded type.

Scenario 1 is the desired case, where a syscall invocation in the replaying phase is equivalent to the one from the recording phase. Since no difference occurred in this case, the tracee's execution continues.
For the other two scenarios, we distinguish between *acceptable differences* and *unacceptable differences*.

Acceptable Differences

This section introduces acceptable differences of the identified syscalls of cat. The identified acceptable differences are caused by differently provided environments. They are eliminated by replacing the differing value in the replaying phase with the recorded value from the recording phase. Thereby, we implement a recording/replaying mechanism and thus force the tracee's execution to continue the execution deterministically as recorded. The following shows the acceptable differences, which are assumed not to be significant for cat's execution.

For the `stat` struct, we accept differences for values of the members `st_dev`, `st_ino`, `st_atime`, `st_mtime`, and `st_ctime`. For the `utsname` struct, we accept different values for all of its members.

To apply the concept to further syscalls or environments, other acceptable differences must be defined if necessary. Details on environment-specific differences are discussed in greater detail in Section 4.3.4.

Unacceptable Differences

The following shows a list of explicit unacceptable differences in the execution of cat. We define aborting the tracee's execution on any encountered unacceptable difference.

1. Occurrence of aforementioned Scenario 3. It indicates a different execution flow and thus an obvious difference in program execution.

2. Difference in any numerical value, as classified in Table 4.1.

3. Difference in any semantical usage of a pointer. This is the case if a pointer does not point to the equivalent location (with respect to ASLR) as in the recording phase. Details on this are discussed in Section 4.3.3.

4. Difference in any buffer content, pointed to by a pointer as classified in Table 4.1 with the exception of the aforementioned accepted differences in `struct stat` and `struct utsname`.

Detailed Conventions for Argument Comparison

The following gives detailed conventions for the equality of the data types as presented in Table 4.1 on Page 20. The comparison happens in the replaying phase between the recorded value and the one captured in the replaying phase. Therefore, we refer to an argument value fetched in the recording phase as the *first*, while we call an argument value retrieved in the replaying phase as the *second*.
The conventions are as follows:

1. Numerical values are equal if the first value matches the second one.

2. `void*` pointers are equal if the second value corresponds to the first value under consideration of ASLR. This is further elaborated in Section 4.3.3.

3. `const char*` values are considered to be equal if the memory content, starting from the first address up to the first appearing null-byte, is equal to the memory content, starting from the second address up to the first appearing null-byte.

4. `struct utsname*` values are considered to be equal if all members of the first referenced struct are equal to all members of the second referenced struct with the exception of the mentioned acceptable differences.

5. `struct stat*` values are considered to be equal if all members of the first referenced struct are equal to all members of the second referenced struct with the exception of the mentioned acceptable differences.

6. `const void*` values are considered to be equal if the memory content, starting from the first address up to the following n bytes, is equal to the memory content, starting from the second address up to the m next bytes. Where n is the argument value of the corresponding argument of the first syscall and m is the argument value of the corresponding argument of the second syscall. Further, n must be equal to m.

For Definition 6, we leverage the fact that the buffer's size is determined by its corresponding argument (compare Syscalls 6, 8 and 13 in Section 4.2). Since in contrast to character buffers (`const char*`), they do not have an implicit indication of their content length.

Due to the nature of `mmap`'s behavior, we define that its return value must both be treated as `void*` and `const void*`. For determining the page's length, `mmap`'s `length` argument is used. For `read`, `buf` is treated as a type of `const void*` after `read` has been executed.

4.3 Issues and Practical Challenges

This section deals with issues and practical challenges of recording and replaying the inner process state. First, we discuss the technical details for an issue. Then we propose the selected solution for our conception. The addressed issues primarily belong to the syscalls from the cat binary, however, some also relate to other aspects beyond those of cat. In particular, theoretical considerations to the previously excluded sources of non-determinism (see Section 3.1) are also addressed.

4.3.1 Interception and Handling of System Calls

One key principle of the SPERRIPS approach is intercepting all syscalls that are invoked by the tracee. Then, depending on the intercepted syscall type, different handling is required.

The handling includes the correct parsing of all syscall arguments and verification of the syscall's performed action. Due to the calling convention's design, a syscall can retrieve up to six different arguments [12].

In Section 4.2 we have seen that numerical argument values can be fetched directly with access to the processor registers. In contrast, buffer and struct content must be retrieved from the tracee's memory by reading from the pointer location as stored in the processor's register. Furthermore, some syscalls like `fstat` use *return arguments*. These are arguments that the syscall alters through its execution in favor of returning information. In the case of `fstat`, the syscall writes the file information into the struct, whose address is given as a syscall argument.

From these observations, we can derive the following requirements for an implementation of SPERRIPS regarding the syscall handlers:

1. Access to an interface for intercepting the tracee's syscalls for both prior and after the syscall' invocation.

2. Primitives for reading and writing to a processor register and the tracee's virtual address space.

3. Handlers that define the argument types of a syscall in order to implement appropriate argument retrieval and syscall behavior.

Requirements 1 and 2 can be resolved by using the ptrace feature of Linux (see Section 2.1.3). It allows intercepting syscalls and provides mechanisms for reading and writing to the tracee's virtual address space and the processor registers. In this work, we refer to an interception as *on entry* when it happened before the system retrieved the syscall and *on exit* when the interception happened after the system performed the syscall. Regarding the third requirement, a mapping of a syscall to its list of argument types can be retrieved from the syscall's signature. It can either be looked up on Page 2 in the manual pages or directly in the kernel's source code.

Nested Data Structures

The previously described issue of correctly understanding and processing a system calls' argument might be even more complex. Despite the `stat` structure, other data structures may contain pointers as its members that reference additional structures or buffers. An example for such a nested structure is the `msghdr` struct, which is an argument of the `recvmsg` syscall. It posses the three members `msg_name`, `msg_iov`, `msg_control`, which are pointers to other structs and data as illustrated in Figure 4.4.

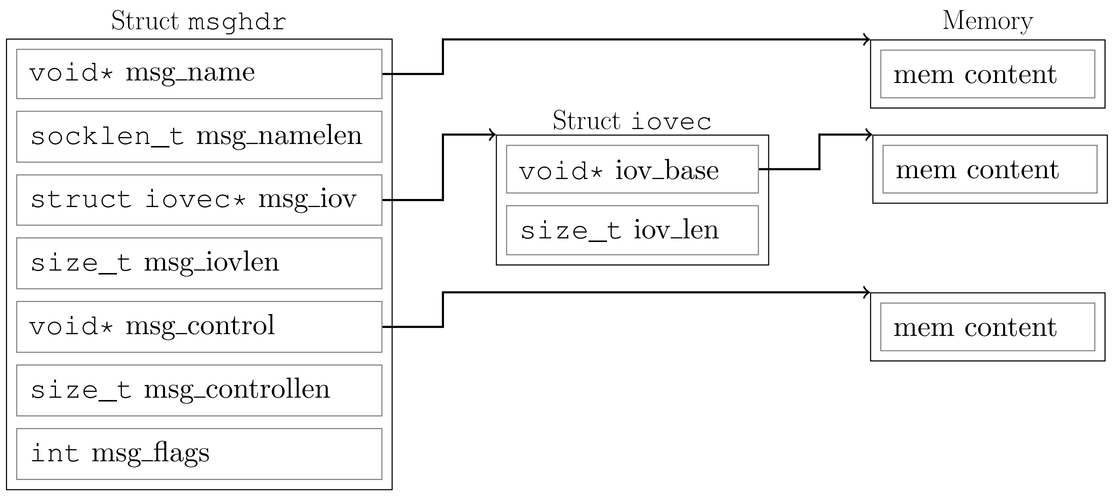

Figure 4.4 The nested structure of the `msghdr` struct as defined in `sys/socket.h` in the Linux kernel.[4]

For recording the inner process state, the actual referenced data must also be saved to be available in the replaying phase. This requires that the implementation is able to parse encountered structs properly and is adapted to retrieve referenced data from memory.

4.3.2 Potential Memory Corruption

This section deals with potential memory corruptions through replaying recorded values. We have already seen that replaying struct and buffer contents requires actively altering the tracee's memory. One of a struct's characteristics is its constant size within memory[5]. Hence, replaying a struct within the tracee's memory by overwriting the corresponding memory content with the recorded struct is not problematic, as their sizes are of equal length. However, this might not be the case for recorded buffer content of "arbitrary" size. Therefore, this section deals with the case where there is a discrepancy in the size of recorded memory content and available space for replaying it. Further, this section specifies how the SPERRIPS approach deals with this scenario.

An example for replaying buffer content for the `read` syscall is already demonstrated in Section 4.2.1. However, in this particular example, the contents of the buffers were of equal size in both phases (`Hello world!\n` and `Hello world!\n`).
In the following, the two scenarios are considered, where the sizes are different:

[4]https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/socket.h?h=v5.15.1

[5]This is the case if the recording and replaying phase happen on the same target platform due to the reason given in Section 2.2.2.

1. The size of the value to replay is less than the available memory space.
2. The size of the value to replay is greater than the available memory space.

Replay Fewer Bytes than Allocated Buffer Size

We consider a scenario where we replay fewer bytes into the tracee's memory than the buffer in memory was allocated for. From a replaying perspective, this case is trivial to handle.

We consider the following example: An application has two buffers allocated, as shown in Listing 4.2. The first buffer begins at `0x7fffe2759020` and holds the content `Hello there!\n.`[6] and the second is located at `0x7fffe275902e` and contains `What's up?\n.` Let us assume we recorded `Hi there!\n.`[6] for the first buffer instead and thus need to replay this content into the first buffer. This content is three bytes shorter than the original one (`Hi` vs. `Hello`). Hence, it is unproblematic to overwrite the memory content, starting from address `0x7fffe2759020`.

```
0x7fffe2759020 48 65 6C 6C 6F 20 74 68   Hello th
0x7fffe2759028 65 72 65 21 0A 00 57 68   ere!..Wh
0x7fffe2759030 61 74 27 73 20 75 70 3F   at's up?
0x7fffe2759038 0A 00 00 00 00 00 00 00   ........
```

Listing 4.2: Exemplary memory layout within the tracee's process. Displayed in big endian notation, for readability sake.

Therefore, we define the following for scenarios like this, where more space is available than required. First, the memory is overwritten with the recorded value, and null-bytes replace the remaining bytes. In our example, this leads to the memory content illustrated in Listing 4.3.

```
0x7fffe2759020 48 69 20 74 68 65 72 65   Hi there
0x7fffe2759028 21 0A 00 00 00 00 57 68   !.....Wh
0x7fffe2759030 61 74 27 73 20 75 70 3F   at's up?
0x7fffe2759038 0A 00 00 00 00 00 00 00   ........
```

Listing 4.3: Overwritten memory content as defined in this work. Displayed in big endian notation, for readability sake.

Replay More Bytes than Allocated Buffer Size

This scenario deals with the case where the content to replay is greater than the actual buffer in the tracee's memory. We begin with the same toy example from the previous section as listed in Listing 4.2. Now we assume that we recorded `Hello everyone!\n.`[6], instead of `Hello there!\n.` for buffer one at address `0x7fffe2759020` and have to replay it. Overwriting the memory with the recorded value would lead to the memory layout as illustrated in Listing 4.4.

```
0x7fffe2759020 48 65 6C 6C 6F 20 65 76   Hello ev
0x7fffe2759028 65 72 79 6F 6E 65 21 0A   eryone!.
0x7fffe2759030 00 74 27 73 20 75 70 3F   .t's up?
0x7fffe2759038 0A 00 00 00 00 00 00 00   ........
```

Listing 4.4: Overwritten memory with corrupted consecutive buffer. Displayed in big endian notation, for readability sake.

[6]The trailing dot represents a non-printable null-byte.

This results in a memory corruption since the replayed content affects the consecutive memory. The second buffer, which begins at `0x7fffe275902e` was formerly holding `What's up?\n.` and is now overwritten with three bytes from the replayed content. Memory corruptions cause severe problems in the process, which most likely result in a crash or wrong behavior. Hence, replaying too much data into too small buffers is no suitable solution. Therefore, we evaluated the following two strategies for coping with this scenario.

1. Allocating new memory on behalf of the tracee.
2. Abort the tracee's execution.

For Strategy 1, the `mmap` syscall can be leveraged to allocate new memory on behalf of the tracee. However, this requires control flow manipulation within the tracee processes, such that the process invokes the mmap syscall. Then, the kernel provides a new memory page that can be used for storing the recorded value. However, this does not have any effect on the inner process state, as long as the former buffer address `0x7fffe2759020` is not replaced with the newly allocated one, for each reference, where the program uses the old buffer address. However, since we operate on the level of syscalls, we do not trace every memory access that takes place within the tracee's process. Intercepting memory access is out of the work's scope. Hence, it is no appropriate solution to our conception as we cannot guarantee that the tracee operates on data of the newly allocated buffer.

Therefore, we define aborting the tracee's execution if there is not enough memory space to write a recorded value. However, since we eliminate non-determinism, we assume that the cases shown in this section will not appear.

4.3.3 Address Space Layout Randomization

Modern OSs implement the exploit mitigation technique ASLR. It increases the system's security and hence is recommended to be enabled. Nevertheless, it also increases the complexity of recording and replaying inner process states. Recorded data cannot simply be replayed into the child's memory. Due to the randomized address space layout, two executions of the same application have two different address layouts, which affect syscalls argument values and the inner process state. In this section, we describe different effects of ASLR. After analyzing them, we define SPERRIPS behavior, such that it can record and replay data on systems with enabled ASLR successfully.
We identified two different types of address usages. In the following, we refer to them as *directly used addresses* and *indirectly used addresses*.

Directly Used Addresses

Using addresses as an argument value or return value of a syscall we refer to as *directly used addresses*. An example for such a syscall is `mmap`[7], which maps a file to a given address location. On success, the kernel returns the actual address of the mapped memory location. A possibly subsequent call of `munmap`[8] could unmap the file and requires the previously

[7]`void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
[8]`int munmap(void *addr, size_t len);`

mapped address as its argument. With activated ASLR, two separate executions of an application using `mmap` result in two differently randomized memory locations and hence in two different argument and return values.

Concerning our aim of verifying correct application behavior, it is not essential to have the exact same numerical values for addresses but their semantical usage. If we recorded a sequence of `mmap(..) = 0x123, munmap(0x123)` it should be considered equal with `mmap(..) = 0x456, munmap (0x456)`, since the differing randomized addresses are used in the same manner.

To realize this, our proposed approach uses a *pointer mapper*. Figure 4.5 on Page 37 illustrates a flow chart of the mapping logic and provides an example for detecting wrong pointer usage. This key-value store maps the address from the recording phase (*rec addr*) to the address from the replaying phase (*repl addr*). Additionally, in a second map, the addresses are mapped vice versa. Whenever a memory address appears as an argument or return value during the replaying phase, it is first checked whether the recorded address has an existing entry in its map. If so, it is checked whether the replayed address is equal to the mapped address, and on success, it is used for SPERRIPS operation. If the recorded address has no existing mapping, it is checked whether the replayed address has an existing entry in its map. If not, the addresses are mapped. But if so, it indicates that an address has falsely been used as an addresses has been used that does not match the recorded sequence. This procedure allows verifying the same semantical usage of differently randomized addresses. On a detected difference, we define aborting the tracee's execution.

Indirectly Used Addresses

Addresses embedded in data, e.g., when using nested data structures (see Section 4.3.1), are referred to as *indirectly used addresses* in this work. Indirectly used addresses require additional effort for the conception and implementation since the affected data structures must be parsed, such that the addresses are available to SPERRIPS. In case of replaying, the indirectly used addresses must be updated within the replayed struct such that they point to the correct randomized addresses of the replaying phase. Moreover, involved nested data structures require that the data behind the referenced locations must also be updated with respect to the current randomized address space layout. This procedure must possibly be executed recursively if a nested data structure also contains a nested data structure as the `msghdr` struct, for example, does (see Figure 4.4 on Page 24).

Listing 4.5 illustrates an example of indirectly used addresses in a `msghdr` struct. We have seen this struct as an example of nested data structures earlier in Section 4.3.1. We assume we obtained the data as shown in Listing 4.5 in the recording phase from a `recvmsg` syscall[9] for its `struct msghdr *msg` argument from memory address `0x7ffc2a0179f0`.

```
struct msghdr { // 0x7ffc2a0179f0
    void          *msg_name       = 0x7ffc2a017a30;
    socklen_t      msg_namelen    = 0x80;
    struct iovec *msg_iov         = 0x7ffc2a0179d0;
    size_t         msg_iovlen     = 0x1;
    void          *msg_control    = 0x7ffc2a017ab0;
    size_t         msg_controllen = 0x0;
```

[9] `ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);`

```
    int             msg_flags      = 0x0;
};
```

Listing 4.5: Pseudo code representation of a msghdr struct in the recording phase.

The three members msg_name, msg_iov, msg_control are pointers to other data or structs (see Figure 4.4 on Page 24). Hence, when replaying this msghdr struct into memory with another address space layout randomization, the pointers must be updated, or they will point to invalid locations.

Now, we assume we have captured the struct in the replaying phase as presented in Listing 4.6. For this example we purposely modified the values of msg_namelen, msg_iovlen, msg_controllen and msg_flags, such that they are different. Due to ASLR, the pointers are different anyway.

```
struct msghdr { // 0x7ffeb7e35a90
    void          *msg_name       = 0x7ffeb7e35ad0;
    socklen_t      msg_namelen    = 0x40;
    struct iovec *msg_iov        = 0x7ffeb7e35a70;
    size_t         msg_iovlen     = 0x0;
    void          *msg_control    = 0x7ffeb7e35b50;
    size_t         msg_controllen = 0x10;
    int            msg_flags      = 0x40;
};
```

Listing 4.6: Pseudo code representation of a msghdr struct in the replaying phase.

We can illustrate our defined behavior for the SPERRIPS approach with these two examples. In the first step, we replay the recorded struct exactly the same as recorded into the memory location of the replayed struct. In this example this is at 0x7ffeb7e35a90. Thereby we ensure that all member fields have the same values as recorded. This is important for all non-pointer members like msg_namelen, msg_iovlen, msg_controllen, msg_flags. However, now the address fields msg_name, msg_iov, msg_control must be updated with the addresses from the struct, as obtained in the replaying phase, such that the randomized addresses are correct. Additionally, at these locations (0x7ffeb7e35ad0, 0x7ffeb7e35a70, 0x7ffeb7e35b50) the nested data structures' content must be replayed. After all, this results in a struct like shown in Listing 4.7.

```
struct msghdr { // 0x7ffeb7e35a90
    void          *msg_name       = 0x7ffeb7e35ad0;
    socklen_t      msg_namelen    = 0x80;
    struct iovec *msg_iov        = 0x7ffeb7e35a70;
    size_t         msg_iovlen     = 0x1;
    void          *msg_control    = 0x7ffeb7e35b50;
    size_t         msg_controllen = 0x0;
    int            msg_flags      = 0x0;
};
```

Listing 4.7: Pseudo code representation of a replayed msghdr struct with activated ASLR.

Note that all non-pointer members have their recorded values set, while the pointers have the values of the replayed struct. Moreover, the memory content behind these pointers has been updated accordingly.

Handling indirectly used memory addresses is significantly more complex, as it requires adapting the SPERRIPS implementation precisely to the data structures used by the kernel. Hence, each nested data structure needs individual code for parsing the structure and updating the contained pointer members.

4.3.4 Environment State

Comparing two application executions on two different systems with varying environment setups can lead to differences in syscall return values. Therefore, it is required to identify environment-specific differences that can influence an application's verification. Possibly such differences can be defined as acceptable differences, as seen in Section 4.2.2. However, this section points out identified dependencies and presents appropriate conceptual solutions for mitigating them on two systems

Influence of Linked Libraries

Using strace on different systems revealed that running the same version of cat[10] on different systems (Debian 11 vs. Ubuntu 20.04) leads to a different sequence of syscalls.
For example, the `arch_prctl` syscall is invoked in the beginning when cat is executed on a Ubuntu 20.04 system, but not on a Debian 11 system. In Listing 4.8 a truncated output of `strace cat /etc/hostname` from a Ubuntu 20.04 is shown, which includes the `arch_prctl` syscall[11].
Debugging the execution revealed that the dynamic linker invokes the syscall. However, both systems use libc' linker in version 2.31, with minor adaptions by the distribution maintainers. Apparently, Debian does not compile the code of libc with the set `CET_ENABLED` flag[12] while Ubuntu does[13] and this impacts calling `arch_prctl` or not.

```
# strace cat /etc/hostname
execve("/usr/bin/cat", ["cat", "/etc/hostname"], 0x7fffffffe4a8 [..]) = 0
brk(NULL)                                  = 0x555555560000
arch_prctl(0x3001 /* ARCH_?? */, 0x7fffffffe3e0) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
[..]
```

Listing 4.8: Shortened strace output of `cat /etc/hostname` on Ubuntu 20.04 with libc 2.31.

This shows that depending on libc's version, the application executes different syscalls. This effect probably applies to other dynamically linked libraries. Therefore, we must ensure that the same shared library versions are used on both systems for recording and replaying. This can be achieved by either having the same shared library files on the systems or statically linking the required libraries into the tracee during compilation. However, this requires adaption of the tracee.

[10]cat (GNU coreutils) 9.0.12-f60a3

[11]As visible in Listing 4.8, argument `0x3001` is not understood by the system, because it belongs to the hardware shadow stack security feature, which is not available on this CPU.

[12]https://buildd.debian.org/status/fetch.php?pkg=glibc&arch=amd64&ver=2.31-13%2Bdeb11u2&stamp=1633188416&raw=0 *Last accessed on 2021-12-09*

[13]https://launchpad.net/ubuntu/+source/glibc/2.31-0ubuntu9.2/+build/20580861/+files/buildlog_ubuntu-focal-amd64.glibc_2.31-0ubuntu9.2_BUILDING.txt.gz *Last accessed on 2021-12-09*

Process Identifiers

On Linux systems, each process is identified by a unique process identifier (PID) [8]. PIDs are assigned by the kernel to each newly created process, e.g., through the `fork` or `clone` syscall. Each call to `fork` creates a new process that adds to the system's process tree. In Linux, the assigned PID depends on the current PID counter and remaining available PIDs[14]. As a consequence, the tracee's PID will be different on two systems if not all processes have been executed in the exact same order up to the tracee's execution, which is unlikely.

Examples of syscalls that take a PID as an argument are `getpid` and `wait`. Replaying a recorded PID to those syscalls thus either requires to translate the recorded PID to the actual one of the tracee or to ensure that the PID values are equal on both systems.

Conveniently, the Linux kernel implements the so-called *namespaces* concept, which we can leverage to mitigate the described problems of PID numbering [28]. Namespaces separate different resources from a process, such that the process is isolated from other processes regarding a particular specified resource type. One of the supported types are PID namespaces.

On creating a new PID namespace, the kernel separates it from other ones on the system and starts numbering with one again. Then, the PID numbering is maintained independently of other namespaces, and processes will only be visible to those in the same namespace. By using PID namespaces, the tracee will always be the first process in the namespace, as the tracer executes it. Hence, the tracee's PID and the ones of its possible children will be deterministic on each execution. This allows us recording and replaying PIDs.

System Configuration

As seen, the tracee's execution flow is influenced by the version of the dynamically linked libc library, which comes as a part of the used Linux distribution. However, other files might also influence the tracee's execution flow depending on the traced application. For example the configuration file `~/.vimrc` influences, which features of the vimeditor are activated and used. Different configurations thus lead to different behavior, and hence different potential execution flows of vim. As a consequence, it is required to provide equal system configurations and file systems to the tracees.

For this purpose, the namespaces concept can also be used. The `mount_namespaces` syscall can control which file system mounts are available to a process. In combination with `pivot_root` it is possible to exchange the file system's root (`/`) with a different directory.

This can be used to exchange the file system's root with a minimal Debian installation[15] on both systems. Thereby, the same Debian environment can be provided for each tracee, independently of their actual host system. It guarantees that both environments provide the same shared libraries and file systems.

Inode Numbers

The `fstat` syscall[16] takes a file descriptor as an argument and returns a struct with information about the corresponding file in its second return argument (see Section 4.3.1).

[14] https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/kernel/pid.c?h=v5.15.1

[15] The command-line utility `debootstrap` installs a Debian distribution into a given directory.

[16] `fstat(int fd, struct stat *buf);`

This information includes the birth, access, and modification timestamps of the file and its *inode* number. Inodes are the file system's internal representation of filesystem elements like directories or files [8]. Each of these elements is identified through a unique inode number assigned sequentially in ascending order. This means that the same path like, e.g., `/tmp/example-file` can exist on two systems. Still, they have different inode numbers representing the mentioned file if not all filesystem elements have been created and deleted in the exact same order.

For this work, two different strategies have been evaluated for dealing with differing inode numbers during the recording and replaying phase.

1. Using an inode mapper, similar to the pointer mapper as for ASLR.
2. No specific handling and accept differences.

Regarding Strategy 1, each recorded inode number would map to a corresponding one in the replaying phase, similar to pointer addresses for ASLR. Further, like for the ASLR mapper, the semantical usage of inodes must be preserved.

We decided to use Strategy 2 for our implementation since inodes are a filesystem-specific implementation abstraction detail that should not affect cat's execution. Hence, inode numbers are treated as acceptable differences and are thus replayed in this work.

However, in general, differences in inode numbers can be avoided by providing identical file system images on both systems. Then, both kernels must return the same inode and timestamp information on both systems, and differences are not acceptable anymore.

File Descriptors

In Section 4.2 we have seen that multiple syscalls (e.g., `openat`, `fstat`, `write`) require a file descriptor number as an argument. In contrast to inode numbers, file descriptor values do not depend on the system's environment state. Instead, they depend on the application's execution flow, as the kernel holds file descriptors in a table per process [8]. On file opening, the kernel adds a new entry to the table together with its file descriptor number, which is assigned in ascending order, starting with zero. Hence, a differing sequence or usage of file descriptors in the recording and replaying phase indicates different application behavior. Thereby, file descriptor numbers should not only be equal, but they must be equal during the recording and replaying phases.

4.3.5 Virtual Dynamic Shared Object

With the activated VDSO feature (see Section 2.1.1), our suggested usage of ptrace (see Section 4.3.1) needs adaption since function calls to the VDSO are not visible to ptrace as a syscall. This has to be considered for our conception and implementation of SPERRIPS. To still trace these calls technically, either the usage of the VDSO can be disabled or calls to the VDSO functions must be intercepted. However, since function interceptions are out of the scope of this work, we disable the usage of the VDSO system-wide by setting the kernel boot parameter `vdso=0`. Then, this complies with our conception as outlined in Section 4.3.1.

4.3.6 Other Sources of Non-Determinism

This section deals with the three other sources of non-determinism, despite syscalls themselves, as given in Figure 3.1 on Page 12. Namely, these are network responses, received interrupts, and shared data between multiple threads [5] [27]. In terms of reproducible application execution, they constitute a more significant challenge to our approach. Therefore, they require a different design and implementation to work with our concept and goals. The following sections explain the reason for the introduced non-determinism, the difficulties they bring, and how an adapted conception of SPERRIPS can treat them anyhow. However, these adaptions are not currently implemented and are theoretical considerations, party based on other recording/replaying systems research.
Additionally, special considerations regarding the `getrandom` syscall are discussed in this section. It deals with the challenge of using the same secure random on two systems under the assumed trust model and the two-phases conception.

Network Connections

When an application sends data to a network endpoint, it loses control over the processing and the received response. A network endpoint's behavior can change anytime, without the application's influence. Hence, a network connection must be treated conceptually as a blackbox, whose internal behavior we neither control nor fully understand. Even if two applications send the same requests to the same endpoint, they might receive different responses. For example, two applications send a request "`gettime`" to a server. Influenced by network routing, connection speed, and processing order of the requests, the server will receive and process one of the two messages first. As the response, it sends its current time, however, due to the different processing times, the responses are different. This example demonstrates that executing two applications consecutively on two different systems does not work for network connections.

Therefore, we suggest the following modifications to the current concept to make it work with network operations. Figure 4.6 on Page 38 illustrates the procedure described in the following. Instead of running the tracees consecutively in two phases, we now require them to run in parallel on the two systems. Further, both SPERRIPS instances must communicate with a trusted third party that we refer to as "*decisioner*". Each time a syscall is intercepted on a system, both instances send the retrieved argument values and return types to the decisioner. The decisioner evaluates the equality of the argument values and returns its result to the instances. Then, it performs network requests on behalf of the two instances, if and only if the request's data of both systems are identical. The corresponding response is sent back to both SPERRIPS instances, which use the data for continuing the tracee's executions. Thereby it is ensured that both parties send the same data and retrieve the same response.

We also propose a new trust- and attacker model with this conception. It is assumed that an attacker can compromise one of the systems through a placed backdoor that implements malicious behavior. However, the attacker cannot interact with the compromised system or leak data from it as the system has no input/output interface other than to the decisioner. Nonetheless, the decisioner's output is observable, including issued network requests, application abortion, and verification results.

Timers and Signals

The `setitimer` syscall allows a process to specify a signal type and a time, after which the OS kernel sends a signal of the given type to the process. When receiving a signal, the process's execution is interrupted to handle the received signal through an appropriate signal handler [8]. Once a `setitimer` call has been executed, the kernel runs the timer asynchronously in the background. This means the process's execution continues independently of the timer until it receives the signal.
These timed asynchronous events and all other asynchronously received signals or interrupts challenge the conception of SPERRIPS. From a recording/replaying perspective, they introduce non-determinism since the signal reception is not deterministic with regard to the position of the currently interrupted instruction within the instruction stream [23]. On signal reception, the application's flow changes to the signal handler routine to handle the signal. If the signal handler itself uses syscalls or if the signal's handling is influenced by the currently interrupted position (e.g., depending on a value in memory), then the differently interrupted executions lead to a differing sequence of recorded syscalls.

Different strategies can be thought of to solve this problem.

1. Replay at exact position: A SPERRIPS implementation counts the amount of executed CPU instructions until the first signal is received. When replaying, it must ensure that the second signal occurs at the exact same position by counting the number of instructions and manually sending the signal to the process at this position. This relates to the mechanism implemented by SimuBoost for replaying asynchronous events for full system replay [23].

2. We require an application not to use signal handlers, which either use syscalls themselves or interact with the process state.

3. Do not support applications with timers or signal handling.

Threading

Applications can use multiple threads for running code concurrently on multiple CPU cores. So, instead of a single main thread, multiple threads run in parallel. The execution order, however, of the threads is managed by the operating system's scheduling mechanism [8]. The application has no control over it, and the thread order will probably vary on two different systems. The SPERRIPS conception needs considerations for two aspects of threading. First, being capable of handling different orders of syscall sequences. Second, respecting data dependencies among multiple threads.

Regarding the first, we require a capability for our intercepting mechanism (see Section 4.3.1) to assign a captured syscall to the corresponding thread. For the second, we see how other recording/replaying mechanisms deal with data dependencies among multiple threads and adapt them to our approach in the following.

Threads can use one of two fundamentally different ways of exchanging data. Either by using a shared resource (i.e., memory) or leveraging message passing [8].

Shared Resources

In the case of shared resources, multiple threads work on the same shared resource, i.e., shared memory. This leads to the commonly known problem of *critical sections* [8]. Critical sections are code passages in which multiple threads work on the same resource. Without proper synchronization mechanisms, not the application's logic would manage the accesses to this resource, but the thread's scheduling order. This can lead to false application behavior. Furthermore, threads could operate in the critical section concurrently and thereby damage the data.

As a mitigation to this problem, OSs supply synchronization techniques such as semaphores. With semaphores, only one thread at a time can enter the critical section and thus access the resource. Other threads which want to enter the critical section are enqueued into a waiting list [8]. The OS permits access to a critical section once another thread leaves it by signaling it to the OS.

As stated by Rossee et al., for a correct replay, it is sufficient to execute the synchronization mechanisms (semaphores) in the replaying phase in the same order as in the recording phase. Thereby, the semaphores guarantee correct access order when assuming a race conditions free program [27].

On Linux, semaphore primitives are provided through syscalls. Therefore, it is possible for SPERRIPS to trace the individual calls for entering a critical section (`wait`) and leaving it (`signal`). In conjunction with Rossee's approach, using Lamport timestamps enables to keep the strict ordering of recorded syscalls with the syscalls in the replaying phase [27].

Message Passing

The second mechanism for transferring data among threads is message passing. Contrary to semaphores, now threads do not work on a shared resource. Instead, they explicitly communicate with each other over a message passing interface. This interface consists of two primitives for sending and receiving messages [8]. For synchronous message reception, we can adapt SPERRIPS. In the case of asynchronous messages, the challenges are then the same as for asynchronous signals, as stated in Section 4.3.6.

On recording synchronous messages, the recorder can assign each message to the corresponding thread [27]. Therefore, it is sufficient to replay the recorded message in the replaying phase without waiting for the actual thread to send it. This is possible as the message content is already available through recording. Netzer and Miller even improved this by only recording the order of *racing* messages [29].

The `getrandom` Syscall

This section deals with the particular `getrandom` syscall. It enables applications to request securely generated random bytes from the kernel. As defined in Section 3.1, we assume that one of the systems is not trustworthy and thus might not return secure random values in either the recording or the replaying phase. Applying the current concept of two consecutive recording/replaying phases thus grants implicitly more power to the system from which the value has been recorded, as this is replayed into the other. Since we do not know which of the two systems is not trustworthy, we might replay insecurely generated randomness and thus also introduce insecure randomness to the other system. From a security point of view, this is not desirable since we want to provide secure randomness, regardless of the trustworthiness of the systems.

Fortunately, this is what the Blum-Coin-Toss method enables. It allows two parties to agree on a secure random value if at least one of the two parties draws secure random [30]. It can be integrated into the SPERRIPS conception, such that on both systems, the Blum-Coin-Toss method is used for agreeing on a secure random number. Like for network operations, this requires interactive communication between the instances. After executing the Blum-Coin-Toss method and thus agreeing on a common value, both instances use this as a seed for a pseudo-random number generator (PRNG). Then, each intercepted invocation of the `getrandom` syscall is answered with a generated random number from the initialized PRNG. PRNGs generate deterministic results for the provided same seed and thus same random bytes on both systems. By applying the Blum-Coin-Toss method, we ensure a secure random seed and thus equal secure random pseudo numbers on both systems.

With the conception of a third-party decisioner for network support and equal pseudo-randomness, it is possible to support authenticated end-to-end encrypted network connections via the decisioner while being fully transparent to the applications. Conceivably, two parties can initiate a network connection, secured by the transport layer security (TLS) protocol. However, since the decisioner acts conceptually as an attacker-in-the-middle, authenticated cryptographic schemes must verify the endpoint's authenticity. Implemented support for pseudo-randomness, as described in the previous section, makes key-generation on both systems deterministic and thus verifiable for the decisioner.

4.3.7 Attacks on Security Goals

This section considers attacks on the security goals confidentiality, integrity, and availability (CIA) under the assumed trust model (see Section 3.2). In particular, it compares the security of the current implementation (the two phases approach) to the conception using a third-party decisioner (see Section 4.3.6 and Figure 4.6 on Page 38). We refer to the former as Implementation 1 and to the latter as Implementation 2. Again, for Implementation 2, it assumed that an attack can alter an application's behavior on his compromised system through a placed backdoor but cannot transfer data from the system other than to the decisioner (see Section 4.3.6).

Confidentiality

Confidentiality aims to prevent the attacker from leaking data from the system. However, in the security model for Implementation 1, the attacker is not restricted from accessing the network and can thus leak data. It does not protect confidentiality.

On the other side, with Implementation 2, an attacker can construct a side-channel through the execution time of the decisioner for leaking information. It leverages that the decisioner needs to wait for both instances' synchronous executions and that the decisioner's result is observable. On the attacker's compromised system, the backdoor delays the execution of a syscall by a specific time depending on the data to leak, e.g., depending on the first few bits. For example, the execution is delayed by 10 ms for the first bits `00`, 20 ms for `01`, 30 ms for `10`, and continuing. As a result, the decisioner's execution delays too, which is observable by the attacker. Thereby, the attacker can draw conclusions about the data by measuring the execution delay. Repeated runs of this endeavor can leak further bits. However, possible mitigation to this attack is that the decisioner accepts only a maximum execution delay (*timeout*) and additionally adds a random delay to its execution. Thereby, the attacker cannot draw conclusions about the data anymore.

Integrity

The integrity, i.e., the correctness of both the collected syscall data and SPERRIPS execution, is crucial for correctly verifying an application's execution. However, with Implementation 1 SPERRIPS verification procedure entirely takes place on an attacker-controlled system. Thus, he can either manipulate the data that SPERRIPS receives or even manipulate its result. Hence, it is insufficient to evaluate an application's execution based on two systems with Implementation 1. A possible solution to this problem is conducting a majority decision by not using two systems for evaluating but three. Thereby, recorded data of System 1 are used for the replaying phase on two distinct and independent Systems 2 and 3. Assuming an attacker only has control over one of the three systems, the result of the majority determines an application's correct execution.

On the other hand, with Implementation 2, the attacker has no control over the decisioner's result. In this case, an attacker could still alter the system's data through its backdoor, however, this is detected by the third-party decisioner on retrieving them. Thus, the decisioner detects manipulated data and notices differing application behavior.

Availability

Regardless of Implementation 1 or 2, an attacker could abort SPERRIPS' execution on his compromised system. Aborting SPERRIPS prevents verifying an application's execution. However, defining a system as defect/compromised if SPERRIPS is not executable on it mitigates this attack.

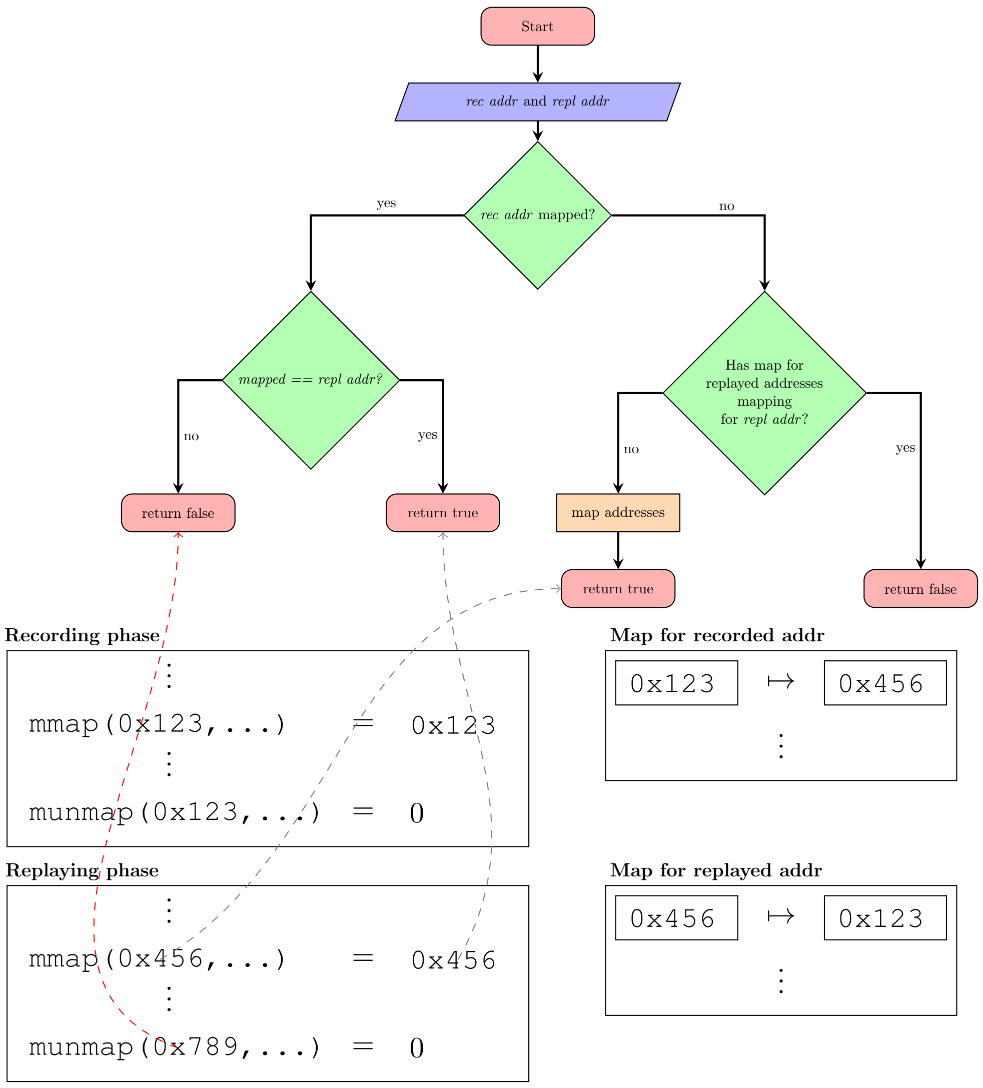

Figure 4.5 Flow chart of the ASLR mapping procedure and an example of identified pointer misuse (red arrow) with simplified addresses. The dashed arrows illustrate where the flow chart terminates for corresponding *rec* and *repl* addresses input.

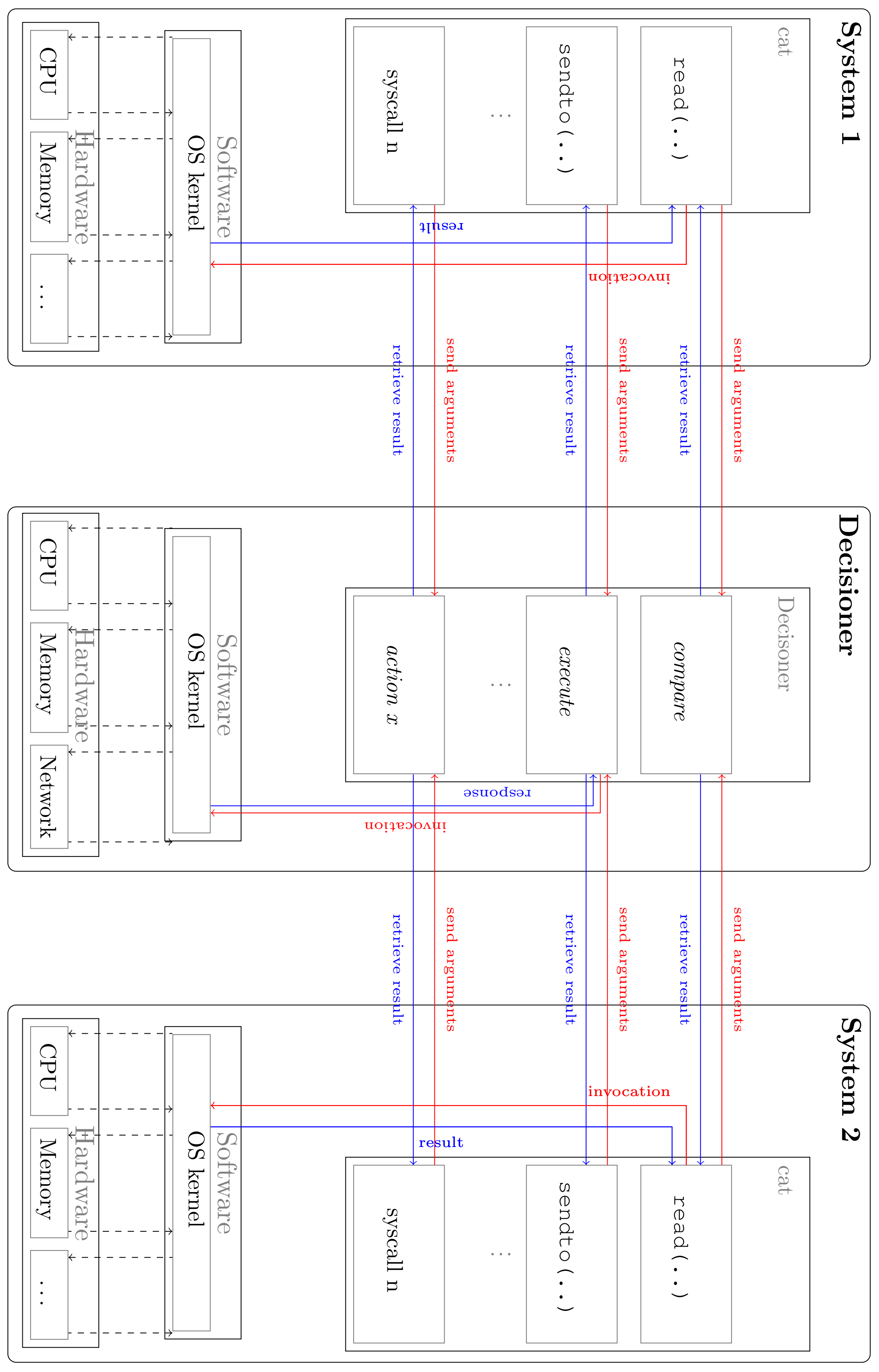

Figure 4.6: Two interactive instances of SPERRIPS, using a decisioner for comparison and network operations.

5. Implementation

This chapter describes certain aspects of the implementation. It discusses some design decisions, key aspects of the architecture, and solutions to the encountered and discussed issues and challenges from Section 4.3.
Regarding the vast amount of different syscalls (more than 350[1]), the implementation does not make claims to be complete in terms of supporting each syscall with its different argument types and effects. Moreover, it is understood as a proof-of-concept for demonstrating the approach's feasibility.

The implementation[2] is written in the C++ programming language, as it allows importing the Linux system header files. This enables direct access to system definitions like the syscall opcodes and kernel interface structure types.

5.1 Overview and Architecture

This section describes the high-level architecture of the given SPERRIPS implementation. The SPERRIPS application is started with the desired configuration, which indicates which target application is executed as the tracee and whether SPERRIPS should operate in either recording or replaying mode. Then, a separated container is initialized for creating and providing a new environment for the tracee (see Section 4.3.4). Subsequently, a call to `fork` executes the actual tracee, whose syscall invocations are constantly monitored and manipulated, according to the defined concept as given in Chapter 4. Arguments for the tracee can either be passed as regular command-line arguments or through `stdin`. In case the `stdin`-input contains control characters, it can be given as a hexadecimal encoded string, which the tracer decodes on passing to the tracee. All output that the tracee writes to `stdout` or `stderr` will be redirected to customizable file paths through a duplicated file descriptor via the `dup2` syscall.

[1]https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/x86/entry/syscalls/syscall64.tbl?h=v5.15.1
[2]https://git.scc.kit.edu/iti-crypto/lehre/sperrips and https://github.com/maik-s/sperrips

5.1.1 Container

As pointed out in Section 4.3.4, the tracee's environment can significantly influence several aspects of the tracee's execution. Therefore, the tracer provides an environment that is reproducibly the same, regardless of the tracee's host system. It utilizes Linux' `unshare` syscall to achieve this. With `unshare`, the tracer process can specify resources through appropriate flags that it wants to disassociate from the current execution context. Depending on the unshared resource, the kernel provides an appropriate new resource to the process, e.g., a new PID namespace. Figure 5.1 on Page 48 illustrates the containerized context created by the tracer. After the `unshare` syscall has been executed, both the tracee and parts of the tracer run in the containerized environment. The current SPERRIPS implementation uses three flags for the invocation of `unshare`. Based on unshare's manpage entry, they are briefly explained in the following.

CLONE_NEWPID

The `CLONE_NEWPID` flag has the effect that all children of this process will be executed within a new PID namespace (see Section 4.3.4). Consequently, the PID numbering starts with one. Since the tracee is the first process created in the new namespace, it will always get PID assigned to it.

CLONE_NEWNS

This flag enables the process to have a different mount namespace than the rest of the system. This means it can mount or unmount file systems without affecting other processes on the system. It is used for swapping the file system's root mount point with a containerized environment. The swap is performed with the `pivot_root` syscall, in combination with an overlay file system. An overlay file system provides a combined view of two underlying mount points, named *lower* and *upper* directory [31]. The lower directory does not need to be writable, whereas the upper directory does. All write operations on the overlay filesystem affect the upper directory. Our implementation uses the overlay file system feature for installing a Debian system into the lower directory. Thereby, the tracee receives the view on a Debian system, and file changes are saved into the upper directory. This enables creating the same system environment on different host systems.
Furthermore, new instances of the special, virtual file systems `proc`, `devtmpfs` and `sysfs` are mounted within the newly created mount namespace.

CLONE_NEWUTS

By setting this flag, processes inside the namespace can set a new hostname. Consequently, the `uname` syscall returns the same set hostname across different host systems.

5.1.2 Tracer and Tracee

SPERRIPS' main component is the tracer, which intercepts each syscall that the tracee executes. Depending on the active phase (recording or replaying phase), different tasks have to be performed (see Section 4.1), but both require a capability to intercept syscalls. For this purpose, Linux's powerful ptrace interface is used (see Section 4.3.1) [32]. With ptrace, the tracer can access the tracee's memory, manipulate memory content or process

register contents and most importantly intercept syscalls on entry and on exit.
A ptrace call expects a so-called request, which specifies the requested action together with belonging data. In this work, we use ptrace by first forking from the current SPERRIPS process and then requesting the child to become a tracee by specifying the `PTRACE_TRACEME` flag. Subsequently, the target application is executed in the child process by calling the `execve` syscall. Now, the child process can be *ptraced*. Different types of requests are used for this implementation, including, but not limited to: `PTRACE_TRACEME`, `PTRACE_PEEKTEXT`, `PTRACE_POKETEXT`, `PTRACE_GETREGS`, `PTRACE_SETREGS`, `PTRACE_SYSCALL`, `PTRACE_GET_SYSCALL_INFO`.

`PTRACE_PEEKTEXT` and `PTRACE_POKETEXT` enable the tracer to read (peek) and write (poke) data to the tracee's virtual address space. Similarly, the requests `PTRACE_GETREGS` and `PTRACE_SETREGS` enable the tracer to retrieve (get) the configuration of the current processor registers or to set an alternative one. `PTRACE_GET_SYSCALL_INFO` provides detailed information about the currently intercepted syscall. We use it to determine whether the tracee is about to enter a syscall or leave it, i.e., if the syscall is intercepted on entry or on exit.

5.2 System Call Handling

Each syscall type requires an appropriate handler for handling the syscall's arguments. As mentioned in Section 4.3.1, each syscall must be evaluated according to its argument types. With a focus on broad support for all syscalls and easy extendability, this implementation chooses a generic approach. First, derived from Table 4.1 on Page 20, an *enumeration* type named `argument_type` has been declared for argument types in file `TypeDefs.h`. Further, all syscall handlers are derived from a base `Handler` class and register themselves in a `HandlerRegistry`, identified by their syscall's opcode. This enables the tracer to retrieve the required handler from the registry, depending on the intercepted syscall. Due to the derived subclasses and their uniform handler interface, all handlers can universally be used. Each handler implements the method `getArgumentTypes`, which returns a vector with `argument_type` enum values, which define the syscall's argument types. Listing 5.1 exemplarily shows the argument type definition of the write syscall handler.

```cpp
std::vector<argument_type> Write::getArgumentTypes()
{
    return std::vector<argument_type>({AT_INT, AT_CHAR_PTR, AT_SIZE_T});
}
```

Listing 5.1: Argument type definition of the `write` syscall handler

On a syscall interception, further detailed information about the intercepted syscall can be requested by issuing a `PTRACE_GET_SYSCALL_INFO` ptrace request[3]. The retrieved `__ptrace_syscall_info`-struct includes the opcode of the intercepted syscall and all of its argument values. Further, it indicates whether the interception happened before the syscall's execution (`PTRACE_SYSCALL_INFO_ENTRY`) or afterward (`PTRACE_SYSCALL_INFO_EXIT`). Intercepting it on entry enables actively altering the execution by manipulating the argument values and memory state before the kernel handles the

[3]This request is available since Linux 5.3. Hence, the implementation requires a system with at least Linux 5.3.

syscall. As mentioned in the previous section, `PTRACE_SETREGS` and `PTRACE_POKETEXT` requests are utilized for altering processor registers and memory content.
Figure 5.2 on Page 49 shows a flow chart to illustrate the syscall handling process as implemented in the `handleSyscall` method in the tracer. This method is invoked on every *on entry* or *on exit* of each syscall. If a syscall is encountered for which no suitable handler is implemented, the interception will be skipped for both recording and replaying, as illustrated in the flow chart. However, the occurrence of the unknown syscall is saved to the file, such that the syscall sequence is preserved to match in the replaying phase.

Implementing the handler for the `write` syscall revealed that it requires specific adaptions to the side-effects introduced by the libc wrapper function for the write syscall. The wrapper function ensures that the amount of passed *count* bytes is actually written to `fd`. However, if SPERRIPS modifies the count argument for the `write` syscall, it may result in fewer written bytes than initially called for the wrapper function. As a consequence, the wrapper function reissues the `write` syscall with the remaining amount of bytes. Since the wrapper function is called before SPERRIPS intercepts the syscall, the passed *count* argument value cannot be modified for the wrapper function prior. However, debugging revealed that the `RBX` register holds the argument value for the wrapper function's argument. Therefore, when replaying a `write` syscall, the `RBX` register must also be updated with the correct value. Nonetheless, this is specific to the libc version and might change in the future due to possible code changes or different compiler output.

Adding Further Syscall Support

Despite of the syscalls of the cat utility (see Section 4.2), the current implementation has handlers implemented for a total of 63 different syscalls. Implementing new handlers consists of the following steps:

1. Create a new class and header file[4]
2. Defining the argument types
3. May modify the return type (default is integer)
4. May implement handling logic for new argument type

If the newly created syscall uses an argument type that has not been implemented before, a new argument type must be added to the enum definition in `TypeDefs.h`. Depending on the argument, different implementation logic has to be implemented, too. This includes argument-value retrieval logic from the tracee, value comparison, recording, and replaying logic. Even though the generic implementation aims to be easily expandable, it might require significant code adaptions. Internally, argument types are grouped by classification (e.g., integer types and structs) to reduce code changes. For example `int` and `size_t` are both atomic integer values and hence grouped as such. Code sections that process atomic integer values, e.g., when reading from registers or comparing values, do this group-wise. Therefore, adding a new atomic number type does not require huge adaptions.

[4]As this is laborious and repetitive, a utility script has been built to ease this work.

5.3 Argument Value Retrieval, Comparison, and Replaying

Section 4.2.2 introduced conventions for comparing argument values. Each argument type needs its individual logic for value retrieval, comparison, and replaying. While integer values can be directly be retrieved from ptrace's `__ptrace_syscall_info` struct, buffer content must be read from the tracee memory. This section explains the approaches of argument value retrieval, comparison, and replaying for different kinds of argument types.

5.3.1 Retrieval

Values for atomic data like `int`, `size_t`, `off_t` and other numeric values including `void*` pointer addresses are most simply to retrieve. They are directly passed via one of the processor registers. Hence, on a ptrace interception on entry, their values can be accessed through the `args` array in the `__ptrace_syscall_info` struct. In case of `const char*`, `struct stat*`, `const void*` the processor registers contain the pointers to the actual data. Thus, they must be retrieved from the tracee's memory. This retrieval is implemented with a `PTRACE_PEEKTEXT` request. Thereby, we can read memory from the tracee's virtual address space in `word`-sized chunks. According to the Conventions 3, 5 and 6 of Section 4.2.2, we implemented the methods `read_string`, `read_struct` and `readnbytes`.

The method `read_string` expects a virtual memory address of the tracee as its only argument. Then, it reads as many bytes from the memory until a null-byte occurs. As a result it returns a `std::string` object, containing the respective string value.

The method `readnbytes` works slightly differently. It also expects a virtual memory address of the tracee and n bytes to fetch. After reading them from the tracee's memory, the method also returns a `std::string` object.

As the name suggests, `read_struct` reads a struct from the tracee's memory. The C++ implementation leverages that the system headers, which contain the struct's definition, can be included directly. This eases determining the necessary amount of bytes to read from memory. Listing 5.2 shows the generic implementation of reading a struct from the tracee's memory. With the help of the active syscall handler, the struct's length is retrieved and passed to `read_struct`, which essentially calls `readnbytes` and organizes the result according to the internal data structure.

```
if (Utils::isStruct(type)) {
    size_t length = handler->getStructLength(type);
    [..] // check for sanitation and an edge case
    read_struct((void*) val, length, argVal);
    [..] // check for related argument value data
    return;
}
```

Listing 5.2: Generic implementation for fetching a struct from the tracee's memory.

Method `read_struct` is of type `void`, but saves the result in one of its return arguments, which stores the read struct as a `std::string` object. When required, the actual struct is crafted by typecasting a `char*` pointer from the `std::string` object (method `c_str()` is utilized) to the corresponding struct's type.

5.3.2 Comparison

When operating in replaying mode, SPERRIPS compares the retrieved data as defined in Section 4.2.2. For implementing the comparing logic, depending on the argument's type, corresponding `Comparers` are implemented. They are derived from the base class `ArgumentComparer` and are specialized for a particular argument type. For instance, the `StructStatComparer` takes two `struct stat*` arguments and compares each member as described in Convention 5 of Section 4.2.2. Implementing a specialized `Comparer` is also the recommended way for future support of other struct types. `BufferComparers` internally use the `compare` method of `std::string`.

5.3.3 Replaying

Similar to value retrieval, values must be replayed differently according to their type. Again, numerical values can be easily altered in the processor registers. Ptrace provides the two requests `PTRACE_GETREGS` and `PTRACE_SETREGS`. With `PTRACE_GETREGS`, a structure is available with members for each available processor register and their current set values. For the manipulation, members of this struct can be altered and then passed the struct to a `PTRACE_SETREGS` request. In the case of buffers and structs, not the processor register values are modified but the referenced memory content within the tracee's virtual address space. Therefore, we implemented a method `write_to_child`, which expects, among others, the target virtual memory address and a `std::string` of bytes to write. Then it issues `PTRACE_POKETEXT` requests with word-sized byte values to write to the tracee's memory. In the case of structs, not single members of a struct are written, but a complete byte representation of the struct. Since structs are stored and replayed as they were fetched from memory, it is essential to note their dependence on the target platform. As mentioned in Section 2.2.2, a compiler may optimize the struct for the target platform by, e.g., introducing padding. Therefore, one needs to make sure that the SPERRIPS instances run on the same target platform since no conversion between different platform representations are implemented yet.

5.4 Data Structures and File Formats

This section describes details of the implemented file formats to store SPERRIPS information. Depending on the active phase, SPERRIPS produces different output files. In the case of the recording phase, it saves all relevant data of the syscall interceptions. These include the order of these syscalls, together with their argument types and values and their return value. In the case of the replaying phase, it outputs a file, which summarizes all encountered differences during the tracee's executions.

To define the custom file formats, Google's protocol buffer (protobuf) library is employed [33]. The library is designed for easily creating platform-independent file exchange formats focusing on small file size, compatibility, and simplicity. It reduces the complexity and effort to correctly implement data objects' serialization and deserialization procedures by generating the necessary primitives by a compiler. To create a custom file format, the desired data structure must be defined with the *protobuf language* in a `.proto` file. Then, the protobuf compiler compiles this file to necessary APIs of the target language for reading and writing data to files. The compiler supports generating code for `C++`, `Python`, `Java` and more languages. This enables to implement compatibility across different applications very simply. After compiling, it is only required to include the generated classes in the project and add the protobuf library as a dependency. Then, creating objects and reading and writing files are handled by the generated protobuf API.

```
syntax = "proto3";
package msc;

message Recording {
  Metadata metadata = 1;
  repeated Syscall syscall = 2;
}

message Syscall {
    SyscallOpcode type = 1;
    ArgumentList args_before = 2;
    ArgumentList args_after = 3;
    Argument return_value = 4;
    bool isExitCall = 5;
}

message ArgumentList {
    repeated Argument args = 1;
}

message Argument {
    argument_type type = 1;
    uint32 int32 = 2;
    uint64 int64 = 3;
    uint64 ptr = 4;
    bytes buffer = 5;
    Related related = 6;
}
```

Listing 5.3: Excerpt from the implemented file format specification with protobuf.

Listing 5.3 shows an excerpt of the file format specifications used for the SPERRIPS implementation. The protobuf compiler translates the `message` blocks to classes of the target language. Hence, we can instantiate objects from them. The inner members of a message are called fields and can be understood as a class attribute. The protobuf language comes with default types for fields, such as `bool`, `uint64` and `bytes` (which is also used for strings), but custom message types can be used as field types as well. Fields that have the `repeated` keyword set are lists that store multiple field items. The protobuf compiler automatically generates the necessary code to add, get and remove list items.
The definition given in Listing 5.3 is used for storing the recorded syscalls, together with their arguments. Each syscall is identified by its opcode. Its argument values and return value are saved both on entry and on exit of the syscall invocation. An argument is identified by an `argument_type` which uses the same definition as in `TypeDefs.h`. Different fields are used to save the argument value depending on its type. For example, when saving an argument from type integer, the type is set to `AT_INT` and the actual value is stored in the `int64` field. Strings or buffer content are stored in the `buffer` field.

Beyond all the syscall data, SPERRIPS also saves some metadata of the system. These include the git commit id of the commit from which the SPERRIPS binary has been built, the time of the execution, the executed tracee with its arguments, and system information retrieved from the uname syscall. The information is collected and stored for easing potential debugging or investigations.

After the replaying phase, SPERRIPS outputs a file that contains all identified differences during the executions. Listing 5.4 shows an excerpt from the protobuf definition for storing the differences. For each differently executed syscall, a DiffPair is created. Such a DiffPair contains all information of the syscall as it was recorded in the recording phase and all information of its appearance in the replaying phase. This enables analyzing the differences afterward.

```
message Differences {
  Metadata metadata = 1;
  repeated DiffPair differences = 2;
  bool equal = 3;
}

message DiffPair {
  Syscall recorded = 1;
  Syscall replayed = 2;
}
```

Listing 5.4: Excerpt from the implemented protobuf specification for saving differences.

5.5 Replaying Nested Structures

To prove the feasibility of replaying nested structures (see Section 4.3.1), support for the msghdr struct has been implemented. This struct is an argument of the recvmsg syscall, which is, e.g., used by the ping application. This SPERRIPS implementation supports recording and replaying executions of ping, with activated ASLR on the host systems. The msghdr struct has three pointer members to other memory regions, which must be recorded for the replaying phase. In the following, they are referred to as *related arguments*. As shown in Listing 5.3, the file format supports saving related argument information within an argument message. Additionally, the file format has been enhanced to save explicitly msghdr structs and their related arguments. Listing 5.5 shows the involved protobuf message types.

```
message Related {
    Struct_msghdr msghdr = 1;
}

message Struct_msghdr {
    bytes msg_name = 1;
    bytes msg_control = 2;
    Struct_iovec iovec = 3;
}

message Struct_iovec {
    bytes iov_data = 1;
}
```

Listing 5.5: Excerpt from the implemented protobuf specification for saving related arguments of the msghdr struct.

The `Struct_msghdr` message owns three fields, one for each related argument. Similar to the arguments message, these fields are set on recording and read from on replaying. With activated ASLR, the SPERRIPS implementation behaves as described in Section 4.3.3. Hence, replaying the recorded original struct data into the tracee's memory updates the pointers to the related arguments with the actual ones from the tracee's randomized address space layout. Subsequently, it updates the memory at these locations with the content of the related arguments, which are `msg_name`, `msg_control` and `iov_data` from `Struct_iovec`.

This demonstrates the feasibility of recording and replaying nested structs with activated ASLR. However, even though ping is a program with network functionality, the SPERRIPS implementation does not comply at this point with the networking conception as composed in Section 4.3.6.

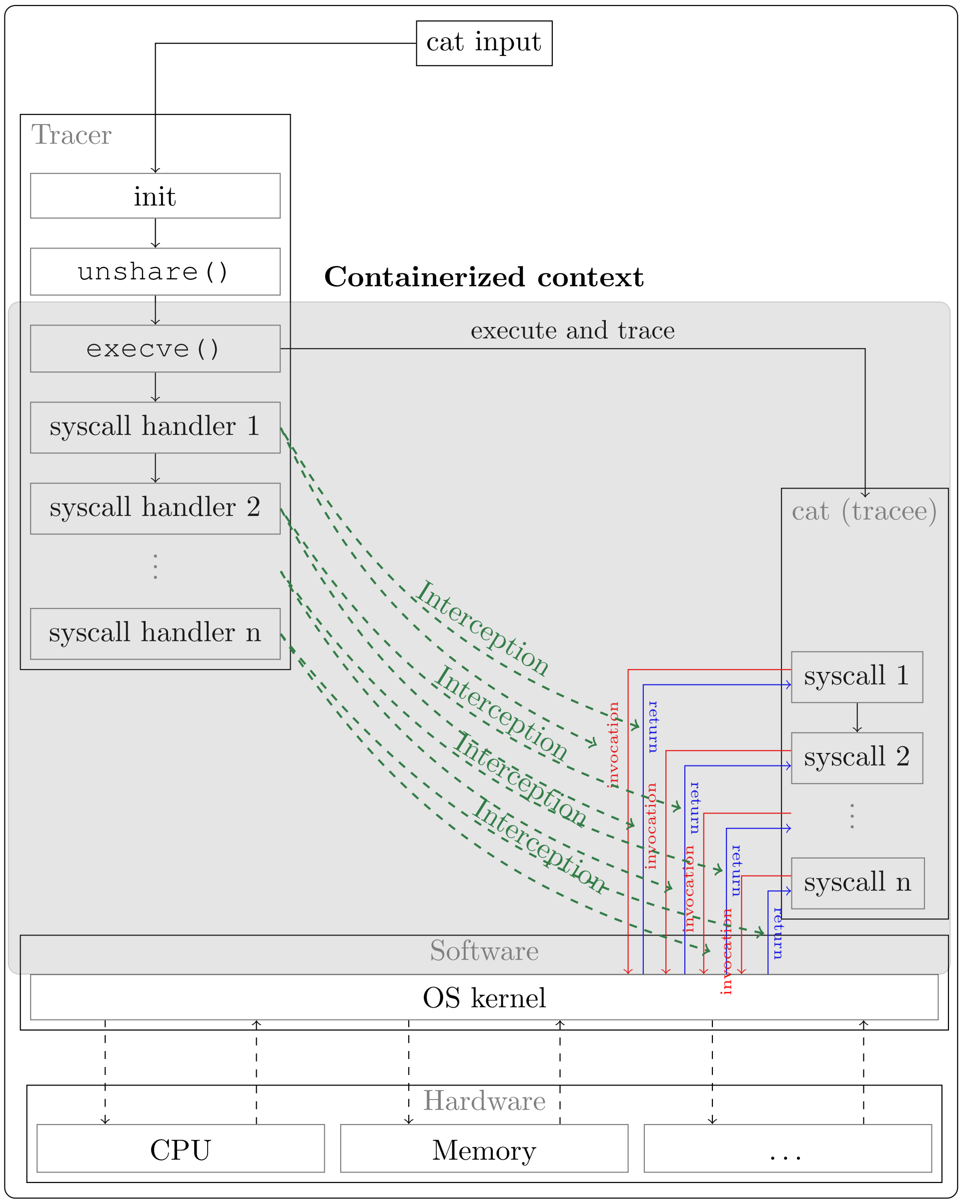

Figure 5.1 Illustration of the scope of the created containerized environment.

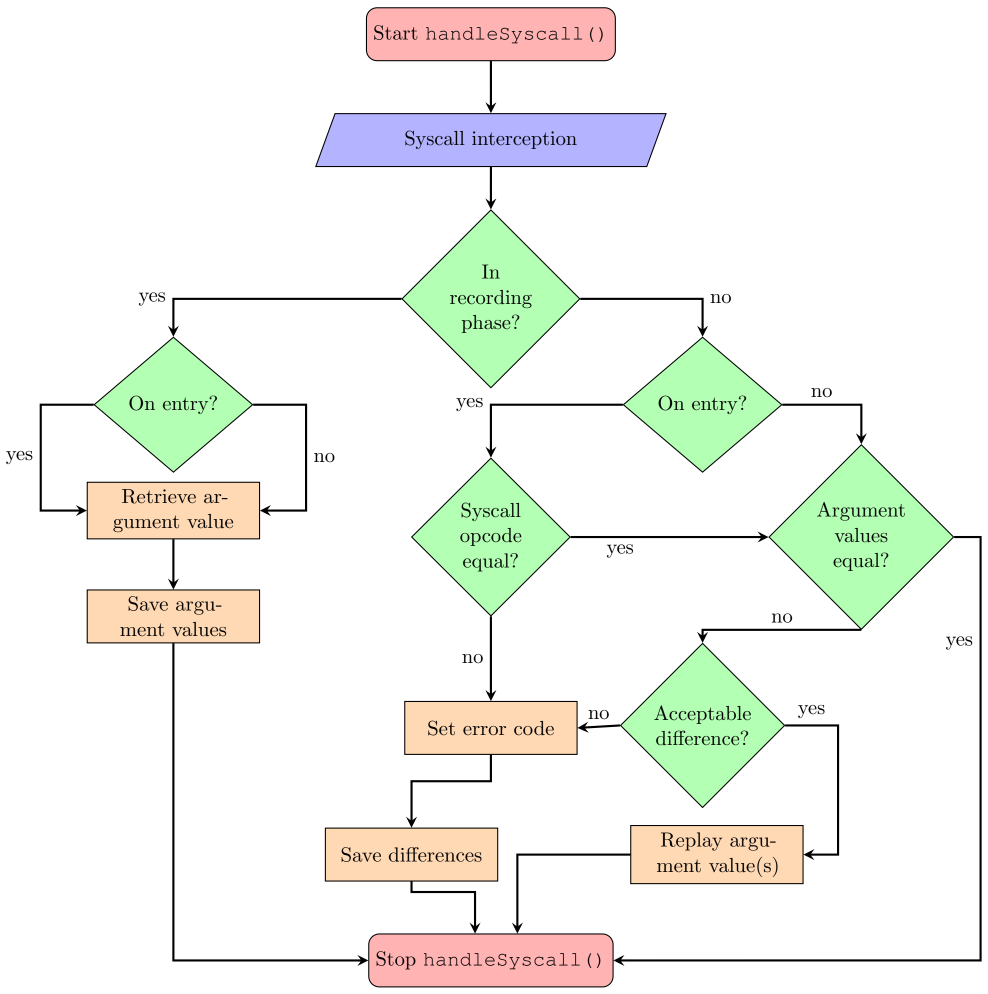

Figure 5.2 Flow chart of the syscall handling process.

6. Evaluation

In this chapter, the implementation of SPERRIPS is evaluated by performing multiple verifications of different applications. The following section describes the evaluation setups and used methodology. Then, two sections discuss the results of both evaluation scenarios. The raw data of the evaluation are available in the project's code repository[1].

6.1 Testing Setup and Methodology

The conducted evaluation separates into two different testing scenarios. In the first setup, SPERRIPS runs on two different systems with varying Linux kernel versions, and one of the kernels has malicious functionality included. In the second setup, SPERRIPS runs on two systems with the same Linux kernel version, but with CPUs of different vendors.

Beyond the initial aim of supporting the cat application, the given implementation supports at least the following applications: echo, cat, hostname[2] and partly ping. The applications are used for testing the concept's and implementation's feasibility. Programs echo and cat are part of the coreutils[3] collection, while ping belongs to iputils[4]. All applications are compiled from their source code. The coreutils applications are compiled from version 9.0 (git tag v9.0) and iputils is compiled as version `20210722-31-ge70a786` from git commit `e70a786e8`.

Originally, ping setups a timer before sending the last package to get notified in case the packet will be lost during transmission. As seen in Section 4.3.6, timers require specific handling, which is currently unavailable in the given implementation. Therefore, we introduced a minor modification to ping, such that it does not use the `setitimer` syscall. This modification circumvents the code section that setups a timer without changing ping's desired behavior. The applied modification is given in Listing B.1 in Appendix B. The execution of the ping application happens with an eased ruleset since not all syscalls of the ping application have a full-functioning handler implemented, and the network conception

[1]https://git.scc.kit.edu/iti-crypto/lehre/sperrips and https://github.com/maik-s/sperrips
[2]http://deb.debian.org/debian/pool/main/h/hostname/hostname_3.23.tar.gz
[3]https://github.com/coreutils/coreutils
[4]https://github.com/iputils/iputils

(see Section 4.3.6) is currently not implemented. Therefore, differences in the execution of ping are overcome by always replaying the recorded values.

As mentioned in Section 4.3.4, we use the debootstrap utility on both systems to install a Debian 11 System into the containerized environment to provide equal system environments.

6.2 Scenario 1: Different Kernel Versions

In this scenario, two different kernel versions are installed on the two systems with the specifications given in Table 6.1. Furthermore, the kernel of System B also includes intentional modifications which introduce incorrect behavior. The modifications are documented in Section 6.2.1 and are used to demonstrate SPERRIPS' capability of detecting their effects.

The kernel versions 5.4.158 and 5.15.1 have been selected because they have the greatest gap between their version numbers and are both stable and long-term releases. Further, version 5.4.158 still fulfills the implementation requirement to run on Linux 5.3 or newer. The kernels are compiled from their official source code[5]. Before compilation, it has been verified that the source code's signatures are correct.

System	Phase	Kernel Version	Modifications
System A	Recording	5.4.158	None
System B	Replaying	5.15.1	Includes malicious modifications

Table 6.1 System setups details for evaluation Scenario 1.

6.2.1 Kernel Modifications

In total, two modifications were introduced to the source code of kernel 5.15.1 on System B. The first change affects the `read` syscall, such that it always returns `This is content from a malicious kernel, modified in ksys_read(); !!\n`, if the read happens on a file in a specific directory. This affected directory is hardcoded into the kernel by identifying it through its inode number. Consequently, this modification is dependent on the system that the kernel runs on. The inode numbers were determined by the stat utility. The complete modification is given in Listing A.1 in Appendix A.
On every read, the kernel checks whether the file's directory has the inode number `131105` and its parent inode number `1421940`. If this is the case, the above content is returned instead of the original file content. The check on the parent's inode number is done to match only the desired directory. Otherwise, each directory with the inode number `131105` would be affected by the modification on any filesystem.

The second modification touches the `execve` syscall. It has the effect that the kernel executes a different binary instead of `/usr/bin/ping`. If `/usr/bin/ping` is issued to be executed via the execve syscall, then the kernel executes `/root/modping` instead. We use this modification to simulate an attacker that exchanges a trustworthy binary with a malicious one. The exchanged `ping` binary is modified such that it always pings the IP address `127.1.1.1`[6]. The kernel modification is documented in Appendix A.2 and ping's modification in Appendix B.2.

[5]https://www.kernel.org

[6]This modification applies if ping's arguments are of the form `ping -c4 <ipaddr>`

6.2.2 Evaluation

The recording phase is performed on System A and the replaying phase on System B. We store the file-to-print in the directory affected by the kernel patch as mentioned in Section 6.2.1. In particular, this is the `/root/dir` directory in the containerized environment on System B. Therefore, we expect to detect both the modified file read and the differently pinged target IP addresses on System B.

The evaluation shows that all syscall sequences were equal on both systems for all applications. However, on some invocations, differences in the argument values have been detected by our tool either on entry or on exit of a syscall. The differences between the application's executions on System A and System B are given in the following tables. For all applications, the `uname` syscall, induced by the dynamic linker due to the disabled VDSO (see Section 4.3.5), returned different values for the `utsname` struct. This difference is expected (and accepted), as the kernel version name strings are different. All other differences are given and discussed in greater detail in the following.
For the executions of echo and hostname, our tool only detected differences in the result of the `uname` syscall. Table 6.2 summarizes the differences of the hostname binary. For echo, the `uname` syscall happened only once on application startup and is thus not listed in a separate table.

Syscall	Occurrence	Note	System A	System B
`uname`	On exit	The returned kernel version strings and build times are different	5.4.158-maiks	5.15.1-maiks-malicious
`uname`	On exit	The returned kernel version strings and build times are different	5.4.158-maiks	5.15.1-maiks-malicious

Table 6.2 Differences of hostname's execution.

For the execution of cat, our tool detected one difference, besides the differing `uname` syscall as listed in Table 6.3. On System B, the `read` syscall did not return the content of the given file but the string from the introduced kernel patch. Therefore, it successfully detected the malicious behavior induced by the kernel modification.

Syscall	Occurrence	Note	System A	System B
`uname`	On exit	The returned kernel version strings and build times are different	5.4.158-maiks	5.15.1-maiks-malicious
`read`	On exit	The kernel of System B returned the string form the malicious modification	`Hello world!\n`	`This is content from a malicious kernel, modified in ksys_read(); !!\n`

Table 6.3 Differences of cat's execution.

The execution of ping revealed the most differences, as Table 6.4 and Page 56 shows. Three `prctl` syscalls had different values in their second argument. One of these `PR_CAPBSET_READ`[7] option calls even returned different return values, namely `-22 (EINVAL) Invalid argument` vs. `1` (success[8]). The different behavior can be explained after debugging the process and studying the involved source code. When executing ping, the dynamic linker includes the library libcap. On library initialization, it searches the available capabilities on the system by performing a binary search on the `prctl` syscall and checking for negative return values to determine whether a capability is available or not[9] [10]. The kernel returns `-22 (EINVAL)`, if a tested capability is not available on the system. Since different kernel versions run on the systems, they thus contain varying definitions of `CAP_LAST_CAP`. On Linux 5.4.158 it is set to `CAP_AUDIT_READ`[11] (37) and on Linux 5.15.1 to `CAP_CHECKPOINT_RESTORE` (40)[12]. Therefore, System A returns `-1` for tested capabilities greater than 37 and System B for those greater than 40. Further, as we replay the recorded values of the `prctl` syscall together with their return values, the binary search retrieves wrong feedback of the available capabilities and hence falsely increases the counter.
We see on the arguments of the `connect` and `sendto` syscalls that the kernel executed the altered version of ping instead of `/usr/bin/ping` because System B connects to the altered IP address `127.1.1.1` instead of the given parameter `127.0.0.1`. The other differences result from varying runtime-dependent addresses due to activated ASLR or varying system times. For readability's sake, they are not listed in the table.

System	**Phase**	**Kernel**	**CPU**
System A	Recording	5.15.1	Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
System B	Replaying	5.15.1	AMD EPYC Processor @ 2445.406 Mhz

Table 6.5 System setups details for evaluation Scenario 2. CPU information is retrieved from `/proc/cpuinfo`.

6.3 Scenario 2: CPUs of Different Vendors

In the second scenario, both systems run the same Linux kernel, namely version 5.15.1, but consist of varying hardware specifications. System A is powered by an Intel processor and System B by an AMD processor, as Table 6.5 summarizes. Again, the kernel's source code is verified through its signature and built from source code without modifications.

The executions of echo, hostname, and cat happened without any differences. However, ping encountered those listed in Table 6.6. Most differences are similar to the ones from the first evaluation scenario. But the `prctl` syscall lead again to a difference, even though the kernel and all shared libraries are in the same version.

[7]Value 23 as the first argument resolved to `PR_CAPBSET_READ`.
[8]Success means, the thread posses the capability, identified by the second argument.
[9]https://git.kernel.org/pub/scm/linux/kernel/git/morgan/libcap.git/tree/libcap/cap_alloc.c?h=libcap-2.32#n20
[10]https://git.kernel.org/pub/scm/linux/kernel/git/morgan/libcap.git/tree/libcap/libcap.h?h=libcap-2.32#n212
[11]https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/uapi/linux/capability.h?h=v5.4.158#n370
[12]https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/uapi/linux/capability.h?h=v5.15.1#n420

On the Intel system, the fifth argument of the syscall or the `R8` register holds the value `0`, while it is set to `0x800000` on the AMD system. The root cause was identified by debugging the application. The libc library in version 2.3 determines the vendor of the used CPU on initialization[13]. Further, it calculates the available caches among the threads. As the library executes different operations, depending on the CPU vendor, the `R8` register contains `0x800000` as a left-over on AMD systems. Later, on invocation of the `prctl` syscall, the value is still present and technically the fifth argument of the syscall. However, since its first argument is `PR_CAPBSET_READ` the syscall only expects and consumes the corresponding second parameter[14] and not the fifth one. Therefore, it does not affect the syscall's execution.

Syscall	**Occurrence**	**Note**	**System A**	**System B**
`readlink`	On exit	The buffer, filled by the kernel, contains runtime-dependent addresses.	-	-
`prctl`	On Entry, on exit	Differences in the fifth parameter	0	0x800000
`getrandom`	On exit	Different returned random	-	-
`getsockname`	On exit	Different port number	32174	692
`rt_sigaction` (repeats two more times)	On exit	Struct sigaction contains different runtime depended addresses	-	-
`clock_gettime`	On exit	Struct timespec contains different clock times	-	-
`clock_gettime`	On exit	Struct timespec contains different clock times	-	-
`gettimeofday`	On exit	Struct timeval contains different times	-	-
`recvmsg`	On exit	Struct msghdr contains different addresses, iovdata contain different IP msgheader data	-	-
`clock_gettime`	On exit	Struct timespec contains different clock times	-	-
`clock_gettime`	On exit	Struct timespec contains different clock times	-	-
`gettimeofday`	On exit	Struct timeval contains different times	-	-

Table 6.6 Differences of ping's execution with removed duplicate entries.

[13]https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/x86/cacheinfo.c;h=e3e8ef27bb0bb78028637df586906a6947d4cf09;hb=9ea3686266dca3f004ba874745a4087a89682617#l1703

[14]https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/security/commoncap.c?h=v5.15.1#n1275

Syscall	Occurrence	Note	System A	System B
`uname`	On exit	The returned kernel version strings and build times are different	5.4.158-maiks	5.15.1-maiks-malicious
`readlink`	On exit	The end of the buffer, filled by the kernel, contains runtime-dependent addresses.	-	-
`prctl`	On entry, on exit	Differences in the second parameter	36	44
`prctl`	On entry, on exit	Differences in the second parameter and return value.	Second parameter: 38, Return value: -22 (EINVAL) Invalid argument	Second parameter: 46, Return value: 1
`prctl`	On entry, on exit	Differences in the second parameter.	37	45
`getrandom`	On exit	Different returned random	-	-
`connect`	On exit	Different target addresses	Connects to 127.0.0.1	Connects to 127.1.1.1
`getsockname`	On exit	Different port number	27100	6119
`rt_sigaction` (repeats two more times)	On exit	Struct sigaction contains different runtime depended addresses	-	-
`clock_gettime`	On exit	Struct timespec contains different clock times	-	-
`clock_gettime`	On exit	Struct timespec contains different clock times	-	-
`gettimeofday`	On exit	Struct timeval contains different times	-	-
`sendto`	On exit	Struct sockaddr contains different ip addresses	127.0.0.1	127.1.1.1
`recvmsg`	On exit	Struct msghdr contains different addresses, iovdata contain different IP msgheader data	-	-
`clock_gettime`	On exit	Struct timespec contains different clock times	-	-
`clock_gettime`	On exit	Struct timespec contains different clock times	-	-
`gettimeofday`	On exit	Struct timeval contains different times	-	-

Table 6.4 Differences of ping's execution.

7. Related Work

This section presents other research that is related to our approach. Their covered aspects include recording/replaying systems, reproducibly executing applications, and moving processes to another system. Their main differences lay in the motivation, the trust model, and thus differing conceptions or outdated technical standards.

Cornelis et al. published their TORNADO approach for replaying applications on a syscall level, including an implementation [24]. However, their motivation, assumptions, and goals differ from ours and thus the conception. Their motivation is driven by creating a recording/replaying tool for debugging purposes, such that an application can be relaunched deterministically. Hence, they always replay the recorded data, such that the second execution behaves the same as the recorded one. Our approach, though, intends to verify the exact same behavior. Hence, it rather verifies that both executions compute the same results on a syscall level, and only acceptable differences are mitigated by replaying recorded values. Further, due to the age of the TORNADO approach, ASLR was not present at this time. Cornelis et al. implemented TORNADO for Linux 2.4, while ASLR was introduced with Linux 2.6.12 in 2005 [15]. Hence, our work further contributes support for today's prevalent enabled ASLR.

Jockey is a user-mode library for recording and replaying Linux applications to re-execute them deterministically for debugging purposes [25]. Target applications must be compiled together with the Jockey library and thus require modification. Then, it records the invocation to a selected subset of the Linux syscalls and CPU instructions. It specializes on those syscalls that introduce non-determinism by design, e.g., `gettimeofday`. In contrast to our work, it does not verify the same results of syscall executions. Moreover, it replays the recorded values "without actually executing the calls" [25].

The Echo approach is also designed for deterministically replaying Linux applications with explicit support for multi-threaded applications and is implemented as a kernel module [26]. Similar to Jockey, it does not re-execute syscalls in the replaying phase but returns the recorded data. This is fundamentally different from our approach as we explicitly compare the results of re-executed syscalls. Like TORNADO, Jockey and Echo do not deal with activated ASLR as it was not broadly available at the time of their creation or publishing.

Steven et al. developed the JRapture approach for recording/replaying on the abstraction level of Java Virtual Machine (JVM) to host system interaction [7]. Again, they are motivated by relaunching a Java application deterministically for the aims of software testing. JRapture implements the recording and replaying mechanisms at the interaction between the JVM and the operating system.

Ročkai et al. developed a system to execute POSIX compatible applications reproducibly [34]. For this purpose, they designed and implemented the POSIX-compatible DiOS operating system. It is a specially designed operating system and guarantees that multiple application executions execute the same deterministic trace of CPU instructions if the application receives the same input. Their operating system supports a wide range of POSIX syscalls and even multi-threaded applications. Creating such an approach is driven by software testing and verification goals. Our work separates from DiOS in multiple aspects. We do not require or guarantee with SPERRIPS that two application executions are equal in their CPU instruction sequence. We instead focus on the semantical usage of syscalls. Further, we compare the recorded syscall sequences explicitly to detect potential differences.

The CRIU project[1] is weakly related to our approach, as it aims to migrate a running process or container from one Linux host to another. CRIU is an acronym and stands for "checkpoint/restore in userspace". As the name suggests, it freezes the current state of a process for restoring it on another host, with user-space capabilities only. Implementing CRIU deals with similar challenges as SPERRIPS, regarding the different environment states. For example, restoring the exact same PID on the target host. Reber describes the challenging procedure of recreating the PID in [35]. Before the clone3 syscall enabled to specify a custom PID for a new process, the implementation called as many fork syscalls as needed to imitate the forking tree on the source hosts.

From a theoretical point of view, a competing approach to ours is Multi Party Computation. With MPC, multiple parties compute the result of a function without gaining information about the data of other involved parties [6]. For our goal, two applications could use MPC for calculating results of individual processing steps to agree on equal values. Thus, the target application requires implementing MPC functionality for its domain-specific tasks and hence requires adaptions. In contrast, SPERRIPS represents a generic approach, which does not require modifications to the target application.

[1]http://criu.org

8. Summary and Future Work

This work demonstrates the feasibility of a new generic approach for verifying the execution of applications across two different systems by verifying the inner process state on the abstraction level of system calls. This work proposes a conception and implementation for such an approach. However, the implementation is not understood as a complete solution yet but rather to demonstrate its feasibility on the example of cat. The work is motivated by the goal of being independent of trusting the vendors of involved hardware and software components for the correctness of the components. Instead, it helps convince oneself by verifying the application's execution. The approach's basic principle is detecting differences in the execution of an application on two different systems. It assumes at most one of the two systems to behave incorrectly, as it might contain bugs or backdoors affecting either hardware or software components.

Therefore, it operates on the abstraction level of system calls in two phases, similar to recording/replaying systems for deterministic program execution. The first phase traces each executed syscall of the target application and obtains the syscalls' argument values. In the second phase, it compares the recorded values to the actual ones from the re-executed application on a different system and thus verifies the equality of the executions. In contrast to recording/replaying systems, our approach explicitly re-executes each syscall in the second phase to produce the same side-effects in the kernel as in the first phase for comparing both invocations and results. Just replaying is explicitly not desired and happens only on acceptable differences, caused by different environment specifics. Definitions of acceptable and unacceptable differences are given for syscalls leveraged by the cat application as a proof-of-concept.

To reduce potential differences within the two executions in the first place, possible sources of non-determinism for program execution have been identified and analyzed. Then, applicable mechanisms for removing the non-determinism have been designed and implemented. Furthermore, the approach supports activated ASLR on the systems, introducing differences to the applications address space layout by design.

The implementation works on modern operating systems with Linux kernel version 5.3 and newer. Even though the implementation is generic, it does not support all currently available syscalls yet. As a proof-of-concept, it allows to verify the execution of echo, hostname, cat, and partly ping. Adding support for new syscalls requires manual labor for adding a new syscall handler. It is required to correctly handle the involved argument types, which might require new logic to operate on involved data structures of the kernel. Requiring this effort results from the vast amount of different kernel structure types available, which are not all implemented yet.

Even though the conception and evaluation utilize a Debian 11 environment for the containerized environment, this does not restrict evaluable applications. It is also possible to configure other containerized environments that are suitable to the tracee.

The limitations of the current implementation are lacking support of network connections, signals and timers, and multiple threads within the target application. However, this work proposed theoretical conceptions for these sources of non-determinism nonetheless. The precise details and implementation are left for future work since it requires a different architecture, including a trusted third-party decisioner. Moreover, such an architecture also increases the approach's security regarding the CIA security goals. An interesting research question for future work is implementing and performing TLS-secured network connections over the third-party decisioner.

A conducted evaluation proved the approach's feasibility of detecting differences in application executions. It successfully detected all effects of intentionally introduced malicious modifications into a Linux kernel. Moreover, it detected previously unknown side-effects within the microarchitectural state induced by libc's implementation, depending on the CPU's vendor Intel or AMD.

A. Kernel modifications

```
diff --git a/linux-5.15.1/fs/read_write.c b/linux-5.15.1-mod/fs/read_write.c
index af057c5..0fdd516 100644
--- a/linux-5.15.1/fs/read_write.c
+++ b/linux-5.15.1-mod/fs/read_write.c
@@ -620,10 +620,25 @@ ssize_t ksys_read(unsigned int fd, char __user *buf, size_t count)
   pos = *ppos;
   ppos = &pos;
  }
- ret = vfs_read(f.file, buf, count, ppos);
- if (ret >= 0 && ppos)
-   f.file->f_pos = pos;
- fdput_pos(f);
+                // /root/evils/root
+ if (f.file->f_path.dentry->d_parent->d_inode->i_ino == 131105 && f.file->f_path.dentry->d_parent->d_parent->d_inode->i_ino == 1421940) {
+   const char* malicious = "This is content from a malicious kernel, modified in ksys_read(); !!\n";
+   size_t actual_count = 69;
+   if (f.file->f_pos) {
+     f.file->f_pos = 0;
+     ret = 0;
+   } else {
+     copy_to_user((void*)buf, (void*)malicious, actual_count);
+     f.file->f_pos = actual_count;
+     ret = actual_count;
+   }
+
+ } else {
+   ret = vfs_read(f.file, buf, count, ppos);
+   if (ret >= 0 && ppos)
+     f.file->f_pos = pos;
+   fdput_pos(f);
+ }
 }
 return ret;
}
```

Listing A.1: Modifications to the read syscall.

```
diff --git a/linux-5.15.1/fs/exec.c b/linux-5.15.1-mod/fs/exec.c
index a098c13..d2799f8 100644
--- a/linux-5.15.1/fs/exec.c
+++ b/linux-5.15.1-mod/fs/exec.c
@@ -2065,6 +2065,13 @@ SYSCALL_DEFINE3(execve,
    const char __user *const __user *, argv,
    const char __user *const __user *, envp)
 {
+ const char kfilename[13] = {};
+ copy_from_user((void*)kfilename, filename, 13);
+ if (strncmp(kfilename, "/usr/bin/ping", 13) == 0) {
+   const char* maliciousfile = "/root/modping";
+   int ret = copy_to_user((void*)filename, (const char* __user) maliciousfile,
     13);
+ }
+
  return do_execve(getname(filename), argv, envp);
 }
```

Listing A.2: Kernel modification, to execute a different binary than `/usr/bin/ping`.

B. Ping modifications

```
diff --git a/ping/ping_common.c b/ping/ping_common.c
index 357c39d..f5713d4 100644
--- a/ping/ping_common.c
+++ b/ping/ping_common.c
@@ -571,7 +571,7 @@ int main_loop(struct ping_rts *rts, ping_func_set_st *fset, socket_st *sock,
                /* Check exit conditions. */
                if (rts->exiting)
                        break;
-               if (rts->npackets && rts->nreceived + rts->nerrors >= rts->npackets)
+               if (rts->npackets && rts->ntransmitted >= rts->npackets)
                        break;
                if (rts->deadline && rts->nerrors)
                        break;
```

Listing B.1: Ping patch to avoid calling the `setitimer` syscall.

```
diff --git a/ping/ping.c b/ping/ping.c
index 0655bf4..a764748 100644
--- a/ping/ping.c
+++ b/ping/ping.c
@@ -609,6 +609,8 @@ int ping4_run(struct ping_rts *rts, int argc, char **argv, struct addrinfo *ai,
        char hnamebuf[NI_MAXHOST];
        unsigned char rspace[3 + 4 * NROUTES + 1];      /* record route space */
        uint32_t *tmp_rspace;
+       char* modified_target = "127.1.1.1";
+       argv = &modified_target;

        if (argc > 1) {
                if (rts->opt_rroute)
```

Listing B.2: Ping modification to always ping IP-Address `127.1.1.1`.

Bibliography

[1] Intel cooperation. *5th Generation Intel® Core™ Processor Family, Intel® Core™ M-Processor Family, Mobile Intel® Pentium® Processor Family, and Mobile Intel® Celeron® Processor Family*. Accessed on 2021-12-07. 2020. URL: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/5th-gen-core-family-spec-update.pdf.

[2] Peter H. Hochschild et al. "Cores that don't count". In: *Proc. 18th Workshop on Hot Topics in Operating Systems (HotOS 2021)*. 2021.

[3] Jordan Robterson and Michael Riley. *The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies*. Accessed on 2021-12-07. 2018. URL: https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies.

[4] Dhwani Mehta et al. "The Big Hack Explained: Detection and Prevention of PCB Supply Chain Implants". In: *J. Emerg. Technol. Comput. Syst.* 16.4 (Aug. 2020). ISSN: 1550-4832. DOI: 10.1145/3401980. URL: https://doi.org/10.1145/3401980.

[5] Frank Cornelis et al. "A Taxonomy of Execution Replay Systems". In: (Jan. 2003).

[6] Ronald Cramer and Ivan Damgård. "Multiparty Computation, an Introduction". In: *Contemporary Cryptology*. Basel: Birkhäuser Basel, 2005, pp. 41–87. ISBN: 978-3-7643-7394-8. DOI: 10.1007/3-7643-7394-6_2. URL: https://doi.org/10.1007/3-7643-7394-6_2.

[7] John Steven et al. "JRapture: A Capture/Replay Tool for Observation-Based Testing". In: *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '00. Portland, Oregon, USA: Association for Computing Machinery, 2000, pp. 158–167. ISBN: 1581132662. DOI: 10.1145/347324.348993. URL: https://doi.org/10.1145/347324.348993.

[8] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. 9th. Wiley Publishing, 2012. ISBN: 1118063333.

[9] Intel cooperation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Accessed on 2021-12-02. 2016. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

[10] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*. Accessed on 2021-12-02. 2021. URL: https://www.amd.com/system/files/TechDocs/24594.pdf.

[11] Mojtaba Bagherzadeh et al. "Analyzing a decade of Linux system calls". In: *Empirical Software Engineering* 23.3 (June 2018), pp. 1519–1551. ISSN: 1573-7616. DOI: 10.1007/s10664-017-9551-z. URL: https://doi.org/10.1007/s10664-017-9551-z.

[12] H.J Lu et al. *System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models) Version 1.0*. Accessed on 2021-12-07. 2021. URL: https://gitlab.com/x86-psABIs/x86-64-ABI.

[13] Michael Kerrisk. *Linux manual pages: section 2*. Accessed on 2021-10-28. URL: https://man7.org/linux/man-pages/dir_section_2.html.

[14] Michael Kerrisk. *vdso(7) - Linux manual page*. Accessed on 2021-11-27. 2021. URL: https://man7.org/linux/man-pages/man7/vdso.7.html.

[15] diegocalleja. *Linux_2_6_12 - Linux Kernel Newbies*. Accessed on 2021-12-01. 2019. URL: https://kernelnewbies.org/Linux_2_6_12.

[16] The kernel development community. *Programming Language*. Accessed on 2021-11-30. 2021. URL: https://www.kernel.org/doc/html/latest/process/programming-language.html.

[17] glibc developers. *The GNU C Library (glibc)*. Accessed on 2021-12-07. 2021. URL: https://www.gnu.org/software/libc/.

[18] Peter Prinz and Tony Crawford. “C in a Nutshell: The Definitive Reference”. In: 2015.

[19] Kostya Serebryany et al. “SiliFuzz: Fuzzing CPUs by proxy”. In: 2021. URL: https://github.com/google/fuzzing/blob/master/docs/silifuzz.pdf.

[20] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[21] Loïc Duflot. “CPU bugs, CPU backdoors and consequences on security”. In: *Journal in Computer Virology* 5.2 (May 2009), pp. 91–104. ISSN: 1772-9904. DOI: 10.1007/s11416-008-0109-x. URL: https://doi.org/10.1007/s11416-008-0109-x.

[22] Christopher Domas. “Hardware Backdoors in x86 CPUs”. In: (July 2018). Accessed on 2021-12-07. URL: https://i.blackhat.com/us-18/Thu-August-9/us-18-Domas-God-Mode-Unlocked-Hardware-Backdoors-In-x86-CPUs-wp.pdf.

[23] Marc Rittinghaus. “SimuBoost: Scalable Parallelization of Functional System Simulation”. PhD thesis. Karlsruher Institut für Technologie (KIT), 2019. 259 pp. DOI: 10.5445/IR/1000097700.

[24] Frank Cornelis, Michiel Ronsse, and Koen De Bosschere. “TORNADO: A Novel Input Replay Tool.” In: Jan. 2003.

[25] Yasushi Saito. “Jockey: A User-Space Library for Record-Replay Debugging”. In: *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*. AADEBUG’05. Monterey, California, USA: Association for Computing Machinery, 2005, pp. 69–76. ISBN: 1595930507. DOI: 10.1145/1085130.1085139. URL: https://doi.org/10.1145/1085130.1085139.

[26] E. A. Itskova. “Echo: A deterministic record/replay framework for debugging multi-threaded applications”. In: 2006.

[27] Michiel Ronsse et al. “Record/Replay for Nondeterministic Program Executions”. In: *Commun. ACM* 46.9 (Sept. 2003), pp. 62–67. ISSN: 0001-0782. DOI: 10.1145/903893.903895. URL: https://doi.org/10.1145/903893.903895.

[28] Michael Kerrisk. *namespaces(7) — Linux manual page*. Accessed on 2021-12-07. 2021. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html.

[29] Robert H. B. Netzer and Barton P. Miller. “Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs”. In: *J. Supercomput.* 8.4 (Dec. 1994), pp. 371–388. ISSN: 0920-8542. DOI: 10.1007/BF01901615. URL: https://doi.org/10.1007/BF01901615.

[30] Manuel Blum. “Coin Flipping by Telephone a Protocol for Solving Impossible Problems”. In: *SIGACT News* 15.1 (Jan. 1983), pp. 23–27. ISSN: 0163-5700. DOI: 10.1145/1008908.1008911. URL: https://doi.org/10.1145/1008908.1008911.

[31] Neil Brown. *Linux_2_6_12 - Linux Kernel Newbies*. Accessed on 2021-12-02. 2021. URL: https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html.

[32] Michael Kerrisk. *ptrace(2) - Linux manual page*. Accessed on 2021-10-26. 2021. URL: https://man7.org/linux/man-pages/man2/ptrace.2.html.

[33] Google Developers. *Protocol Buffers | Google Developers*. Accessed on 2021-10-28. URL: https://developers.google.com/protocol-buffers.

[34] Petr Ročkai et al. “Reproducible execution of POSIX programs with DiOS”. In: *Software and Systems Modeling* 20.2 (Apr. 2021), pp. 363–382. ISSN: 1619-1374. DOI: 10.1007/s10270-020-00837-y. URL: https://doi.org/10.1007/s10270-020-00837-y.

[35] Adrian Reber. *CRIU and the PID dance*. Accessed on 2021-12-07. 2019. URL: https://lisas.de/˜adrian/criu-and-the-pid-dance-article.pdf.