TU WIEN Informatics

Abfangen und Manipulieren von System-/Funktionsaufrufen in Linux-Systemen

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Lorenz Stechauner
Matrikelnummer 12119052

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner
Mitwirkung: Univ.Ass. Dipl.-Ing. Florian Mihola, BSc

Wien, 4. September 2025

Lorenz Stechauner	Peter Puschner

TU WIEN Informatics

Intercepting and Manipulating System/Function Calls in Linux Systems

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Lorenz Stechauner
Registration Number 12119052

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner
Assistance: Univ.Ass. Dipl.-Ing. Florian Mihola, BSc

Vienna, September 4, 2025

Lorenz Stechauner	Peter Puschner

Erklärung zur Verfassung der Arbeit

Lorenz Stechauner

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Das einzige verwendete generative KI-Tool war ChatGPT, welches zum Korrekturlesen, bei der Umformulierung einzelner Sätze und Paragraphen, und zum Erstellen der Kurzfassung verwenet wurde.

Wien, 4. September 2025

Lorenz Stechauner

Kurzfassung

Diese Arbeit stellt Methoden zum Abfangen und Manipulieren von Funktionsaufrufen in Linux-Systemen vor, mit dem Schwerpunkt auf der Unterstützung automatisierter Tests von Studierendenprogrammen im Rahmen der Betriebssysteme-Lehrveranstaltung. Der zentrale Beitrag ist die Entwicklung von `intercept.so`, einer Shared Object-Bibliothek, die über den Mechanismus `LD_PRELOAD` eingebunden wird und so das Abfangen von Funktionsaufrufen zur Laufzeit ermöglicht, ohne dass Änderungen am Programm oder dessen Quellcode erforderlich sind. Die Bibliothek protokolliert detaillierte Informationen über Funktionsaufrufe, einschließlich Argumenten, Rückgabewerten und Aufrufstellen, wodurch eine präzise Rekonstruktion und Analyse der Programmausführung möglich wird. Darüber hinaus erlaubt sie die kontrollierte Manipulation abgefangener Aufrufe – etwa durch das Ändern von Argumenten, das Erzwingen von Fehlern oder das Simulieren von Rückgabewerten – mittels eines einfachen Kommunikationsprotokolls. Dies ermöglicht robuste automatisierte Tests zur Überprüfung von Fehlerbehandlung und Ressourcenverwaltung. Leistungsmessungen zeigen, dass der Ansatz in praktischen Szenarien nur minimale Laufzeitkosten verursacht und sich somit gut für den Einsatz in der Lehre eignet. Im Vergleich zu bestehenden Werkzeugen wie `strace` oder `ltrace` bietet die Lösung eine größere Flexibilität, da sie das Abfangen mit Manipulationsmöglichkeiten kombiniert. Insgesamt stellt der Ansatz eine flexible und transparente Grundlage für automatisierte Tests von Low-Level-C-Programmen dar, verbessert deren Zuverlässigkeit und verursacht dabei nur vernachlässigbaren Laufzeit-Overhead.

Abstract

This thesis presents methods for intercepting and manipulating function calls in Linux systems, with a focus on supporting automated testing of student programs in the Operating Systems course. The central contribution is the development of `intercept.so`, a shared object preloaded via the `LD_PRELOAD` mechanism, which enables function call interception at runtime without requiring modifications to the program or its source code. The library records detailed information about function calls, including arguments, return values, and call locations, allowing precise reconstruction and analysis of program execution. Furthermore, it introduces controlled manipulation of intercepted calls—modifying arguments, forcing failures, or mocking return values—through a simple communication protocol, enabling robust automated tests of error handling and resource management. Performance evaluations show minimal overhead in practical scenarios, making the approach suitable for educational use. Compared to existing tools such as `strace` or `ltrace`, this solution offers greater flexibility by combining interception with manipulation capabilities. Overall, the approach provides a flexible and transparent foundation for automated testing of low-level C programs, improving reliability while imposing negligible runtime overhead.

Contents

CHAPTER 1

Introduction

Intercepting (also known as Hooking, or Tracing) system or function calls allows one to trace what a given program does. This information is useful for security analysis or when testing or verifying a program. This chapter gives a general overview about what the motivation and goal for this work were (Section 1.1), and what the difference between system calls and function calls is (Section 1.2).

1.1 Motivation and Goal

When teaching students about Operating Systems, their interfaces, and standard libraries, C is still a widely used language, especially when using Linux. Therefore, it is obvious why many university courses still require students to write their assignments and exams in C. The problem when trying to verify whether students have correctly implemented their assignment, is that low-level OS constructs (like semaphores, pipes, sockets, memory management) make it hard to run automated tests, because the testing system needs to keep track, set up, and verify the usage of these resources.

The goal of this work was to find a way to easily intercept system or function calls, and to verify if students called the right functions, with the right arguments, at the right time. This restriction in scope allows focusing on simple binary programs without having to think about complex or I/O heavy programs. Furthermore, in this setting the source code of the student's programs is obviously available, because this is what they need to deliver. The availability of source code is a key concern when trying to intercept function or system calls, as will be clear in the next chapters.

1.2 Definitions

First, function calls, system calls, and their differences need to be defined. The following subsections concern these definitions.

1.2.1 Function Calls

Generally, a function in C (and also most other programming languages) is a piece of code that may be called, and therefore executed, from elsewhere. Functions have zero or more arguments and return a single value. When calling a function, the caller places the return address onto the stack. This address indicates where the function should continue executing when it is finished.

Functions are used to structure programs, reuse functionality, or expose functionality in libraries. Other languages than C differentiate between functions, methods, procedures, and so on. A function written in the source code is almost always compiled to a function in the resulting binary.

Intercepting calls to functions allows one to see the function name, arguments, return value, and return address.

1.2.2 System Calls

In contrast to functions, system calls are calls to the kernel itself. Many operations on a modern operating system require special privileges, which a simple user-space process does not have. By invoking a system call, the (user-space) process hands control over to the (privileged) kernel, and requests an operation to be performed. [16, Chapter 10]

How exactly these system calls work depends on the architecture and operating system. But generally, the process places the system call number and its arguments in defined registers and then executes a special system call opcode. Then the kernel executes the requested operation and places the return value inside another register, and lastly hands the execution back to the process. [16, Chapter 10]

Intercepting system calls allows one to see the system call number, arguments, and return value. One has to keep in mind, that many system-related functionalities are not, in fact, translated to system calls one-to-one. For example, `malloc` [10] has no dedicated system call, it is managed by the C standard library internally. Many system calls have corresponding wrapper functions in the C standard library (like `open`, `close`, `sem_wait`).

CHAPTER 2

Related Work

This chapter gives a rough overview on techniques and methods to intercept or hook system calls and function calls. See also Section 3.1. Some relevant methods will be discussed there in more detail.

2.1 System and Function Call Hooking in Literature

The following subsections explore some applications of system and function call hooking. There are possibly many other use-cases, but the following were deemed most important.

2.1.1 Classification of Hooking Techniques

Lopez et al. [25] classify subroutine hooking techniques as follows:

- Subroutine Type: Function / System call.
- Hook Insertion: Static (before execution) / Dynamic (during execution).
- Instrumentation Type: Active ("manipulation") / Passive ("interception").
- Hooking Location: On-device / Off-device (most used for mobile devices).
- Hooking Scope: Inner Functions / Exported Functions (e.g., libraries).
- OS Modification: Required / Not Required.
- Availability of Source Code: Open-source / Closed-source.
- Pricing Model: Free / Paid.

The technique developed in this work would be classified as follows: Function, Static, Active+Passive, On-device, Exported Functions, OS Modification Not Required, target program may be Closed-source, Free.

2.1.2 Linux Systems

Yasukata et al. [28] introduced zpoline, a system call hooking technique using binary rewriting. See Subsection 2.3.3 for more details. Hong et al. [22] developed DataHook, a system call hooking technique based on glibc. See Subsection 2.3.4 for more details.

2.1.3 Windows Systems

Hunt and Brubacher [23] developed Detours, a library for instrumenting arbitrary Win32 functions on x86 machines. Detours intercepts Win32 functions by re-writing target function images. Based on this method, Sze and Sekar developed Spif [27] (see next subsection).

2.1.4 Security Applications

Fraser et al. [19] introduced a general mechanism for securing unmodified commercial software by wrapping system calls at the library interface. They hook system calls by replacing the standard library entry points with wrapper functions, similar to `LD_PRELOAD`. The wrapper functions are able to perform security checks or other security measures.

Garfinkel et al. [20] developed Ostia, a sandboxing system, which uses system call hooking to secure applications. They implemented their own ELF binary loader to load their emulation library into memory before the sandboxed program starts. To communicate between this library and their *agent*, they use Unix domain sockets. The *agent* then responds, according to its policies. This is a similar approach to the one of this work (see Chapter 4).

Sze and Sekar [27] introduced Spif, an approach that defends against malware by tracking code and data origin on Windows systems. They use Detours [23] to intercept low-level Windows API calls.

Kern [24] discusses the use of `LD_PRELOAD` in cloud environments for HTTP deception. This is done to analyze malware or other adversaries in real environments without their knowledge and without any risk of danger. Examples are to override the `send` and `recv` functions of libc. With some modifications, the technique presented in this work may also be used in this context to intercept and manipulate `send` and `recv` calls.

2.1.5 Software Distribution

Guo and Engler [21] use system call hooking for creating portable software. They developed CDE, which logs all files a program accesses during execution, including shared libraries. All accessed files and the environment are bundled together and may now be executed on any other system having a compatible kernel without having to install any dependencies. This would also be possible with the use of logged function calls like in this work (e.g., see Section 3.8).

2.1.6 Rapid Prototyping

Spillane et al. [26] use `ptrace` to hook system calls of another process to simulate these calls using a user-space program. This is useful for rapid prototyping (e.g., file systems) by developing a user-space program first, and then, using the gained insight, porting it into kernel-space.

2.2 Function Call Hooking

All underlying techniques for function call interception on Linux systems are mentioned in Section 3.1. See `ltrace` (Subsection 3.1.3), wrapper functions (Subsection 3.1.5), and `LD_PRELOAD` (Subsection 3.1.6).

2.3 System Call Hooking

This section discusses further techniques regarding system call interception. This excludes techniques discussed in Section 3.1, like `ptrace` (Subsection 3.1.1), and `strace` (Subsection 3.1.2). Almost all following methods use binary rewriting to replace system calls with other instructions (except SUD, Subsection 2.3.2). This is one of the reasons why they are not mentioned in Section 3.1. Another reason is that this work focuses on function call interception rather than system call interception.

2.3.1 `int3` Signaling

`int3` is a one-byte instruction (`0xcc`) that invokes a software interrupt. On Linux, the kernel handles it and raises `SIGTRAP` to the user-space process that executed `int3`. The `int3` signaling technique exploits this behavior to hook system calls; it replaces `syscall`/`sysenter` with `int3` and employs the signal handler for `SIGTRAP` as the hook function. Since `int3` is one byte, it can replace an arbitrary instruction without breaking neighboring instructions. This technique is traditionally used in debuggers to implement breakpoints. However, signal handling incurs a large overhead because it involves context manipulation by the kernel. [28]

2.3.2 Syscall User Dispatch (SUD)

Syscall User Dispatch (SUD) [14] was added in Linux 5.11, and it offers a way to redirect system calls to arbitrary user-space code. For the SUD feature, the kernel implements a hook point at the entry point of system calls. A user-space process can activate SUD via the `prctl` interface. When SUD is activated, the hook point raises `SIGSYS` to the user-space process. This mechanism allows a user-space program to leverage the `SIGSYS` signal handler as the system call hook. However, similarly to the `int3` signaling technique, SUD imposes a significant performance penalty on the user-space program due to the overhead of the signal handling. [28]

2.3.3 zpoline

zpoline is a system call hook mechanism for x86-64 CPUs. Binary rewriting is used to replace (two-byte) `syscall`/`sysenter` instructions with a (two-byte) `callq *%rax` instruction. Because this instruction jumps to `rax`, where also the syscall number is stored, the trampoline code has to be initialized beginning at virtual address 0. zpoline is exhaustive and achieves very low performance reduction (28–761 times less overhead compared to other exhaustive system call hooking techniques). [28]

2.3.4 DataHook

DataHook is a system call hooking technique for 32-bit programs based on glibc running on x86 or x86-64 machines. It relies on glibc's way of performing system calls, namely a `call *%gs:0x10` instruction to call the `__kernel_vsyscall` function. The content of `gs:0x10` is backed up and modified to jump to a given hook function. DataHook is only exhaustive when used on glibc-based programs. It achieves a very low performance reduction (5–1429 times less overhead compared to existing hooking techniques). [22]

CHAPTER 3

Intercepting Function Calls

In this chapter, all steps on how to intercept function calls in this work are discussed. An example of what the resulting interception looks like may be found in Section 3.8. Furthermore, an overview on how to test given programs is presented in Section 3.9. How these function calls may be manipulated is discussed in Chapter 4.

3.1 Identified Methods for Intercepting Function and System Calls

First, one has to answer the question on *how exactly* to intercept function or system calls. At the beginning of this work, it was not yet determined if the interception of function calls, system calls, or both should be used to achieve the overarching goal (see Section 1.1). This first section tries to list all possible and relevant methods on how to intercept function or system calls but does not claim exhaustiveness. The order of the following subsections is roughly based on the thought process on finding the most appropriate method suitable for this work.

3.1.1 `ptrace` System Call

The first thing that pops up when researching on how to intercept system calls in Linux is the `ptrace` ("process trace") system call. This system call allows one process to observe and control the execution of another process (including memory and registers). The control is handed from the traced process to the tracing process each time any signal is delivered. [11]

To make use of this system call, a corresponding command already exists. See Subsection 3.1.2.

3.1.2 `strace` Command

The `strace` ("system call/signal trace") command may be used to run a specified command and to thereby intercept and record the system calls which are made. Each system call is recorded as a line and either written to the standard error output or a specified file. [13]

Listings 3.1 and 3.2 give a simple example of what this output looks like. It is clearly visible that only ("pure") system calls are recorded, and calls to library functions (like `malloc` or `free`) do not appear. Also note that arguments to the calls are displayed in a "pretty" way. For example, string arguments would be simple pointers, but `strace` displays them as C-like strings.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(const int argc, char *const argv[]) {
    char *str = malloc(10);
    strcpy(str, "Abc123");
    printf("Hello World!\nString: %s\n", str);
    free(str);
}
```

Listing 3.1: Contents of `main.c`.

```
execve("./main", ["./main"], 0x7ffd63b32bb0 /* 71 vars */) = 0
[-- 32 lines omitted --]
write(1, "Hello World!\n", 13)          = 13
write(1, "String: Abc123\n", 15)        = 15
exit_group(0)                           = ?
+++ exited with 0 +++
```

Listing 3.2: Output of `strace ./main`.

This approach works well for debugging and other use cases, but intercepting system calls alone does not satisfy the requirements of this work.

3.1.3 `ltrace` Command

The `ltrace` ("library call trace") command may be used to trace dynamic library calls instead of system calls. It works similarly to `strace` (see 3.1.2). [8]

Listings 3.1 and 3.3 illustrate what the output of `ltrace` looks like. In contrast to the output of `strace`, now only "real" calls to library functions are included in the output. Therefore, a lot less "noise" is generated (see omitted lines in Listing 3.2). Again,

the function arguments are displayed in a “pretty” way. This command uses so-called prototype functions [9] to format function arguments.

```
malloc(10)                                      = 0x55624164b2a0
printf("Hello World!\nString: %s\n", "Abc123") = 28
free(0x55624164b2a0)                            = <void>
+++ exited (status 0) +++
```

Listing 3.3: Output of `ltrace ./main`.

This method fits the requirements for this work a lot better than `strace` (see Subsection 3.1.2), but it is not very flexible and offers no means to modify the intercepted function calls.

3.1.4 Kernel Module

Another possibility to intercept system calls is to intercept them directly in the kernel via a kernel module. However, this work did not explore this approach further due to time constraints and other, better-fitting alternatives. See [18, Section 7.2] for more details on how to intercept system calls using kernel modules.

3.1.5 Wrapper Functions in gcc

A different approach to intercepting function calls is to tell the compiler directly which functions should be intercepted. The compiler, and the linker respectively, then directly link calls to the specified functions to wrapper functions. (See Subsection 3.1.6 for more details.)

The default linker `ld` includes such a feature. See the ld(1) Linux manual page [6, Section OPTIONS]:

> **`--wrap=`*`symbol`*** Use a wrapper function for *`symbol`*. Any undefined reference to *`symbol`* will be resolved to `__wrap_`*`symbol`*. Any undefined reference to `__real_`*`symbol`* will be resolved to *`symbol`*.
>
> This can be used to provide a wrapper for a system function. The wrapper function should be called `__wrap_`*`symbol`*. If it wishes to call the system function, it should call `__real_`*`symbol`*. [...]

The gcc compiler also supports this by allowing passing options to the linker. See the gcc(1) Linux manual page [3, Section OPTIONS]:

> **`-Wl,`*`option`*** Pass *`option`* as an option to the linker. If *`option`* contains commas, it is split into multiple options at the commas. You

> can use this syntax to pass an argument to the option. For example, `-Wl,-Map,output.map` passes `-Map output.map` to the linker. When using the GNU linker, you can also get the same effect with `-Wl,-Map=output.map`. [...]

This means, by specifying `-Wl,--wrap=`*`symbol`* when compiling using gcc, all calls from the currently compiled program to *`symbol`* are redirected to `__wrap_`*`symbol`*. To call the real function inside the wrapper, `__real_`*`symbol`* may be used. Listings 3.4 and 3.5 illustrate this by overriding the `malloc` function of the C standard library.

```c
#include <stddef.h>

extern void *__real_malloc(size_t size);

void *__wrap_malloc(size_t size) {
    // before call to malloc
    void *ret = __real_malloc(size);
    // after call to malloc
    return ret;
}
```

Listing 3.4: Contents of `wrap.c`.

```
gcc -o main_wrapped main.c wrap.c -Wl,--wrap=malloc
./main_wrapped
```

Listing 3.5: Compile `main.c` and `wrap.c` and run the resulting program.

This approach allows wrapping any function in a relatively clean way. But it is not possible to override functions in any given binary program. It is required to re-compile (or to re-link) a given program to use this feature of ld. Therefore, the source code (or the corresponding `*.out` files) needs to be available. Note, only calls from the targeted source code will be redirected, calls from other libraries won't.

Theoretically, it should be possible to re-link a given binary without having access to its source code. But due to other more straight-forward methods (see Subsection 3.1.6), this has not been investigated further.

3.1.6 Preloading using `LD_PRELOAD`

To execute binary files on Linux systems, a dynamic linker is needed at runtime. (Unless the binaries were statically linked at compile-time.) Usually, `ld.so` and `ld-linux.so` are used as dynamic linkers. They find and load the shared objects (shared libraries) needed by a program, prepare the program and finally run it. [7]

As the overwhelming majority of programs are dynamically linked, most function calls to other libraries (like to the C standard library) reference a shared object, which has to be loaded by the linker at runtime. Therefore, it would be possible to "hijack" (or intercept) these function calls when the linker would allow loading other functions instead of the proper ones.

Luckily, `ld.so` allows this so-called "preloading". See the ld.so(8) Linux manual page [7, Section ENVIRONMENT]:

> **LD_PRELOAD** A list of additional, user-specified, ELF shared objects to be loaded before all others. This feature can be used to selectively override functions in other shared objects. [...]

This means, by setting the environment variable `LD_PRELOAD`, it is possible to override specific functions. Listings 3.6 and 3.7 illustrate this by overriding the `malloc` function of the C standard library.

```c
#include <stdlib.h>
#include <dlfcn.h>
#include <errno.h>

void *malloc(size_t size) {
    // before call to malloc
    void *(*_malloc)(size_t);
    if ((_malloc = dlsym(RTLD_NEXT, "malloc")) == NULL) {
        errno = ENOSYS;
        return NULL;
    }
    void *ret = _malloc(size);
    // after call to malloc
    return ret;
}
```

Listing 3.6: Contents of `preload.c`.

```
# ./main is already compiled and ready
gcc -shared -fPIC -o preload.so preload.c
LD_PRELOAD="$(pwd)/preload.so" ./main
```

Listing 3.7: Compile `preload.c` and run a program with `LD_PRELOAD`.

The function `dlsym` is used to retrieve the original address of the `malloc` function. `RTLD_NEXT` indicates to find the next occurrence of `malloc` in the search order after the current object. [2]

By using this method, it is possible to override, and therefore wrap, any function as long as the targeted binary was not statically linked. However, one must be aware that, not only function calls inside the targeted binary, but also calls inside other libraries (e.g., `malloc`), are redirected to the overriding function.

3.1.7 Conclusion

During the research on different approaches to intercepting system and function calls, it has been found, that the most reliable way to achieve the goals of this work (see Section 1.1) is to intercept function calls instead of system calls. This is because—as long as the programs to test are dynamically linked—intercepting function calls allows one to intercept many more calls and in a more flexible way. Therefore, from now on this work only considers function calls and no system calls directly.

In this work, preloading (see Subsection 3.1.6) was chosen to be used because it is simple to use ("clean" source code, easy to compile and run programs with it) and offers the means to arbitrarily execute code when the intercepted function call is redirected. The following sections concern the next steps in what else is needed to create a powerful "interceptor".

3.2 Fundamental Project Structure

After deciding to use the preloading method to intercept function calls, a more detailed plan is needed to continue developing. It was decided to have one single `intercept.so` file as a resulting artifact which then may be loaded via the `LD_PRELOAD` environment variable. The easiest and most straightforward way to structure the source code was to put all code in one single C file. Listing 3.8 gives an overview of the underlying code structure. For each function that should be intercepted, this function simply has to be declared and defined the same way `malloc` was.

3.3 Retrieving Function Argument Values

Now that the first steps have been done, one needs to think about what exactly to record when intercepting. A simple notification that a given function was called would not be sufficient. Within the following subsections, effort is put into getting as much information as possible from each function call.

As already mentioned, `ltrace` uses prototype functions to format its function arguments. This allows `ltrace` to "dynamically" display function arguments for any new or unknown functions without the need for recompilation. [9]

However, due to implementation complexity reasons and the need for "complex" return types for string/buffer and structure values (see Section 3.4) a statically compiled approach has been used for this work. This means that each function formats its arguments and return values itself without any configuration option.

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <dlfcn.h>

static void *(*__real_malloc)(size_t);

static int mode = 0;

static void fin(void) {
    if (mode > 0) fprintf(stderr, "intercept: stopped\n");
    mode = 0;
}

static void init(void) {
    if (mode) return;
    mode = -1;
    if (((__real_malloc) = dlsym(RTLD_NEXT, "malloc")) == NULL) {
        fprintf(stderr, "intercept: unable to load symbol '%s': %s",
            "malloc", strerror(errno));
        return;
    }
    atexit(fin);
    fprintf(stderr, "intercept: intercepting function calls\n");
    mode = 1;
}

void *malloc(size_t size) {
    init();
    // before call to malloc
    void *ret = __real_malloc(size);
    // after call to malloc
    return ret;
}
```

Listing 3.8: Contents of `intercept-preload.c`.

The reason for retrieving as much information as possible from each function call is that at a later point in time it is possible to completely reconstruct the exact function calls and their sequence. This allows an analysis on these records to be performed independently of the corresponding execution of the program. It should always be possible to fully parse the recorded calls without any specific knowledge of specific functions, their argument types, or return value type.

3.3.1 Numbers

The simplest types of arguments are plain numbers, like integers (`int`, `long`, ...) or floating point numbers (`float`, `double`). (In fact, *all* arguments are represented as numbers or integers. See the following subsections for examples.) Plain numbers may be formatted simply as what they are, in base 10 notation, or with a prefix like `0x` for hexadecimal or `0` for octal representation.

Example: `malloc(123)` (or `malloc(0x7B)`).

3.3.2 Unspecific Pointers

Pointers without additional type information (such as `void *`) can essentially be treated as integers.

Example: `free(0x55624164b2a0)`.

3.3.3 Strings and Buffers

Strings in C are simple pointers to a place in memory which is null-terminated. This means that the strings end with the first occurrence of the null-byte (`0x00`). To distinguish unspecific pointers from pointers to strings, it was chosen to use a colon (`:`) after the pointer numerical value. The colon is followed by the contents of the string with beginning and ending quoted (`"`). Special values inside the string are escaped with a backslash.

Example: `sem_unlink(0x1234:"/test-semaphore")`.

Another type of string-like data in C is a buffer with a known length. When buffers are used, usually another argument is passed to the function which indicates the length of the buffer. This fact may be used to print out the contents of the buffer in the same way as normal C strings.

Example: `write(3, 0x1234:"Test\x00ABC", 8)`.

3.3.4 Flags

Some functions have one of their arguments dedicated to flags which may be combined by bitwise XOR. These arguments are also of type integer. To distinguish flag arguments from others, a pipe symbol (`|`) is used after the colon and between the flags.

Example: `open(0x1234:"test.txt", 0102:|O_CREAT|O_RDWR|, 0644)`.

3.3.5 Constants

For some functions, constants are used. These constants are typically implemented as C macros (`#define`) in the source code. This makes the source code more readable (and portable). Constants are represented as an integer, again followed by a colon, this time without any special characters to distinguish them from other types.

Example: `socket(2:AF_INET, 1:SOCK_STREAM, 6)`.

3.3.6 Pointers to Arrays

Sometimes arrays are used as arguments. Arrays in C work similar to strings, they are either null-terminated (by an element being of value 0), or their length is explicitly given. So to represent them, two brackets are used (`[]`) and a comma (`,`) to separate the respective elements. Each element may be represented as an "argument" on its own (as illustrated by the example).

Example:
`getopt(2, 0x7f0b8:[0x7feb3:"./main", 0x7fee6:"arg"], 0x123:"v")`.

3.3.7 Pointers to Structures

In rare cases, structures (`struct`) are used as argument types. Two curly brackets (`{}`) are used to indicate structures. Then the field names are displayed plainly, followed by a colon and then the value of that field. Commas are used to separate the fields respectively.

Example: `connect(2, 0x123:{sa_family: 2:AF_INET, sin_addr: "1.1.1.1", sin_port: 80}, 16)`.

3.4 Retrieving Function Return Values

It might seem that retrieving return values of functions is as straightforward as retrieving their arguments, but this is not entirely the case. Most libc functions return -1 on error and set `errno` to indicate the exact type of error. Other functions (like `read`, `pipe`, or `sem_getvalue`) even store their output in a pointer which was given to them as an argument. The following examples illustrate how this challenge was solved.

Example (`malloc`):
`return 0x1234; errno 0`,
`return -1; errno ENOMEM`.

Some libc functions return their results via a pointer, which was previously given to them as an argument. The `pipe` function is called with an `int` array of size two as an argument and stores its two pipe ends into this array. The `read` function is called with a pointer to a buffer and a corresponding size and stores its read data into this buffer.

Example (`pipe`):
`return 0; errno 0; fildes=[3,4],`
`return -1; errno ENFILE.`

Example (`read`):
`return 12; errno 0; buf=0x7fff70:"Hello World!",`
`return -1; errno EINTR.`

3.5 Determining Function Call Location

Besides argument values and return values, it would be interesting to know from where inside the intercepted program the function call came. At first this seems quite impossible. But a function always knows at least the return address, the address to set the instruction pointer to when the function finishes. With this information, it may be estimated where the call to the current function came from.

3.5.1 Return Address and Relative Position

As already mentioned, the return address of a function is vital for estimating where the call came from. Luckily, GCC provides the means to get the return address of the current function. See in the manual of GCC [15, Section 7.6]:

> **`void *__builtin_return_address(unsigned int level)`**
>
> This function returns the return address of the current function, or of one of its callers. The *level* argument is number of frames to scan up the call stack. A value of `0` yields the return address of the current function, a value of `1` yields the return address of the caller of the current function, and so forth. [...]

The return address on its own is of limited use. Because, among other things, of Address Space Layout Randomization (ASLR) in almost all modern programs. ASLR is a security feature that randomly places shared objects (libraries) in the virtual memory of a program on each execution. In contrast to always positioning the same object at the same address each time, this makes it harder to exploit internal memory structures.

Fortunately, the dynamic linking library includes a function to translate a given virtual memory address to symbolic information without having to worry about ASLR and other obstacles. See the dladdr(3) Linux manual page [1]:

> **`int dladdr(const void *addr, Dl_info *info)`**
>
> The function `dladdr()` determines whether the address specified in *addr* is located in one of the shared objects loaded by the calling application. If it is, then `dladdr()` returns information about the shared object and

symbol that overlaps *addr*. This information is returned in a `Dl_info` structure:

```
typedef struct {
  const char *dli_fname; /* Pathname of shared object
                            that contains address */
  void       *dli_fbase; /* Base address at which
                            shared object is loaded */
  const char *dli_sname; /* Name of symbol whose
                            definition overlaps addr */
  void       *dli_saddr; /* Exact address of symbol
                            named in dli_sname */
} Dl_info;
```

[...]

Using information from `Dl_info`, it is possible to exactly determine the (shared) object where the call came from (`dli_fname`). Furthermore, it is possible to calculate the relative position inside this (shared) object using `dli_fbase` and the return address itself. Keep in mind that the return address may only be used as an estimation for the origin of the call. Especially heavily optimized programs might use the same return address for functions in different code paths. Optionally, a name of a "symbol" (function) may be retrieved, indicating where the function call came from.

3.5.2 Source File and Line Number

DWARF is a file format used for storing debugging information (like source file, line number) inside compiled binaries. This allows various debuggers and other analysis programs to better give feedback to the user. [17]

This also helps to find the origin of a given function call. When a program is compiled with GCC using the flags `-g` or `-gdwarf` GCC includes the DWARF debug section in the resulting binary. Using the readelf tool, it is possible to make use of this debug section. See the readelf(1) Linux manual page [12, Section OPTIONS]:

> **`--debug-dump`** Displays the contents of the DWARF debug sections in the file, if any are present. [...] The letters and words refer to the following information:
>
> [...]
>
> **`=rawline`** Displays the contents of the `.debug_line` section in a raw format.
>
> **`=decodedline`** Displays the interpreted contents of the `.debug_line` section.
>
> [...]

Using the resulting output, which sets relative address and source file and line number in relation, it is possible to retrieve both values from any given relative address inside the binary. If this information is present, it is printed within the meta-information of the function call (see Section 3.8).

3.6 `intercept.so` Library

The time has come for putting it all together. As mentioned in Section 3.2, almost the whole project exists in one source file, `intercept.c`. This file is compiled to `intercept.so`, which may be preloaded using `LD_PRELOAD` and controlled with other environment variables. These other environment variables are described in the following:

`INTERCEPT` This variable has to be set to enable function call interception. The value decides where to output/print/write/send the recorded function calls. Values may be `stdout`, `stderr`, `file:<path>`, `unix:<path>`.

`INTERCEPT_VERBOSE` This variable indicates whether string and structure types should be printed fully or empty. Possible values are `0` and `1` (default).

`INTERCEPT_FUNCTIONS` This variable is used to specify which function calls should be intercepted. It is a list separated by commas, colons, or semicolons. Wildcards (`*`) at the end of function names are possible. A prefix of `-` indicates that the following function should not be intercepted. Example: `*,-sem_*` intercepts all functions except those which start with `sem_`. By default, all (implemented) functions are intercepted.

`INTERCEPT_LIBRARIES` This variable is used to specify which libraries' function calls should be intercepted. It is a list separated by commas, colons, or semicolons. Wildcards (`*`) at the end of library paths are possible. A prefix of `-` indicates that the following library path should not be intercepted. Example: `*,-/lib*,-/usr/lib*` intercepts only function calls originating from binaries outside `/lib*` or `/usr/lib*` which in most cases is the executed program itself. By default, function calls from everywhere are intercepted.

The shared object currently supports intercepting the following functions: `malloc`, `calloc`, `realloc`, `reallocarray`, `free`, `getopt`, `exit`, `read`, `pread`, `write`, `pwrite`, `close`, `sigaction`, `sem_init`, `sem_open`, `sem_post`, `sem_wait`, `sem_trywait`, `sem_timedwait`, `sem_getvalue`, `sem_close`, `sem_unlink`, `sem_destroy`, `shm_open`, `shm_unlink`, `mmap`, `munmap`, `ftruncate`, `fork`, `wait`, `waitpid`, `execl`, `execlp`, `execle`, `execv`, `execvp`, `execvpe`, `execve`, `fexecve`, `pipe`, `dup`, `dup2`, `dup3`, `socket`, `bind`, `listen`, `accept`, `connect`, `getaddrinfo`, `freeaddrinfo`, `send`, `sendto`, `sendmsg`, `recv`, `recvfrom`, `recvmsg`, `getline`, `getdelim`.

3.7 intercept Command

To make the usage of the aforementioned shared object easier, a simple python script has been put together. This script may be used as a command line tool. See Listing 3.9.

The synopsis of the command is as follows:

```
intercept [-h] [-F FUNCTIONS] [-s] [-o | -L LIBRARIES] \
  [-l LOG | -i INTERCEPT] [--] COMMAND [ARGS...]
```

-F, --functions A list of functions to intercept. See Section 3.6 for more details. Default value is `*`.

-s, --sparse Indicates that strings and structures should be printed empty to save bandwidth.

-o, --only-own A shorthand for `-L *,-/lib*,-/usr/lib*`. This has the effect that only function calls from the executed binary itself are recorded.

-L, --libraries A list of library paths to intercept function calls from. See Section 3.6 for more details. Default value is `*` (except when `-o` is present).

-l, --log Used to specify in which file the recorded function calls should be logged. Shorthand for `-i file:<arg>`.

-i, --intercept Decides where to output/print/write/send the recorded function calls. Values may be `stdout`, `stderr`, `file:<path>`, `unix:<path>`. See Section 3.6 for more details.

3.8 Example

To make it easier for the reader, Listing 3.10 provides some recorded function calls. Most lines had to be broken up into multiple lines for better readability. The recorded calls stem from a program written by myself as a solution for an assignment in the Operating Systems course at university. It is a simple HTTP client. The program was invoked using `./intercept -o -- ./client http://www.complang.tuwien.ac.at/`.

The first number on each line indicates unix time with nanosecond precision. The second and third numbers correspond to the process ID and thread ID respectively. Each line contains either a recorded call to a function or a recorded return of a function. After the arguments of each function call a colon (`:`) indicates the beginning of meta-information. This information always includes the return address to where the function jumps when completed. If available, the interpretation of the return address is also provided. This includes the offset relative to the calling binary and a source file and line number combination if the binary was compiled using `gcc -g` or `gcc -gdwarf`.

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# File: intercept
# Author: Lorenz Stechauner <e12119052@student.tuwien.ac.at>
#         Lorenz Stechauner <lorenz.stechauner@necronda.net>

import argparse
import subprocess
import os
import sys


def main() -> None:
    parser = argparse.ArgumentParser()
    parser.add_argument('-F', '--functions')
    parser.add_argument('-s', '--sparse', action='store_true')
    libs = parser.add_mutually_exclusive_group()
    libs.add_argument('-o', '--only-own', action='store_true')
    libs.add_argument('-L', '--libraries')
    mode = parser.add_mutually_exclusive_group()
    mode.add_argument('-l', '--log')
    mode.add_argument('-i', '--intercept')
    args, extra = parser.parse_known_args()
    if len(extra) > 0 and extra[0] == '--':
        extra.pop(0)
    if len(extra) == 0:
        parser.error("command expected after arguments or '--'")

    if args.intercept:
        intercept = args.intercept
    elif args.log:
        intercept = 'file:' + args.log
    else:
        intercept = 'stderr'
    subprocess.run(extra, stdin=sys.stdin, env={
        'LD_PRELOAD': os.getcwd() + '/intercept.so',
        'INTERCEPT': intercept,
        'INTERCEPT_VERBOSE': '0' if args.sparse else '1',
        'INTERCEPT_FUNCTIONS': args.functions or '*',
        'INTERCEPT_LIBRARIES': '*,-/lib*,-/usr/lib*' if
            args.only_own else args.libraries or '*',
    })


if __name__ == '__main__':
    main()
```

Listing 3.9: Contents of `intercept`.

```
1747639484.855979238 17036 17036 \
  getopt(2, 0x7ffdff7b20b8:[0x7ffdff7b3eb3:"/home/lorenz/client", 0x7ffdff7b3ee6:\
  "http://www.complang.tuwien.ac.at/"], 0x61520b0190f2:"hp:o:d:"): 0x61520b017ac5 \
  (/home/lorenz/client+0x1ac5, client.c:186)
1747639484.856009998 17036 17036 \
  return -1
1747639484.859018930 17036 17036 \
  getaddrinfo(0x7ffdff7b0e70:"www.complang.tuwien.ac.at", 0x61520b019052:"http", 0x7ffdff7b0c30:\
  [{ai_flags: 0x0:|, ai_family: 0:AF_UNSPEC, ai_socktype: 1:SOCK_STREAM, ai_protocol: 0, \
  ai_addrlen: 0, ai_addr: (nil):{}, ai_canonname: (nil):"", ai_next: (nil)}], 0x7ffdff7b0c10): \
  0x61520b01747b (/home/lorenz/client+0x147b, client.c:74)
1747639484.870971294 17036 17036 \
  return 0:SUCCESS; errno 0; res=0x615238e79e00:[{ai_flags: 0x0:|, ai_family: 2:AF_INET, \
  ai_socktype: 1:SOCK_STREAM, ai_protocol: 6, ai_addrlen: 16, ai_addr: 0x615238e79e30:{sa_family: \
  2:AF_INET, sin_addr: "128.130.173.64", sin_port: 80}, ai_canonname: (nil):"", ai_next: (nil)}]
1747639484.870983698 17036 17036 \
  socket(2:AF_INET, 1:SOCK_STREAM, 6): 0x61520b0174f2 (/home/lorenz/client+0x14f2, client.c:81)
1747639484.870991734 17036 17036 \
  return 7; errno 0
1747639484.870998006 17036 17036 \
  connect(7, 0x615238e79e30:{sa_family: 2:AF_INET, sin_addr: "128.130.173.64", sin_port: 80}, 16): \
  0x61520b0175f3 (/home/lorenz/client+0x15f3, client.c:104)
1747639484.877322756 17036 17036 \
  return 0; errno 0
1747639484.877333157 17036 17036 \
  freeaddrinfo(0x615238e79e00): 0x61520b017638 (/home/lorenz/client+0x1638, client.c:114)
1747639484.877358736 17036 17036 \
  return
1747639484.877364678 17036 17036 \
  send(7, 0x7ffdff7b0f70:"GET / HTTP/1.1\r\nHost: www.complang.tuwien.ac.at\r\nUser-Agent: \
  osue-12119052/1.0\r\nConnection: close\r\n\r\n", 101, 0x0:|): 0x61520b017f5c \
  (/home/lorenz/client+0x1f5c, client.c:277)
1747639484.877385048 17036 17036 \
  return 101; errno 0
1747639484.877390719 17036 17036 \
  recv(7, 0x7ffdff7b0f70, 4095, 0x2:|MSG_PEEK|): 0x61520b017fa1 (/home/lorenz/client+0x1fa1, \
  client.c:284)
1747639484.885364636 17036 17036 \
  return 2674; errno 0; buf=0x7ffdff7b0f70:"HTTP/1.1 200 OK\r\n\
  Date: Mon, 19 May 2025 07:24:44 GMT\r\n\
  Server: Apache/2.4.62 (Debian) OpenSSL/3.0.15\r\n\
  Last-Modified: Thu, 25 Aug 2022 14:41:10 GMT\r\n\
  ETag: \"944-5e711c9dd0ce5\"\r\nAccept-Ranges: bytes\r\nContent-Length: 2372\r\n\
  Vary: Accept-Encoding\r\nConnection: close\r\nContent-Type: text/html; charset=UTF-8\r\n\r\n\
  <!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01//EN\"\n                \
  \"http://www.w3.org/TR/html4/strict.dtd\">\n<HTML lang=\"de\">\n\
[-- omitted --]
  </HTML>\n"
1747639484.889134948 17036 17036 \
  recv(7, 0x7ffdff7b0f70, 302, 0x0:|): 0x61520b018062 (/home/lorenz/client+0x2062, client.c:300)
1747639484.889148325 17036 17036 \
  return 302; errno 0; buf=0x7ffdff7b0f70:"HTTP/1.1 200 OK\r\n\
  Date: Mon, 19 May 2025 07:24:44 GMT\r\n\
  Server: Apache/2.4.62 (Debian) OpenSSL/3.0.15\r\n\
  Last-Modified: Thu, 25 Aug 2022 14:41:10 GMT\r\n\
  ETag: \"944-5e711c9dd0ce5\"\r\nAccept-Ranges: bytes\r\nContent-Length: 2372\r\n\
  Vary: Accept-Encoding\r\nConnection: close\r\nContent-Type: text/html; charset=UTF-8\r\n\r\n"
1747639484.889156551 17036 17036 \
  recv(7, 0x7ffdff7b0f70, 4096, 0x0:|): 0x61520b018442 (/home/lorenz/client+0x2442, client.c:360)
1747639484.889160779 17036 17036 \
  return 2372; errno 0; buf=0x7ffdff7b0f70:"\
  <!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01//EN\"\n\                \
  \"http://www.w3.org/TR/html4/strict.dtd\">\n<HTML lang=\"de\">\n\
[-- omitted --]
  </HTML>\n"
1747639484.889196809 17036 17036 \
  recv(7, 0x7ffdff7b0f70, 4096, 0x0:|): 0x61520b018442 (/home/lorenz/client+0x2442, client.c:360)
1747639484.889200556 17036 17036 \
  return 0; errno 0; buf=0x7ffdff7b0f70:""
1747639484.889203532 17036 17036 \
  close(7): 0x61520b018489 (/home/lorenz/client+0x2489, client.c:375)
1747639484.889214523 17036 17036 \
  return 0; errno 0
```

Listing 3.10: Recorded function calls from `./client`.

3.9 Automated Testing on Intercepted Function Calls

The recorded function calls of a program run may now be used to perform checks and tests on them. It is trivially possible to check which functions were called and in what order. Furthermore, it is possible to check various pre- and post-conditions for each function call. This is beneficial because many library functions in C rely on these pre- and post-conditions, which are not enforced by the compiler or in any other way.

For example, the `malloc` function has the post-condition that the returned value later needs to be passed to `free` to avoid memory leaks. The `free` function, on the other hand, has the pre-condition that the passed value was previously acquired using `malloc` and may not be yet freed. Any violation of such pre- and post-conditions may be reported as non-compliant behavior. [10]

This means that intercepted function calls allow a tester to check whether programmers use library functions in compliance with their specifications. Other checks may also include guards to calls to "forbidden" functions, or that specific functions must be called exactly three times. Another important post-condition of most library functions is the return value, which in most cases indicates success or failure of an operation. However, intercepting of calls alone may not be able to verify if a program really checks the return value of a function and acts accordingly. Chapter 4 shows how this problem may be solved.

3.9.1 Validating Memory Management

The most basic memory management functions in the C standard library are the following.

`malloc`, `calloc` Allocate memory. [10]

`realloc`, `reallocarray` Change the size of a previously allocated memory block and possibly move the block to another position in virtual memory. [10]

`free` Free previously allocated memory. [10]

`getaddrinfo` Allocate and initialize a linked list of `addrinfo` structures. [4]

`freeaddrinfo` Frees memory previously allocated by `getaddrinfo` for the dynamically allocated linked list. [4]

`getline`, `getdelim` Used to split strings. Allocate memory on their own, which must be freed afterward. [5]

By only intercepting these functions, it is possible to check if all allocated memory blocks in a simple program were properly allocated and freed.

3.9.2 Validating Resource Management

In addition to memory management, the proper use of other resources—most notably file descriptors—can also be checked. Many functions in the C standard library rely on file descriptors. It may be checked if file descriptors were properly acquired, if only previously acquired file descriptors are used, and if these file descriptors are closed after their use. Relevant for this work are also semaphores because they do not rely on file descriptor in their API. Due to time restrictions, no detailed list for validating resource management has been put together.

CHAPTER 4

Manipulating Function Calls

This chapter discusses how to manipulate function calls and how this may be used to test programs. How function calls may be intercepted at all has been discussed in Chapter 3. This chapter builds on the basis of the previous one and expands its functions. In this context, "manipulation" means changing the arguments of a function before calling it with the modified arguments, or skipping the execution of the real function completely and simply returning a given value ("mocking"). These techniques allow in-depth testing of programs.

In contrast to simply recording and logging function calls, which may be controlled via environment variables, manipulation of such function calls requires some other process to indicate how to handle each call. This work uses simple Unix domain sockets to communicate between the process of the program to be tested, and a "server", which decides what action to perform for each function call. Currently, only communication over Unix domain sockets is implemented, but communication over TCP sockets is also easily possible. This approach is similar to the one used in [20] to communicate with the *agent*.

Figure 4.1 illustrates the control flow for manipulating function calls.

4.1 Defining a Protocol

When using a socket to communicate with another process, a protocol definition is needed. This work defines a text-based protocol in which line breaks denote the end of a message. The following subsections describe the defined message types.

4.1.1 *Init* Message (Client → Server)

This message is the first message sent in a newly established connection. The client (`intercept.so`) uses it to identify the running program to the server (PID, path, ...).

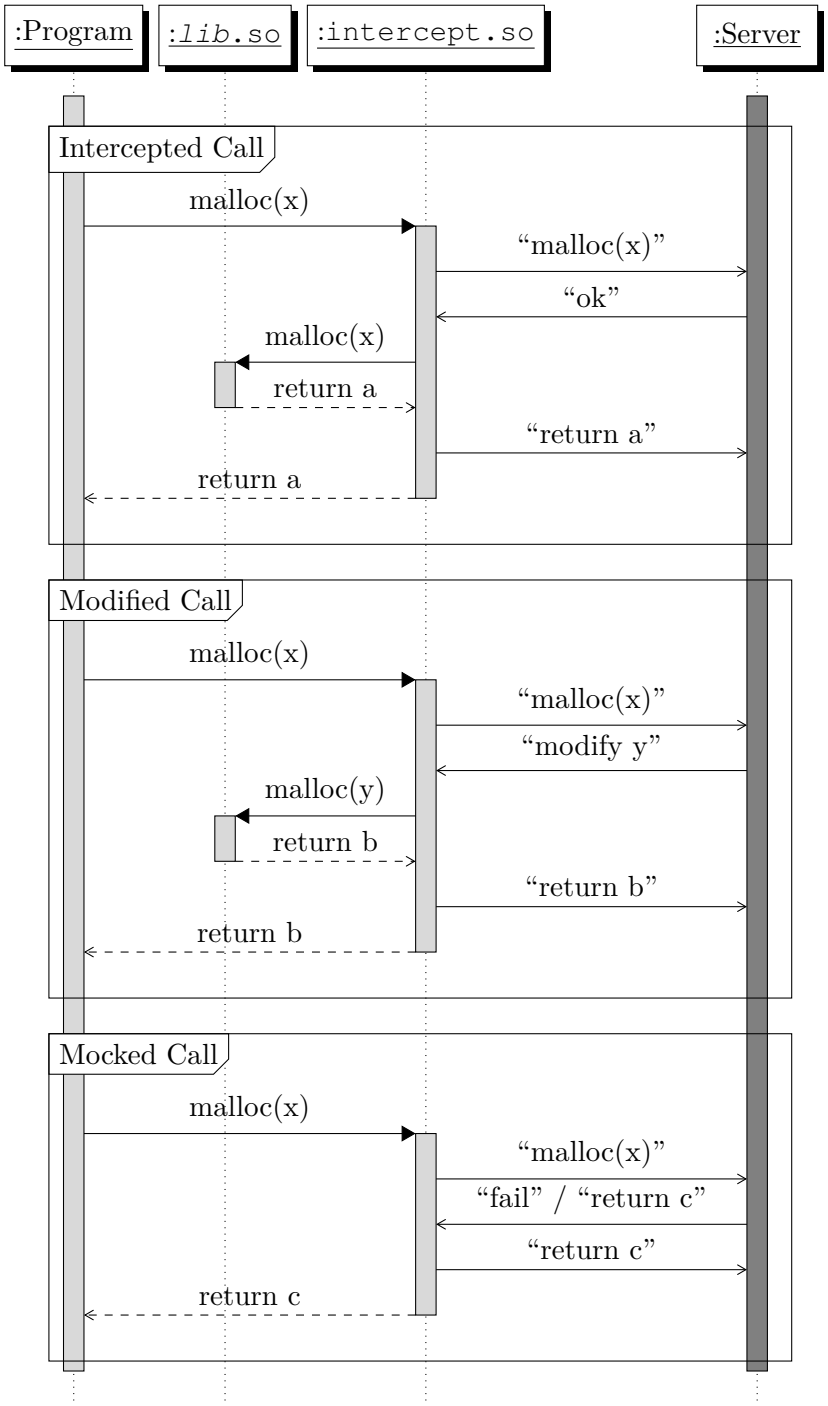

Figure 4.1: Simplified Control Flow for Function Call Manipulation.

4.1.2 *Call* Message (Client → Server)

For each function call, the client sends this message to the server and waits for a reply (*Action* message). The contents of this message type correspond to the first line of an intercepted function call (see Section 3.9).

4.1.3 *Action* Message (Server → Client)

After receiving a *Call* message from the client, the server decides what the client should do with this call. The server responds in one of four possible ways:

`"ok"` indicates that the function should be called normally.

`"modify ARG..."` indicates that the arguments of the function should be changed according to the message before the call to the function.

`"fail ERROR"` indicates that the function should not be called and instead should fail with the given error code.

`"return VALUE"` indicates that the function should simply return the provided value without calling the real function.

4.1.4 *Return* Message (Client → Server)

This message informs the server about the resulting return value. The server does not acknowledge this message. The contents of this message type correspond to the second line of an intercepted function call (see Section 3.9).

4.2 Automated Testing using Function Call Manipulation

As seen in Figure 4.1 function call manipulation allows for mocking individual calls. Mocking may be used to see how the program behaves when individual calls to a function fail, or return an unusual, but valid, value. The simplest way to automatically test programs is to run them multiple times, allowing a single function call to fail in each run. The resulting sequence of function calls now may be put together to a call sequence graph (or tree). By analyzing this call graph, it is possible to decide if a program correctly terminated, when faced with a failed function call. This may be the case when the following function calls differ from those which were recorded on a default run (without any mocked function calls).

4.2.1 Testing Return Value Checks

Figure 4.2 shows the simplified and collapsed call sequence graph of the prior example in Section 3.8. Each edge between two nodes without any label indicates the next function call on a normal run of the program. Edges labeled with “fail” indicate the next function

call after a mocked failed call. In reality, there are multiple failing paths, one for each possible error return value. However, in this example they all yield the same resulting path, and have therefore been collapsed.

To test if a programmer always checked the return value of a function and acted accordingly, this resulting call sequence graph now may be analyzed. At first glance, this test appears trivial. The simplest approach is to verify that after a failing function call only "cleanup" function calls (`free`, `close`, `exit`, ...) follow. For simple programs, this assumption may hold, but there are many exceptions. For example, what if the program recognizes the failed call correctly as failed but recovers and continues to operate normally? Or what if the "cleanup" path is very complex and includes function calls not priorly marked as valid cleanup functions? However, for simple programs (like those mentioned in Section 1.1), the simplest approach from above suffices.

4.2.2 Testing Correct Handling of Interrupts

Many functions (like `read`, `write`, or `sem_wait`) are interruptable by signals. When this happens, they return a value indicating an error and set `errno` to `EINTR`. Usually, the program is expected to repeat the call until it gets a real return value or error other than `EINTR`. Therefore, testing correct handling of interrupts is a different type of test in contrast to general tests on return value checks as seen in Subsection 4.2.1.

It is relatively simple to test if a program correctly handles interrupts. On any function call that may yield `EINTR` mock the call and return exactly that error. Afterward, check if the same function is called again. To increase confidence in the result, one may repeat this process multiple times. As in the test in Subsection 4.2.1, the handling of the interrupt may involve calls to other functions, so this method is not always the right choice. But for simple programs, it totally suffices.

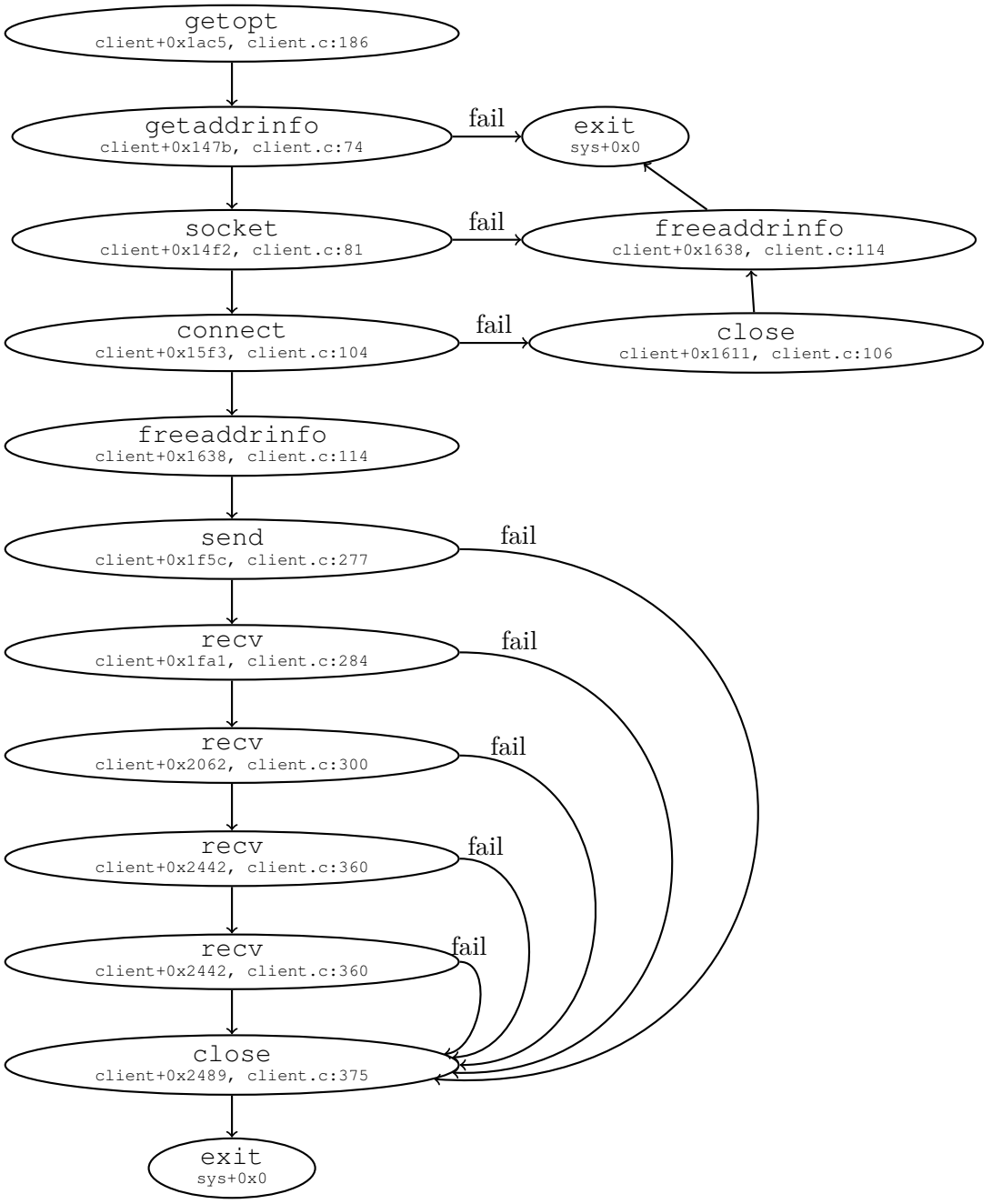

Figure 4.2: Simplified Call Sequence Graph of `./client`.

CHAPTER 5

Evaluation

This chapter evaluates the developed approach in terms of both its practical usefulness and its performance. Section 5.1 examines how well the library and tools support automated testing in the context of the Operating Systems course. Section 5.2 analyzes the performance overhead introduced by intercepting function calls.

5.1 Usefulness for the Operating Systems Course

Up until recently the Operating Systems course (mentioned in Section 1.1) was split into three exercise blocks: Files, Shared Memory, Semaphores; Related Processes and Inter-Process Communication via Unnamed Pipes; and Sockets. Table 5.1 lists all functions presented in the course and their implementation status in `intercept.so`. As one may see, simple file stream functions are not currently implemented in `intercept.so`. This is because of time restrictions on this work, and the fact that simple file operations may be tested easily in the conventional way of checking the resulting output. Note that the future implementation of single functions is not very complex. All other functions have at least interception and mocking (returning, failing) implemented. For some functions the modification of function arguments has been implemented.

5.2 Performance

Although high performance was not a primary goal of this work, the performance degradation caused by interception and manipulation should not be excessive. The following two subsections test and discuss the performance degradation of `intercept.so` compared to running a program without any interception or hooking.

Ex. Block	Functions	Implementation
-	malloc, calloc, realloc, reallocarray, free	icmrf
-	getopt	ic-rf
-	sigaction	ic--f
-	mmap	ic-rf
-	munmap	icmrf
-	close	icmrf
1	open, fopen, fdopen	-----
1	read, pread, write, pwrite	ic-rf
1	fread, fgets, fgetc, fwrite, fputs, fputc, fprintf, fseek	-----
1	ferror, feof, clearerr, fileno, fflush, fclose	-----
1	getline, getdelim	ic-rf
1	shm_open, ftruncate, shm_unlink	icmrf
1	sem_open, shm_close, sem_unlink	icmrf
1	sem_wait, sem_post	icmrf
1	sem_trywait, sem_timedwait	icmrf
1	sem_getvalue, sem_destroy	icmrf
2	fork, wait, waitpid	icmrf
2	exec*, fexecve	ic-rf
2	exit	icm--
2	pipe	ic-rf
2	dup, dup2, dup3	icmrf
3	socket, bind, accept, connect	ic-rf
3	listen	icmrf
3	getaddrinfo	ic-rf
3	freeaddrinfo	icmrf
3	send, recv	ic-rf
3	sendto, sendmsg, recvfrom, recvmsg	ic-rf
3	setsockopt	-----

[i] Function may be intercepted.
[c] Complex function arguments or return value(s) are recorded fully.
[m] Function arguments may be modified.
[r] Function may be mocked using a specified return value.
[f] Function may be mocked using a specified error value.

Table 5.1: List of relevant functions and their implementation status.

5.2.1 Performance when Intercepting

To test the performance of `intercept.so`, the following measurement environment was set up. On an x86-64 machine with an AMD Ryzen 7 7700X 8-Core processor, a simple program was called with an increasing number of iterations it had to perform. The program simply called the `pipe` function and then closed the created pipes in a for loop.

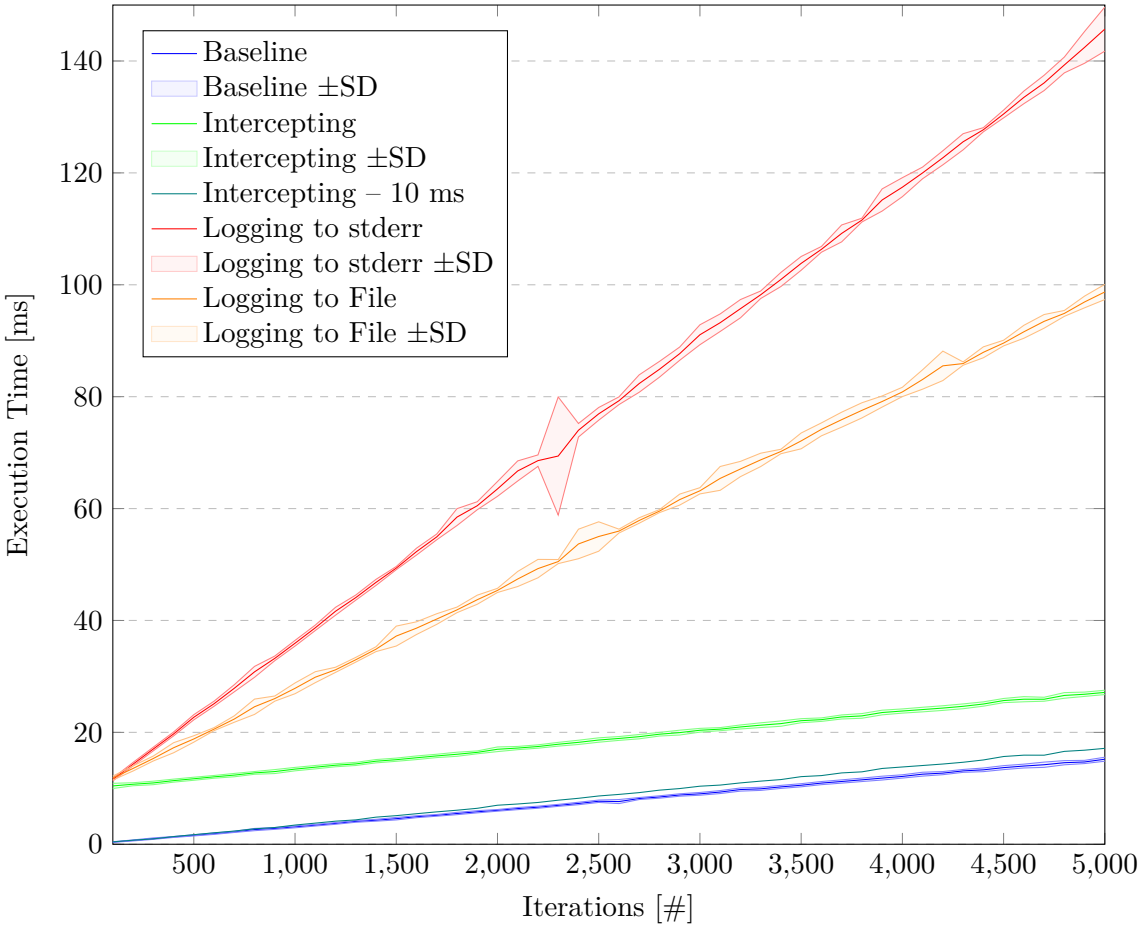

Figure 5.1: Execution times of a simple program using `intercept.so` with different output modes.

At first execution time of the program was measured without using `intercept.so` ("Baseline"). Then `intercept.so` was preloaded, but without any action to perform when intercepting ("Intercepting"). After that, logging to `stderr` was enabled ("Logging to stderr"). Finally, the logs were written to a file ("Logging to file"). For each of the four variants, the program was called with an iteration count beginning at 100 and increasing in steps of 100 up to 5000. Each measurement was taken 30 times, with one second between program executions to rule out statistical outliers. Figure 5.1 illustrates the results.

It is clearly visible that the initialization step of `intercept.so` always takes around 10 ms. In comparison to that, the performance degradation of the intercepting procedure alone is negligible, only around +13% compared to the baseline execution (see "Intercept – 10 ms" and "Baseline"). Most of the delay is caused by logging the recorded function calls.

5.2.2 Performance when Manipulating

Meaningful performance evaluation of function call manipulation requires a specific server implementation, since the response speed of the server dominates overall performance. As seen in Subsection 5.2.1, most delay comes not from intercepting itself, but from the further processing. This also applies to function call manipulation. The performance degradation heavily depends on the response speed of the used socket. Despite this, a simple performance test has been conducted.

The setup was the same as in Subsection 5.2.1. But this time `intercept.so` was configured to connect to a Unix domain socket. At first, a simple C program was used to respond to the messages on the socket, only using `getline` and `fprintf`. For the first test run the program always responded with the `"ok"` command ("Manipulate (simple ok)"), for the second with the `"return 0"` command ("Manipulate (simple return)"). After that, the default Python implementation developed in this work, which parses the incoming messages automatically, was tested. Again, at first always responding with the `"ok"` command ("Manipulate (Python ok)"), and then with the `"return 0"` command ("Manipulate (Python return)"). Figure 5.2 illustrates the results and some previous measurements for context.

Results of the simple C program show, that the communication over the socket alone has only minimal overhead compared with "Logging to stderr". Due to the parsing of messages the Python program has slightly worse performance degradation. The "return" test is slightly faster compared to the "ok" test, because the `pipe` function normally responds with `return 0; errno 0; fildes=[7,8]`, but when using `"return 0"` it responds with `return 0; errno 0`, which is less data to parse.

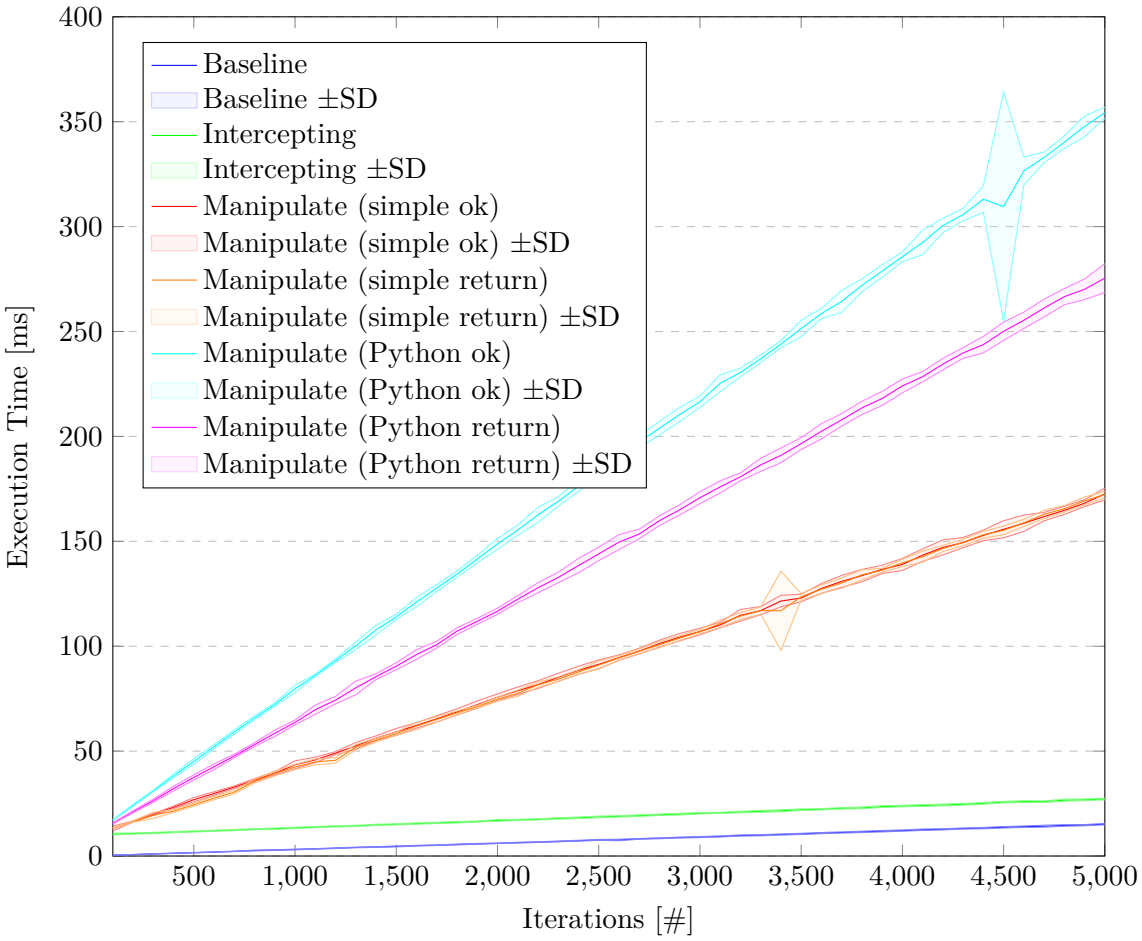

Figure 5.2: Execution times of a simple program using `intercept.so` with different manipulation modes.

CHAPTER 6

Conclusion

The primary goal of this work was to support the Operating Systems course by providing a practical and reliable way to automatically test students' C programs. Exercises in this course often involve low-level concepts such as semaphores, sockets, and shared memory, which are difficult to test automatically with conventional approaches. The motivation was therefore to develop a technique that allows intercepting function calls in order to verify whether students invoked the correct functions with appropriate arguments and in the correct order.

To address these challenges, this work presented `intercept.so`, a shared object file intended to be preloaded using `LD_PRELOAD`, which may be used to intercept function calls on Linux systems. Furthermore, a supporting Python program, `intercept`, was presented to make the shared object easier to use. By using preloading to hook or intercept function calls, the overhead and performance degradation remain negligible for the purpose of testing student submissions. To make use of intercepted function calls, some techniques of automatic testing of simple C programs were discussed.

In addition, the work has shown that creating an automated testing tool for the Operating Systems course assignments is both feasible and practical. The evaluation indicates that performance overhead is negligible and therefore not a hindrance to real-world use. Finally, since the approach relies on standard and portable mechanisms, the solution can be considered future-proof and adaptable to other environments.

The source code of the programs developed in this work is attached below. (Not all PDF viewers may open/download attachments.)

`intercept.so`: `intercept.c` (source code), and `Makefile`.

`intercept` (Python program).

Automatic testing: `auto-test.zip` (zipped Python programs).

List of Figures

List of Tables

List of Listings

Bibliography

[1] *dladdr(3) – Library Functions Manual – Linux manual pages.*

[2] *dlsym(3) – Library Functions Manual – Linux manual pages.*

[3] *GCC(1) – GNU – Linux manual pages.*

[4] *getaddrinfo(3) – Library Functions Manual – Linux manual pages.*

[5] *getline(3) – Library Functions Manual – Linux manual pages.*

[6] *ld(1) – GNU Development Tools – Linux manual pages.*

[7] *ld.so(8) – System Manager's Manual – Linux manual pages.*

[8] *LTRACE(1) – User Commands – Linux manual pages.*

[9] *ltrace.conf(5) – ltrace configuration file – Linux manual pages.*

[10] *malloc(3) – Library Functions Manual – Linux manual pages.*

[11] *ptrace(2) – System Calls Manual – Linux manual pages.*

[12] *READELF(1) – GNU Development Tools – Linux manual pages.*

[13] *STRACE(1) – General Commands Manual – Linux manual pages.*

[14] *Syscall User Dispatch – The Linux Kernel documentation.*

[15] *Using the GNU Compiler Collection (GCC).*

[16] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly, 3rd edition.

[17] DWARF Committee. Dwarf debugging information format.

[18] Nitesh Dhanjani and Justin Clarke. *Network Security Tools.* O'Reilly.

[19] T. Fraser, L. Badger, and M. Feldman. Hardening cots software with generic software wrappers. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 323–337 vol.2, 2000.

[20] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2004.

[21] Philip J. Guo and Dawson Engler. Cde: Using system call interposition to automatically create portable software packages. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[22] Quan Hong, Jiaqi Li, Wen Zhang, and Lidong Zhai. Datahook: An efficient and lightweight system call hooking technique without instruction modification. *Proc. ACM Softw. Eng.*, 2(ISSTA), June 2025.

[23] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Windows NT 3rd symposium*, 1999.

[24] Patrick Kern. Injecting shared libraries with ld_preload for cyber deception, 2023.

[25] Juan Lopez, Leonardo Babun, Hidayet Aksu, and A. Selcuk Uluagac. A survey on function and system call hooking approaches. *Journal of Hardware and Systems Security*, 1(2):114–136, 2017.

[26] Richard P. Spillane, Charles P. Wright, Gopalan Sivathanu, and Erez Zadok. Rapid file system development using ptrace. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, page 22–es, New York, NY, USA, 2007. Association for Computing Machinery.

[27] Wai Kit Sze and R. Sekar. Provenance-based integrity protection for windows. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC '15, page 211–220, New York, NY, USA, 2015. Association for Computing Machinery.

[28] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. zpoline: a system call hook mechanism based on binary rewriting. In *2023 USENIX Annual Technical Conference (USENIX ATC '23)*, pages 293–300, Boston, MA, July 2023. USENIX Association.