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, 1. August 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, August 1, 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. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 1. August 2025

Lorenz Stechauner

Kurzfassung

Lorem Ipsum.

Abstract

Lorem Ipsum.

Contents

CHAPTER 1

Introduction

Lorem Ipsum.

1.1 Motivation and Goal

Lorem Ipsum.

1.2 Definitions

Lorem Ipsum.

1.2.1 System Calls

Lorem Ipsum.

1.2.2 Function Calls

Lorem Ipsum.

1.3 Related Work

See also Section 2.1.

Lorem Ipsum.

https://dl.acm.org/doi/10.1145/3728874

What other solutions are available? What are the differences? What are the characteristics?

1.3.1 GDB Checker

Lorem Ipsum.

1.3.2 zpoline

Lorem Ipsum. [16]

1.3.3 DataHook

Lorem Ipsum. [15]

CHAPTER 2

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 2.8. Furthermore, an overview on how to test given programs is presented in Section 2.9. This chapter does not discuss how these function calls may be manipulated in any way. For that see Chapter 3.

2.1 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 methods on how to intercept function or system calls but does not claim completeness. The order of the following subsections is roughly based on the thought process on finding the most appropriate method suitable for this work.

2.1.1 `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. [9]

To make use of this system call, a corresponding command already exists. See Subsection 2.1.2.

2.1.2 `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. [11]

Listings 2.1 and 2.2 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 2.1: 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 2.2: Output of `strace ./main`.

This approach works great for debugging and other use-cases, but only intercepting system calls does not statisfy the requirements for this work.

2.1.3 `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 2.1.2). [6]

Listings 2.1 and 2.3 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 2.2). Again,

the function arguments are displayed in a “pretty” way. This command uses so-called prototype functions [7] to format function arguments.

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

Listing 2.3: Output of `ltrace ./main`.

This method fits the requirements for this work a lot better than `strace` (see Subsection 2.1.2), but it is not very flexible and offers no means to modify the intercepted function calls.

2.1.4 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 [14, Section 7.2] for more details on how to intercept system calls using kernel modules.

2.1.5 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 2.1.6 for more details.)

The default linker `ld` includes such a feature. See the ld(1) Linux manual page [4, 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 2.4 and 2.5 try to 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 2.4: Contents of `wrap.c`.

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

Listing 2.5: 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 2.1.6), this has not been further investigated.

2.1.6 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. [5]

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 [5, 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 2.6 and 2.7 try to 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 2.6: 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 2.7: 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. Although, one has to be aware that not only function calls inside the targeted binary, but also calls inside other libraries (e.g., to `malloc`) are redirected to the overriding function.

2.1.7 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 2.1.6) 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".

2.2 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 2.8 gives an overview over the grounding code structure. For each function that should be intercepted, this function simply has to be declared and defined the same way `malloc` was.

2.3 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 be too less. Within the following subsections it is tried to get 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. [7]

However, due to implementation complexity reasons and the need for "complex" return types for string/buffer and structure values (see Section 2.4) 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 2.8: 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 analysis on these records to be performed independently of the corresponding execution of the program. It should always be possible for any parser to fully parse the recorded calls without any specific knowledge of specific functions, their argument types, or return value type.

2.3.1 Numbers

The most simple types of argument 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)`).

2.3.2 Unspecific Pointers

Pointers with no further information known about (like `void *`) are essentially integers. Therefore, they may be treated as such.

Example: `free(0x55624164b2a0)`.

2.3.3 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" 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)`.

2.3.4 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)`.

2.3.5 Constants

For some functions constants are used. These constants are typically used C macros 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 disdinguish them from other types.

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

2.3.6 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")`.

2.3.7 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)`.

2.4 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.`

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.`

2.5 Determining Function Call Location

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

2.5.1 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 [12, 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 from 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 from where the function call came from.

2.5.2 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. [13]

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 [10, 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 2.8).

2.6 `intercept.so` Library

The time has come for putting it all together. As mentioned in Section 2.2, 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`.

2.7 `intercept` Command

To make the usage of the aforementioned shared object more easy, a simple python script has been put together. This script may be used as a command line tool. See Listing 2.9.

The synopsis of the command is as follows:

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

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 2.9: Contents of `intercept`.

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

-F, --functions A list of functions to intercept. See Section 2.6 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 2.6 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 2.6 for more details.

2.8 Example

To make it easier for the reader, Listing 2.10 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`.

2.9 Automated Testing on Intercepted Function Calls

The recorded function calls of a program run now may 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

```
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 2.10: Recoreded function calls from `./client`.

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 free'd. Any violation of such pre- and post-conditions may be reported as incompliant behavior. [8]

This means that intercepted function calls allow a tester to check if programmers use library function in compliance to their specification. 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 3 shows how this problem may be solved.

2.9.1 Validating Memory Management

Lorem Ipsum. (malloc, calloc, realloc, free, getaddrinfo, freeaddrinfo).

2.9.2 Validating Resource Management

Lorem Ipsum. (open, close, socket, ...).

CHAPTER 3

Manipulating Function Calls

This chapter discusses how to manipulate function calls and how this may be used to test programs. For how function calls may be intercepted at all, see Chapter 2. This chapter builds on the basis of the previous one and expands its functions. "Manipulation" in this context means to change the arguments of a function then running it normally, 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 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 sockets is implemented, but communication over TCP sockets is also easily possible.

Figure 3.1 illustrates the control flow for manipulating function calls.

3.1 Defining a Protocol

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

3.1.1 *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, ...).

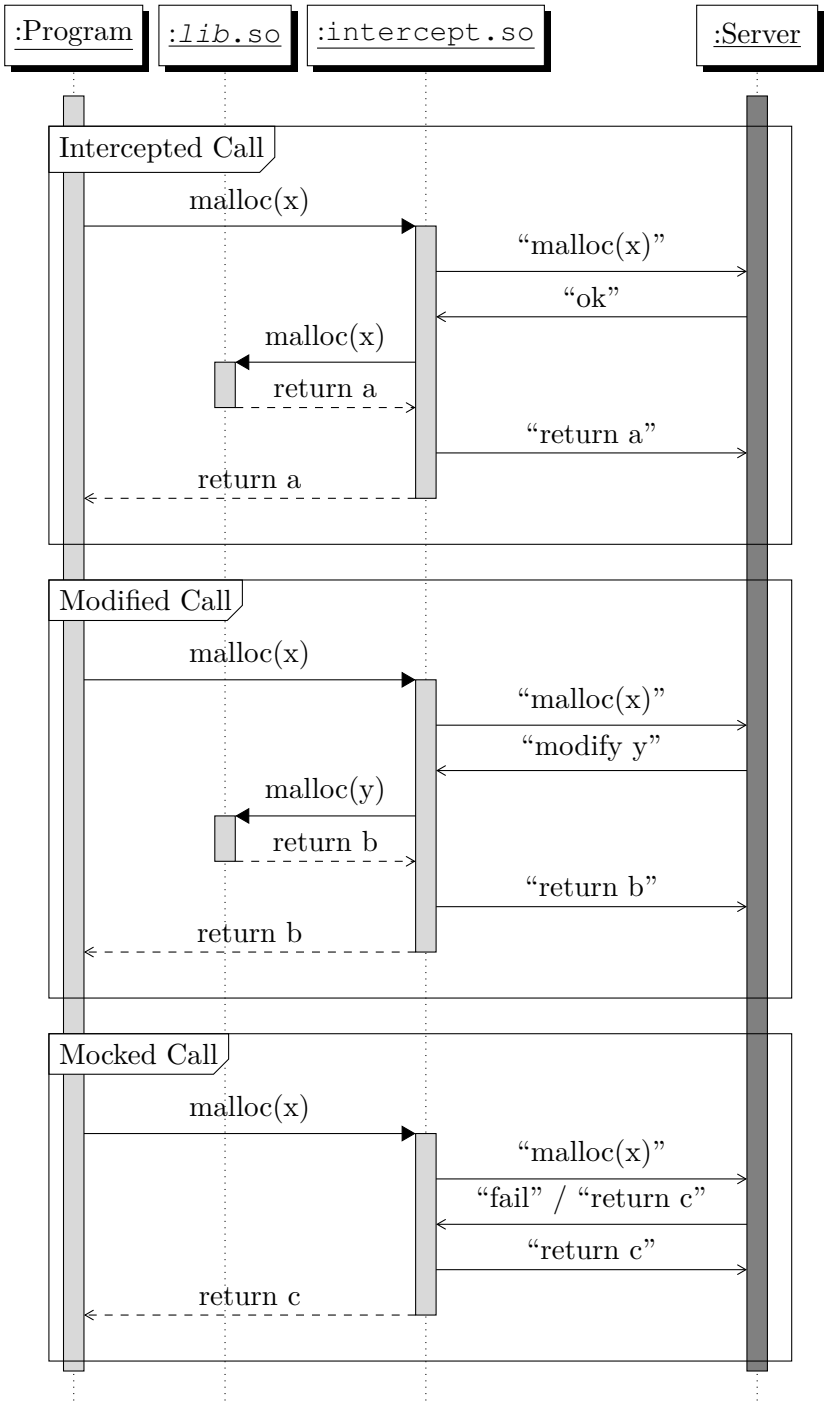

Figure 3.1: Simplified Control Flow for Function Call Manipulation.

3.1.2 *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 2.9).

3.1.3 *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.

3.1.4 *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 2.9).

3.2 Automated Testing using Function Call Manipulation

As seen in Figure 3.1 function call manipulation allows for mocking individual calls. Mocking may be used to see how the program behaves when individual calls to function fail or return an unusual, but valid, value. The simplest way to automatically test programs is to run them multiple times and on each run let a single function call fail. 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).

3.2.1 Testing Return Value Checks

Figure 3.2 shows the simplified and collapsed call sequence graph of the prior example in Section 2.8. 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, each for every possible error return value, but in this example they all yield the same resulting path, therefore, they have 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. This test seems trivial at first. 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.

3.2.2 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 3.2.1.

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 3.2.1, 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.

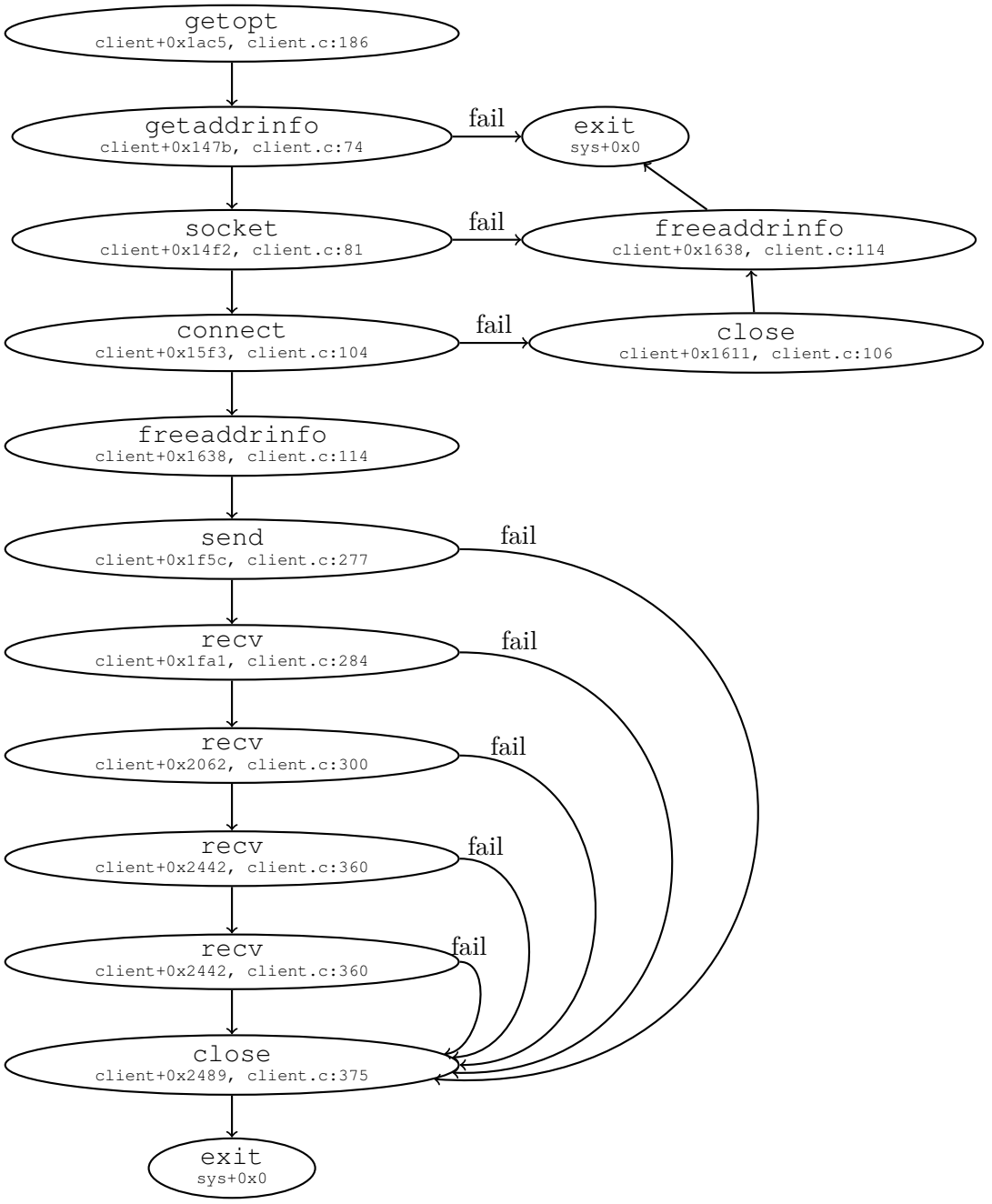

Figure 3.2: Simplified Call Sequence Graph of `./client`.

CHAPTER 4

Conclusion

Lorem Ipsum.

Perhaps do some study/“research” on performance (CPU/memory/...).

Overview of Gen. AI Tools Used

No generative AI tools were used in and for this work whatsoever.

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

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

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

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

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

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

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

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

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

[13] DWARF Committee. Dwarf debugging information format.

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

[15] 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.

[16] 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.