University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Krzysztof Małysa
Student no. 394442

Multi-process sandbox for unprivileged users on Linux

Master's thesis
in COMPUTER SCIENCE

Supervisor:
dr Janina Mincer-Daszkiewicz
Institute of Informatics

Warsaw, December 2022

Abstract

TODO

Keywords

sandboxing, security, Linux, secure execution, arbitrary code execution, judging system,

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

Security and privacy – Systems security – Operating systems security

Tytuł pracy w języku polskim

Sandbox wielu procesów dla nieuprzywilejowanych użytkowników systemu Linux

Contents

1. Introduction 5
- 1.1. Background 5
- 1.2. Goal of the thesis 5
  - 1.2.1. Requirements 6
  - 1.2.2. Existing solutions 6
- 1.3. Structure of the Thesis 7

2. Literature overview 9
- 2.1. Overview of Sandboxing Techniques 9
- 2.2. Existing Implementations 9
  - 2.2.1. Modifying OS kernel 9
  - 2.2.2. `ptrace`-based 11
  - 2.2.3. Using Linux namespaces 11
  - 2.2.4. Other 12
- 2.3. Conclusion 12

3. Design and Architecture 13
- 3.1. Client-server architecture 13
  - 3.1.1. Sandboxing request handling 14
- 3.2. Cgroups 14
- 3.3. Linux namespaces 15
- 3.4. Inter-process communication 16
- 3.5. Capabilities 16
- 3.6. Hardening 17
- 3.7. Conclusion 17

4. Implementation 19
- 4.1. Interface 19
- 4.2. Time limits 20
  - 4.2.1. Real time limit 20
  - 4.2.2. CPU time limit 20
- 4.3. Runtime statistics 21
- 4.4. Error handling 21
- 4.5. Request sending and receiving 21
- 4.6. File descriptors 21
- 4.7. Canceling the request 22
- 4.8. Killing the request 22
- 4.9. Sandbox server upon client death 22

4.10. PID 1 process upon supervisor death 22
4.11. Signals 23
    4.11.1. Tracee signals 23
    4.11.2. `SIGPIPE` in the supervisor process 23
    4.11.3. Undefined Behavior Sanitizer 23
4.12. Running as superuser 24
4.13. Performance optimizations 24
    4.13.1. Seccomp filter of the PID 1 process 24
    4.13.2. Seccomp filter as file descriptor 24
    4.13.3. Unsharing network, ipc, uts and time namespace 24
4.14. Integration with Online Judge Platform 24
    4.14.1. Interactive problems 25
    4.14.2. Non-interactive problems 26
    4.14.3. Conclusion 27
4.15. Testing and validation 27
4.16. Challenges faced 27
    4.16.1. `execveat` returns `EINVAL` 27
    4.16.2. `execveat` returns `ENOENT` 27
4.17. Conclusion 28

5. Performance Evaluation 29
5.1. In the context of the online judge 29
    5.1.1. Compilation 29
    5.1.2. The test program 29
5.2. Short-running programs 30
5.3. Comparison with nsjail 30
5.4. Impact of some optimizations 30

6. Future work 33

7. Conclusion 35

Chapter 1

Introduction

1.1. Background

Secure execution environments are commonplace these days, from containers and virtual machines on servers to sandboxes on laptop and smartphones — most of which run on Linux. They are used to securely execute untrusted code, as well as trusted programs to prevent damage escalation in the event of unknown vulnerabilities. Their key features are isolation, limiting resource usage, and accounting for resource consumption.

The features of Linux allow the creation of simple yet effective and efficient secure environments. They work at application runtime, so in most cases existing software does not need to be adapted to use them. This makes them easily applicable, and explains why their adoption is growing.

In this thesis, the most important application of sandboxing are online judge systems. Online judge systems have beneficial role in programming education and competitive programming. They allow testing user-provided solution to a specific problem. The solution is run on a predefined test cases in order to check if it is valid. In such platforms isolating the compilation and running of the tested program is essential to provide security and robustness of the platform itself.

Historically, isolation techniques evolved together with the online judge platforms. The most primitive (yet insecure) was usage of `chroot(2)` [29] to restrict access to part of the filesystem. To increase isolation virtual machines were used [2]. Later, containerization became a new way to provide isolation [20, 40].

Online education platforms greatly facilitate teaching and learning programming. They provide quick feedback on the correctness of the code the user submits. They are used in schools and universities and provide great learning opportunities for all.

Moreover, a versatile sandbox has applications outside online judging platforms. For example, it can be used to sanitize compiling a PDF form LaTeX sources or for safe execution of untrusted server-side scripts in web applications.

1.2. Goal of the thesis

The goal of the thesis is to design, implement and integrate a new sandbox for the Sim project [16]. The Sim project is an online platform for preparing people for and carrying out algorithmic contests. The project started in 2012 and is developed by me since the beginning. It is used at the XIII High School in Szczecin and programming camps to teach young people programming and algorithms. It has an online judge with a sandbox specially developed for this use case. Over the years the sandbox became a limitation. It only allows running a single-threaded statically linked executable of programs written in C, C++ or Pascal. The new sandbox will allow supporting more programming languages and improve security of the tested program compilation stage.

1.2.1. Requirements

The new sandbox needs to be optimized for running short-running programs as well as have minimal runtime overhead. Most of the test cases the tested program is run on are small and it completes them in less than 10ms. The goal is to allow hundreds of such sort-running runs per second, hence optimizing for short-running programs is important. However, minimizing overhead of the sandbox during the run is also important i.e. if the program runs X ms normally, the objective is that the program inside the new sandbox will also run approximately X ms.

The new sandbox needs to be versatile. It will be used to secure the compilation of the tested programs as well as running of the tested programs. Compilation is a complicated process that involves parsing, translating, optimizing and linking the final program. For languages like C, C++ and Rust it involves running several executables in coordination e.g. compiler and linker i.e. more than one process at a moment — the sandbox needs to support that.

Sandboxing needs to have a low overhead. Apart from small test where tested program runs quickly (a matter of milliseconds), almost always the tested program is run on big test cases, where it may need several seconds for it to solve the problem. Increasing this time as little as possible while the tested program is running inside the sandbox is one of the primary objectives.

It often requires running several executables e.g. compiler and linker, so allowing a single process inside sandbox is not enough. Sandboxing the tested program is simpler, because it is a single process. But since it is often short-running, the overhead needs to be minimal.

The sandbox needs to allow limiting resources. Real time, CPU time, memory – these need to be limited not only for the robustness of the platform, but specific problems require different limits. The goal of some problems is to solve it with very restricted memory e.g. find a missing integer in a random permutation of integers $1, \ldots, n$ without one element, but in constant memory.

The sandbox needs to account resource usage. For every test, the user is presented with consumed memory and CPU time by their solution. The sandbox needs to provide this information.

The last requirement is the sandbox will not require any privileges. There is a tool called Sip [17] for preparing the problem packages for the Sim platform. One of the purposes of the tool is to run the solutions inside the same secure environment as on the Sim platform. The user should not need any privileges to run this tool, so the sandbox should not require them either.

1.2.2. Existing solutions

Approaches to form a secure execution environments differ. One of them is virtualization or emulation e.g. QEMU [27] and KVM [26], VirtualBox [28], VMWare Workstation [44]. Although powerful and effective, they come with an enormous overhead i.e. booting up an entire operating system. Moreover, emulation noticeably slows down the runtime of an emulated application, rendering such solutions inapplicable.

Containers provide much lower overhead: setup of an order of milliseconds and negligible runtime overhead. But, Docker [22], LXC [1] require root privileges to create a container. systemd-nspawn [42] requires root privileges to run.

Rootless are containers [38] that can be created and run by an unprivileged user are the almost perfect solution to the problem. They provide almost all of the functionality of the normal containers but without the need to engage a privileged user. However, they often use setuid binaries and that is undesirable [39]. Also they are not optimized to run sequences of short-running programs. In this thesis we will create a sandbox that uses the same techniques as rootless containers but will be optimized for running sequences of short-running programs.

1.3. Structure of the Thesis

Chapter 2 contains overview of sandboxing techniques and existing implementations and comparative analysis of them. Details of design and architecture are described in chapter 3. Implementation is described in 4. Chapter 5 contains performance evaluation of the final implementation and impact of some optimizations. Later, in chapter **??** use cases and applications are discussed. Chapter **??** details integration with online judge and challenges involved. In chapter 6 future work and opportunities are discussed. Finally, chapter 7 contains the conclusion.

Chapter 2

Literature overview

2.1. Overview of Sandboxing Techniques

During the first programming competitions, the human judges manually read and verified the source code of the contestants' solutions [43]. Over time this became infeasible and gave birth to automatic judge systems.

To prevent people from interfering with the normal workflow of the competition e.g. Denial of Service Attack by exhausting memory resources, the automatic judge systems need a secure way to compile and execute a contest's solution. This is where sandboxes come into place.

First sandboxes required modification of the OS kernel [36, 7, 8, 10, 13]. While they had little run-time overhead, some of them were limited to single-threaded applications [24].

Later, as support for process tracing matured, `ptrace`-based sandboxes arose [19, 12, 11]. The problem with those solutions is the overhead that varies from around 75% [23] to 160% [20] for syscall-intensive programs. This overhead however, does not affect programming contest fairness much [18]. Supporting multi-threaded and multi-process programs while using `ptrace` is tricky, but possible [11], because of Time of Check/Time of Use (TOCTOU) problem [5]. `ptrace`-based sandbox needs to inspect syscall arguments. To do so it has to read them, but the multi-threaded or multi-process program can change the indirect argument after the reading but before the kernel uses the argument. This creates a dangerous race condition that has to be addressed.

Finally, after the kernel support for containerization materialized, namespace and cgroup based sandboxes came into place [20, 25, 37, 9, 6, 40]. Contrary to ptrace-based sandboxes, namespace-based sandboxes have negligible runtime overhead [20]. Moreover, they don't require modifications of the Linux kernel and work on major Linux distributions out of the box.

2.2. Existing Implementations

2.2.1. Modifying OS kernel

Systrace

Systrace [36] intercepts all system calls in the kernel. It then decides if the syscall is safe by first checking a static list of safe syscalls. This step exists to reduce sandboxing performance overhead. If the syscall is not on the list, Systrace consults user space for a decision.

The system avoids TOCTOU problem [5] by copying syscall arguments to kernel memory before asking user space for a decision.

Janus

Janus [7] adds a module to extend Linux `ptrace` API. Policies are defined using configuration files. By default all syscalls are denied. The configuration directive refers to the policy module that provides the logic for deciding whether to allow a particular system call or not. For example, `path` module could be used to restrict IO on certain file paths.

Ostia

Ostia [8] instead of filtering system calls it delegates them to an external agent that performs syscalls on behalf of the sandboxed process. Authors emphasize that such architecture simplifies the system and protects from TOCTOU problems [5].

Ostia is implemented as two components: a small kernel module and a user space part. The module intercepts the syscall and copies its arguments via IPC link to the user space agent. The agent decides whether the call should be allowed, executes it and returns the results back over the IPC link. Worth noting is that not all syscalls have to delegated — some can be always allowed while others always denied.

TxBox

TxBox [10] introduces system-level transaction support. Impact of the untrusted insecure code is limited by rolling back the system state after the execution. This provides strong isolation and works with arbitrary executables but requires significant out-of-tree patches of the OS kernel.

MiniBox

MiniBox [13] is a two-way sandbox that protects operating system from the application as well as application from the operating system. A modified version of TrustVisor [21] hypervisor runs OS and sandboxed application separately in a Mutually Isolated Execution Environment. The hypervisor is the only communication channel between the isolated application and the regular OS. This way application is protected from the malicious operating system. To protect the OS from the application, MiniBox uses Software Fault Isolation techniques from NaCl [45].

SACO sandbox

South African Computer Olympiad (SACO) sandbox [24] inserts a custom kernel module that hooks up to Linux Security Module infrastructure. Although it has negligible time and memory overhead, it only supports single-threaded programs.

2.2.2. `ptrace`-based

MO sandbox

MO sandbox [19, 12] allows only single-threaded programs. It simply inspects arguments using `ptrace` and uses `setrlimit` [31] to limit resources. It is used by USA Computer Olympiad (USACO).

MBOX

MBOX [11] requires no superuser privileges. It makes use of seccomp BPF system call filtering to restrict allowed syscalls. BPF filtering is effective only for non-indirect arguments. To address this issue, the installed BPF filter notifies the `ptrace` monitoring process if further argument inspection is necessary to make a decision. To avoid TOCTOU problem [5], the MBOX allocates a read-only page to which it copies the indirect arguments before inspecting them and rewrites the syscall to use the rewritten arguments. The copied arguments are protected against modification because changing page access permissions is impossible without a syscall.

2.2.3. Using Linux namespaces

Firejail

Firejail [25] uses seccomp BPF system call filtering and mount namespaces to restrict filesystem access. Similarly it uses process namespaces to limit view of running processes and network namespaces to restrict access to network devices. However, Firejail uses a `setuid` [34] helper binary to achieve that. It allows resource limiting through `prlimit` [31].

nsjail

nsjail [9] uses Linux namespaces, seccomp BPF system call filtering, `setrlimit` [31] and cgroups to limit resources. It does not require superuser privileges. However, it is not optimized for running short-running programs. Also, it does not provide statistics of the run.

nsroot

nsroot [37] does not support resource limiting. It only makes use of Linux namespaces to restrict view of the file system, IPC and network devices.

Flatpak

Flatpak [6], previously xdg-app, is a software packaging and sandboxing tool. Internally, it uses Bubblewrap sandbox. The Bubblewrap [14] is a `setuid` [34] program that uses Linux namespaces and seccomp filters.

New Contest Sandbox

New Contest Sandbox [20] uses Linux namespaces and cgroups but not seccomp filters. It is used by Moe modular contest system (2012) [20]. Linux namespaces and cgroups have negligible overhead compared to `ptrace`.

APAC

APAC (Automatic Programming Assignment Checker) [40] uses Docker for sandboxing. It sets up a container for each run. Docker uses runC under the hood. While runtime overhead of Docker is low, the setup phase is primary source of overhead for short-running programs.

runC

runC [3] uses the same features of Linux kernel as nsjail. However, configuration is stored as files instead of passed as command-line arguments. It has a special `rootless` mode which does not require superuser privileges. Given all of the above however, it is not optimized for running short-running programs. runC is used internally by Docker.

2.2.4. Other

Google Native Client

Native Client (NaCl) [45] uses static analysis and Software Fault Isolation. After the static analysis, the program runs at native speed but requires recompilation with special compiler and libraries. NaCl only works for x86 architecture.

2.3. Conclusion

Many sandboxing solutions exist. From all of the above, closest to our requirements is nsjail (see Section 2.2.3). However, it is not optimised for running short-running programs. In fact, none of the above solutions is optimised to run hundreds of short-running programs per second.

Considering the similarities of nsjail and our solution, in the performance analysis we will compare our sandbox to nsjail sandbox.

Chapter 3

Design and Architecture

3.1. Client-server architecture

From the start the sandbox was based on the client-server architecture. This was the choice to minimize process cloning overhead [15], since it is a costly operation and happens for every sandboxing request. fork/clone needs to clone the whole address space of the process and the client process could have a large address space. Server process that is executed from a separate executable has a minimal required address space size therefore the cloning overhead is minimal for every request. Moreover, this architecture easily and safely allows sharing as much work as possible between the sandboxing requests which is the key to low overhead of running short-running programs.

The client spawns the sandboxing server and sends sandboxing requests via UNIX domain socket to the server. This is illustrated on Figure 3.1. The request contains executable, arguments, namespace configuration, resource limits, seccomp BPF filters and a pipe through which the result will be sent back.

At startup, the server creates cgroup hierarchy and some namespaces so that they won't have to be created later or their creation will be faster. Other utilities are also setup here to do it once instead of for every request. Then it starts accepting requests.

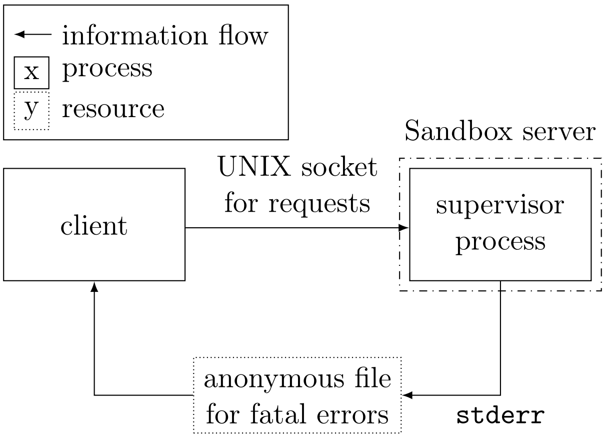

Figure 3.1: Sandbox server waits for requests. Client sends requests through the UNIX socket. Sandbox server will die on fatal error leaving the error message for the client in the anonymous file.

3.1.1. Sandboxing request handling

For each request, the server process (aka supervisor) spawns the PID 1 process of the new PID namespace. Then the init process setups namespaces and some of the resource limits. Finally the PID 1 process spawns the tracee process that finishes configuration and executes the requested executable. So for each sandboxing request we spawn exactly 2 processes. However, the executed program can spawn new processes — each of them is referred to as a "tracee process". The PID 1 process is necessary for a couple of reasons:

- It reaps the zombie processes in the tracee PID namespace.
- It allows locking mount-points in the mount namespace. The tracee process is spawned in a new user and mount namespace. Mounts are performed by the PID 1 process, therefore all mounts become locked together and cannot be individually unmounted by the tracee [32]. These mounts cannot be performed by the supervisor process instead, because it would alter the mount namespace for subsequent requests.
- Inside a PID namespace, sending signals to the PID 1 process is allowed only for signals that the PID 1 process installed signal handler for. This could change the behavior for some programs, therefore a helper PID 1 process is needed.

This is shown on Figure 3.2.

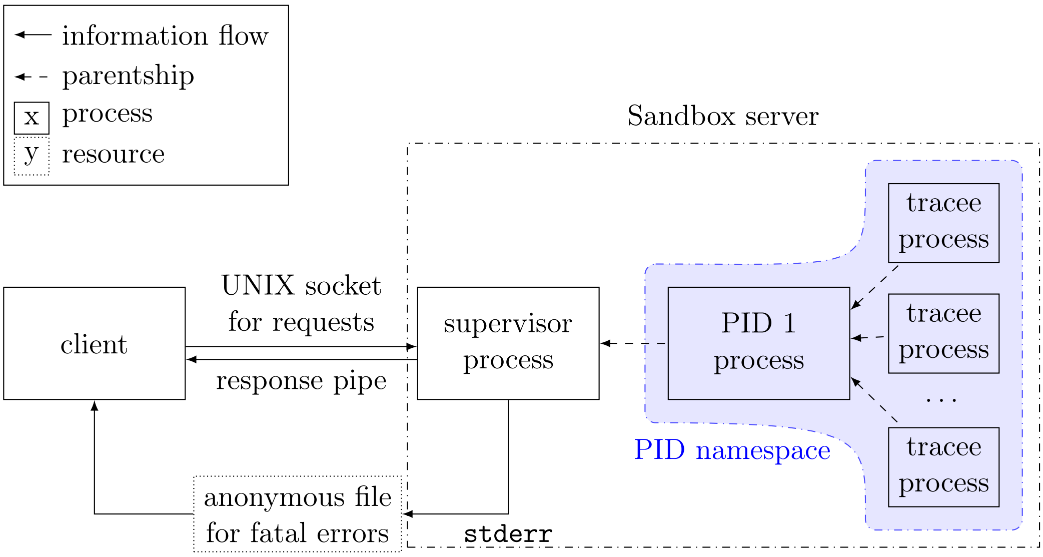
information flow
parentship
x process
y resource
Sandbox server
tracee process
UNIX socket for requests
client
supervisor process
PID 1 process
tracee process
response pipe
...
PID namespace
tracee process
anonymous file for fatal errors
stderr

Figure 3.2: Sandbox server handles a request, at the moment after executing the requested executable. Sandbox server will die on fatal error leaving the error message for the client in the anonymous file. Sandbox server consist of the supervisor process and its child — the PID 1 process that is spawned for each request. The PID 1 process performs a role of the init process in the PID namespace of the tracee processes.

3.2. Cgroups

The server gains write access to cgroup hierarchy by being executed through `systemd-run --user --scope --property=Delegate=yes --collect`. It enables `pid`, `memory`, and `cpu` controllers for the below subgroups.

The server process creates at startup the cgroup v2 hierarchy that looks as follows:

- `/supervisor` — cgroup of the supervisor process,
- `/pid1` — cgroup of the pid1 process,
- `/tracee` — cgroup of the tracee processes.

After creation of the hierarchy it places the supervisor process in its cgroup. Subsequent processes are placed in their cgroups by making use of `CLONE_INTO_CGROUP` flag.
`/tracee` cgroup allows:

- Killing all tracee processes by writing 1 to `/tracee/cgroup.kill` file.
- Reading CPU user and CPU system time via `/tracee/cpu.stat` file.
- Reading peak memory usage via `/tracee/memory.peak` file.
- Setting process/tread number limit by writing `/tracee/pids.max`.
- Setting memory hard limit by writing `/tracee/memory.max`.
- Setting CPU usage limit by writing `/tracee/cpu.max`.
- Disabling PSI accounting to reduce the sandboxing overhead by writing 0 to `/tracee/cgroup.pressure` file.

`/tracee` cgroup needs to be deleted and recreated after each request to reset `/tracee/cpu.stat` and `/tracee/memory.peak` files.

3.3. Linux namespaces

Linux allows unprivileged users to create user namespaces only. However, after entering a new user namespace the process gains all privileges inside the namespace and can create other namespaces.
The supervisor process creates the following namespaces:

- user namespace — in order to create other namespaces and hide user ID and group ID,
- mount namespace — to allow mounting detached cgroups v2 hierarchy,
- cgroup namespace — to allow mounting detached cgroups v2 hierarchy,
- network namespace — to disconnect every tracee from network devices, done once, as it is costly,
- IPC namespace — to isolate every tracee from other processes' IPC, done once, for optimization,
- UTS namespace — to isolate every tracee from host's hostname, done once, for optimization,
- time namespace — to isolate every tracee from host's time namespace, done once, for optimization.

The pid1 process creates the following namespaces:

- user namespace — in order to create other namespaces and hide user ID and group ID,
- mount namespace — to allow mounting requested mount-point hierarchy,
- PID namespace — to isolate tracee from accessing other processes.

The tracee process creates the following namespaces:

- user namespace — in order to create other namespaces and hide user ID and group ID and lock the mount tree,
- mount namespace — in lock the mount tree created by pid1.

The listed namespaces hierarchy is illustrated on Figure 3.3.

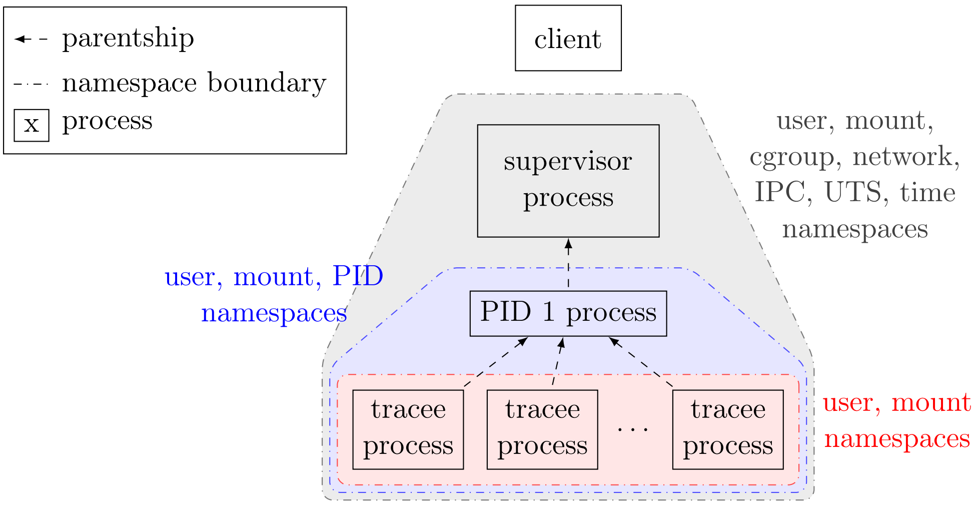
parentship
namespace boundary
x
process
client
supervisor process
user, mount, cgroup, network, IPC, UTS, time namespaces
user, mount, PID namespaces
PID 1 process
tracee process
tracee process
. . .
tracee process
user, mount namespaces

Figure 3.3: Namespaces hierarchy of the sandbox server processes.

3.4. Inter-process communication

The client sends requests via UNIX domain socket to the supervisor process. The results are sent via a pipe attached to the request. The pipe is attached to the request as a file descriptor using `SCM_RIGHTS` control message [35].

The supervisor, pid1 and tracee process communicate via shared anonymous memory page. Figure 3.4 illustrates this communication. Such communication requires no syscalls, is fast and reliable. This page is automatically unmapped upon `execveat` syscall [30] in the tracee process, so it is protected from the tracee access.

3.5. Capabilities

The supervisor process drops all capabilities, sets securebits and `NO_NEW_PRIVS` flag. This ensures minimal possible capabilities in its and ancestor user namespace and prevents gaining any new privileges.

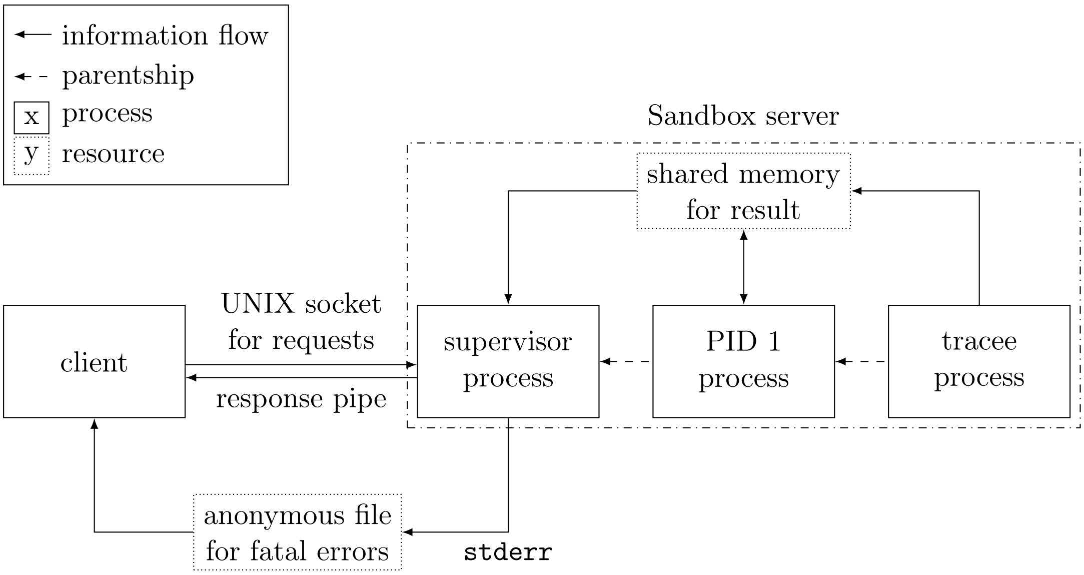
information flow
parentship
x
process
y
resource
Sandbox server
shared memory for result
client
UNIX socket for requests
response pipe
supervisor process
PID 1 process
tracee process
anonymous file for fatal errors
stderr

Figure 3.4: Sandbox server handles a request, at the moment before executing the requested executable. Sandbox server will die on fatal error leaving the error message for the client in the anonymous file. Sandbox server consist of the supervisor process and its child — the PID 1 process that is spawned for each request and its child — the tracee process that will execute the requested executable. The tracee process saves in the shared memory the time just before `execveat` and signals the PID 1 process. The PID 1 process reads the time saved in the shared memory and starts the real and CPU timers. After the tracee process dies, the PID 1 process writes exit code and status of the tracee process and the time it died. Moreover, the shared memory is used to communicate fatal errors to the supervisor process.

3.6. Hardening

The pid1 process, after spawning the tracee enters a new cgroup namespace to limit view of other namespaces i.e. if the tracee somehow takes control of the pid1 process, it will not be able to raise its PID and memory limit. Moreover a seccomp filter is installed to limit allowed syscalls only to those needed for reaping the orphaned zombie process, managing time limits and exiting upon tracee death.

3.7. Conclusion

Client-server architecture allows time-performance optimizations. Furthermore, it allows more common work to be done once and simplifies implementation. For instance, file descriptors do not leak to other processes because there are no threads that could fork a new process. Request handling requires creation of 2 processes — the PID 1 process and the tracee process that later executes the requested program. Resource limits and accounting is mostly performed by cgroups. Isolation is achieved by deft usage of Linux namespaces.

Chapter 4

Implementation

The project is written in C++, as it is a low-overhead, low-level language, but more convenient than C. Git is used as a Version Control System to track incremental implementation. Invaluable tool used during development was `strace` [41]. It allowed easy inspection of system calls and their return values without any modification to the source code.

4.1. Interface

The client has to spawn the sandbox server — the supervisor process. It then operates on the connection handle. Using the connection handle, the client can send requests to execute programs in the sandbox.

After sending a request, a request handle object is constructed. It can be used to obtain result of the execution, cancel execution or kill the tracee processes. Canceling execution is useful in case of errors in the client, where the result of the execution as well as the execution itself are no longer needed. Upon cancellation of the request, tracee process are immediately killed. Result of running the request is discarded. Canceling an already finished request discards the result. Canceling the request before the server started handling it causes it to be skipped. Killing an already finished request is no-op. Killing the request before the server started handling it causes tracee to be immediately killed after the `execveat`.

The client can then await the result of the request using the request handle. The result can either be a successful result or an error with a textual description. This error is not fatal to the supervisor process i.e. new requests can be sent. The successful result consists of the exit code and status, and runtime statistics:

- real time,
- CPU user and CPU system time,
- peak tracee cgroup memory usage.

Each request has a set of accompanying options:

- Optional `stdin`, `stdout`, and `stderr` file descriptors. If optional is specified as empty, `/dev/null` is opened as the file descriptor.
- Environment as an array view of string views.
- Linux namespace configuration:

  - user ID and group ID mapping,
  - mounts and new root mount,

- Cgroup resource limits: process and thread limit, memory limit, CPU maximum bandwidth.

- `prlimit` hard limits.

- Real time limit.

- CPU time limit.

- Seccomp BPF filter as a file descriptor. The decision to pass it as a file descriptor is that it lowers the overhead of repeatedly using the same filter — a common scenario in a judge system. Only the file descriptor needs to be sent with each request instead of the whole BPF filter content. This allows the filter to be compiled once and passed for multiple requests with minimal overhead. An alternative is to extend the API to save seccomp filters but it was considered unnecessary given how small is the overhead of passing a single file descriptor.

4.2. Time limits

The PID 1 process controls the time limits. The tracee process, just before `execveat` saves current real time from `CLOCK_MONOTONIC_RAW` and CPU time from `cpu.stat` tracee cgroup file to the shared memory (see Figure 3.4). The problem with `cpu.stat` file is that it is updated infrequently. For a young tracee process, this file often reports consumed CPU time equal to 0 microseconds instead of a few hundred microseconds. Fortunately, executing `sched_yield()` system call forces recalculation of the file and the values are no longer 0. This is why this syscall is required as allowed in the seccomp BPF filter.

4.2.1. Real time limit

After saving the current real time the tracee process signals the PID 1 process with `SIGUSR2`. The PID 1 process reads the saved real time and sets up a POSIX timer to expire at the moment of saved time + real time limit. When the timer expires, `SIGUSR1` is sent by the kernel to the PID 1 process and it terminates all tracee processes by writing 1 to the `cgroup.kill` file of the tracee cgroup.

4.2.2. CPU time limit

In case the tracee is not restricted to one process, the setup is analogous to real time except that there is no CPU timer for a cgroup of processes. Instead we calculate minimal period of time in which the CPU time limit could expire as follows: $\frac{\text{remaining cpu time}}{\text{max parallelism}}$, where max parallelism equals: $\min(\text{available threads}, \texttt{process\_num\_limit}, \texttt{cpu\_max\_bandwidth}\text{ in threads})$. Upon the timer expiration the remaining cpu time is recalculated and the timer is rescheduled if the remaining cpu time is greater than 0. Timer expiration is signaled by the kernel as signal `SIGXCPU`. To prevent polling, the minimal timer expiration period is capped to have minimum value of 1ms — this gives at most 1000 checks per second.

In case the tracee is restricted to one process, the setup is different. After saving the current CPU time the tracee process signals the PID 1 process through a pipe. A signal cannot be used because `timer_create` syscall and `clock_getcpuclockid` library function are not marked async-signal-safe — they are not specified to be safe to call inside a signal handler. An `eventfd` cannot be used either, because if tracee dies before writing to the `eventfd`, the PID 1 process will wait indefinitely on the `read` syscall. With a pipe, `read` syscall returns 0 when the other end becomes closed. With the limit of one process we use the CPU timer of the tracee process and set up a timer to expire when the tracee exceeds the CPU time limit.

In both cases, when the CPU time limit is exceeded, the PID 1 process is signaled about it and it terminates all tracee processes by writing 1 to `cgroup.kill` file of the tracee cgroup.

4.3. Runtime statistics

After the main tracee process (the first spawned process) exits, the PID 1 process saves the current real time and the exit status in the shared memory, unless the tracee set an error, and exits. The kernel kills all remaining tracee processes (because the PID namespace's init process died). After the PID 1 process exits, the supervisor process reads the shared memory (see Figure 3.4). It checks if there is an error of either tracee or PID 1 process. If there is one, it becomes the result of the request. If there is none, the supervisor process calculates:

- real time using formula: time of tracee death − saved `execveat` real time,
- CPU time using formula: CPU time read from `cpu.stat` file−saved `execveat` CPU time,
- Peak memory usage by reading `memory.peak` tracee cgroup file.

4.4. Error handling

Errors in the supervisor process are considered fatal and are reported by writing to `stderr`. After writing errors, the supervisor process exits immediately. When the client tries to read the request result, the read will fail with `read` returning unexpected value 0. The client then ensures the supervisor process is dead (in case the communication failed) and tries to read the error the supervisor wrote. If the client finds one it throws exception with this error, otherwise it throws the exception with the `read` error.

The PID 1 process and tracee process write error to the shared memory (see Figure 3.4) and exit immediately. The supervisor process reports these errors as a request result.

4.5. Request sending and receiving

The request is sent via UNIX domain socket (see Figure 3.1). The request consists of a constant-length with file descriptors and a variable length body. The request header contains only the length of the request body. The request body contains all parameters of the request that are serialized to a custom binary format.

4.6. File descriptors

The sandbox server closes all file descriptors except the UNIX socket fd and opens `/dev/null` as `stdin`, `stdout`, and `stderr`. This is a small optimization, in case the request does not specify a standard file descriptor. For instance, if the request is to execute a program without `stdin`, the sandbox server has to set up the `stdin` of the tracee process to be `/dev/null` opened for reading. However, the tracee process inherits the file descriptors of the PID 1 process that inherits the file descriptors of the supervisor process. The supervisor process has already opened `/dev/null` as the `stdin` file descriptor. Therefore no action is needed in the PID 1 process and the tracee process for `stdin` of the tracee process to be `/dev/null` opened for reading. The same principle applies to `stdout` and `stderr` file descriptors.

All file descriptors are opened with `O_CLOEXEC` flag so that they will not leak to the executed process. A unit test to check if any file descriptor leaks to the sandboxed program is in the test suite.

The PID 1 process inherits all request standard file descriptors and passes them to the tracee process. It has to close them after spawning the tracee process. To see why, let's consider a pipe of which one end is passed as a `stdin` to the sandboxed program. A pipe is broken if all file descriptors of one end become closed. If the PID 1 process did not close the standard file descriptors of the tracee, the pipe could not become broken until the PID 1 process dies. This changes the semantics of the pipe if the program is run inside the sandbox and is undesirable. Moreover, for hardening purposes the PID 1 process closes all unnecessary file descriptors after spawning tracee — in case, the tracee somehow takes control of the PID 1 process.

4.7. Canceling the request

The response to the request is passed via pipe that is provided alongside the request. If the pipe becomes broken i.e. the client closes the read end of the pipe, the request is considered cancelled and is immediately discarded if currently handled and omitted otherwise.

4.8. Killing the request

Alongside request is send an `eventfd` file descriptor. The supervisor process monitors this file descriptor and if it becomes readable i.e. the client writes a value to it, the tracee is killed immediately. To avoid false-positive errors (the tracee process is killed unexpectedly), if the request is killed before `execveat` syscall executing the requested program, killing of the tracee is delayed to the `execve` moment.

4.9. Sandbox server upon client death

The supervisor process monitors the UNIX socket through which the requests flow in. If the other end becomes read and write closed, the supervisor recognises it as a death of the client process and dies immediately. The PID 1 process is configured to die upon the supervisor process death, and the kernel kills all tracee processes when the PID 1 process dies. Therefore, all server processes die.

4.10. PID 1 process upon supervisor death

The PID 1 process configures kernel to kill it upon the supervisor process death. This is done using `prctl`'s option `PR_SET_PDEATHSIG`. However, if the supervisor dies before the kernel configures the PID 1 process to die, the PID 1 process will still be alive and waste the resources. To solve this, one could check if the `getppid()` returns the expected PID of the supervisor process. However, this will not work, since the PID 1 process is in a new PID namespace, and `getppid()` will always return 0. A reliable solution is to pass a pidfd file descriptor of the supervisor process and check if the supervisor process is dead by checking if the pidfd file descriptor became readable. This way, upon supervisor process death, the PID 1 process is either killed by the kernel or it detects the death of the supervisor process and kills itself.

4.11. Signals

Signals are another way the processes can communicate with each other. They have their nuances and have to be isolated as well.

4.11.1. Tracee signals

Tracee can signal only to the visible processes and those are limited by the PID namespace. However, it can also send signals to its process group that can span multiple PID namespaces. Figure 4.1 illustrates this situation. Therefore it is necessary to set new process group for the tracee processes. Furthermore, as a hardening, a new process group is set for the PID 1 process as well, in case the tracee takes control of it.

However, it is better to also set a new session id using `setsid` syscall instead of just the process group id to avoid vulnerabilities connected to the current controlling terminal [4].

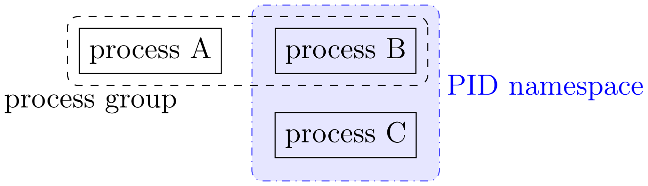

Figure 4.1: Process group can span multiple PID namespaces.

4.11.2. `SIGPIPE` in the supervisor process

Sending response to the client may generate `SIGPIPE` signal for the supervisor process if the client cancels the request approximately in the same moment. We have to ignore this signal in the supervisor process. However, this cannot be done using `SIG_IGN` because this signal disposition is not reset upon `execveat` system call and it had to be reset manually. As an alternative, it was chosen to install an empty signal handler for `SIGPIPE` so that the disposition of the signal handler is reset upon `execveat` in the tracee process automatically by the kernel.

4.11.3. Undefined Behavior Sanitizer

The code of the sandbox may be compiled with the Undefined Behavior Sanitizer (UBSan) enabled. UBSan installs signal handlers for `SIGBUS`, `SIGFPE` and `SIGSEGV` signals. This is problematic, because tracee could send these signals to the PID 1 process. For the init process in the PID namespace, the kernel only allows sending signals for which the init process (here the PID 1 process) has installed the signal handlers [33]. Therefore the PID 1 process resets signal dispositions of these handlers if the UBSan is used to prevent tracee from sending these signals to the PID 1 process.

4.12. Running as superuser

The sandbox is not safe to be run by the superuser. If this is needed, then you have to switch to some unprivileged user first. This is because many global system resources are still available, even after dropping the capabilities, e.g. rising privileges works. The check is done in the supervisor process at startup. To make it user namespace-proof it is checked if `/dev/null` is the null device and if the effective user id of the process equals the owner of the `/dev/null` file.

4.13. Performance optimizations

Everything that can be done is done in the supervisor process at startup, before handling requests e.g. creating cgroups, entering the network namespace, opening `/dev/null` as standard file descriptors. Sharing this work between requests ensures minimal overhead of handling the request i.e. it increases throughput (handled requests per second). Some of the optimizations are described in this section.

4.13.1. Seccomp filter of the PID 1 process

The seccomp filter of the PID 1 process is created and compiled in the supervisor process. Therefore it is done once instead of for every request.

4.13.2. Seccomp filter as file descriptor

The seccomp filter in a request is sent as a file descriptor. This avoids unnecessary copies of the seccomp filter contents in case the filter is large. Moreover, the gain is more evident if the same filter is used for subsequent requests.

4.13.3. Unsharing network, ipc, uts and time namespace

Unsharing of network, ipc, uts and time namespace is done in the supervisor process, only once, at startup. This avoids doing it for every request in the PID 1 process and has non-negligible impact on the performance (see Chapter 5).

4.14. Integration with Online Judge Platform

To integrate the new sandbox with the Online Judge Platform, a suite for each language was needed. The suite sandboxes the compiler if the language is compiled and sandboxes the runtime of the tested program. The following suites were implemented:

- C, C++, Pascal and Rust — fully compiled languages, the suite has to sandbox the compilation process and a fully compiled executable.
- Python, Bash — fully interpreted languages, the suite does not have a compilation stage, but requires sandboxing the interpreter when it runs the solution.

The Bash language is used only for testing due to its short start-up time.

Each of the compilation and run stages requires creating a root file system and a seccomp BPF filter. Root file system has to include the following bind mounts (due to dynamically linked executables):

- `/lib`
- `/lib64`
- `/usr/lib`
- `/usr/lib64`

Additionally, C, and C++ compilers require `/usr/bin` and `/usr/include`. Pascal compiler requires `/usr/bin` and `/tmp`. Rust compiler requires `/usr/bin`, `/tmp`, and on Debian `/proc`. Bash and Python require no additional bind mounts.

4.14.1. Interactive problems

Interactive problems require the tested program to communicate with the checker program i.e. the standard input of the tested program is the standard output of the checker program and the standard output of the tested program is the standard input of the checker program. Figure 4.2 illustrates this configuration. To accomplish this we need two pipes, one for each communication channel. However, the judge needs to know which process died first to provide a reasonable verdict of checking the tested program on the test.

To see why, lets consider the two examples. In the first one, the checker decides early that the tested program answered wrong, it terminates with a message ”Wrong answer”. Then, the pipe closes and the tested program may get terminated by `SIGPIPE` for trying to write to the closed pipe. If this happens, the tested program's abnormal death is caused by the checker exiting early. In this situation verdict ”Wrong answer” is the expected verdict. In the second example, an incorrect tested program terminates early and abnormally. In this case, the pipe closes after tested program's death and the checker sees the output of the tested program as incomplete and decides ”Wrong answer”. However, in this example an expected verdict would be ”Runtime error” because the tested program's death caused checker to decide ”Wrong answer”, therefore tested program's abnormal death takes precedence here. If we don't know who died first in such cases, we cannot reliably deduce the primary cause and therefore cannot decide what is more important, a tested program's abnormal death or the checker's verdict.

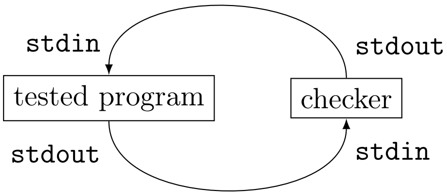

Figure 4.2: Schema of the communication between the tested program and the checker in an interactive problem.

To solve this, one could monitor the ends of the two pipes and see which end closes first. This is possible with e.g. `poll` syscall. However, it is prone to a race condition because the process monitoring the ends of the pipes may be scheduled after the checker and the tested program process and see them as if they died at the same moment. For example, the tested program process dies abnormally, then checker decides ”Wrong answer” and exits and only then the `poll` syscall returns reporting all ends of the pipes as closed without the information which closed first. To avoid this race condition and decide reliably 4 pipes are needed and a process that glues both pairs of pipes together and detects which end is closed first. Figure 4.3 illustrates this configuration. As long as, the third process holds open inner four ends of the pipe pairs, the tested program and the checker will not see their `stdin` and `stdout` as broken and will not proceed (to terminate, either normally or not). This way we can reliably detect who dies first and give a correct verdict in the scenarios where one's death causes the other's death. To efficiently pass messages in the third process, the `splice` syscall is used.

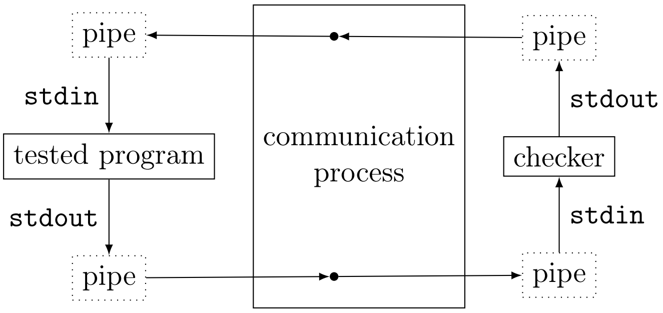

Figure 4.3: This is how communication between the tested program and the checker is implemented. Two pairs of pipes are used. This allows detection which process dies first — the tested program or the checker. Because the communication process does not close its ends of the pipes first, it can detect which process died first without causing the second to die because of a broken pipe. To efficiently pass messages in the communication process, the `splice` syscall is used.

4.14.2. Non-interactive problems

In the non-interactive problems, the semantics of the input is read-once and of the output is write-once. To achieve this without disallowing `dup`'ing, `close`'ing, `mmap`'ping, `pread`'ing etc. of the standard input and output file descriptors, we use pipes that are read-once and write-once. Input file is piped to `stdin` of the tested program, and the tested program's output is piped to the output file. Figure 4.4 illustrates this configuration. To efficiently pass messages between a pipe and a file the `splice` syscall is used.

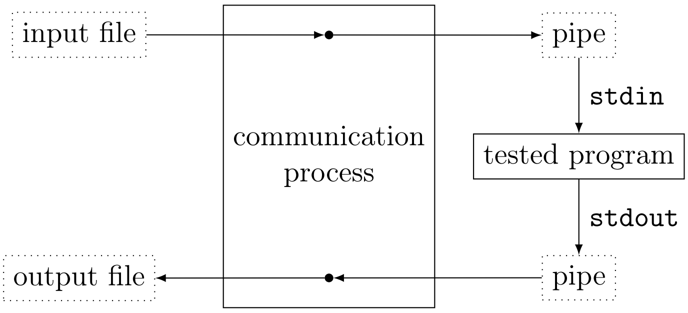

Figure 4.4: To provide the read-once semantics of the standard input and the write-once semantics of the standard output the pipes are used. The communication process passes contents of the file to the input pipe that outputs to the tested program's `stdin`. The tested program's `stdout` is piped to the output file. To efficiently pass data between files and pipes in the communication process, the `splice` syscall is used.

4.14.3. Conclusion

Apart from the above difficulties, the integration was rather easy. It required changing usage of the old sandbox to the new sandbox knobs. A lot of code was simplified along the way.

4.15. Testing and validation

To test and validate the new sandbox a comprehensive set of unit tests was developed. Tests check that namespaces are used appropriately, the interface is implemented correctly, the limits are enforced, runtime statistics are provided and are correct, no file descriptor leaks to the traced process, etc. Error reporting was also tested e.g. that unexpected supervisor death is reported, the supervisor terminates as soon as the socket connection with the client becomes broken etc. These tests ensured no regressions during the development and in the end eased the development process.

4.16. Challenges faced

There were many challenges during the development of the sandbox. First was to understand the semantics of the kernel's interfaces i.e. namespaces, cgroups and capabilities, and nuances in every one of them. One of the great achievements was to desist from using `ptrace` and all its complexity, while controlling a group of processes. This reduced the overhead and simplified things a lot. It was possible thanks to the Linux namespaces and cgroups.

Another challenge was to orchestrate everything to work together: resource limits, file descriptors, communication between processes, setting up namespaces and cgroups, dropping capabilities etc. It all has to be done in the right order and was often unobvious how to do it right.

The hardest of all was to debug very obscure errors happening during setup of the namespaces and cgroups. Often configuration failed with `EPERM` or `EINVAL` and it required figuring out what was wrong with just such vague information. For example, mounting cgroup2 file system is not allowed without unsharing cgroup namespace first. It also requires unsharing mount namespace and user namespace, but this is completely reasonable. In Sections 4.16.1 and 4.16.2 two examples are shown of the hard to debug errors.

4.16.1. `execveat` returns `EINVAL`

Executing a dynamically linked executable may fail with an error `EINVAL`. The executable needs the dynamic linker, so one reason of the error is the missing dynamic linker in the new root file system, on x86_64 it is at `/lib64/ld-linux-x86-64.so.2`. Another reason may be a missing shared library that usually resides in the directory `/usr/lib/`. It is important to bind mount all of these paths during creation of the root file system for the dynamic executables to work.

4.16.2. `execveat` returns `ENOENT`

Executing a file may fail with `ENOENT`, even though the file exists. This may happen because the file is a symbolic link and the destination does not exist, or if the symbolic link is recursive (refers to a symbolic link) and one of the intermediate files is non-existent in the new root file system. One of the solutions is to bind mount the executable without `AT_SYMLINK_NOFOLLOW`.

4.17. Conclusion

Despite challenges and complexity the sandbox was implemented and tested successfully. The usage of Linux namespaces provides isolation while cgroups and prlimits limit the resources. The sandbox is versatile enough to be used both as a sandbox for running a tested program as well as for running the compiler. The goal of optimizing the implementation for short-running programs was achieved with several optimizations.

Chapter 5

Performance Evaluation

All performance tests were made on a laptop with Intel i5-8250U processor and 16 GB of RAM. The purpose of this chapter is to verify the efficiency of the sandbox in the context of the online judge platform and briefly compare it with nsjail.

5.1. In the context of the online judge

For the testing we use problem Ptaszek from the XXI Polish Olimpiad in Informatics. The evaluation is done only on the test `pta9a`, because it is the test the solution runs the slowest on.

5.1.1. Compilation

Table 5.1 contains measured real time of the compilation of the solution. Table 5.2 contains measured CPU time of the same compilation. From the measurements we see slowdown in the sandbox of around 14% compared to time outside the sandbox. However the slowdown is caused solely by the seccomp BPF filters, as without them the compilation takes around the same time as outside the sandbox. Command used in testing: `/usr/bin/g++ -std=c++17 -static source.cc -o exe`. Compiler version: 13.2.1. It is worth noting that the compilation required around 109 000 syscalls, as reported by the `strace` tool [41] in a separate run.

Benchmark	Mean real time	Std. dev.	Std. err. on the mean	Slowdown
No sandbox	1.983 s	0.048 s (2.40%)	0.015 s (0.76%)	0.00%
In sandbox	2.267 s	0.035 s (1.52%)	0.011 s (0.48%)	14.32%
In sandbox without BPF	1.977 s	0.038 s (1.92%)	0.012 s (0.61%)	-0.30%

Table 5.1: Real time of compilation of the solution in different benchmarks. N = 10. First row presents time without sandbox, second with full sandbox and third with full sandbox without BPF filters.

5.1.2. The test program

Table 5.3 contains measured CPU time of the tested program. From the data it is clear that there is no meaningful overhead for running the tested program inside the sandbox.

Benchmark	Mean CPU time	Std. dev.	Std. err. on the mean	Slowdown
No sandbox	1.968 s	0.046 s (2.32%)	0.014 s (0.73%)	0.00%
In sandbox	2.249 s	0.031 s (1.40%)	0.010 s (0.44%)	14.28%
In sandbox without BPF	1.950 s	0.036 s (1.84%)	0.011 s (0.58%)	-0.91%

Table 5.2: CPU time of compilation of the solution in different benchmarks. N = 10. First row presents time without sandbox, second with full sandbox and third with full sandbox without BPF filters.

Benchmark	Mean CPU time	Std. dev.	Std. err. on the mean	Slowdown
No sandbox	0.608 s	0.005 s (0.89%)	0.002 s (0.28%)	0.00%
In sandbox	0.609 s	0.010 s (1.70%)	0.003 s (0.54%)	0.16%

Table 5.3: CPU time of the solution in different benchmarks. N = 10. First row presents time without sandbox, second with full sandbox.

5.2. Short-running programs

An example of a short-running program is `/bin/true` — we will use it in the benchmark. The metric we will use is the number of requests per second. Table 5.4 contains the measurements. While the slowdown is huge (more than twofold), it still allows for hundreds of runs per second and that was the goal of this thesis.

Benchmark	Mean requests per second	Std. dev.	Std. err. on the mean	Slowdown
No sandbox	1138.6	23.70 (2.08%)	7.50 (0.66%)	0.00%
In sandbox	472.9	7.36 (1.56%)	2.33 (0.49%)	58.47%
In nsjail	91.8	4.32 (4.70%)	1.36 (1.49%)	91.94%

Table 5.4: The statistics were collected from 10 runs, each run executed 300 requests and divided the request number by the time it took to complete all requests to calculate the requests per second statistic.

5.3. Comparison with nsjail

nsjail does not reuse the resources between the runs. Therefore the requests per second metric is drastically lower than for our sandbox. This is presented in table 5.4. The nsjail can handle around 5 times less requests per second than our solution. Moreover, our sandbox allows easy collection of the runtime statistics that nsjail is incapable of doing. Command used for benchmarking the nsjail: `/usr/bin/nsjail -q -Mo --chroot / --use_cgroupv2 -- /bin/true`.

5.4. Impact of some optimizations

From the Table 5.5 it is clear that unsharing namespaces once instead of for every request has positive impact on the performance. The most meaningful was the network namespace that if unshared for every request resulted in 19% performance degradation.

Benchmark	Mean requests per second	Std. dev.	Std. err. on the mean	Slowdown
Baseline	472.9	7.36 (1.56%)	2.33 (0.49%)	0.00%
New network namespace for each request	382.2	4.39 (1.15%)	1.39 (0.36%)	19.18%
New IPC namespace for each request	452.3	7.66 (1.69%)	2.42 (0.54%)	4.36%
New UTS namespace for each request	464.1	7.40 (1.59%)	2.34 (0.50%)	1.86%

Table 5.5: The statistics were collected from 10 runs, each run executed 300 requests and divided the request number by the time it took to complete all requests to calculate the requests per second statistic.

Chapter 6

Future work

Chapter 7

Conclusion

Bibliography

[1] David Beserra et al. "Performance Analysis of LXC for HPC Environments." In: *CISIS*. IEEE Computer Society, 2015, pp. 358–363. ISBN: 978-1-4799-8870-9. URL: `http://dblp.uni-trier.de/db/conf/cisis/cisis2015.html#BeserraMEBSF15`.

[2] Sander van der Burg and Eelco Dolstra. "Automating System Tests Using Declarative Virtual Machines". In: *2010 IEEE 21st International Symposium on Software Reliability Engineering*. 2010, pp. 181–190. DOI: `10.1109/ISSRE.2010.34`.

[3] Henrique Zanela Cochak et al. "RunC and Kata runtime using Docker: a network perspective comparison". In: *2021 IEEE Latin-American Conference on Communications (LATINCOM)*. IEEE. 2021, pp. 1–6.

[4] *CVE-2017-5226 – Bubblewrap escape*. URL: `https://security-tracker.debian.org/tracker/CVE-2017-5226` (visited on 2023-10-22).

[5] *CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition*. URL: `https://cwe.mitre.org/data/definitions/367.html` (visited on 2023-10-19).

[6] Flatpak. *Flatpak - the future of application distribution*. URL: `https://flatpak.org/` (visited on 2023-10-17).

[7] T Garfinkel. "Janus: A practical tool for application sandboxing". In: *http://www. cs. berkeley. edu/daw/janus* (2004).

[8] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. "Ostia: A Delegating Architecture for Secure System Call Interposition." In: *NDSS*. 2004.

[9] Google. *A light-weight process isolation tool, making use of Linux namespaces and seccomp-bpf syscall filters*. URL: `https://github.com/google/nsjail` (visited on 2023-10-17).

[10] Suman Jana, Donald E Porter, and Vitaly Shmatikov. "TxBox: Building secure, efficient sandboxes with system transactions". In: *2011 IEEE Symposium on Security and Privacy*. IEEE. 2011, pp. 329–344.

[11] Taesoo Kim and Nickolai Zeldovich. "Practical and effective sandboxing for non-root users". In: *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 2013, pp. 139–144.

[12] Rob Kolstad. "Infrastructure for contest task development". In: *Olympiads in Informatics* 3 (2009), pp. 38–59.

[13] Yanlin Li et al. "{MiniBox}: A {Two-Way} Sandbox for x86 Native Code". In: *2014 USENIX annual technical conference (USENIX ATC 14)*. 2014, pp. 409–420.

[14] *Low-level unprivileged sandboxing tool used by Flatpak and similar projects*. URL: `https://github.com/containers/bubblewrap` (visited on 2023-10-19).

[15] Redis Ltd. *Diagnosing latency issues: Latency generated by fork.* 2011-09-08. URL: https://redis.io/docs/reference/optimization/latency/#latency-generated-by-fork (visited on 2022-09-08).

[16] Krzysztof Małysa. *Sim project.* URL: https://github.com/varqox/sim (visited on 2023-03-15).

[17] Krzysztof Małysa. *Sip – a tool for preparing problem packages for the Sim platform.* URL: https://github.com/varqox/sip (visited on 2023-03-15).

[18] Martin Mareš. "Fairness of Time Constraints." In: *Olympiads in Informatics* 5 (2011), pp. 92–102.

[19] Martin Mareš. "Perspectives on grading systems". In: *Olympiads in Informatics* (2007), pp. 124–130.

[20] Martin Mareš and Bernard Blackham. "A New Contest Sandbox." In: *Olympiads in Informatics* 6 (2012), pp. 100–109. URL: https://ioi.te.lv/oi/pdf/INFOL094.pdf.

[21] Jonathan M McCune et al. "TrustVisor: Efficient TCB reduction and attestation". In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 143–158.

[22] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* 2014.239 (2014-03). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2600239.2600241.

[23] Bruce Merry. "Performance analysis of sandboxes for reactive tasks". In: *Olympiads in Informatics* 4 (2010), pp. 87–94.

[24] Bruce Merry. "Using a Linux security module for contest security". In: *Olympiads in Informatics* 3 (2009), pp. 67–73.

[25] netblue30/firejail. *Linux namespaces and seccomp-bpf sandbox.* URL: https://github.com/netblue30/firejail (visited on 2023-10-17).

[26] *Official website of Kernel Virtual Machine.* URL: https://www.linux-kvm.org/ (visited on 2022-11-23).

[27] *Official website of QEMU — A generic and open source machine emulator and virtualizer.* URL: https://www.qemu.org/ (visited on 2022-11-23).

[28] Oracle. *Official website of VirtualBox.* URL: https://www.virtualbox.org/ (visited on 2022-11-23).

[29] Vassilis Prevelakis and Diomidis Spinellis. "Sandboxing Applications." In: *Usenix annual technical conference, freenix track.* Citeseer. 2001, pp. 119–126.

[30] man-pages project. *execveat - execute program relative to a directory file descriptor.* URL: https://man7.org/linux/man-pages/man2/execveat.2.html (visited on 2023-10-20).

[31] man-pages project. *getrlimit, setrlimit, prlimit - get/set resource limits.* URL: https://man7.org/linux/man-pages/man2/prlimit.2.html (visited on 2023-10-18).

[32] man-pages project. *mount_namespaces - overview of Linux mount namespaces.* URL: https://man7.org/linux/man-pages/man7/mount_namespaces.7.html (visited on 2023-10-20).

[33] man-pages project. *pid_namespaces - overview of Linux PID namespaces.* URL: https://man7.org/linux/man-pages/man7/pid_namespaces.7.html (visited on 2022-11-13).

[34] man-pages project. *setuid - set user identity.* URL: https://man7.org/linux/man-pages/man2/setuid.2.html (visited on 2023-10-18).

[35] man-pages project. *unix - sockets for local interprocess communication*. URL: https://man7.org/linux/man-pages/man7/unix.7.html (visited on 2023-10-20).

[36] Niels Provos. "Improving Host Security with System Call Policies." In: *USENIX Security Symposium*. 2003, pp. 257–272.

[37] Inge Alexander Raknes, Bjørn Fjukstad, and Lars Ailo Bongo. "nsroot: Minimalist process isolation tool implemented with linux namespaces". In: *arXiv preprint arXiv:1609.03750* (2016).

[38] rootlesscontaine.rs. *Rootless Containers*. URL: https://rootlesscontaine.rs (visited on 2022-11-28).

[39] Giuseppe Scrivano. *Rootless containers with Podman and fuse-overlayfs*. 2019-06-04. URL: https://indico.cern.ch/event/757415/contributions/3421994/attachments/1855302/3047064/Podman_Rootless_Containers.pdf (visited on 2022-11-28).

[40] František Špaček, Radomír Sohlich, and Tomáš Dulík. "Docker as Platform for Assignments Evaluation". In: *Procedia Engineering* 100 (2015). 25th DAAAM International Symposium on Intelligent Manufacturing and Automation, 2014, pp. 1665–1671. ISSN: 1877-7058. DOI: https://doi.org/10.1016/j.proeng.2015.01.541. URL: https://www.sciencedirect.com/science/article/pii/S1877705815005688.

[41] *strace - trace system calls and signals*. URL: https://man7.org/linux/man-pages/man1/strace.1.html (visited on 2023-10-20).

[42] systemd. *systemd-nspawn — Spawn a command or OS in a light-weight container*. URL: https://www.freedesktop.org/software/systemd/man/systemd-nspawn.html (visited on 2022-11-28).

[43] Tocho Tochev and Tsvetan Bogdanov. "Validating the Security and Stability of the Grader for a Programming Contest System." In: *Olympiads in Informatics* 4 (2010), pp. 113–119.

[44] VMWare. *Official website of VMWare Workstation*. URL: https://www.vmware.com/products/workstation/ (visited on 2022-11-23).

[45] Bennet Yee et al. "Native client: A sandbox for portable, untrusted x86 native code". In: *Communications of the ACM* 53.1 (2010), pp. 91–99.