Thesis Statement

Thesis topic

(An/Two) approach(es) to develop and evaluate emulation of seL4 in Linux operating system.

Description of the topic

In this research project, we are going to find a way to emulate the seL4 kernel so that seL4 applications can run on Linux-based operating systems. The final product will be an emulation layer that allows seL4 applications to run directly on Linux-based operating systems without having to resort to hardware virtualization. This provides many advantages, such as enabling the use of Linux debugging and profiling tools for the emulated seL4 applications.

Motivation

- Providing an easy way for the seL4 application developers to do rapid prototyping in a Linux environment. The unmodified seL4 application source code can be targeted to the seL4 Linux emulation environment in addition to targeting the real seL4 environment.
- Enables seL4 applications to run on Linux, which enables us to:
    - Leverage Linux tools for debugging. Although it is possible to use hardware virtualization to debug seL4 applications such as by using Qemu, debuggers are often confused because the stack frames might be occupied by different code caused by context switching done by the seL4 kernel, which typically results in breakpoints unexpectedly triggered or missed and wrong values on the inspected variables.
    - Access to the rich inputs and outputs provided by the host's Linux system, such as files and networking, without having to develop a device-specific driver first.

Survey the literature

Existing solution

Qemu provides a hardware virtualization feature, allowing arbitrary operating systems to run safely in a host system without affecting the host system. Qemu also provides a debugging interface (GDB server). However, debugging under such interfaces is typically difficult because of the guest operating system's context switching. Hardware virtualizations will also unquestionably creates an overhead.

Related Work

- For the non-binary compatible approach:
    - Cygwin provides a method to run Unix applications on top of Windows natively. This is achieved by compiling Linux application source code under Cygwin with minimal modification. Cygwin provides a dynamic-link library as an API compatibility layer. Inspired by this idea, the first approach will provide the emulation at the seL4 syscall layer.
- For the binary compatible approach
    - Wine is a project which is a compatibility layer that allows applications targeted for Windows to run on Unix-like operating systems. To provide a Windows runtime system

compatibility layer, Wine also translates the Windows system calls into POSIX-compliant system calls, implements the Windows application binary interface (ABI) entirely in userspace, rather than a kernel module. Wine mostly mirrors the hierarchy to provide the services which are normally provided by the Windows kernel. Now those services are provided by the daemon called wineserver which implements basic Windows functionality, as well as integration with the X Windows System.

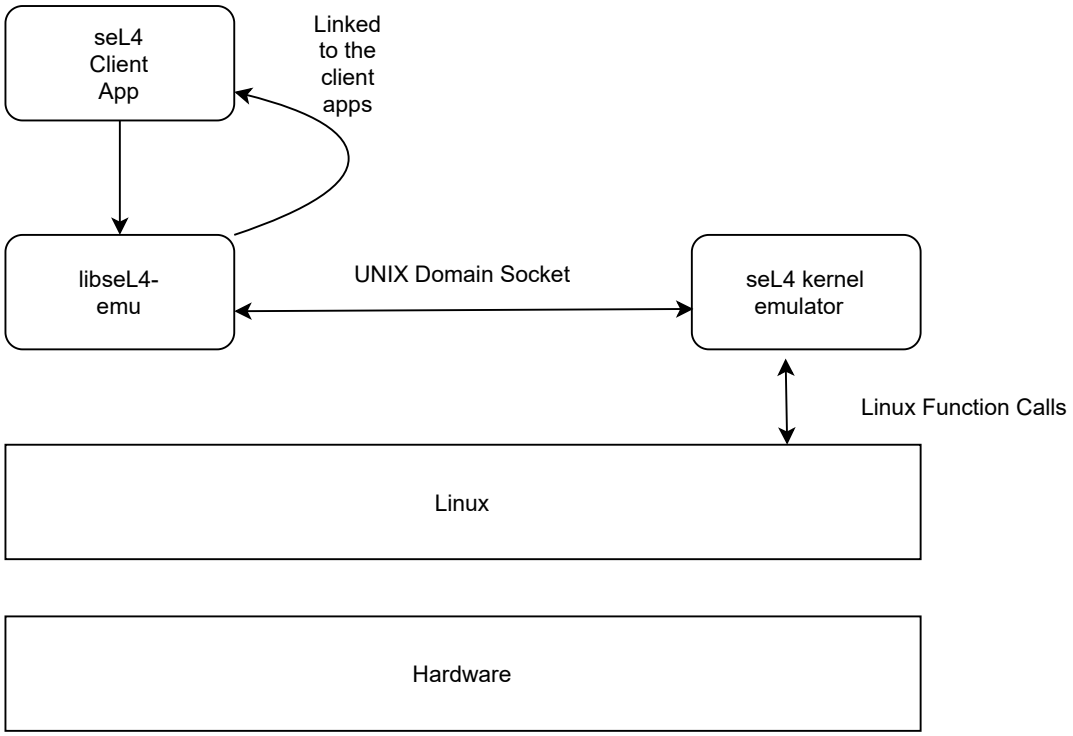
- The main point from studying those related projects is: it might be a good idea to target the seL4 emulation layer at the seL4 syscall layer, and let the seL4 applications on top of the seL4 syscall layer be able to run in Linux OS without noticing that it's no longer running on top of seL4 kernel.

Options of the implementation

To achieve the emulation functionalities the following approaches can be considered:

Approach 1:

- Providing an alternative seL4 syscall wrapper libraries that can be linked to seL4 client apps. Instead of invoking the real seL4 syscalls, this wrapper library will instead dispatch the syscall request to a server app that behaves like a seL4 kernel and provides seL4 syscall semantics.
- The model is as below:



- The above figure illustrates the main program flow of this approach. A seL4 client app can be linked with the libseL4 emulation layer called libseL4-emu and other helper libraries. Assuming that the seL4 client app exclusively uses the libseL4 library to interact with the seL4 kernel, the application will proceed to perform as expected even though it is running as a regular Linux process since the libseL4-emu provides the same semantics as the real libseL4 library running on the real seL4 kernel.
- The emulation of this approach uses a client-server model. The libseL4-emu on the seL4 client app side acts as a client. And the seL4 kernel emulator on the other side acts like a server. The client and the server communicates using UDS (Unix Domain Socket) and other IPC mechanisms provided by Linux. The kernel emulator provides the emulation of seL4 capability space, IPC, and scheduling context management.

Pros

- Relatively easy to implement.
- ISA independent for pure C seL4 app (except seL4 apps that does memory management).
- Good performance, since:
    - All user-mode codes are run as-is without any sort of virtualization or hardware emulation.
    - seL4 syscalls will only invoke regular Unix socket communications, which is also very performant.
    - No emulation is involved in the seL4 kernel emulator. The seL4 kernel emulator is just a regular Linux application implementing the seL4 semantics.

Cons

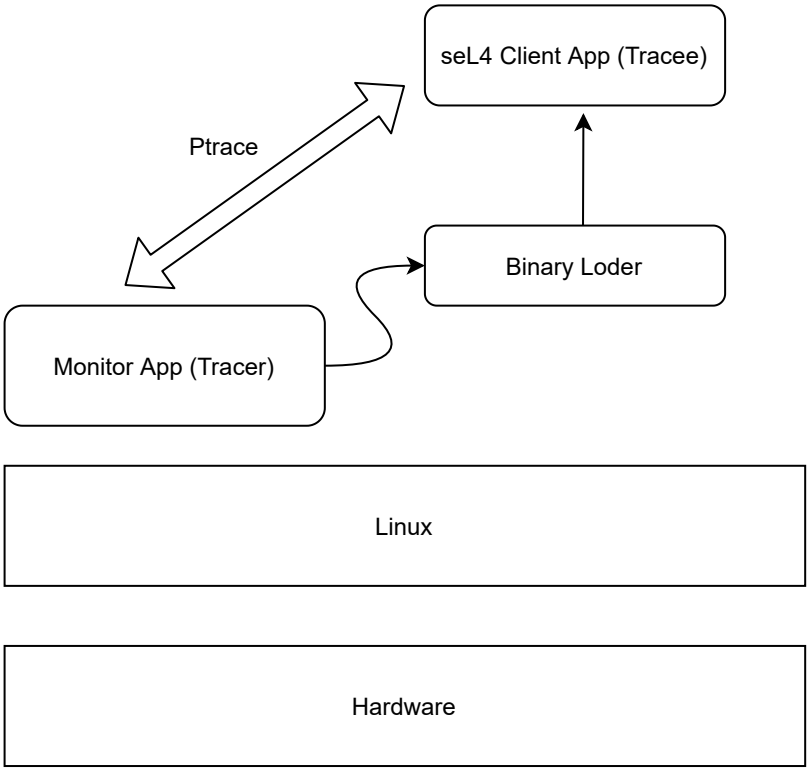
- No binary compatibility: seL4 applications that wish to be emulated have to be recompiled from their source code.
- Not compatible with seL4 applications that does not use libseL4 (e.g. those that do the seL4 system calls directly).

Approach 2 (Might consider this approach to be a future work if only manage to do approach one)

The motivation for implementing this approach is to achieve one-to-one binary compatibility. Ideally, the emulation will work even without recompiling the seL4 client app's source code.

To achieve this goal, we can leverage the capability of `ptrace()` syscall in Linux. It allows us to mutate syscall arguments, return values, or intercept the syscall. Also, The `ptrace()` syscall introduced in the Linux has a feature that it can stop at the entry of every syscall and let the tracer decide how to service that syscall before allowing the client app to continue by using the `PTRACE_SYSEMU` parameter. With those features, we can emulate the seL4's semantics on Linux efficiently, completely from user mode.

- In this binary compatible approach the following model is used:



- In the above model we provide a monitor app that is able to load and run seL4 client apps as well as monitor and intercept syscalls of the client apps via `ptrace()`. This can be done by following steps:
    - Fork a child process inside the monitor app which then invokes a custom ELF loader library to parse and load the seL4 ELF file to the memory.
    - In the next step, the loader will set up the appropriate memory regions which can be done by `mmap()` syscall in Linux.
    - After everything is set up, the loader will block and notify the monitor app to perform the ptrace.
    - Once the monitor app finished `ptrace`, it will ask the tracee to continue execution, which will begin from the entry point specified in the seL4 client app's ELF file.
- Now the monitor app and the seL4 client app can be defined as a tracer and tracee relationship. Everytime the tracee performs a syscall, Linux kernel will suspend the execution of the tracee and notify the tracer, so that the tracer can then do the appropriate actions to simulate the semantics of a real seL4 kernel.

Pros

- Binary compatibility. Emulated seL4 applications can be run as-is directly on a Linux system without recompilation.

Cons

- IPC heavy seL4 applications will be slower due to the performance overhead involved in trapping syscalls by `ptrace()`.
- Have to provide our debugging interface as most Linux debuggers also utilise `ptrace()`. And `ptrace()` only allows one tracee to be attached to one tracer at any time.