School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

seL4 API/ABI emulation approaches in Linux

by

Jiawei Gao

Thesis submitted as a requirement for the degree of

Master's of Information Technology

Submitted: April 2021 Student ID: z5242283

Supervisor: A/Prof. Gernot Heiser Topic ID:

Abstract

This thesis examines different methodologies and designs that aim to realize the idea of emulating seL4 in Linux. This report will mainly focus on reviewing the related projects. The first three are QEMU, Cygwin, and Wine, which provide solutions to run either foreign operating systems or programs targeting the foreign operating systems on the host operating system. And the last UML project provides an idea of porting kernel to the userspace. This report also proposes ideas of providing emulation at the sel4 syscall API layer and at the seL4 kernel ABI layer. They all share similar goals but tackle the problem from different perspectives. An analysis will be provided to reveal similarities and differences between them the techniques. Evaluation approaches and applications will also be proposed at the end to demonstrate the underlying technical solutions and outcomes of this project.

Acknowledgements

The work of this thesis has been inspired by Prof. Gernot Heiser and Axel Heider who encourage me to explore the scope of this project leading to an enhancement in research skills and understanding of emulation technologies. Further support and encouragement have been given by James Nakoda Nugraha who provides me the recommendation, advice, and information about Linux. At last, I would like to thank you, my families and friends, who always support me.

Abbreviations

ABI Application Binary Interface

API Application Programming Interface

ASLR Address Space Layout Randomization

DLL Dynamic-link library

IPC Inter Process Communication

KVM Kernel-based Virtual Machine

OS Operating System

PID Process Identifier

POSIX Portable Operating System Interface

RPC Remote Procedure Call

TCB Trusted Computing Base

UML User-mode Linux

Contents

List of Figures

List of Tables

Chapter 1

Introduction

seL4 provides a very secure environment to run any untrusted applications as separate components on top of it. What's more, it maintains a good performance as well as ensures the strong isolation of the applications and the kernel to keep the entire system robust and safe. However, developing seL4 systems is not trivial because of the developing ecosystems of seL4. Because Linux has a much larger and more complete ecosystem than seL4 does, therefore, in this project, we are going to explore some approaches to emulate the seL4 user-mode applications and components which provide different functionalities in the Linux environment. The outcome of this project will let us leverage helpful tools in Linux to make developing seL4 systems much easier and faster. Sometime in the future, we will port those systems back to seL4 without modifying the source code. Hence, we are can fully leverage the advantages that seL4 and Linux provide us.

Chapter 2 explains the related projects that have been done before, followed by a detailed explanation of each related project. Chapter 3 states the rough design models and implementation ideas of this project. Chapter 4 explains the success criteria defined for this project and the methods that will be used for the evaluation. Chapter 5 states the plan for the future research project and the timeline for the next semester.

Chapter 2

Background

seL4 is a formally verified and the fastest microkernel in the world with lots of awesome features ([KEH+09]). The componentized system architecture that seL4 implements enforce the isolation between those untrusted components and other trusted components running on top of the kernel. seL4 was born for safety. The fine-grained capability-based access control model carefully manages the access to the hardware resources from the software components.

The secure and well design of the seL4 microkernel together with the formal verification ensure the kernel itself is robust and is the ideal foundation to build a secure system on top of it ([KAE+10]). What's more, seL4 is not only secure but also fast. With its performant IPC mechanisms and most advanced mixed critical real-time systems ([LH14]), the seL4 is capable to handle a wide range of real-world scenarios. However, seL4 is relatively young comparing to other mainstream kernels, such as Linux. And its ecosystem is in growing. For those seL4 developers, there are limited tools that can be used while developing seL4 systems. However, Linux provides us a powerful developing environment with lots of useful tools. Therefore, the main motivation of this project is to explore some ways to leverage the Linux features in terms of development to make developing seL4 systems much easier and faster.

2.1 Challenge

Developing applications targeted for seL4 in Linux is challenging because seL4 as a microkernel has different semantics and OS model based on seL4 comparing with Linux (In figure 2.1).

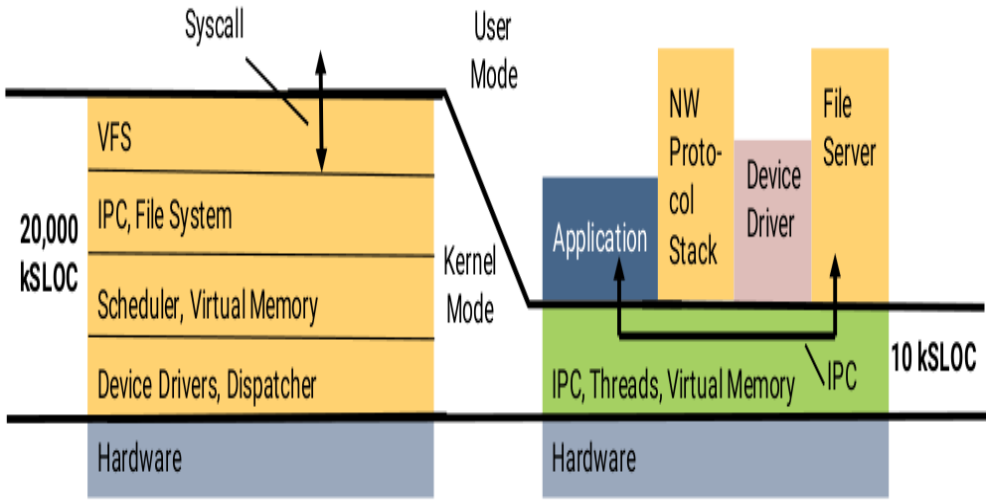

Figure 2.1: Linux based OS model vs seL4 based OS model.

Linux as a monolithic kernel, the kernel itself provides a wide range of OS services. The drivers, OS components are implemented inside of the kernel. All those OS services execute in kernel mode. While in seL4, to guarantee the strong isolation of different OS components, seL4 itself is just a thin wrapper of the underlying hardware providing minimal critical services of the hardware resources. And the OS services are implemented at the user level as separate components. Such design minimizes the TCB(Trusted Computing Base).

In a monolithic kernel, the user-level applications request the OS services through the system call interfaces that are exported to the user level. A system call will cause a context switch from the user mode to the kernel mode. And the kernel will serve the OS requests by itself. For example, an I/O request from the user level will trigger the

system call and the kernel will dispatch such request to the corresponding handler. In Linux, this might be a handler in the VFS layer. Then the VFS layer handler will find the particular File system handler and finally, it will invoke the driver to perform the I/O. After the system call getting triggered, everything happens in the kernel mode until the requests have been served. However, in the seL4 based OS model, it works differently. First, to clarify the term, we are going to call anything that runs in seL4 user space the seL4 applications. Also, we distinguish those applications into client applications that request OS services and those server applications that serve the OS services. And the rest of the article, we will use the seL4 application as the term to make the description easier. When the seL4 client applications request OS services, it will invoke IPC mechanisms provided by the seL4 kernel. The seL4 IPC is a protected remote procedure call that can invoke the function in another seL4 application, which usually is the seL4 server applications securely. The IPC message passing is the main system call in the seL4, it triggers the context switch to the kernel mode, and the sel4 microkernel will then transport the input to the authorized seL4 server application via an exported entry point. Those requests are served by the seL4 serve application at the user level. The seL4 client applications and seL4 server applications are isolated components, which ensures the security of the whole system.

Since, the main problem here is that seL4 and Linux have different semantics and program flow of serving OS services, the final goal of this project will be providing methods for developing seL4 user-level systems which can either run on Linux or seL4. In other words, we need to provide compatibility between seL4 applications and Linux.

2.2 Related Work

2.2.1 QEMU

One way to achieve compatibility is to leverage hardware emulation or virtualization technology ([Wik21b]). QEMU provides us both. It's a machine emulator supporting a wide range of ISA as well as a virtualizer providing us the hardware virtualization

feature if the guest's ISA is the same as the host's ISA. With those features, we are capable of running the seL4 microkernel on top of the Linux. And from the seL4 user-level applications' view, they are still running on top of the seL4 microkernel. And the QEMU provides the compatibility between the seL4 microkernel and the underlying Linux kernel.

QEMU uses portable dynamic translation to emulate the full system including processors and peripherals. By leveraging the advantages of hardware extensions such as Intel VT-X, QEMU can be used with KVM to run a virtualized guest OS in near-native speed. Moreover, the QEMU uses a dynamic binary translation approach, which means it's binary compatible. The way it works is QEMU will convert the guest instruction set into the host instruction set, then it will split instructions into fewer simpler instructions and use the dynamic code generator to assemble instructions into functions.

Although QEMU is an efficient dynamic translator ([QEM]), which makes QEMU significantly faster than other emulators, the overhead of doing that can't be ignored. On the other hand, QEMU provides its debugging interfaces which can be used to debug the guest system. However, the design goal of QEMU focuses on low-level machine emulation. It can be useful for inspecting each instruction's execution of the CPU, but it's difficult to use from a high-level seL4 developers' perspective. Because the debugger is not OS aware, the guest OS's context switch might confuse the debugger.

2.2.2 Cygwin

The Cygwin is a compatibility layer that allows UNIX-like applications to run on top of Windows ([Wik21a]). This is achieved by introducing a DLL called cygwin1.dll which acts as an emulation layer providing substantial POSIX system call functionalities providing a Linux look and feel. While Cygwin uses newlibc as its C library. With Cygwin, users can access several standard UNIX utilities such as bash, etc.

Cygwin began development in 1995 at Cygnus Solutions. In the project, developers provided interfaces called Cygwin API to add the missing UNIX-like functionalities in

Win32 API, such as fork, signals, select, etc ([Cyg]). Cygwin will not magically make any UNIX-like application's binary code runnable on top Windows directly. To make it work, the source code of the application is required. The source code can be compiled under the Cygwin and linked with the shared library which implements the POSIX system call semantics using the Win32 APIs and native NT APIs. After executing the application, the Cygwin DLL will be loaded into the application's test region so that it has full access to the whole process. Next, shared memory is created containing the instances of the resources that the shared library can access. Therefore, the OS resources such as file descriptors can be tracked. Besides such shared memory regions, each process also has its resource bookkeeping structures, such as signal masks, PID, etc.

Cygwin has several nice designs to implement the POSIX features in Windows. For example, it handles signals by starting a separate thread from the library for only signal handling purposes. This thread waits for the Windows event used to pass signals to the process, and scan through its signal bitmask to handle the signals appropriately. While since such thread resides in the same address space as the executing program, the signal sending function for sending a signal to other processes is wrapped with a mutex, and the signal sending function which sends the signal to itself is wrapped by separate semaphore or event pair to avoid them being interrupted.

Another example is the implementation of sockets. It's mapped on top of the Winsock which is implemented as Berkeley sockets by Microsoft with lots of modification. Since UNIX domain socket is not provided in Windows, Cygwin implements it with the local IPv4 as the address family. Besides, Cygwin provides the Winsock initialization on the fly, as the Winsock requires to be initialized before the socket function is called. For implementing the POSIX select, Cygwin implements the polling of file handles besides socket type handles by sorting the file descriptors into different types and creating a thread for each type of the file descriptors present to poll those file descriptors with Win32 API. Such a design is because the Winsock only works on socket type file descriptors. Although Cygwin implements lots of POSIX features, not all of them mapped well into Windows due to the huge difference in terms of semantics and underlying

design between UNIX-like OS and Windows. For example, in UNIX-like systems, the $fork()$ system call provides a way to create a child process that copies the parent's address space. While there are no appropriate process creation functions in Windows which can be mapped on top of it.

On the other hand, implementing $fork()$ semantics in Windows requiring to copy all of the executable binary and all the DLLs loaded statically or dynamically to be identical as to when the parent process has started or loaded a DLL. This can be problematic as Windows allows the binaries to be renamed to even removed to the recycle bin while the binary is executing, which means they can reside in a different directory or have a different file name. While to allow an executing process to fork requires access to the binary file via their original filenames. The solution to this problem is that Cygwin will try to create a private directory that contains the hardlink to the original files and remove it when no process is using it. When the parent process wants to fork a child process, it will first initialize a space in Cygwin process table for the child and create a suspended child process using Win32 CreateProcess system call.

After that, the parent process will use setjump to save the context and set a pointer to the current context in the Cygwin shared memory, and fill the child process's .data and .bss section with its own address space. Next, the parent process will block on a mutex and the child process will run and use the longjump to jump to the saved jump buffer. Then child process will release the mutex that the parent is blocked on and waits for another mutex. The parent process will copy its stack and heap into the child process space then release the mutex that the child is blocking on and return from the fork call. Finally, the child process will wake up and recreates any memory-mapped areas that passed to it and return from the fork call. However, such implementation is not perfectly reliable as in Windows, Windows implements the ASLR starting from Vista, which means the stack, heap, text, and other regions may be placed in different places in each process. This behavior interferes with the POSIX fork's semantic that is the child process has the same address space as the parent process. In that case, Cygwin will try to compensate the movable memory regions at the wrong place but can't do anything with those unmovable regions such as the memory heap.

In summary, adding compatibility at the API layer can solve most of the problems when attempting to run applications from foreign systems. While the downside is that, apparently this will only work when the user of Cygwin can obtain the source code of the application. Meanwhile, it's very difficult to implement every POSIX system call in Windows correctly due to the huge difference between the internal design and the semantics.

Even though Cygwin is used to make UNIX-like applications compatible with the Windows environment, we can still leverage the nice idea that Cygwin uses. To emulate seL4 in Linux we can link applications to a specific library that remaps the system calls with the underlying host's system calls.

2.2.3 Wine

As mentioned before Cygwin is an API-compatible solution for providing compatibility between Windows and Linux. The next related project we are going to introduce is called Wine which is ABI compatible. Wine is a project that provides the compatibility layer to run Windows applications in UNIX-like operating systems ([Wik21c]). However, Wine doesn't emulate the internal Windows logic like a virtual machine or an emulator. Instead, it directly translates the Windows ABI into POSIX-compliant calls. Besides, Wine also provides various Windows services through Wineserver as well as other Windows components. In Wine's architecture, it implements Windows's ABI entirely in the user space, rather than a kernel module.

A system call from a Windows application usually invokes a particular DLL library, which in turn invokes the user-mode GDI/user32 libraries, and then finally invokes the system calls via Win32 subsystem. However, since the architecture of Windows OS is a hybrid model of monolithic kernel combined with the microkernel, some OS services run as separate processes, so applications need to invoke RPCs to communicate the user-mode services. In that case, this is somehow similar to how seL4 applications request OS services. Although Wine implements the Wineserver to provide services that are provided by the Windows kernel, as well as other OS functionalities, it's impossible

to implement all the aspects of the Windows kernel as well as to use native Windows drivers due to the internal architecture of Wine.

With a similar idea as the Wine project, we can target the emulation layer of seL4 applications at the kernel ABI layer to achieve the binary compatibility as Wine does.

2.2.4 User-mode Linux

UML is an old project used to port a Linux kernel to the userspace. This can be helpful to the system developers as it provides a nice way to develop and debug the kernel as well as to make several interesting applications possible for Linux ([Dik06]).

In UML's architecture, it treats Linux as a platform to which the kernel can be ported. All the implementation fully leverages the Linux system calls without any modification of the host kernel. And all the user-level code can run natively on the processors without any instruction emulation overhead. The implementation creates a separate thread that uses the *ptrace*() system call to trace all the other threads running in UML.

System Call

Since the transition between user mode and the kernel mode is controlled by the tracing thread, it will intercept all the signals and system calls that the running process issues and then transit from the user mode to the kernel mode and continue executing the particular process without tracing it. The distinguishment of the user mode and the kernel mode is whether the process is being traced by the tracing thread. After changing the register which contains the syscall number into the syscall number for *getpid*() and restoring a previously saved thread registers state, the tracing thread will invoke the syscall handler to accomplish the syscall request, then finally return the process to the user mode.

Trap and Fault

UML not only virtualizes system calls but also implements several other Linux OS services and functionalities. For example, to handle the processor traps or faults, it implements those with signals and installs handlers for each. Once the tracing thread captures the signals that the process received, it will switch to the kernel mode and continue executing the process in the handler.

For the memory fault, UML implements with the $SIGSEGV$ and invokes the handler to figure out whether the fault was because of legal access to an unmapped page or illegal memory access.

Interrupt

UML also implements external interrupts and timer interrupts using $SIGIO$ and $SIGALARM$ or $SIGVTALARM$ depending on whether the interrupted process is idling or not. For the external interrupts, it uses *select* to check which file descriptor has received input then invokes the IRQ handler. For the timer interrupt, UML also treats it similarly to treating the external interrupt and invokes the particular IRQ handler.

Scheduling

For process scheduling, UML implements scheduling processes by stooping an outgoing one and running an ingoing one as each process has its thread belongs to it. Since each process has its own address space, UML also manages the page mapping of each process as well as the $SIGIO$ queued in each process.

Virtual memory

In UML, the kernel and process's virtual memory is implemented as a physical memory-sized file mapping into their address spaces. However, some regions in the address space

will be reserved by placing kernel code and data which is unusable.

Host filesystem access

UML also implements a virtual file system called $hostfs$ which translates the directly into the functions in the host's *libc*. Hence, it provides direct access to the host filesystem.

UML was a nice project showing a novel way to use Linux interfaces to implement itself, also it demonstrated that porting a kernel in the userspace can be beneficial to both kernel and application development.

From this project's perspective, UML gives us the hint of how to leverage *ptrace* to intercept the system call from the process being traced as well as how to simulate the kernel in the userspace.

Chapter 3

Methods

At this stage, we are going to purpose a software development approach to tackle the problem. Running seL4 applications in Linux requires introducing an extra compatibility layer between the application and the Linux OS. Hence we are going to purpose two approaches to achieve this. The API layer emulation approach and the ABI layer emulation approach.

3.1 API Layer Emulation Approach

In the original seL4 system, a userland seL4 client application can request OS services from another seL4 server application which also runs in the userland through IPC mechanisms provided by the seL4 kernel. The seL4 system call library provides those IPC interfaces.

However, if in our case those seL4 userland applications will run on top of Linux OS, and the semantics of system calls of Linux and seL4 are totally different. Directly using the original seL4 system call libraries will not work. Hence, we will provide a custom system call library that exports the same interfaces as the original seL4 system call libraries to seL4 userland applications, but instead of invoking the seL4 IPC directly, we will use Linux IPC mechanisms as a replacement.

3.1.1 Design Model

Figure 3.1 shows the model of the API layer emulation approach.

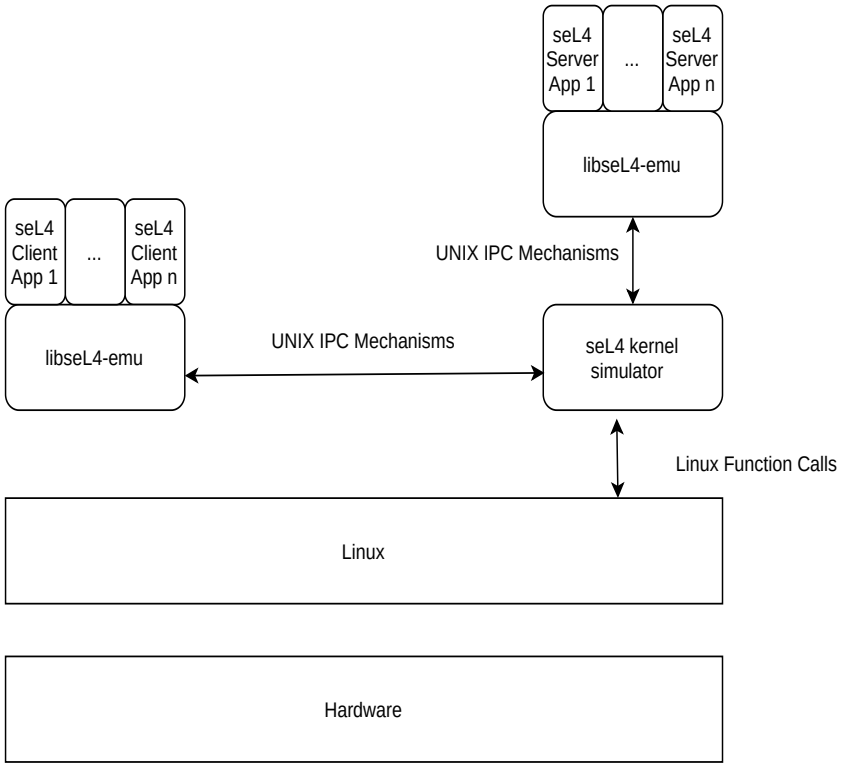

Figure 3.1: The model of API layer emulation approach

In general we need to develop two components in this model.

1. An seL4 emulation library
2. An seL4 kernel simulator

The seL4 emulation library is a compatibility layer named libseL4-emu in the figure which can be linked to other seL4 applications. Therefore, the seL4 applications will invoke the emulated seL4 system calls instead of the real original seL4 system calls.

And the libseL4-emu library will dispatch those system call requests to the seL4 kernel simulator.

The seL4 kernel simulator runs at the user level on top of Linux. Hence the dispatching processes can be implemented using Linux IPC mechanisms, such as UDS(Unix Domain Socket) or shared memory. The seL4 kernel simulator mainly provides the real seL4 semantics and management of capability space, IPC mechanisms, scheduling context management, etc.

In this model, all the seL4 applications and the seL4 kernel simulator are just regular Linux processes running at the user level. And the seL4 applications act as clients, communicating with the server which is our seL4 kernel simulator.

3.1.2 Advantages and Disadvantages

Providing an API layer emulation approach has several advantages. First of all, this approach is ISA independent. In the model, all the seL4 applications and the seL4 kernel simulator all run at the user level, and the seL4 system call emulation layer is at a high layer providing the interface. From the perspective of seL4 applications, everything below the system call emulation library is abstracted away. It will never notice that it is no longer running on top of the real seL4 kernel. And since simulating seL4 system calls and seL4 kernel use functions provided by Linux. The real interactions with hardware are abstracted away by Linux. Therefore, using this approach only focuses on implementing the correct semantics of the real seL4 kernel.

Secondly, in the API layer approach, the implementation of simulating the seL4 IPC mechanisms uses Linux IPC mechanisms which is also very performant. Ideally, we won't introduce too much overhead. Moreover, since everything runs at the user level including the seL4 kernel, no hardware-level virtualization or emulation is involved at all. Meanwhile, regarding the complexity of implementation, this approach is relatively easy to implement.

However, the main disadvantage of the API layer emulation is that it is not binary

compatible. This is because the emulation will work only if the seL4 applications are linked with the seL4 system call emulation library. Hence, we have to obtain the source code of the application first and recompile the source with the emulation library every time we want to use the emulation.

3.2 ABI Layer Emulation Approach

In the former approach, we have introduced the API layer emulation which is not binary compatible, while in this approach we are going purpose a method to achieve binary compatibility.

3.2.1 Design Model

Figure 3.2 shows the model of the ABI layer emulation approach.

In the figure, mainly we have two components:

1. An seL4 application monitor
2. An seL4 application binary loader

In the rest of the article, we will call the seL4 application monitor as the monitor, and the seL4 application binary loader as the binary loader for simplicity. The main challenge in this approach is that since the seL4 application is an unmodified binary code, hence it will invoke the real seL4 system calls. In that case, we have to redirect the system calls and observe what is the seL4 application requesting then serve the system call by ourselves according to the real seL4 semantics. To achieve this goal, we are going to use the *ptrace*() system call in Linux.

ptrace() allows a tracer process to observe and control the execution of the other process. The "tracee" process can be attached to the "tracer" process, so that the

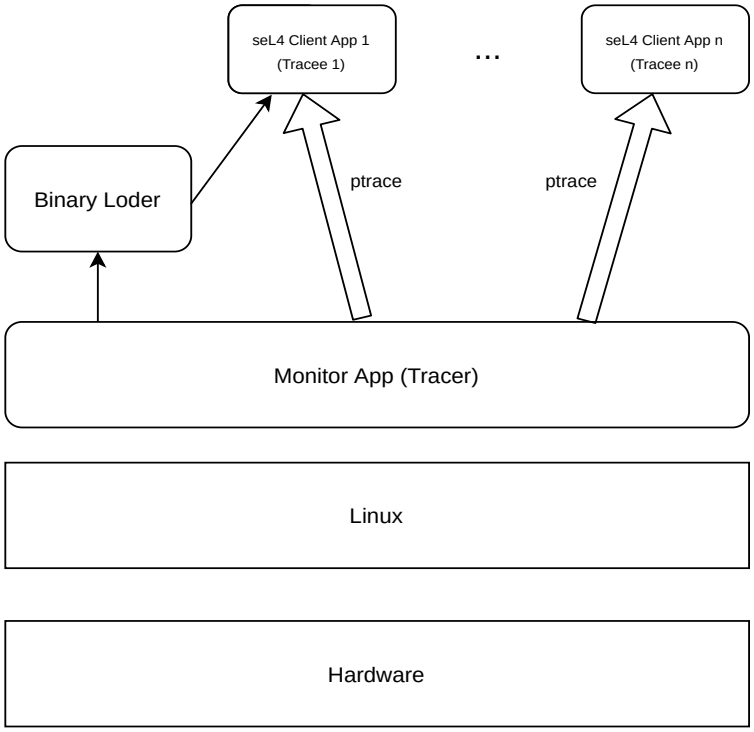

Figure 3.2: The model of ABI layer emulation approach

"tracer" can modify the "tracee" process's memory, registers, etc. Also, the *ptarce*() semantics allow us to establish a one-to-many relationship, which means we can have several "tracees", but can only have one "tracer". In this approach, we can leverage such a feature to intercept the system call invocation from the seL4 applications.

In the model described above, the monitor acts as our "tracer", and the seL4 applications are our "tracees". The monitor is also a seL4 kernel simulator providing the same seL4 semantics and kernel mechanisms as the real seL4 kernel does. The emulation starts from the monitor forks a binary loader process that can load the seL4 application's ELF file into the memory and set up the appropriate memory regions for it. Next, the binary loader will block and notify the monitor that the initialization has finished. The monitor can use *ptrace*() to control the seL4 application. Finally, the seL4 application can continue executing after the *ptrace*() process finishes. Now, every

time the seL4 application invokes a seL4 system call, the monitor will be notified and suspend the execution of the seL4 application, then the monitor will inspect the system calls requests and do appropriate actions to serve them.

3.2.2 Advantages and Disadvantages

The main advantage of this approach is that the kernel ABI approach is binary compatible. As long as, we obtain the binary code of the seL4 applications that we would like to emulate, no recompilation is required. However, using the ABI layer emulation can introduce some overhead. This is because every time the seL4 application invokes a seL4 system call. The monitor will use *ptrace*() to intercept it and modify the arguments, return value if necessary, and the *ptrace*() itself is a system call, which requires trapping into the kernel. The trade-off here is leading to a penalty of the performance. Besides, *ptrace*() is usually used to implement the Linux debuggers, and it only allows on "tracer" to exist at any time. Therefore, we have to provide our own debugging interfaces in some way, which can introduce implementation complexity.

3.3 Comparison Between API and ABI Layer Emulation

Table 3.1 listed the trade-offs of implementing the API and the ABI layer emulation approaches. The API layer approach is relatively easier to implement as we don't need to develop the binary loader and the debugging interfaces. The API layer emulation can fully leverage the native Linux debugging tools such as GDB, LLDB, etc. In terms of portability, the API layer approach is ISA compatible as all the implementation is at the seL4 system call library level and doesn't involve the ISA dependent code. However, the API layer approach doesn't achieve binary compatibility as it requires recompiling the source code to work, while the ABI layer emulation approach is fully binary as we intercept the system calls using *ptrace*().

Last but not least, in terms of the performance, the API layer approach should be

better than the ABI layer approach due to the internal implementation. As mentioned before, intercepting system calls and modifying the system calls' arguments or return values require extra context switches to the Linux kernel, which introduces overhead.

Table 3.1: Comparison between API and ABI layer emulation

Approach	Implementation Complexity	Potability	Binary Compatibility	Debugging Support	Kernel IPC Performance
Syscall Library API emulation	Intermediate	Good, as the solution is ISA potable	No	Can use native Linux debugging tools	First
Kernel ABI emulation	Hard	Not as good as the API layer emulation	Yes	Extra debugging interface required	Second

Chapter 4

Evaluation

In this chapter, we will purpose some evaluation approaches regarding to both API layer emulation and ABI layer emulation approaches. Despite the fact that the real implementation and the benchmark haven't been finished yet we can still set up some criteria in terms of the implementation in both a qualitative aspect and a quantitative aspect.

4.1 Qualitative Evaluation

One way to evaluate whether the implementation of the emulation approaches is successful or not is to test the emulation framework with some existing seL4 applications. In this project, we are going to use applications from the following source:

- seL4 tutorials
- seL4 test suit
- SOS(simple operating system)

The seL4 tutorials contain several trivial seL4 applications used for demonstration and education purposes.

Those applications mainly focus on demonstrating seL4 features and API usage including printing "Hello World", simple IPC message passing, setting up runnable threads in seL4 userland, etc. Those applications can be used to use to test each of the emulated seL4 features separately.

The seL4 test suit contains several unit tests that focus on low-level kernel APIs. As those tests were used to prove the correctness of the implementation of the real seL4 kernel and some seL4 helper libraries. Therefore, we can use those tests to evaluate the implementation of the seL4 kernel simulator. However, since the seL4 kernel simulator fully running in the Linux userspace, it doesn't have the access to the hardware and performs privilege options. We don't expect the kernel simulator can pass all of the tests. We will focus more on testing critical functionalities, such as the IPC mechanisms, cspace management. vspace management and scheduling, etc, and skip those tests that are not quite important to the emulation project or those are complex or even impossible to emulate correctly such as cache tests, multicore tests, etc.

The SOS is a simple multitasking operating system running on the seL4 kernel which has an interactive shell, and several subsystems including I/O, memory management, process management, etc. Since it has a relatively complex structure and more comprehensive functionalities implemented with seL4 semantics. Hence, emulating it using the emulation approach can be a nice way to both verify and demonstrate the emulation approaches function well.

4.2 Quantitative Evaluation

From the qualitative perspective, we are going to evaluate the two approaches via microbenchmark and macrobenchmark. Table 4.1 is the expected quantitative benchmarking outcome.

Table 4.1: Expected success criteria

	syscall API emulation (A1)	kernel ABI emulation (A2)
On the emulated hardware by QEMU (B1)	A1 should be significantly faster than B1 because no hardware emulation is involved	A2 should be significantly faster than B1 but slower than A1 due to syscall interception
Native hardware running seL4 (B2)	A1 should be less than an OoM slower than B2 because seL4 is highly optimized	A2 should be significantly slower than B2 due to syscall interception

4.2.1 Microbenchmark

The microbenchmark mainly focuses on evaluating the performance of each emulated system call. We can measure the number of system calls that the seL4 application invokes per second. The higher the result, the better the performance. Besides, to make the result more representative, we should measure the result several times and make a statistical analysis.

The ideal expectation is listed in the table 4.1. In general, we expect the API emulation approach will not introduce too much overhead, and the result should be slower compared to the result of running the same application on the native seL4 kernel. In the worst case, the overhead upper bound shouldn't exceed an order of magnitude. While comparing to the result of using ABI layer emulation, hardware virtualization, or emulation approach, the API layer emulation should be faster. This is because, in the API layer approach, all the real seL4 system calls are emulated, which can save some of the overhead of trapping into the kernel.

On the other hand, the ABI layer emulation approach should be faster than using any hardware emulation or virtualization, but slower than the API layer emulation approach, because intercepting system calls will introduce relatively high overhead.

4.2.2 Macrobenchmark

The macrobenchmark can be commenced under two circumstances and the result will be the measurement of the total system calls' elapsed time. On one hand, we can simulate a

situation with a heavy system call workload. The result can emphasize the expectation that using either API or ABI layer emulation approach will not introduce unacceptable overhead. On the other hand, we are going to simulate another situation with a low system call workload, the result emphasizes that using API or ABI emulation approach is more performant than using hardware virtualization or emulation approaches such as QEMU.

Chapter 5

Project Plan

So far the work of this project has been done is the background research of the related projects as well as construct the basic solution model in a theoretical aspect. This report summarizes the related projects' ideas that can be borrowed in this project. QEMU is an existing solution for emulating the entire foreign system. While Cygwin and Wine project give the idea that we can avoid hardware virtualization of emulation to execute applications that target foreign systems by adding an extra compatibility layer at API or ABI layer. And the final project UML introduced an approach to port Linux kernel into Linux user space and intercept the system calls from the userspace by using *ptrace*(). With those ideas, we purposed the implementation of the project using the API layer emulation approach by modifying the system call libraries and using the ABI layer emulation approach by using *ptrace*() system calls in Linux. The design and implementation details are not yet finalized at this stage. It's only a rough prototype that could potentially work.

5.0.1 Timeline

Below is the timeline for future thesis research work.

1. Port the seL4 kernel to the Linux userspace, and implement the basic framework

for the API. This will take roughly 2 weeks, and will be done during the break of the semester 21T1

2. Refine the emulation framework, complete seL4's critical kernel functionalities, and pinpoint parts that are hard to emulate. At this stage, the emulation framework should be mostly functional. The plan is to achieve this stage before week3 in semester 21T2.

3. Evaluate the API emulation framework using the methods listed in this report and modify the implementation if necessary. We should complete evaluating the API emulation approach two weeks before the week5 of the semester 21T2.

4. Depending on the time left in the semester 21T2. The next stage will be either implementing the ABI emulation approach or refining the theoretical part of the ABI emulation approach and adding details of implementation in the Thesis Report as future work.

5. Preparing for the final demonstration and the presentation.

Bibliography

[Cyg] A brief history of the cygwin project. `https://cygwin.com/cygwin-ug-net/brief-history.html`. Accessed: 2020-04-25.

[Dik06] Jeff Dike. *User Mode Linux*. Prentice Hall, 1rd edition, 2006.

[KAE+10] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

[LH14] Anna Lyons and Gernot Heiser. Mixed-criticality support in a high-assurance, general-purpose microkernel. In Rob Davis and Liliana Cucu-Grosjean, editor, *Workshop on Mixed Criticality Systems*, pages 9–14, Rome, Italy, December 2014.

[QEM] Qemu internals. `https://stuff.mit.edu/afs/sipb/project/phone-project/share/doc/qemu/qemu-tech.html`. Accessed: 2020-04-25.

[Wik21a] Wikipedia contributors. Cygwin — Wikipedia, the free encyclopedia, 2021. [Online; accessed 26-April-2021].

[Wik21b] Wikipedia contributors. Qemu — Wikipedia, the free encyclopedia, 2021. [Online; accessed 26-April-2021].

[Wik21c] Wikipedia contributors. Wine (software) — Wikipedia, the free encyclopedia, 2021. [Online; accessed 26-April-2021].