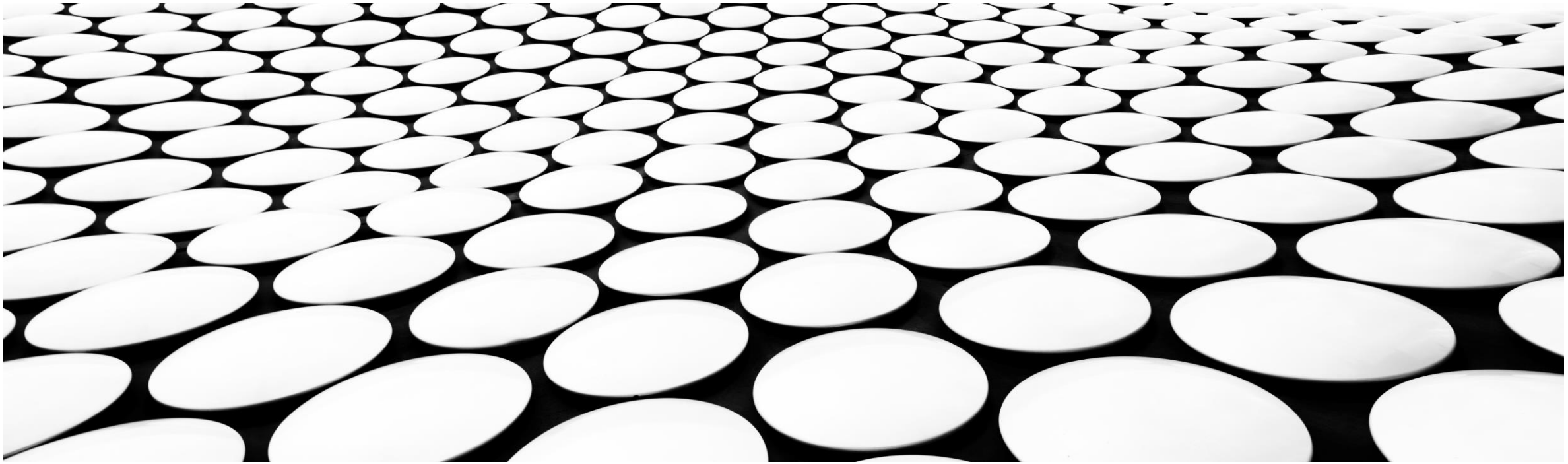
seL4 ABI/API EMULATION IN LINUX

JIAWEI GAO



OVERVIEW

- What is seL4?
- What is this project and why do we need this?
- How can we think about this problem?
- What is the existing solution?
- What are my approaches and Trade-offs?
- How to evaluate different approaches?

WHAT IS SEL4?

- A Microkernel and a Hypervisor but doesn't provide OS services.
- Formally verified to be correct and secure.
- Improves security with fine-grained access control through capabilities.
- The world's most advanced mixed-criticality system ensuring the safety of time-critical systems.
- The world's fastest microkernel.

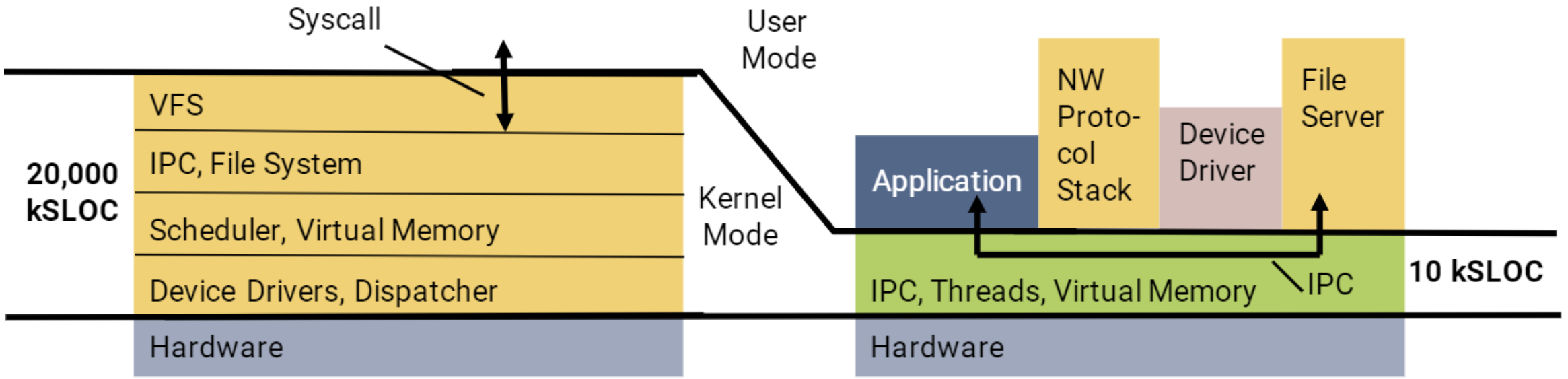
DIFFERENCE BETWEEN SEL4-BASED OS AND MONOLITHIC OS

Monolithic Kernel

- A high-level virtual interface over computer hardware. The higher layer abstracts the lower layer.
- Operating system services are implemented in the kernel such as file storage in the yellow part.
- Large trusted computing base (TCB).

seL4 Microkernel

- A minimal wrapper of hardware.
- OS functionalities are provided by the user-mode servers. Clients invoke servers by kernel IPC mechanisms.
- Minimal trusted computing base (TCB).



WHAT DO WE NEED TO SOLVE IN THIS PROJECT?

The main goal of this project is to explore and evaluate approaches to run seL4 applications in the Linux user space.

WHY DO WE NEED THIS PROJECT?

- Can be beneficial to seL4 developers!
  - Easy way for rapid prototyping an seL4 application in Linux environment.
  - Leverage Linux tools which can't be used directly in seL4 such as debugging tools (e.g. GDB, LLDB, etc.) or profiling tools (e.g. perf, Valgrind, etc.).
  - Access to Linux system's rich input and output (e.g. Files, Networking etc.). Instead of developing a device-specific driver, we can implement a special seL4 sever application which has the direct access to Linux I/O.

HOW CAN WE THINK ABOUT THIS PROBLEM?

- Running seL4 applications transparently in Linux can be thought of as providing a way to virtualize at different layers:
  - At the ISA layer.
  - At the kernel ABI layer.
  - At the syscall library API layer.
  - At the OS personality API layer or the CAmkES API layer (Not considered in this project).

WHAT IS THE EXISTING SOLUTION?

- An ISA level emulation
  - Qemu can
    - Emulate hardware platforms if the guest OS's ISA is different from the host's.
    - Perform hardware virtualization and be used with the KVM to run guest OS at native speed if the guest's ISA is the same as the host's ISA.
- Pros
  - Binary compatible.
- Cons (From a high level seL4 developer's view)
  - If the targeted ISA to be emulated is not the same as the host's or the host does not support hardware virtualization then using Qemu is not a preferred method as it will fully emulate the CPU which is slow. It may use a binary translation to optimize speed.
  - Debugging user land applications using Qemu's debugging interface is difficult as it has no understanding of the guest OS. Hence it can confuse the debugger when a context switch happens.

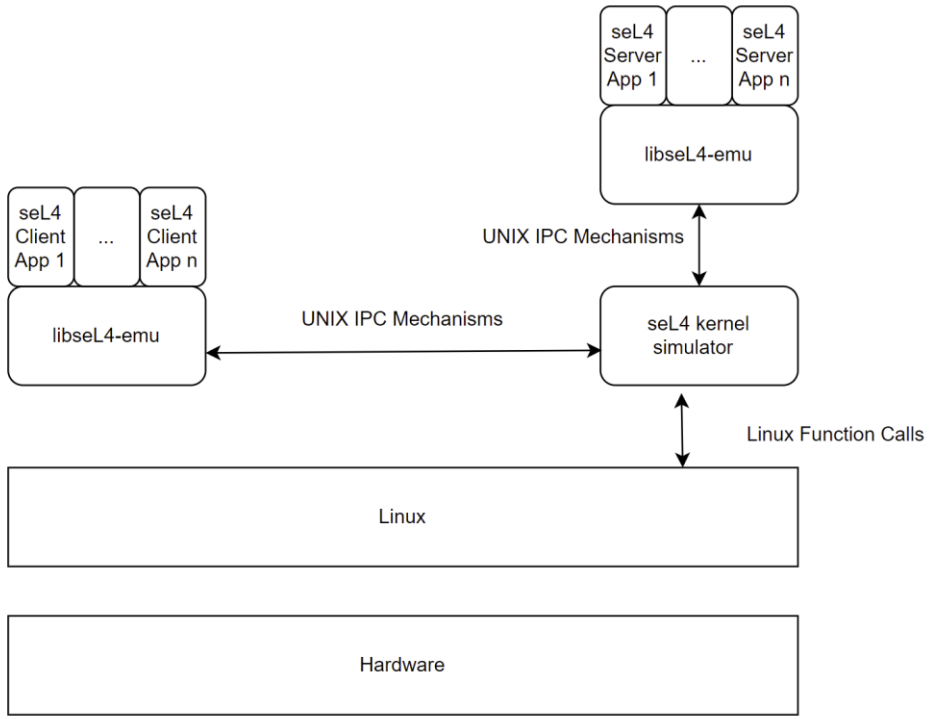
WE CAN DO BETTER THAN THAT!

- Approach 1
  - A syscall library API layer emulation.
- Approach 2
  - A kernel ABI layer emulation.

SYSCALL LIBRARY LAYER EMULATION

- Related project
  - **Cygwin**
- How to implement?
  - Provide an alternative seL4 syscall wrapper library which can be linked by seL4 client apps.
  - Provide a seL4 kernel simulator which can behave like a real seL4 kernel:
    - Providing seL4 syscall semantics.
    - Emulating the Cspace management.
    - IPC.
    - Context scheduling management.

MODEL



- By using this model, we can
  - Simulate the IPC between the seL4 applications and the seL4 kernel.
  - Treat each seL4 application as a regular Linux application and let them run on top of Linux without noticing that they are no longer running on seL4.
  - Interact with Linux instead of the real hardware so that we don't care about the underlying ISA.

IMPLEMENTATION

- A client-server model used in this implementation. The client refers to our seL4 applications and the server refers to the seL4 kernel simulator.
- A seL4 Client/Server Apps are linked with the libseL4-emu layer so that they can run **efficiently** as they are like regular Linux Apps.
- A seL4 kernel simulator serves the client's requests with the **same semantics** as the real seL4 syscalls do.
- Communication between seL4 apps and seL4 kernel emulator can use **Linux IPC mechanisms** (Unix Domain Socket/Share Memory etc.).

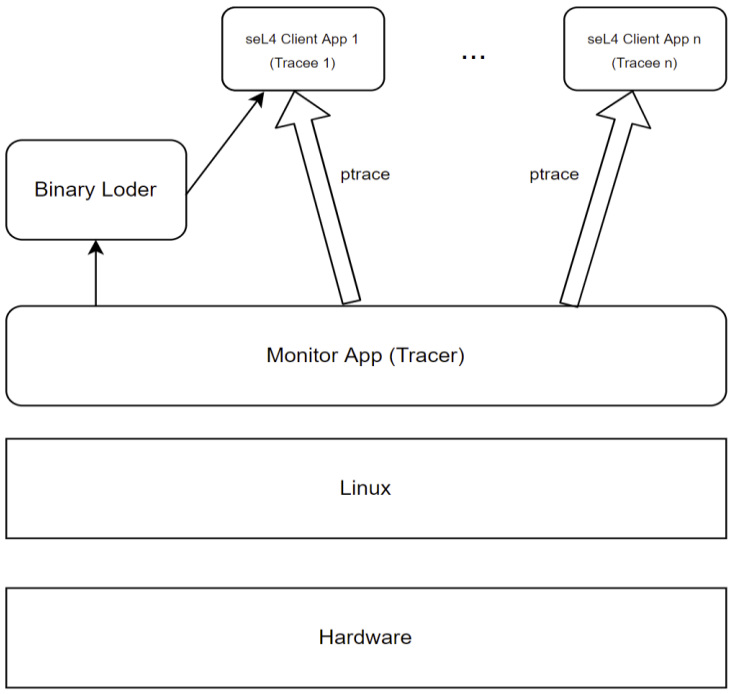
TRADE-OFFS

- What do we achieve here?
  - Relatively easy to implement.
  - ISA independent, since the emulation layer is on top of Linux, we don’t care about the underlying ISA.
  - Good performance.
    - Reduce trappings into the kernel.
    - The user-mode code will run as-is without any sort of virtualization or emulation on hardware level.
    - Regular Linux IPC is performant.
    - No emulation involved in the seL4 kernel emulator.
- What are the issues?
  - No binary compatibility achieved.
  - Require to recompile the source code of the seL4 application to perform the emulation.
  - Don’t work if the seL4 applications don’t use the libseL4-emu.

KERNEL ABI LAYER EMULATION

- Related project
  - **Wine**
- How to implement?
  - Need to redirect the syscall via ptrace syscall in Linux.
  - Provide a monitor app which is able to:
    - Load and launch a seL4 client app.
    - Using ptrace to intercept the seL4 syscalls.
    - Simulate the semantics of the real seL4 kernel.

MODEL



- By using this model, we can
  - Run unmodified seL4 binaries as long as the seL4 binaries are targetting the same ISA as the host.
  - Intercept the syscalls generated by the seL4 applications so that we can serve the syscalls as the real seL4 kernel does.

IMPLEMENTATION

- A Tracer-Tracee model is used in this approach.
- The monitor app (tracer) will do the following:
  - Fork a child process to load the seL4 client app binary code into the memory.
  - Set up the appropriate memory regions for the seL4 client app.
  - The loader will block and ask the monitor app to trace the seL4 client app (tracee).
  - Continue execute the seL4 client app.
- Every time the tracee does a syscall, the Linux kernel will suspend the tracee and notify the tracer so that the tracer can service the syscall by itself.

TRADE-OFFS

- What do we achieve here?
  - Binary compatibility.
  - No recompilation required to run seL4 client app.
- What are the issues?
  - Introduced overhead due to trapping into the ptrace syscalls.
  - Requires own debugging interface due to a tracee can be only attached to only one tracer every time and Linux debugger utilizes ptrace usually.

COMPARISONS BETWEEN DIFFERENT APPROACHES

Approach	Implementation Complexity	Implementation Portability	Binary Compatible	Debugging Support	Performance (Kernel IPC)
Syscall library API emulation	Intermediate	Good, as the solution is ISA potable	No	Yes and can use native Linux debugging tools	1st
Kernel ABI emulation	Hard	Good, but not as good as the above approach because this approach is ISA dependent	Yes	Yes, but extra debugging interface needed	2nd

HOW DO WE EVALUATE ALL THE APPROACHES QUALITATIVELY ?

	Qualitative Evaluation		
seL4 syscall layer	seL4 tutorials	seL4Test	AOS project
kernel ABI layer			
Success Criteria	Pass (Including CAmKEs tutorials)	Pass seL4 tests that we need	Successfully run SOS

- seL4 tutorials for trivial testing and demonstration such as printing “helloworld”, passing IPC message between two threads etc.
- seL4 Test for more specifically low level kernel APIs testing, Here we can focus on testing whether emulated kernel APIs function correctly such as syscalls, cspace management, scheduling and vspace management etc. Not considering test cases like caches, multicores, etc.
- AOS project is a simple multitasking operating system running on seL4 which has an interactive shell, and several subsystems including I/O, memory management, process management etc. This can be used for both testing and demonstration.

HOW DO WE EVALUATE ALL THE APPROACHES QUANTITATIVELY ?

Microbenchmark:

- Focus on evaluating the performance of each emulated seL4 syscall.
- Measure the number of syscalls per second.
  - In that case faster = more syscalls / sec, slow = fewer syscalls /sec.
- The result should be measured many times to make it more representative.

Success Criteria	syscall API emulation (A1)	kernel ABI emulation (A2)
Each seL4 syscall on 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
Each seL4 syscall on native hardware (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

HOW DO WE EVALUATE ALL THE APPROACHES QUANTITATIVELY ?

Macrobenchmark:

- Focus on evaluating the overall performance of emulation approaches.
- Consider simulating a situation where an app has a heavy syscall workload and a situation where an app has a low syscall workload.
  - The heavy syscall workload case emphasizes the comparison between the performance of the emulation approach and the native performance.
  - The low syscall workload case emphasizes comparing the performance of the emulation approach with the performance of Qemu.
- Measure the elapsed time of simulating the above situations.

Success Criteria	syscall API emulation (A1)	kernel ABI emulation (A2)
seL4 apps running on emulated hardware by Qemu (B1)	A1 should be significantly faster than B1 as no hardware emulation is involved	A2 should be significantly faster than B1 but slower than A1 due to syscall interception
seL4 apps running on native hardware (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

THANKS!