Hypervisor State Comparison: Linux vs seL4 for FreeRTOS Virtualization

PhD Research Project

August 18, 2025

Abstract

This document outlines an experimental design to compare the initialization state differences between Linux and seL4 hypervisors when hosting the same FreeRTOS binary. The research employs a triangulation methodology to identify what hardware and software state configurations may be causing differences in FreeRTOS behavior under different hypervisor environments. Three implementation approaches are analyzed with detailed pros and cons to determine the optimal experimental strategy.

1 Introduction

1.1 Research Context

Recent successful integration of FreeRTOS virtualization on seL4 microkernel has demonstrated that formally verified hypervisors can support real-time operating systems. However, subtle differences in hypervisor initialization may affect guest OS behavior in ways that are not immediately apparent.

1.2 Research Question

Primary Question: What initialization state differences exist between Linux and seL4 as hypervisors when hosting the same FreeRTOS binary?

Secondary Questions:

- Do different hypervisors configure the Generic Interrupt Controller (GIC) differently?
- Are Memory Management Unit (MMU) and page table configurations identical?
- Does CPU privilege level and register state differ between hypervisors?
- Are ARM generic timers initialized to the same state?
- Do cache configurations (L1/L2 data/instruction caches) match?

1.3 Methodology Rationale

By comparing FreeRTOS execution under two different hypervisors on identical hardware platforms, we can isolate hypervisor-specific initialization differences. This triangulation approach will reveal what state seL4 VM might be missing compared to Linux virtualization.

2 Experimental Design

2.1 Platform Consistency Requirements

To ensure valid comparison, all experiments must use identical platform configurations:

- **Hardware Platform:** QEMU ARM virt machine (`qemu-system-arm -M virt`)
- **CPU Model:** ARM Cortex-A15 or Cortex-A53 (consistent across tests)
- **Memory Layout:** Base address 0x40000000, 512MB allocation
- **UART Configuration:** PL011 UART at 0x09000000
- **Interrupt Controller:** ARM GICv2 configuration
- **FreeRTOS Binary:** Identical `freertos_image.bin` used in both scenarios

2.2 Test Scenarios

2.2.1 Scenario A: FreeRTOS under Linux Hypervisor

Linux kernel with virtualization support hosts the FreeRTOS binary using one of three implementation approaches (detailed in Section 3).

2.2.2 Scenario B: FreeRTOS under seL4 (Baseline)

Existing working seL4 + CAmkES VM configuration serves as the reference implementation. Same QEMU platform and FreeRTOS binary as Scenario A.

2.3 Comparison Metrics

2.3.1 Hardware State Analysis

- **CPU Registers:** CPSR, SCTLR, TTBR0/1, DACR, etc.
- **GIC Configuration:** Distributor registers, CPU interface state, interrupt priorities
- **MMU State:** Page table base addresses, domain settings, memory protection
- **Cache Configuration:** L1/L2 cache enable state, cache maintenance operations
- **Timer Setup:** ARM generic timer frequency, enable state, interrupt configuration

2.3.2 Memory Layout Analysis

- **Virtual Memory Mapping:** Address translation differences
- **Page Table Structure:** PTE permissions, memory type attributes
- **Memory Protection:** Access permissions for code/data/device regions
- **Device Memory Regions:** MMIO mapping configuration

2.3.3 Interrupt Configuration Analysis

- **GIC Distributor:** Interrupt enable/disable state, priority configuration
- **GIC CPU Interface:** Priority mask, interrupt acknowledgment setup
- **Interrupt Routing:** Which interrupts are routed to which CPU interfaces
- **Timer Interrupts:** ARM generic timer interrupt configuration

3 Implementation Approaches

3.1 Option 1: Linux + QEMU User Mode Emulation

3.1.1 Implementation Description

Boot a minimal Linux system, then use QEMU's user-mode emulation (`qemu-arm`) to execute the FreeRTOS binary within the Linux userspace environment.

3.1.2 Detailed Implementation Steps

1. Boot minimal Linux kernel (Buildroot or Alpine Linux)
2. Install QEMU user-mode emulation tools
3. Map FreeRTOS binary to appropriate memory location
4. Execute: `qemu-arm -cpu cortex-a15 freertos_image.bin`
5. Capture execution behavior and state

3.1.3 Pros

- **Simplicity:** Fastest to implement and debug
- **Tooling:** Excellent debugging support through QEMU
- **Isolation:** User-mode emulation provides controlled environment
- **Reproducibility:** Easy to script and automate
- **Safety:** Cannot crash host system
- **Portability:** Works on any Linux system with QEMU
- **State Inspection:** Can examine emulated CPU state easily

3.1.4 Cons

- **Abstraction Layer:** QEMU user-mode adds emulation layer
- **Limited Hardware Access:** Cannot examine real GIC/MMU state
- **Syscall Translation:** FreeRTOS system calls translated through Linux
- **Memory Management:** Uses Linux virtual memory, not bare-metal
- **Interrupt Handling:** QEMU translates interrupts to signals
- **Timer Differences:** Uses Linux timer services instead of hardware timers
- **Privilege Levels:** Runs in user mode, not privileged ARM modes
- **Cache Behavior:** Linux cache management affects FreeRTOS execution

3.2 Option 2: Linux + Custom Loader

3.2.1 Implementation Description

Create a minimal Linux kernel module or userspace program that directly loads FreeRTOS binary to physical memory, sets up minimal CPU state, and transitions execution to FreeRTOS entry point.

3.2.2 Detailed Implementation Steps

1. Boot minimal Linux with custom kernel module support
2. Implement kernel module or userspace loader with:
    - Physical memory mapping (`ioremap` or `/dev/mem`)
    - CPU state setup (registers, MMU, cache)
    - GIC configuration matching seL4 expectations
    - Direct jump to FreeRTOS entry point (0x40000000)
3. Capture CPU/hardware state before and after transition
4. Compare state dumps with seL4 VM initialization

3.2.3 Pros

- **Hardware Access:** Direct access to GIC, MMU, and CPU registers
- **State Control:** Full control over CPU state before FreeRTOS execution
- **Realistic Simulation:** Closely mimics hypervisor behavior
- **Debugging Capability:** Can insert state dumps at any transition point
- **Flexibility:** Can experiment with different initialization sequences
- **Bare Metal Transition:** Can transition from supervised to bare-metal mode
- **Memory Management:** Direct control over physical memory layout
- **Interrupt Configuration:** Can configure GIC identically to seL4

3.2.4 Cons

- **Complexity:** Requires kernel module development and low-level programming
- **System Stability:** Kernel module bugs can crash entire system
- **Platform Dependencies:** May require platform-specific MMU/GIC knowledge
- **Debugging Difficulty:** Kernel debugging more complex than userspace
- **Development Time:** Significant implementation effort required
- **Security Risks:** Direct hardware access requires careful programming
- **Linux Dependencies:** Still influenced by Linux kernel initialization
- **State Contamination:** Linux may alter hardware state before handoff

3.3 Option 3: Buildroot + Bare Metal Transition

3.3.1 Implementation Description

Use Buildroot to create a minimal Linux system that initializes hardware completely, dumps comprehensive CPU/hardware state, transitions to bare-metal mode, and jumps to FreeRTOS with minimal Linux interference.

3.3.2 Detailed Implementation Steps

1. Build custom Buildroot image with:
    - Minimal kernel with hardware initialization
    - Custom init program for state capture
    - Hardware debugging tools (devmem, register dump utilities)
2. Implement bare-metal transition sequence:
    - Capture complete system state (CPU, GIC, MMU, timers)
    - Disable Linux services and interrupt handling
    - Reset MMU to identity mapping
    - Configure GIC for bare-metal operation
    - Jump to FreeRTOS entry point
3. Compare captured state with seL4 VM initialization

3.3.3 Pros

- **Complete State Visibility:** Most comprehensive hardware state analysis
- **Minimal Interference:** Custom init minimizes Linux contamination
- **Hardware Realism:** Closest to actual hypervisor handoff behavior
- **Flexible Debugging:** Can capture state at multiple transition points
- **Educational Value:** Reveals complete hardware initialization sequence
- **Reproducible Builds:** Buildroot ensures consistent environment
- **Full Control:** Complete control over Linux initialization process
- **State Documentation:** Generates comprehensive state documentation

3.3.4 Cons

- **Maximum Complexity:** Most complex implementation requiring multiple expertise areas
- **Development Time:** Significant time investment for Buildroot customization
- **Build Dependencies:** Complex build system with many dependencies
- **Cross-compilation:** ARM cross-compilation complexity
- **Debugging Challenges:** Bare-metal debugging without standard tools
- **Hardware Specificity:** Highly platform-dependent implementation
- **State Transition Risks:** Complex state transitions prone to errors
- **Maintenance Overhead:** Custom builds require ongoing maintenance

4 Recommended Approach

4.1 Primary Recommendation: Option 2 (Linux + Custom Loader)

Based on the detailed analysis, **Option 2** provides the optimal balance of implementation complexity and research value:

4.1.1 Rationale

- **Direct Comparability:** Most similar to seL4 VM loading mechanism
- **Manageable Complexity:** Achievable within reasonable development timeframe
- **Hardware State Access:** Enables examination of critical CPU/GIC registers
- **Debugging Capability:** Allows state dumps at key transition points
- **Research Impact:** Directly addresses supervisor's triangulation suggestion

4.1.2 Implementation Priority

1. Start with userspace implementation using `/dev/mem` for hardware access
2. If userspace limitations prevent adequate hardware access, escalate to kernel module
3. Focus on GIC and MMU state comparison as primary objectives
4. Document all state differences discovered during implementation

4.2 Fallback Strategy

If Option 2 proves too complex or time-consuming, implement **Option 1** as a rapid prototype to establish baseline comparison methodology, then evolve toward Option 2 for comprehensive analysis.

5 Expected Deliverables

5.1 Implementation Artifacts

- Custom Linux loader (kernel module or userspace program)
- Hardware state capture scripts
- FreeRTOS execution comparison tools
- Automated testing framework

5.2 Analysis Reports

- Side-by-side hardware state comparison
- CPU register difference analysis
- GIC configuration comparison
- MMU and memory protection analysis
- Performance and timing characteristic comparison

5.3 Research Outcomes

- Identification of seL4 VM initialization gaps
- Recommendations for seL4 VM improvements
- Documentation of hypervisor-specific requirements for FreeRTOS
- Validation or refutation of current seL4 VM approach

6 Success Criteria

6.1 Minimum Success

- Execute identical FreeRTOS binary under both Linux and seL4 hypervisors
- Capture and compare basic CPU register state
- Identify at least 3 concrete differences in hardware initialization

6.2 Full Success

- Comprehensive hardware state comparison (CPU, GIC, MMU, timers)
- Root cause analysis of any FreeRTOS behavioral differences
- Actionable recommendations for seL4 VM configuration improvements
- Reproducible experimental framework for future hypervisor comparisons

7 Risk Analysis and Mitigation

7.1 Technical Risks

- **Hardware Access Limitations:** Mitigation through kernel module development
- **Platform Dependencies:** Mitigation through careful QEMU configuration
- **State Contamination:** Mitigation through minimal Linux configuration
- **Debugging Complexity:** Mitigation through incremental development approach

7.2 Timeline Risks

- **Implementation Complexity:** Mitigation through fallback to simpler approaches
- **Debugging Time:** Mitigation through early prototype development
- **Platform Issues:** Mitigation through proven QEMU ARM virt platform

8 Focused ARM Exception Vector Analysis Implementation

8.1 Problem Identification

Based on detailed analysis of the seL4 VM failure, the root cause has been identified as a missing ARM exception vector table configuration. FreeRTOS fails with a **Page Fault at PC: 0x8** during context switching, where 0x8 is the ARM Software Interrupt (SWI) vector address.

8.1.1 Failure Analysis

- **Location**: `vPortRestoreTaskContext()` in `portASM.S:199`
- **Instruction**: `BX R1` where R1 = 0x8
- **Expected**: R1 should contain valid task function address
- **Actual**: R1 contains ARM SWI vector address (0x8)
- **Cause**: seL4 VM doesn't provide ARM exception vectors for guest OS

8.2 ARM Exception Vector Layout

Address	Offset	Exception Type
0x00	+0	Reset Vector
0x04	+4	Undefined Instruction
0x08	**+8**	**Software Interrupt (SWI/SVC) ← Failure Point**
0x0C	+12	Prefetch Abort
0x10	+16	Data Abort
0x14	+20	Reserved
0x18	+24	IRQ
0x1C	+28	FIQ

Table 1: ARM Exception Vector Table Layout

8.3 Detailed Implementation Steps

8.3.1 Phase 1: Linux KVM Reference Setup (Day 1)

```
# Create working directory
mkdir -p linux_vm_comparison && cd linux_vm_comparison

# Download Alpine Linux ARM images
wget https://dl-cdn.alpinelinux.org/alpine/v3.19/releases/armhf/\
alpine-virt-3.19.1-armhf.iso

# Extract kernel and initramfs
mkdir alpine_extract && cd alpine_extract
7z x ../alpine-virt-3.19.1-armhf.iso
cp boot/vmlinuz-lts ../linux-kernel
cp boot/initramfs-lts ../linux-initramfs.gz
cd ..



# Boot Linux with identical platform configuration as seL4
qemu-system-arm \
  -M virt \
  -cpu cortex-a15 \
  -m 512M \
  -kernel linux-kernel \
  -initrd linux-initramfs.gz \
```

```
  -append "console=ttyAMA0,38400 rdinit=/bin/sh" \
  -nographic \
  -netdev user,id=net0 \
  -device virtio-net-device,netdev=net0
```

```
// vector_analyzer.c - Run inside Linux guest
#include <stdio.h>
#include <stdint.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

void analyze_vectors() {
    int fd = open("/dev/mem", O_RDWR);
    if (fd < 0) {
        perror("Cannot open /dev/mem");
        return;
    }

    // Map exception vector region (0x0-0x1C)
    uint32_t *vectors = mmap(0, 0x1000, PROT_READ,
                             MAP_SHARED, fd, 0x0);

    printf("=== ARM Exception Vector Analysis ===\n");
    for(int i = 0; i < 8; i++) {
        printf("Vector 0x%02x: 0x%08x\n", i*4, vectors[i]);
    }

    // Read VBAR register (requires privileged access)
    uint32_t vbar;
    asm volatile("mrc p15, 0, %0, c12, c0, 0" : "=r"(vbar));
    printf("VBAR = 0x%08x\n", vbar);

    munmap(vectors, 0x1000);
    close(fd);
}

int main() {
    analyze_vectors();
    return 0;
}
```

```
# Inside Linux guest - copy FreeRTOS binary
# Transfer freertos_image.bin to guest filesystem

# Load FreeRTOS to 0x40000000 and analyze state before execution
./vector_analyzer  # Capture baseline state
insmod freertos_loader.ko  # Custom kernel module to load FreeRTOS
```

```
./vector_analyzer  # Capture state after Linux hypervisor setup
```

8.3.2 Phase 2: Vector Table State Comparison (Day 2)

```
// Add to FreeRTOS main.c before scheduler start
void debug_vector_state() {
    uart_puts("=== seL4 VM Vector Analysis ===\r\n");

    uint32_t *vectors = (uint32_t*)0x0;
    for(int i = 0; i < 8; i++) {
        uart_puts("Vector 0x");
        uart_hex(i*4);
        uart_puts(": 0x");
        uart_hex(vectors[i]);
        uart_puts("\r\n");
    }

    // Read VBAR register
    uint32_t vbar;
    asm volatile("mrc p15, 0, %0, c12, c0, 0" : "=r"(vbar));
    uart_puts("VBAR = 0x");
    uart_hex(vbar);
    uart_puts("\r\n");
}

// Call before vTaskStartScheduler()
int main() {
    // ... existing FreeRTOS initialization ...
    debug_vector_state();
    vTaskStartScheduler();  // This will fail at vPortRestoreTaskContext
}
```

Step 2.2: Side-by-Side Comparison Create comparison table documenting the differences found.

8.3.3 Phase 3: seL4 VM Vector Configuration (Day 3-4)

```
// Examine seL4 VM component configuration
// File: projects/vm/components/VM_Arm/src/main.c

// Look for ARM exception vector setup
// Check if vm_install_vgic() configures exception vectors
// Verify if seL4_ARM_VSpace_* calls map low memory region
```

```
// Add to seL4 VM initialization
// File: projects/vm-examples/apps/Arm/vm_freertos/qemu-arm-virt/devices.camkes

assembly {
    configuration {
        // Add low memory mapping for exception vectors
        vm0.untyped_mmios = [
            "0x0:12",           // Exception vectors (4KB at 0x0)
            "0x8040000:12",     // GIC Virtual CPU interface
            "0x40000000:29",    // Guest RAM region (512MB)
            "0x09000000:12",    // UART MMIO region
        ];

        // Ensure vector region is mapped with execute permissions
        vm0.vm_address_config = {
            "ram_base" : "0x40000000",
            "ram_paddr_base" : "0x40000000",
            "ram_size" : "0x20000000",
            "dtb_addr" : "0x4F000000",
            "initrd_addr" : "0x4D700000",
            "kernel_entry_addr" : "0x40000000",
            // Add exception vector mapping
            "vector_base" : "0x0",
            "vector_size" : "0x1000"
        };
    }
}



// Create minimal exception vector table for guest
// File: projects/vm/components/VM_Arm/src/guest_vectors.S

.section .vectors, "ax"
.global guest_exception_vectors
guest_exception_vectors:
    ldr pc, =reset_handler          @ 0x00: Reset
    ldr pc, =undefined_handler      @ 0x04: Undefined instruction
    ldr pc, =swi_handler            @ 0x08: Software interrupt
    ldr pc, =prefetch_abort_handler @ 0x0C: Prefetch abort
    ldr pc, =data_abort_handler     @ 0x10: Data abort
    nop                             @ 0x14: Reserved
    ldr pc, =irq_handler            @ 0x18: IRQ
    ldr pc, =fiq_handler            @ 0x1C: FIQ

// Minimal handlers that delegate to FreeRTOS
swi_handler:
    // Jump to FreeRTOS SWI handler
    ldr pc, =FreeRTOS_SWI_Handler
```

8.3.4 Phase 4: Testing and Validation (Day 5)

```
# Rebuild seL4 VM with vector table support
cd camkes-vm-examples/build
ninja clean
ninja

# Test FreeRTOS execution
./simulate

# Expected output:
# "=== seL4 VM Vector Analysis ==="
# "Vector 0x08: 0x????????"  # Should now contain valid address
# "VBAR = 0x00000000"
# [FreeRTOS should progress past vPortRestoreTaskContext]



// Add success checkpoint after vPortRestoreTaskContext
// File: freertos_vexpress_a9/Source/portable/GCC/ARM_CA9/portASM.S

vPortRestoreTaskContext:
    // ... existing code ...
    BX      R1              // This should now succeed

context_switch_success:
    // Add debug output to confirm success
    LDR     R0, =success_message
    BL      uart_puts_asm
    // Continue with task execution
```

8.4 Expected Results and Success Criteria

8.4.1 Linux KVM Reference Results

Expected vector table with valid handlers at all positions, particularly at 0x8 for SWI.

8.4.2 seL4 VM Before Fix

Expected unmapped or zero values at vector positions, causing the page fault at 0x8.

8.4.3 seL4 VM After Fix

Expected properly configured vector table with valid handlers, allowing FreeRTOS to successfully start.

8.5 Implementation Timeline

Day	Duration	Tasks
1	8 hours	Linux KVM setup, FreeRTOS loader, vector analysis
2	4 hours	seL4 VM vector capture, side-by-side comparison
3	8 hours	seL4 VM vector configuration implementation
4	6 hours	Testing, debugging, refinement
5	4 hours	Validation, documentation, final testing
Total	**30 hours**	**Complete apple-to-apple comparison**

Table 2: Implementation Timeline

8.6 Risk Mitigation

- **Linux KVM Access Issues**: Use Alpine Linux with `/dev/mem` access enabled
- **VBAR Register Access**: Implement kernel module if userspace access insufficient
- **seL4 VM Configuration Complexity**: Start with minimal vector table, expand gradually
- **Memory Mapping Conflicts**: Carefully coordinate low memory mapping with existing seL4 VM setup

This focused implementation provides the exact triangulation needed to identify and resolve the ARM exception vector configuration differences between Linux and seL4 hypervisors.

9 Conclusion

This experimental design provides a systematic approach to understanding hypervisor initialization differences between Linux and seL4 when hosting FreeRTOS. The triangulation methodology addresses the supervisor's insight about configuration state differences, while the focused ARM exception vector analysis provides a direct path to resolving the identified Page Fault at PC: 0x8 issue.

The detailed implementation steps offer a practical, achievable approach that will generate actionable insights for improving seL4 VM configuration and advancing formally verified hypervisor research. The 5-day timeline balances thorough analysis with efficient execution, delivering concrete results for the PhD research objectives.