International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Debugging Early Guest Firmware (OVMF) and Guest Kernel Code Under KVM/QEMU

Ashish Kalra

Abstract: Debugging early-stage guest firmware and kernel code in virtualized environments remains one of the most persistent challenges in cloud infrastructure engineering. Before serial consoles, early printk outputs, or network logging services become operational, standard diagnostic pathways are entirely unavailable, leaving critical boot failures effectively invisible. Typically, to debug early guest firmware, specifically the Open Virtual Machine Firmware (OVMF), and early Linux guest kernel code, the most effective environment uses QEMU as the emulator and GDB as the debugger. However, when early firmware or kernel stages fail before a serial port or standard console is initialized, a specialized hardware emulation approach is needed to gain visibility into the earliest phases of guest execution. This article describes a virtual Port80h debug interface implemented within QEMU, through which guest firmware and kernel code emit low-level diagnostic postcodes that the host emulator traps and logs. The discussion covers the QEMU hardware emulation layer, OVMF build configuration with Port80h support, a checkpointing technique applied to ResetVector and 64-bit mode transitions, and practical instrumentation of the early Linux kernel startup path. Beyond the technical mechanics, the article examines why this capability matters strategically for cloud service providers, addressing security hardening, confidential computing integrity, infrastructure reliability, multi-tenant stability, boot latency reduction, hypervisor regression testing, and VirtIO driver stability. This technique is specific to the x86 architecture and is designed for use before early console or serial port debugging becomes available.

Keywords: *OVMF, KVM, QEMU, Guest Firmware Debugging, Port80h, Early Kernel Debugging, UEFI, Confidential Computing, Virtualization, Boot Diagnostics*

1. Introduction

1.1 The Debugging Challenge in Early Boot Phases

The initialization of a virtual machine in a cloud environment follows a tightly sequenced execution pipeline that begins well before any conventional logging or diagnostic mechanism is active. This early phase stretches from the first executed instruction in the firmware's reset vector through the UEFI Pre-EFI Initialization (PEI) phase and the Driver Execution Environment (DXE) and into the opening stages of the Linux kernel's startup path, specifically the __startup_64() context, which runs before any console is initialized. Throughout this entire window, the usual debugging instruments simply do not exist: serial ports have not been configured, network stacks are inactive, and kernel printk buffers have not yet been established. When failures happen here, because of firmware mismatches, memory encryption initialization errors, page table misconfiguration, or boot protocol violations, the resulting behavior is completely opaque from the perspective of any higher-level tooling.

Typically, to debug early guest firmware (OVMF/UEFI firmware) and early Linux kernel code, the most effective environment uses QEMU as the emulator and GDB as the debugger [5][6]. When early guest firmware or kernel stages fail before a serial port or standard console is initialized, however, this article discusses using specialized hardware emulation to gain visibility and debug early guest firmware and kernel code. The method described here is specific to the x86 architecture and is designed for use before

Independent Researcher, USA

early console or serial port debugging becomes available. It fills the diagnostic gap that exists even within a QEMU and GDB setup, providing an ordered signal from the very first guest instruction that requires no guest console or serial port to be operational.

The virtual Port80h debug interface sits at the heart of this discussion. Port 0x80, historically a diagnostic POST (Power-On Self-Test) code port found on legacy x86 PC hardware, has been accessible since the very first cycles of x86 system initialization, long before any interrupt controller, UART (Universal Asynchronous Receiver-Transmitter), or graphics subsystem is brought up. In a QEMU virtual machine, however, Port 0x80 emulation is off by default. Enabling it via a MemoryRegion registration modification in QEMU's hardware emulation layer creates a lightweight, ordered diagnostic channel that operates independently of any guest initialization state. This technique, known as checkpointing, involves adding checkpoints to guest firmware or kernel code using Port I/O writes to virtual debug port 0x80; these writes are trapped and logged by the QEMU host's Port I/O trap handler backend, facilitating tracing of code flow and helping identify the guest's state prior to a crash.

1.2 Importance for Cloud Infrastructure

Debugging early-stage guest firmware and kernel code is a foundational capability for cloud providers, ensuring that the entire cloud stack is secure, stable, and efficient [1]. As cloud deployments scale to support millions of concurrent virtual machine instances, even a single corrupted guest image or a firmware regression introduced through a VM/guest image update can ripple across multi-region infrastructure, causing widespread instance failures before any alerting system has a chance to react. Therefore, instrumenting and observing the earliest phases of guest execution has become a critical engineering capability, not merely a debugging convenience.

The rise of hardware-enforced Confidential Computing frameworks, including Intel TDX (Trust Domain Extensions), AMD SEV (Secure Encrypted Virtualization) and AMD SNP (Secure Nested Paging) has added an entirely new class of early-boot correctness requirements [7, 8]. In these environments, the host is explicitly untrusted and cannot access guest memory, which means conventional host-side introspection techniques are no longer available for observing early guest execution. Debugging early firmware, specifically OVMF/EDK2 — is the only way to verify that the Virtual Machine Privilege Level (VMPL) and memory encryption are initialized correctly. If the early boot fails in a Confidential Computing context, the secure handshake between the hardware and the guest is broken, and no conventional diagnostic channel remains [2].

The strategic value of early-stage debugging extends across nine distinct operational dimensions for cloud service providers: security and isolation hardening, infrastructure reliability, performance optimization, multi-tenant stability, confidential computing integrity, black-box boot failure resolution, hypervisor update validation, boot latency optimization, and VirtIO driver stability. Each of these is examined in Section 4 of this article. Taken together, they explain why early-stage guest debugging has become a prerequisite engineering discipline for providers operating at cloud scale.

1.3 Article Organization

The remainder of this article proceeds as follows. Section 2 provides background on the KVM/QEMU virtualization architecture and the OVMF firmware stack. Section 3 walks through the implementation of the virtual Port80h debug interface and its application to both early OVMF and Linux kernel code. Section 4 examines the strategic importance of these techniques for cloud service providers across nine operational dimensions. Section 5 covers known limitations and compares alternative approaches. Section 6 presents the conclusion.

2. Background and Related Work

2.1 KVM/QEMU Virtualization Architecture

The Kernel-based Virtual Machine (KVM) is a Linux kernel module that leverages hardware-assisted virtualization through Intel VT-x and AMD-V processor extensions. KVM exposes these capabilities through a device interface (/dev/kvm) consumed by user-space hypervisor processes. QEMU serves as the user-space counterpart in this architecture, handling full machine emulation including virtual CPU management, memory layout, device emulation, and I/O processing. Together, KVM and QEMU constitute one of the most widely deployed open-source virtualization stacks in production cloud infrastructure [1][9].

Of particular relevance to this article is how QEMU handles I/O. Its MemoryRegion-based dispatch framework maps specific I/O port address ranges to registered read and write handler callbacks — a clean, extensible mechanism covering both legacy x86 I/O ports and memory-mapped I/O (MMIO) regions. This same framework is what makes it straightforward to modify the existing Port80h handler. For the standard PC machine type, hw/i386/pc.c is where the machine initialization path lives and, consequently, where this integration is introduced. QEMU's user-space architecture means that all Port 0x80 write trapping and logging happen within the QEMU process itself, with no modification to the KVM kernel module.

Table 2 in Section 3.5 maps the principal components of the KVM/QEMU stack to their functional roles and the specific points at which the Port80h checkpointing methodology integrates with each component. Understanding this component-level breakdown is necessary for placing the debugging technique within the correct architectural context.

2.2 OVMF and the UEFI Firmware Stack

OVMF (Open Virtual Machine Firmware) is a project within the EDK2 (EFI Development Kit II) open-source framework that provides UEFI (Unified Extensible Firmware Interface) firmware specifically for QEMU and KVM virtual machines. OVMF implements the full UEFI specification lifecycle: the Security (SEC) phase, Pre-EFI Initialization (PEI), Driver Execution Environment (DXE), Boot Device Selection (BDS), and Runtime Services. The earliest phases, SEC and the mode transitions from 16-bit Real Mode through 32-bit Protected Mode to 64-bit Long Mode, execute from the reset vector at physical address 0xFFFFFFF0, well before any platform hardware has been initialized [3].

The ResetVector code, located in edk2/UefiCpuPkg/ResetVector/Vtf0/, handles the processor mode transitions that establish the execution environment required by all subsequent UEFI phases. Errors at this stage are notoriously difficult to diagnose because they precede every standard I/O facility. Enabling Port80h debug calls in the OVMF build activates predefined debugShowPostCode macro invocations throughout this early execution path, enabling post codes to be emitted to Port 0x80 at key transition checkpoints. Once this OVMF build is running under QEMU with Port80h emulation active, every default Port80h trace in the early OVMF code appears in the QEMU host log.

3. Implementation Details

3.1 Virtual Port 80h Debug Interface in QEMU

By default, the Port80h emulation code in QEMU is disabled. The MemoryRegionOps structure for Port 0x80 already exists in hw/i386/pc.c; enabling debug output requires modifying the existing MemoryRegion registration to add read and write handler logic. The following patch shows the modifications made to implement the Port80h debug interface:

```
static const MemoryRegionOps ioport80_io_ops = {
    .write = ioport80_write,
    .read  = ioport80_read,
    .endianness = DEVICE_NATIVE_ENDIAN,
    .impl = {
```

```
        .min_access_size = 1,
        .max_access_size = 1,
    },
};
uint64_t port80_mem;
static void ioport80_write(void *opaque, hwaddr addr,
                           uint64_t data, unsigned size)
{
    printf("port80: write 0x%02" PRIx64 "\n", data);
    port80_mem = data;
}
static uint64_t ioport80_read(void *opaque, hwaddr addr,
                              unsigned size)
{
    return port80_mem;
}
```

The write handler logs each diagnostic quadword (64-bit) to QEMU's standard output on the host, producing a real-time ordered trace of every post code the guest emits. The read handler returns the last written value, giving guest code a way to confirm that the emulation layer is active. All of this happens within QEMU's user-space process, no guest kernel driver, no interrupt configuration, and no platform initialization of any kind are needed. The technique is operational from the very first guest instruction.

3.2 Building OVMF with Port80h Support

The OVMF firmware must be explicitly compiled with Port80h diagnostic instrumentation enabled. The EDK2 build system provides a DEBUG_PORT80 build flag that activates predefined debugShowPostCode macro invocations distributed throughout the early firmware codebase. The following build command compiles the standard OVMF IA32X64 platform configuration with Port80h debugging, serial port debugging, Secure Boot, and SMM (System Management Mode) support enabled:

```
# build --cmd-len=64436 \
  -DDEBUG_ON_SERIAL_PORT=TRUE \
  -n $(getconf _NPROCESSORS_ONLN) \
  -a X64 \
  -a IA32 \
  -t GCC5 \
  -DSMM_REQUIRE \
  -DSECURE_BOOT_ENABLE=TRUE \
  -DDEBUG_PORT80=TRUE \
```

-p OvmfPkg/OvmfPkgIa32X64.dsc

The -DDEBUG_PORT80=TRUE flag switches on a series of conditional assembly macros across the ResetVector and SEC phase modules. Alongside it, -DDEBUG_ON_SERIAL_PORT=TRUE enables serial port logging, a complementary output channel that becomes available once the UART initializes — typically after the initial mode transitions complete. The GCC5 toolchain handles both 32-bit and 64-bit cross-compilation targets within the same invocation, which matters because the early firmware runs in 32-bit protected mode before switching to 64-bit long mode. With this OVMF build running under QEMU with Port80h emulation active, all default Port80h traces in early OVMF code are logged. Additionally, GDB can be used to connect to the QEMU monitor and perform I/O port reads on Port80h [5][6].

3.3 Debugging the Early Reset Vector Code

The reset vector code at edk2/UefiCpuPkg/ResetVector/Vtf0/Ia32/Flat32ToFlat64.asm carries out the processor's transition from 32-bit flat protected mode to 64-bit long mode. The following assembly code illustrates a debugShowPostCode macro call placed at the moment of successful 64-bit mode entry:

```
BITS    32
;
; Modified:  EAX
;
Transition32FlatTo64Flat:
    OneTimeCall SetCr3ForPageTables64
    mov     eax, cr4
    bts     eax, 5                  ; enable PAE
    mov     cr4, eax
    mov     ecx, 0xc0000080
    rdmsr
    bts     eax, 8                  ; set LME
    wrmsr
    mov     eax, cr0
    bts     eax, 31                  ; set PG
    mov     cr0, eax                 ; enable paging
    jmp     LINEAR_CODE64_SEL:ADDR_OF(jumpTo64BitAndLandHere)
BITS    64
jumpTo64BitAndLandHere:
    debugShowPostCode POSTCODE_64BIT_MODE
    OneTimeCallRet Transition32FlatTo64Flat
```

Running this code with Port80h emulation active in QEMU produces a host log output such as the following, in strict execution order:

port80: write 0x16

```
port80: write 0x32
port80: write 0xb1
port80: write 0xf1
```

3.4 Instrumenting Early Linux Kernel Code

The same checkpointing approach applies equally well to very early Linux kernel code — in particular, the __startup_64() function in arch/x86/kernel/head64.c. This function runs before the kernel's console, earlyprintk, infrastructure is available, making it one of the hardest sections of kernel code to observe through any standard means [5]. The example below shows outb() instrumentation placed inside the SME (Secure Memory Encryption) initialization path within __startup_64():

```
unsigned long __head __startup_64(unsigned long physaddr,
                                  struct boot_params *bp)
{
   ...
   /* Encrypt the kernel and related (if SME is active) */
   sme_encrypt_kernel(bp);

   if (mem_encrypt_active()) {
      vaddr     = (unsigned long)__start_bss_decrypted;
      vaddr_end = (unsigned long)__end_bss_decrypted;
      for (; vaddr < vaddr_end; vaddr += PMD_SIZE) {
         i = pmd_index(vaddr);
         pmd[i] -= sme_get_me_mask();
      }
         outb(0xCD, 0x80);
         pfn = pmd_pfn(*(pmd_t *)&pmd[pmd_index(vaddr)]);
         outb((pfn >> 24), 0x80);
         outb((pfn >> 16), 0x80);
         outb((pfn >> 8),  0x80);
         outb(pfn,         0x80);
   }
   ...
}
```

3.5 The Checkpointing Methodology

Combining Port80h emulation in QEMU with deliberate post-code instrumentation in guest firmware and kernel code creates a practical, repeatable checkpointing methodology for early-stage debugging. As the original description of this technique states: checkpointing involves adding checkpoints to the guest firmware or kernel code using Port I/O writes to virtual debug port 0x80; these writes are trapped and logged by the QEMU host's Port I/O trap handler backend, facilitating tracing of code flow and helping identify the guest's state prior to a crash.

Several properties make this methodology well-suited to production engineering environments. Host-side operation is entirely non-invasive — no kernel module, no ptrace attachment, and no changes to the VM's memory layout are needed. Output ordering is guaranteed because outb instructions are serializing on x86, meaning the logged sequence accurately reflects instruction execution order. The technique also composes cleanly with GDB: an engineer can attach to the QEMU GDB stub, set breakpoints at expected post code boundaries, and drop into full register and memory inspection at any instrumented checkpoint [5][6]. Table 1 below presents a structured comparison of the primary debugging techniques available for early-stage guest firmware and kernel code.

Table 1: Comparison of Early-Stage Guest Debugging Techniques in KVM/QEMU Environments

Debugging Technique	Pre-Console Availability	Host Invasiveness	CI Pipeline Suitable
Port80h Checkpointing	Yes — operational from first guest instruction	None — user-space QEMU handler only	Yes — output captured as plain text log
GDB Remote Stub (QEMU GDB)	Partial — requires QEMU -s flag; console not needed but setup is manual	Low — QEMU's remote stub (gdbstub) enables debugging guest code by acting as a GDB server, allowing host-side GDB client to connect via TCP and control the VM's CPU and Memory.	Limited — requires custom scripting
ISA-DebugCon Device	Yes — character-oriented channel available early	None — QEMU device emulation only	Moderate — richer output but firmware integration required
Hardware JTAG	Yes — full hardware state visible	High — physical access required; not applicable at cloud scale	No — impractical for server fleet deployment

The comparison highlights that while GDB and hardware JTAG (Joint Test Action Group) offer greater interactive depth, only Port80h checkpointing combines pre-console availability, non-invasive host operation, and pipeline automation in a single approach. Table 2 below maps the principal KVM/QEMU stack components to their functional roles and Port80h integration points, providing a complete architectural picture of where this technique operates within the virtualization stack.

Table 2: KVM/QEMU Virtualization Stack Components and Port80h Integration Points

Stack Component	Functional Role	Port80h Integration Point
KVM Kernel Module (/dev/kvm)	Provides hardware-assisted virtualization via Intel VT-x and AMD-V. Manages virtual CPU creation, VM entry/exit handling, and memory virtualization.	No direct integration. KVM transparently passes Port 0x80 I/O exits up to QEMU's device emulation layer without requiring KVM-level modification.
QEMU Machine Emulation (hw/i386/pc.c)	Implements full x86 PC machine emulation: virtual CPU management, memory layout, legacy I/O port dispatch, and device initialization. The MemoryRegion framework maps I/O port ranges to handler callbacks.	Primary integration point. The ioport80_io_ops MemoryRegionOps structure is modified here, updating read and write handlers bound to I/O port address 0x80. All guest Port 0x80 writes are trapped and logged from the first guest instruction.
OVMF / EDK2 Firmware (ResetVector, SEC, PEI, DXE)	Implements the full UEFI boot lifecycle from the reset vector at 0xFFFFFFF0 through SEC, PEI, DXE, and BDS phases. Manages processor mode transitions and platform hardware initialization.	Guest-side integration. Enabling Port80h debug calls in the OVMF build activates debugShowPostCode macros throughout the ResetVector and SEC modules, each expanding to an outb instruction that emits a phase-specific value to Port 0x80.
Linux Kernel Early Startup (arch/x86/kernel/head64.c)	Executes __startup_64() before any console, printk, or ioremap infrastructure is available. Handles early page table construction and AMD SME/SEV memory encryption initialization.	Guest-side kernel instrumentation. Manual outb() calls inserted at key points within __startup_64() emit sentinel and data bytes to Port 0x80, enabling diagnosis of SME initialization errors and pre-start_kernel() hangs that are otherwise invisible.

4. Strategic Importance for Cloud Service Providers

Debugging the VM/guest launch at an early stage is a specialized discipline that provides several strategic advantages to cloud service providers (CSPs). Table 3 below summarizes nine strategic dimensions through which early-stage guest firmware and kernel debugging delivers operational value.

Table 3: Strategic Importance of Early-Stage Debugging for Cloud Service Providers

Strategic Dimension	Description
Hardening Security and Isolation	The guest kernel represents the critical boundary for modern cloud native environments.

	Preventing Escape Attacks: By debugging guest-virtualization specific drivers (e.g., virtio), providers can close vulnerabilities that might allow a guest to escape into the host hypervisor.
	Root of Trust Verification: Debugging guest firmware ensures that the platform integrity is verified during boot, which is essential for Confidential Computing and secure VM launches.
Improving Infrastructure Reliability	Cloud providers use these techniques to resolve "silent" failures that high-level logs cannot capture.
	Resolving "Zombie" Instances: When a VM hangs during boot (before the network or serial port is ready), low-level debugging identifies if the issue is a kernel panic or a firmware/bootloader bug.
	Minimizing Global Downtime: Rapidly identifying root causes in shared guest images prevents widespread deployment failures across multi-region infrastructure.
Performance Optimization	Eliminating Bottlenecks: Debugging allows providers to observe how guest code interacts with virtualized hardware, helping them optimize resource usage and reduce inefficient memory usage.
	Driver Tuning: Providers can fine-tune the performance of their proprietary guest drivers by tracing execution paths within the guest kernel.
Supporting Multi-Tenant Stability	Isolation Integrity: In a multi-tenant environment, the hypervisor must ensure that one guest's kernel failure does not impact others. Debugging ensures that the Virtual Machine Monitor (VMM) correctly handles all guest states, even during crashes.
	Automated Diagnostics: Large-scale providers use automated debugging mechanisms to troubleshoot thousands of orchestrated VM deployments simultaneously.
Hardening "Confidential Computing"	Modern cloud security relies on Trusted Execution Environments (TEEs) like Intel TDX or AMD SEV, in a confidential VM, the host is "untrusted" and cannot see guest memory.
	Debugging early firmware (like OVMF/EDK2) is the only way to verify that the Virtual Machine Privilege Level (VMPL) and memory encryption are initialized correctly. If the early boot fails, the secure "handshake" between the hardware and the guest is broken.
Eliminating "Black Box" Boot Failures	When a cloud customer launches an instance and it stays in a "Pending" or "Running (No Heartbeat)" state, it is often due to an early kernel panic or a firmware mismatch and early debugging is an essential tool to debug and fix these kernel panics and/or firmware issues.

Validating Hypervisor Updates (Regression Testing)	Cloud providers constantly update their underlying hypervisors (KVM, Xen, or proprietary VMMs), by debugging the early guest kernel, engineers can see exactly how the guest reacts to a new virtual CPU feature or an added/changed device feature.
Optimizing Boot Latency	In "Serverless" or "Function-as-a-Service" (FaaS) environments, boot time is a critical performance metric and identifying bottleneck by using GDB to trace the early guest kernel, providers can find exactly where time is being wasted (e.g., waiting for a non-existent hardware probe) and "prune" the kernel to achieve sub-second boot times.
Driver Stability for Virtualized Hardware	Cloud providers typically use custom VirtIO drivers which often begin initializing very early in the boot process. Debugging at this stage ensures that the communication channel between the guest and the host's physical hardware (via the hypervisor) is stable and performant from the first instruction.

5. Discussion

5.1 Limitations and Constraints

The Port80h checkpointing methodology carries several limitations worth acknowledging. Most immediately, it is specific to the x86 architecture. ARM, RISC-V, and POWER systems have no equivalent legacy I/O port mechanism, so different instrumentation strategies are needed on those platforms. Beyond architecture scope, the technique requires source-level modifications — either adding Port80h debug calls in the OVMF build or inserting outb() calls into the Linux kernel — which means the instrumented debug build diverges from the production binary. Behavioral differences introduced by compiler optimization levels between these two build configurations need to be factored in when drawing conclusions from diagnostic data [3].

5.2 Comparison with Alternative Approaches

Reference [4] discusses using a similar technique through QEMU via a peripheral device called the ISA-DebugCon. The ISA-DebugCon device offers a complementary character-oriented debug channel that can carry multi-character string output — richer than single-byte post codes — but it demands more complex firmware integration and has no native support in the Linux kernel's early startup path. GDB remote protocol attachment to the QEMU GDB stub remains the most capable option for general kernel and firmware debugging, providing full register inspection, memory reads, breakpoints, and single-step execution [5][6]. The tradeoff is setup complexity and limited suitability for automated pipelines without custom scripting.

Also, in confidential computing environments like SEV-SNP or TDX, [7][8] the guest register state and memory is encrypted and inaccessible to host and hence standard debugging mechanisms like register inspection, memory reads/writes, breakpoints and single-step execution can't be used. In such environments, code checkpointing and dumping register/memory contents via a virtual (I/O) debug port is essential for confidential guest debugging.

Hardware JTAG debugging offers complete hardware state visibility with no guest cooperation needed at all, but requires physical machine access — a practical non-starter for cloud-scale deployments across commodity server fleets. The Port80h methodology sits in a useful middle ground: lightweight enough to instrument at scale, automatable in CI pipelines, and fully operational from the first guest instruction with no modifications needed on the host side beyond the QEMU MemoryRegion registration modification described in Section 3.1. For general kernel and module debugging through QEMU, the standard approach remains GDB-based as documented in the Linux kernel's own debugging guides [5].

Conclusion

In the context of cloud computing, "early" debugging refers to the period between the virtual machine

power-on and the point where the operating system's networking or standard logging services are active. It is a narrow window, but it covers some of the most security-sensitive and reliability-critical code in the entire cloud stack — UEFI firmware, processor mode transitions, memory encryption initialization, and early kernel startup. It is also the window where conventional diagnostic tools go silent.

This article has reviewed a systematic approach for making that window observable: a virtual Port80h debug interface implemented in QEMU, combined with post-code instrumentation placed at deliberate points in OVMF firmware and Linux kernel source. The resulting checkpointing methodology is ordered, automatable, and independent of guest console availability. It survives the memory encryption boundary of Confidential Computing environments and works alongside GDB-based interactive debugging when deeper inspection is needed. This technique is specific to the x86 architecture and is most useful before early console or serial port debugging becomes available.

Debugging the VM/guest launch at an early stage is a specialized discipline that provides several strategic advantages to cloud service providers. As cloud deployments grow in scale and hardware-enforced TEE adoption becomes standard practice, maintaining clear visibility into the earliest phases of guest execution is not an optional engineering concern — it is a prerequisite for building infrastructure that is genuinely secure, predictably reliable, and ready for the performance demands that modern cloud workloads place on it.

References

[1] D. Hildenbrand, "Guest operating system debugging," Linux KVM Forum, 2015. [Online]. Available: https://www.linux-kvm.org/images/9/92/01x10-David_Hildebrand-Guest-operating_system_debugging.pdf

[2] The Institute of Internal Auditors, "A Roadmap to Auditing Cloud Security," 2025. [Online]. Available: https://www.theiia.org/en/content/articles/global-best-practices/2025/a-roadmap-to-auditing-cloud-security/

[3] Xeno Kovah and Corey Kallenberg, "Advanced x86: BIOS and System Management Mode Internals Reset Vector," OpenSecurityTraining. [Online]. Available: https://opensecuritytraining.info/IntroBIOS_files/Day1_XX_Advanced%20x86%20-%20BIOS%20and%20SMM%20Internals%20-%20Reset%20Vector.pdf

[4] Zero's Blog, "Debugging in Extreme Context through QEMU+Linux-KVM," 2023. [Online]. Available: https://tangptr.com/2023/debugging-in-extreme-context-through-qemulinux-kvm/

[5] The Linux Kernel Documentation, "Debugging kernel and modules via GDB." [Online]. Available: https://www.kernel.org/doc/html/v4.14/dev-tools/gdb-kernel-debugging.html

[6] QEMU Documentation, "GDB usage," QEMU. [Online]. Available: https://www.qemu.org/docs/master/system/gdb.html

[7] Advanced Micro Devices, "AMD SEV-SNP: Strengthening VM isolation with integrity protection and more," AMD White Paper, 2020. [Online]. Available: https://docs.amd.com/v/u/en-US/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more

[8] Pau-Chen Cheng, et al., "Intel TDX demystified: A top-down approach," arXiv, 2023. [Online]. Available: https://arxiv.org/pdf/2303.15540

[9] M. Rosenblum, et al., "Virtual machine monitors: current technology and future trends," Computer, 2005. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1430630