Debugging Support for Confidential Computing Virtual Machines: Architectural Design and Implementation Framework for QEMU

Ashish Kalra

Independent Researcher, USA

Abstract

Confidential computing virtual machines (VMs) rely on hardware-enforced memory encryption technologies to protect guest data from unauthorized access, including access by the hypervisor itself. While this isolation provides a strong security boundary, it fundamentally breaks the memory-access mechanisms upon which conventional debugging tools depend. QEMU, a widely deployed open-source virtual machine monitor (VMM), exposes built-in debugging capabilities through its gdbstub interface and monitor commands; both rely on direct guest physical memory access via host virtual address (HVA) operations that become unavailable when guest memory is encrypted. This article presents the architectural design of a structured debugging framework for QEMU that restores debugging functionality without compromising the security properties of confidential guests. The framework introduces a layered abstraction comprising an extended memory transaction attributes structure, a new MemoryDebugOps dispatch interface, vendor-specific memory region callbacks, and complementary debug-aware physical memory APIs. Each component is designed to intercept the debug memory-access path, route requests through firmware-assisted or hypervisor-extension-assisted decryption when permitted by guest policy, and leave general-purpose, non-debug code paths entirely unmodified. The design trade-offs, hook placement strategies, page-table walk considerations, and handling of unencrypted memory regions are examined in detail. The framework has practical relevance for cloud operators, hypervisor developers, and security engineers who need to diagnose and validate confidential VM workloads in production and staging environments without relaxing hardware-enforced isolation guarantees.

Keywords: Confidential Virtual Machine Debugging, QEMU Memory Debug Framework, AMD SEV Encrypted Guest Memory, Memory DebugOps Architecture, Secure Encrypted Virtualization Hypervisor

1. Introduction

The proliferation of confidential computing technologies has introduced a new class of virtualization environment in which guest workloads execute with hardware-enforced memory confidentiality [11]. Technologies such as AMD Secure Encrypted Virtualization (SEV) [1] and Intel Trust Domain Extensions (Intel TDX) [2] encrypt the memory contents of a virtual machine; however, the design and implementation discussed in this article are grounded primarily in AMD SEV, such that even a privileged hypervisor cannot read plaintext guest data through ordinary memory operations. This design is deliberate and constitutes the central security guarantee of the confidential computing model: the hypervisor, despite managing the VM lifecycle, is excluded from the trusted computing base of the guest. This exclusion, however, creates a significant operational problem for virtual machine monitors (VMMs) that provide built-in debugging facilities. QEMU, one of the most widely used open-source hypervisors, offers debugging capabilities through two principal mechanisms: a gdbstub that exposes a GDB remote debugging interface and a set of human-readable or machine-readable monitor commands (HMP/QMP) that allow inspection of guest memory and virtual address space [4]. Both mechanisms ultimately rely on the ability of the hypervisor to perform direct reads from guest physical memory using host virtual address (HVA) pointers, typically through operations equivalent to memcpy() from the host side. When a guest's memory is encrypted, these operations yield unintelligible ciphertext, rendering the debugging tools functionally inoperative.

Despite the growing adoption of confidential computing technologies, no systematic architectural framework exists within QEMU for restoring debugger functionality in encrypted guest environments. Prior approaches either compromise the guest security boundary or lack the extensibility required for multi-vendor deployment. This article addresses that gap and makes the following contributions: (1) an extension to the MemTxAttrs structure that propagates a debug-context signal without modifying general-purpose code paths; (2) the MemoryDebugOps dispatch interface, a vendor-neutral mechanism for routing debug memory accesses through firmware-assisted decryption; (3) per-region RAM debug callbacks via MemoryRegionRAMReadWriteOps for handling mixed encrypted and unencrypted memory within a single address space;

and (4) debug-aware physical memory API extensions that ensure all debug access sites within QEMU are correctly redirected. The framework introduces a layered debugging abstraction within QEMU that routes debug-initiated memory accesses through vendor-specific, firmware-assisted, or hypervisor-extension-assisted decryption interfaces when the guest policy explicitly permits debugging [3]. The framework is designed to be extensible, vendor-agnostic at the dispatch layer, and non-intrusive with respect to QEMU's general-purpose memory subsystem. This design is derived from and validated against AMD SEV-based implementations discussed in QEMU development threads and primarily reflects SEV behavior, with broader applicability to other confidential computing technologies remaining conceptual.

2. Background and Problem Context

2.1 Confidential Guest Memory Encryption

In confidential computing architectures, guest physical memory pages are encrypted by hardware using keys that are not accessible to the hypervisor [12][13]. In the AMD SEV model, a separate encryption key is provisioned per virtual machine, and all memory traffic between the guest and DRAM is transparently encrypted and decrypted by the processor's memory controller [1]. The SEV firmware, running in a privileged processor security context, mediates access to these keys and exposes a constrained set of interfaces to the hypervisor through the KVM kernel module and the Linux crypto (CCP/ASP) device driver.

Critically, the AMD SEV firmware is intended to provide two commands specifically designed to assist in debugging: DBG_DECRYPT and DBG_ENCRYPT [3]. These commands instruct the firmware to decrypt or encrypt data at specified guest memory regions on behalf of the hypervisor. Because this operation grants the hypervisor temporary access to plaintext guest memory, it is gated behind a guest-controlled policy bit. If the guest's SEV policy includes the NODBG flag, the firmware will refuse to execute either command, preserving the confidentiality guarantee even in the presence of a cooperative or compromised hypervisor [3]. This policy-gated mechanism forms the trust boundary within which all debugging support must operate.

2.2 QEMU's Existing Debug Memory Path

QEMU's guest memory access for debugging is centralized through the function cpu_memory_rw_debug(), which is invoked by both the gdbstub (via target_memory_rw_debug()) and the monitor subsystem (via memory_dump() in monitor/misc.c) [4]. Internally, this function delegates to the address_space_rw() family of accessor functions, which perform address translation and ultimately invoke memcpy() to transfer data between the host and guest memory regions.

For non-confidential guests, this path is entirely correct and efficient. For encrypted guests, the memcpy() operation at the terminal step reads ciphertext rather than plaintext, and the resulting data is meaningless to any debugger. The architectural challenge is, therefore, to intercept this terminal step and substitute a firmware-assisted or hypervisor-extension-assisted decryption operation while leaving all non-debug code paths unmodified.

3. Initial Design Approach and Maintainer Feedback

An initial approach to the problem involved attaching debug-specific callbacks directly to MemoryRegion objects, the fundamental abstraction QEMU uses to represent regions of the guest physical address space. Under this design, each memory region that required special handling for debugging would carry a pointer to a set of debug operations, and the address_space_read and address_space_write functions would be modified to check for and invoke these callbacks.

This approach was reviewed on the QEMU developer mailing list and received critical feedback from project maintainers [4][5]. The primary objection was architectural: hooking into address_space_read and address_space_write introduces debug-specific logic into general-purpose memory access functions. These functions are performance-critical and are invoked on every guest memory access, not only during debugging sessions. Contaminating them with conditional debug logic was considered unacceptable from both a performance and a code-clarity standpoint. Maintainers specifically requested that the hook mechanism avoid touching general-purpose, non-debug code paths wherever possible [5].

This feedback directly motivated the design described in the subsequent sections: a dedicated debug dispatch layer that intercepts the debug memory-access path before it reaches the general-purpose accessor functions, rather than modifying the accessor functions themselves.

4. Architectural Design of the Debug Framework

4.1 Extension of MemTxAttrs: The debug Flag

The first architectural element of the framework is a minimal but semantically significant extension to the MemTxAttrs structure, which QEMU uses to annotate memory transaction requests with attributes such as requester identity and access type. A single one-bit field, named debug, is added to this structure:

```
typedef struct MemTxAttrs {
  /* ... existing fields ... */
  unsigned int debug : 1;
} MemTxAttrs;
```

This flag is set to one exclusively within cpu_memory_rw_debug() before any downstream memory access is initiated. Because MemTxAttrs is propagated through the entire chain of memory access calls, all functions downstream of the debug entry point can inspect this flag to determine whether the current access is debug-initiated. This provides a clean, non-invasive signal that allows vendor-specific and region-specific handlers to distinguish debug accesses from ordinary memory traffic without requiring separate function signatures or parallel call hierarchies. The design satisfies the maintainer requirement that general-purpose code remain unmodified: the flag is set only at the debug entry point, and non-debug callers never set it.

Table 1 below provides a structured before-and-after comparison of the MemTxAttrs structure and its propagation behavior.

Aspect	Before Extension	After Extension
Struct field	No debug indicator field	unsigned int debug: 1 (1-bit flag added to MemTxAttrs)
Propagation	address_space_rw() called directly	attrs.debug = 1 set in cpu_memory_rw_debug(); propagated to all downstream calls
Memory access path	Single unified path: memcpy() used for all RAM reads	Branching path: debug flag triggers ram_debug_ops if registered; else memcpy()
Impact on non-debug paths	N/A — single path	None; general-purpose code untouched; hook scoped to debug callers only

Table 1: MemTxAttrs Extension: Before and After debug Flag Addition [4, 5]

4.2 MemoryDebugOps: A Vendor-Neutral Dispatch Interface

The central architectural contribution of the framework is the introduction of a new structure, MemoryDebugOps, which represents a logically vendor-neutral dispatch mechanism. This mechanism is currently implemented for AMD SEV-based guests and defines a set of function pointers that are used exclusively on the debug memory-access paths.

```
typedef struct MemoryDebugOps {
  hwaddr (*translate)(CPUState *, target_ulong addr, MemTxAttrs *attrs);
  MemTxResult (*read)(AddressSpace *as, hwaddr phys_addr,
                      MemTxAttrs attrs, void *buf, hwaddr len);
  MemTxResult (*write)(AddressSpace *as, hwaddr phys_addr,
                       MemTxAttrs attrs, const void *buf, hwaddr len);
  uint64_t (*pte_mask)(void);
} MemoryDebugOps;
```

A global pointer, debug_ops, is initialized to a default implementation that delegates to the standard QEMU memory accessor functions. The default assignment is as follows:

```
static const MemoryDebugOps default_debug_ops = {
    .translate = cpu_get_phys_attrs_debug,
    .read      = address_space_read,
    .write     = address_space_write_rom,
    .pte_mask  = address_space_pte_mask,
};
static const MemoryDebugOps *debug_ops = &default_debug_ops;
```

This means that in the absence of any vendor-specific registration, the framework behaves identically to the existing QEMU debug path. Vendor-specific implementations, such as those required for SEV guests, register their own MemoryDebugOps instance, overriding one or more of these callbacks to route requests through firmware-assisted or hypervisor-extension-assisted decryption [4]. Within cpu_memory_rw_debug(), the direct calls to address_space_read and address_space_write_rom are replaced with indirect calls through debug_ops->read and debug_ops->write, as shown below:

```
cpu_memory_rw_debug(.. *cpu, .. addr)
{
    /* ... */
+   attrs.debug = 1;
    if (is_write) {
-       res = address_space_write_rom(.., phys_addr, attrs, ..);
+       res = debug_ops->write(.., phys_addr, attrs, ..);
    } else {
-       res = address_space_read(.., phys_addr, attrs, ..);
+       res = debug_ops->read(.., phys_addr, attrs, ..);
    }
    /* ... */
}
```

Table 2 below describes each MemoryDebugOps callback in detail.

Callback	Signature (simplified)	Default Implementation	Purpose
translate	translate(CPUState*, addr, MemTxAttrs*) → hwaddr	cpu_get_phys_attrs_debug()	Virtual-to-physical address translation with debug context
read	read(AddressSpace*, hwaddr, MemTxAttrs, buf, len) → MemTx Result	address_space_read()	Read guest physical memory; invokes SEV DBG_DECRYPT if policy permits

write	write(AddressSpace*, hwaddr, MemTxAttrs, buf, len) → MemTx Result	address_space_write_rom()	Write guest physical memory; invokes SEV DBG_ENCRYPT if policy permits
pte_mask	pte_mask(void) → uint64_t	address_space_pte_mask()	In the default case, a bitmask is used to strip out flag bits from a page table entry so that only the physical address (PFN) portion remains during address translation. In the SEV-specific case, it additionally strips the C-bit from page-table entries to ensure correct physical address retrieval during the debug page-table walk.

Table 2: MemoryDebugOps Interface Callbacks and Their Roles

4.3 Runtime Flow of a Debug Memory Access

To understand how the proposed framework operates in practice within the SEV-enabled QEMU context, it is useful to examine the end-to-end flow of a debug-initiated memory access. The following sequence describes how a debugger request is processed in a confidential guest environment:

1. **Debugger Request Initiation:** A debugging tool such as the gdbstub or QEMU monitor issues a memory read or write request. This request is routed to the central debug entry point, cpu_memory_rw_debug().
2. **Debug Context Propagation:** Within cpu_memory_rw_debug(), the MemTxAttrs structure is initialized with the debug flag set to 1. This flag is propagated through all subsequent memory access calls, marking the operation as debugger-initiated.
3. **Dispatch via MemoryDebugOps:** Instead of directly invoking standard memory accessors, the request is routed through the MemoryDebugOps interface. Depending on the operation type, either debug_ops->read or debug_ops->write is invoked.
4. **Address Translation and Region Resolution:** The debug read/write handler performs virtual-to-physical address translation (optionally using the translate callback) and resolves the corresponding memory region.
5. **Conditional Region-Level Handling:** If the resolved region is backed by RAM and provides ram_debug_ops, and the debug flag is set, the framework invokes the region-specific debug callback instead of performing a direct memcpy().
6. **Firmware-Assisted Memory Access (SEV Case):** For encrypted guest memory, the region-level handler invokes SEV-specific routines, which ultimately issue a firmware command (DBG_DECRYPT or DBG_ENCRYPT) via the KVM interface. This step temporarily converts ciphertext into plaintext (or vice versa) under the control of the guest policy.
7. **Data Return to Debugger:** The decrypted data is copied into the debugger buffer and returned through the call chain back to the original requester (gdbstub or monitor).

This flow ensures that all debug memory accesses are transparently redirected through a controlled firmware-assisted or hypervisor-extension-assisted decryption path when required, while preserving the original QEMU memory-access behavior for non-debug operations.

This sequence corresponds to the logical flow illustrated in Figure 1

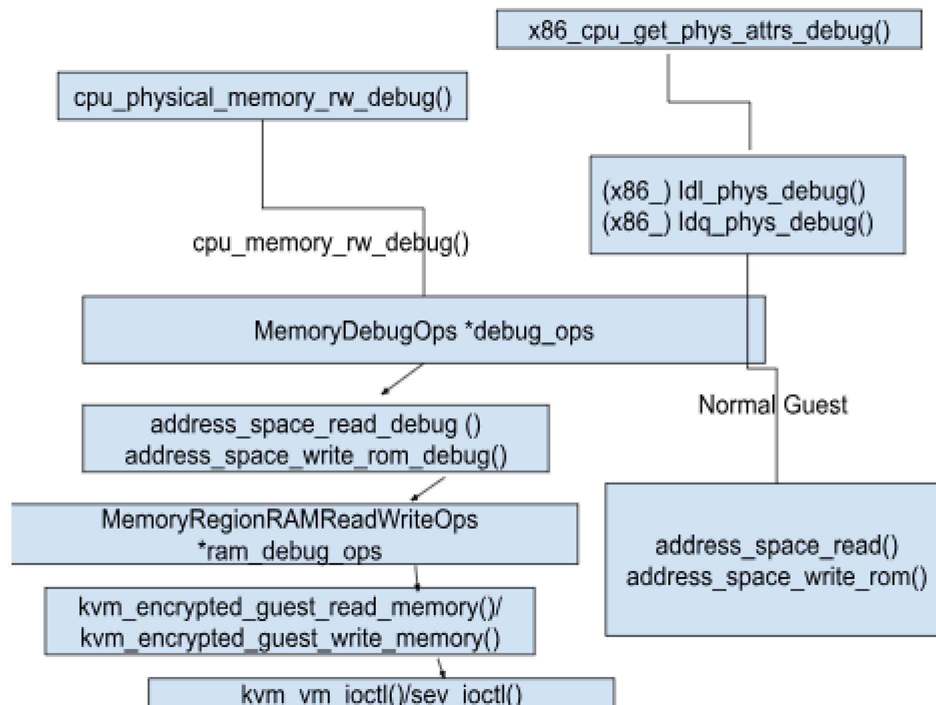

Figure 1: Relationship Diagram of the APIs and Interfaces for Confidential VM Debug Support

4.4 Memory Region RAM Read Write Ops: Region-Level Debug Callbacks

At a finer granularity, the framework also extends the MemoryRegion structure [7] by adding an optional pointer to a set of RAM-specific debug read/write callbacks.

```
struct MemoryRegionRAMReadWriteOps {
    int (*write)(uint8_t *dest, ..src, ..len, MemTxAttrs attrs);
    int (*read)(uint8_t *dest,  ..src, ..len, MemTxAttrs attrs);
};
struct MemoryRegion {
    /* ... existing fields ... */
    const MemoryRegionRAMReadWriteOps *ram_debug_ops;
};
```

Within the address translation and dispatch logic, after a physical address has been resolved to a host RAM pointer, the code checks whether the debug flag in MemTxAttrs is set and whether the resolved MemoryRegion has ram_debug_ops registered. If both conditions are true, the region-specific callback is invoked instead of memcpy(). Otherwise, the standard memcpy() path is taken:

```
ram_ptr = ...;
if (attrs.debug && mr->ram_debug_ops)
    mr->ram_debug_ops->read(buf, ram_ptr, l, attrs);
else
    memcpy(buf, ram_ptr, l);
```

This conditional branch is the only modification to the memory region dispatch path, and it is guarded by the debug flag, ensuring that non-debug accesses never encounter the additional branching.

4.5 Debug-Aware Physical Memory API Extensions

To ensure that all sites within QEMU that access guest physical memory for debugging purposes are routed through the new framework, the design proposes a set of additional debug-specific API functions:

```
cpu_physical_memory_read_debug()
cpu_physical_memory_write_debug()
cpu_physical_memory_rw_debug()
ldl_phys_debug()
ldq_phys_debug()
```

These functions are behaviorally equivalent to their non-debug counterparts, but they internally invoke the MemoryDebugOps dispatch layer rather than the general-purpose accessor functions. All existing call sites within QEMU that perform physical memory reads for debugging purposes, such as those in the page-table walking routines, are updated to call these debug-specific variants. The following example illustrates the substitution at a page-table walking call site:

```
/* Before */
cpu_physical_memory_read(pgd + l1*4, &pde, 4);
/* After */
cpu_physical_memory_read_debug(pgd + l1*4, &pde, 4);
```

5. Page-Table Walking in Encrypted Guests

5.1 The C-bit Problem

In AMD SEV-enabled guests, page-table entries (PTEs) include a C-bit (encryption bit) that marks whether the referenced page is encrypted [1, 3, 8]. When a VMM attempts to walk the guest page table to translate a virtual address to a physical address, it must correctly handle the C-bit: if the C-bit is left set during address computation, the resulting physical address will be incorrect, because the C-bit occupies a position within the physical address field of the PTE. This is not merely a theoretical concern; incorrect page-table walks will produce entirely wrong physical addresses, meaning that the debug framework would attempt to decrypt the wrong guest memory pages.

5.2 The pte_mask Callback

The pte_mask callback in MemoryDebugOps provides the architectural solution to this problem. When invoked, it returns a bitmask with the C-bit position cleared. During page-table walks performed for debugging, each PTE value is ANDed with this mask before the physical address field is extracted:

```
uint64_t me_mask;
me_mask = cpu_physical_memory_pte_mask_debug();  /* debug_ops->pte_mask() */
pdpe = le64_to_cpu(pdpe & me_mask);
/* similarly applied to pde and pte values */
```

This operation strips the C-bit and flag bits from the PTE, producing the correct guest physical address for subsequent memory access. In the default non-SEV case, the bitmask is used to strip out the flag bits from a page table entry so that only the physical address portion remains during address translation.

5.3 Virtual-to-Physical Translation Override

The translate callback in MemoryDebugOps is designed to allow the override of the CPU class's page-table walker with a vendor-specific implementation. For SEV guests, the function sev_cpu_get_phys_attrs_debug() is registered as the implementation of this callback by directly overriding the CPUClass method get_phys_page_attrs_debug [4]. This override

is performed conditionally: if the guest SEV policy includes the NODBG flag, the override is omitted, and debug operations are disabled [3]:

```
sev_set_debug_ops_cpu_state(..*handle, CPUState *cs)
{
    CPUClass *cc;
    if (s->policy & SEV_POLICY_NODBG) {
        return;
    }
    cc = CPU_GET_CLASS(cs);
    cc->get_phys_page_attrs_debug = sev_cpu_get_phys_attrs_debug();
    address_space_set_debug_ops(&sev_debug_ops);
}
```

This is designed to ensure that the debugging framework strictly respects the guest's declared security policy.

6. Handling Unencrypted Guest Memory Regions

Not all guest physical memory in a confidential VM is necessarily encrypted. Certain regions are explicitly designated as unencrypted, including software IOTLB bounce buffers, regions allocated through dma_decrypted() interfaces, and guest memory regions marked with the __bss_decrypted annotation [6]. For these regions, invoking the SEV DBG_DECRYPT firmware command would be incorrect: the firmware would attempt to decrypt data that is already in plaintext, producing corrupted output.

The framework therefore requires a mechanism, in the current SEV-based implementation context, to determine at the point of a debug memory access whether the target physical page is encrypted or not. This is expected to require an interface into the KVM hypervisor layer that can report the encryption status of individual guest memory pages [5][6]. For unencrypted regions, the debug dispatch logic must bypass the vendor-specific debug callbacks and fall back to the standard memcpy() path even when ram_debug_ops are registered.

This distinction is architecturally important because it means the framework cannot be a simple binary switch between encrypted and plaintext memory access. The dispatch logic must be aware of the encryption status of each individual memory region at the time of access. The precise mechanism for querying this status from KVM is identified as a necessary integration point, though the exact interface design requires further elaboration in conjunction with the KVM development community [5][6].

7. Encrypted Register State, Extended Debug Considerations, and Design Trade-offs

The confidentiality guarantees of SEV-ES extend beyond memory to encompass the CPU register state of the guest [9]. In SEV-ES, the guest's general-purpose registers, segment registers, and other architectural states are encrypted and are not accessible to the hypervisor through ordinary VMM mechanisms. The CPUClass structure in QEMU provides methods gdb_read_register() and gdb_write_register() that are invoked by the gdbstub whenever a debugger requests a register read or write [4]. For SEV-ES guests, the proposed architectural response to such requests is to reject them outright because the register state is not available to the hypervisor in plaintext form. The debug framework proposes hooks at the CPUClass level to intercept these register-access calls and enforce this rejection when SEV-ES is active [4][9]; however, this mechanism has not yet been implemented and remains a proposed extension to the framework. This design direction is deliberate: silently returning incorrect or ciphertext register data would be substantially more hazardous to a debugging session than an explicit refusal, as it could lead an analyst to draw entirely false conclusions about guest execution state. The proposed hook would therefore enforce a strict boundary at the register-access layer, mirroring the boundary already enforced at the memory-access layer by the MemoryDebugOps dispatch mechanism.

Several areas of the debug framework remain subjects of ongoing design discussion and represent open architectural problems that the current framework does not yet resolve and for which no implementation exists at this time. These include

the proposed mechanism for virtual INT1/INT3 injection for software breakpoint support in encrypted guests, the prospective role of the GHCB (Guest Hypervisor Communication Block) and the VC# exception handler in SEV-ES environments for communicating debug-related events between the guest and the hypervisor, and the question of guest-side awareness of debug sessions [4]. Each of these problems is structurally distinct from the memory-access and register-access concerns already addressed, as they would require coordinated design across the hypervisor, the firmware, and the guest operating system simultaneously. The GHCB protocol in particular introduces a communication channel between guest and host that would itself need to be designed with the confidentiality model in mind, since any proposed extension of that protocol for debugging purposes must not inadvertently leak guest state to an untrusted hypervisor in configurations where debugging has not been explicitly permitted.

The framework as a whole reflects several deliberate design trade-offs worth articulating explicitly, as they carry implications for any future extension. The decision to introduce a separate MemoryDebugOps dispatch layer, rather than modifying the existing address_space_rw() functions, reflects a strong separation-of-concerns principle rooted in the feedback received during the mailing list review [4, 5]. Debug functionality is inherently exceptional and episodic; it should not impose any overhead on the normal memory access path, which is performance-critical and invoked continuously during guest execution. By confining all debug-specific logic to a dedicated dispatch layer entered exclusively through cpu_memory_rw_debug() and the new debug-specific physical memory APIs, the framework is designed to ensure that the performance of ordinary guest memory access is completely unaffected. The use of a single global debug_ops pointer, rather than per-address-space or per-memory-region debug dispatch, reflects a complementary pragmatic choice: at any given time, a QEMU instance manages a single guest with a single memory encryption policy, making a global pointer sufficient to capture the vendor-specific behavior without introducing unnecessary indirection at finer granularities [4]. The finer-grained MemoryRegionRAMReadWriteOps callbacks then supply the additional granularity required to handle mixed encrypted and unencrypted memory within a single address space, preserving correctness without generalizing the dispatch mechanism beyond what the architecture requires. Finally, the policy-gated nature of the entire framework, mediated through the SEV NODBG policy bit, is designed to ensure that the security model of the confidential computing platform is fully respected at every layer [1, 3]. The framework is designed to propose no mechanism by which a hypervisor could access encrypted guest memory without the guest's explicit consent; this principle is a fundamental design invariant, not an implementation convenience, and it must be preserved in any future extension of the framework to additional confidential computing technologies or additional debugging modalities. The framework described reflects a specific design approach based on SEV-enabled QEMU environments and should not be interpreted as a complete or standardized debugging solution for all confidential computing platforms. Certain aspects, including unencrypted memory detection, device-level debugging, and guest-hypervisor coordination mechanisms, remain dependent on further integration with KVM, firmware interfaces, and guest-side support.

Conclusion

The debugging of confidential computing virtual machines presents a structurally novel challenge: the very security mechanism that protects guest workloads also prevents the hypervisor from performing the memory reads that its built-in debugging tools require. Conventional approaches, which rely on direct host-side access to guest physical memory through HVA pointer operations, are rendered non-functional when guest memory is encrypted by hardware. Addressing this challenge requires not merely a workaround but a principled architectural redesign of the VMM's debug memory-access path. The framework presented in this article achieves this redesign through a set of targeted, minimally invasive additions to the QEMU memory subsystem. The extension of MemTxAttrs with a debug flag is intended to provide a propagating signal that distinguishes debug-initiated memory accesses from ordinary ones throughout the entire call chain. The MemoryDebugOps structure proposes a vendor-neutral dispatch layer that can be overridden by any confidential computing technology to substitute firmware-assisted or hypervisor-extension-assisted memory access for the standard memcpy() path. The MemoryRegionRAMReadWriteOps callbacks provide the finer-grained per-region control necessary to handle mixed encrypted and unencrypted memory within a single guest address space. Together, these components form a coherent layered architecture that restores meaningful debugging capability while respecting the security boundaries mandated by the guest's confidential computing policy. This design gives rise to several architectural principles with broader applicability beyond confidential computing contexts. The maintainer feedback that shaped the final design made it clear that general-purpose code paths should not be changed. This is an important rule for any security-related extension to a performance-critical subsystem. For correctness, testability, and long-term maintainability, it is important to put hooks at well-defined entry points instead of random places in a complex call graph. The use of policy-gated activation, in which

the entire debug capability is withheld if the guest has explicitly prohibited debugging, demonstrates how the security invariants of the underlying hardware platform can be faithfully preserved in the VMM software layer. Several open problems remain subjects of active investigation. The correct handling of unencrypted memory regions within an otherwise encrypted guest address space requires a reliable mechanism for querying per-page encryption status from the KVM hypervisor, which has not yet been fully specified. The support for encrypted register state in SEV-ES environments adds additional complexity at the CPU class level. Device-level debugging, software breakpoint injection, and the integration of guest-side debug communication protocols represent further areas requiring architectural elaboration. Nonetheless, the framework described here provides a rigorous and extensible foundation upon which these extensions can be developed and provides a conceptual vocabulary for future standardization efforts across confidential computing platforms. The foundational design work underlying this framework was formally presented at the Linux Plumbers Conference 2021 Confidential Computing Microconference and represents an important step toward establishing community consensus on debugging interfaces for confidential computing virtual machines. The discussion is therefore intended to capture an implementation-driven architectural perspective rather than a universally standardized model. Future work should prioritize three directions: first, specifying the KVM interface required to query per-page encryption status for correct handling of unencrypted memory regions; second, extending the framework to support SEV-ES register-state interception and GHCB-based guest-hypervisor debug communication; and third, evaluating the framework's applicability to Intel TDX and other emerging confidential computing platforms. From a deployment standpoint, the framework has direct implications for cloud providers operating confidential VM infrastructure, enabling controlled, policy-gated debugging workflows that satisfy both operational and compliance requirements without weakening the security guarantees that tenants rely upon.

References

[1] Advanced Micro Devices Inc., "AMD Secure Encrypted Virtualization (SEV)," AMD Developer Documentation, 2023. Available: https://www.amd.com/en/developer/sev.html

[2] Intel Corporation, "Intel Trust Domain Extensions (Intel TDX)," Intel Developer Documentation, 2023. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html

[3] Advanced Micro Devices Inc., "Secure Encrypted Virtualization API Version 0.24," AMD Technical Information Portal, 2020. Available: https://docs.amd.com/v/u/en-US/55766_PUB_3.24_SEV_API

[4] Ashish Kalra, "SEV Guest Debugging Support for QEMU," QEMU Developer Mailing List, 2020. Available: https://www.mail-archive.com/QEMU-devel@nongnu.org/msg743405.html

[5] Paolo Bonzini, "Re: SEV Guest Debugging Support for QEMU," QEMU Developer Mailing List, 2020. Available: https://www.mail-archive.com/QEMU-devel@nongnu.org/msg744640.html

[6] Ashish Kalra, "Re: SEV Guest Debugging Support for QEMU," QEMU Developer Mailing List, 2019. Available: https://www.mail-archive.com/QEMU-devel@nongnu.org/msg745039.html

[7] QEMU Documentation, "The memory API," QEMU Official Documentation, 2023. Available: https://www.QEMU.org/docs/master/devel/memory.html

[8] Advanced Micro Devices Inc., "AMD64 Architecture Programmer's Manual Volume 2: System Programming (Publication #24593)," AMD Technical Information Portal, 2022. Available: https://docs.amd.com/v/u/en-US/24593_3.44_APM_Vol2

[9] David Kaplan, "PROTECTING VM REGISTER STATE WITH SEV-ES," AMD Technical Information Portal, 2017. Available: https://docs.amd.com/v/u/en-US/Protecting-VM-Register-State-with-SEV-ES

[10] Ashish Kalra, "Debug Support for Confidential VMs," Linux Plumbers Conference (Confidential Computing Microconference), 2021. Available: https://lpc.events/event/11/contributions/959/

[11] Mark Russinovich et al., "Towards Confidential Cloud Computing," Communications of the ACM, 2021. Available: https://dl.acm.org/doi/10.1145/3453930

[12] Sergei Arnautov et al., "SCONE: Secure Linux Containers with Intel SGX," in Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016. https://www.usenix.org/system/files/conference/osdi16/osdi16-arnautov.pdf

[13] Mohamed Sabt et al., "Trusted Execution Environment: What It Is and What It Is Not," in Proc. IEEE Trustcom/BigDataSE/ISPA, 2015. Available: https://ieeexplore.ieee.org/document/7345265