Secure IVSHMEM: End-to-End Shared-Memory Protocol with Hypervisor-CA Handshake and In-Kernel Access Control

Hyunwoo Kim
Intel
Seoul, South Korea
onion.kim@intel.com

Jaeseong Lee
Intel
Seoul, South Korea
jerry.j.lee@intel.com

Sunpyo Hong
Intel
Seoul, South Korea
brandon.hong@intel.com

Changmin Han
Intel
Seoul, South Korea
duke.han@intel.com

Abstract

In-host shared memory (IVSHMEM) enables high-throughput, zero-copy communication between virtual machines, but today's implementations lack any security control, allowing any application to eavesdrop or tamper with the IVSHMEM region. This paper presents *Secure IVSHMEM*, a protocol that provides end-to-end mutual authentication and fine-grained access enforcement with negligible performance overhead. We combine three techniques to ensure security: (1) channel separation and kernel module access control, (2) hypervisor-mediated handshake for end-to-end service authentication, and (3) application-level integration for abstraction and performance mitigation. In microbenchmarks, Secure IVSHMEM completes its one-time handshake in under 100 μs and sustains data-plane round-trip latencies within 5% of the unmodified baseline, with negligible bandwidth overhead. We believe this design is ideally suited for safety and latency-critical in-host domains, such as automotive systems, where both performance and security are paramount.

CCS Concepts

• **Computer systems organization** → *Embedded software*; • **Security and privacy** → **Access control**.

Keywords

IVSHMEM, Inter-VM Shared Memory, End-to-End Security

1 Introduction

The automotive industry is rapidly evolving, driven by advances in semiconductor technology that have shifted system architectures from traditional microcontrollers (MCUs) to powerful Systems-on-Chip (SoCs). This evolution not only enhances computational capabilities but also paves the way for Software-Defined Vehicles (SDVs), where flexibility, scalability, and rapid updates are paramount. In SDVs, virtualization technology plays a crucial role by enabling the coexistence of multiple virtual machines (VMs) on a single hardware platform, ensuring isolated yet efficient execution of diverse applications. For example, modern cockpit domain controllers often deploy separate VMs for real-time operations (RTOS) and infotainment systems, which is essential for balancing performance and safety[6, 15].

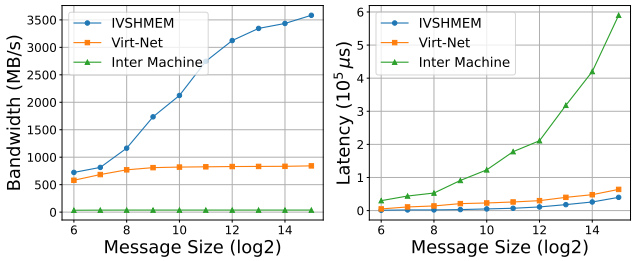

Figure 1: A performance comparison between IVSHMEM and other inter-VM communication methods.

Inter-VM communication in these environments is critical. Traditional approaches, such as TCP/UDP over a network stack or even UART-based messaging, often lack sufficient speed and resource efficiency[16]. Alternative solutions like VirtIO offer a paravirtualized communication mechanism through ring buffers (VirtQueues), but they do not fully leverage the benefits of shared physical memory.[18] IVSHMEM(Inter-VM Shared Memory) addresses these limitations by mapping each VM's virtualized PCI device to a common physical memory region, allowing rapid data exchange through shared memory[9, 16, 27]. As illustrated in Figure 1, IVSHMEM achieves substantially higher bandwidth and lower latency compared to VirtIO-based networking (Virt-Net) and inter-machine TCP communication over Ethernet, highlighting its superior performance for inter-VM communication. Despite its performance advantages, this method introduces significant security challenges; multiple VMs accessing the same memory space creates vulnerabilities where a compromised or malicious VM could potentially access or modify data belonging to another VM[22].

This concern is particularly acute in scenarios where critical systems interact with less secure environments. For instance, when an RTOS communicates with an Android-based infotainment VM, there is a tangible risk that a malicious application within Android might tamper with the shared memory region[15]. Such tampering could result in attacks ranging from man-in-the-middle to eavesdropping, ultimately compromising system stability and safety.

In response to these challenges, we propose a secure protocol designed specifically for IVSHMEM communication. Our approach

introduces robust security measures on top of the IVSHMEM framework, ensuring data integrity and access control even in an environment with inherent vulnerabilities. While our protocol does introduce some performance overhead, we have implemented techniques to mitigate this impact, ensuring that the overhead remains minimal relative to the performance gains achieved by shared memory communication.

In this paper, we provide a detailed analysis of the security threats associated with IVSHMEM, explore the limitations of existing inter-VM communication methods, and describe our protocol's architecture and mitigation strategies. Through comprehensive evaluation, we demonstrate that our secure protocol successfully balances between robust security and the high-performance demands of IVSHMEM applications—such as those found in modern automotive systems.

2 Background

In this section, we examine the IVSHMEM mechanism, outline its challenges, and review recent progress.

2.1 IVSHMEM: Mechanism and Architecture

IVSHMEM is a specialized implementation of shared memory IPC designed for virtualized environments. It emulates a virtual PCI device to expose the shared memory's base address and size to guest VMs[5, 13]. As shown in Figure 2, the hypervisor exposes the shared-memory region to the guest VM, and the UIO kernel driver maps the emulated PCI device, allowing applications to access that memory directly via `/dev/uioX`. The design leverages the standardized PCI configuration to facilitate memory mapping and efficient communication. Specifically, IVSHMEM utilizes:

- **BAR0 (Base Address Register 0):** This region (256 bytes of MMIO) holds the device registers, which control the operation of the virtual device.
- **BAR1:** It contains the MSI-X table and Pending Bit Array (PBA), primarily used by the IVSHMEM doorbell mechanism for signaling interrupts.
- **BAR2:** This is mapped to the shared memory object, providing a direct communication channel between VMs.

The doorbell interrupt mechanism enabled by this configuration allows VMs to notify one another when new data is available, ensuring efficient core utilization and reducing latency in inter-VM communication[7].

2.2 Security Concerns of Shared Memory

Shared-memory IPC within a single OS benefits from well-established protections, including file access controls, sandboxing, and security modules such as SELinux or AppArmor. IVSHMEM, however, presents distinct challenges [1, 21]. In a traditional OS environment, the OS enforces strict access controls over shared memory regions, ensuring that only authorized processes can read or write data. However, these protections doesn't work when multiple, potentially untrusted VMs share the same memory space. A compromised or malicious VM could easily access or tamper with data in the shared region, leading to unauthorized data disclosure, corruption, or even system instability[22].

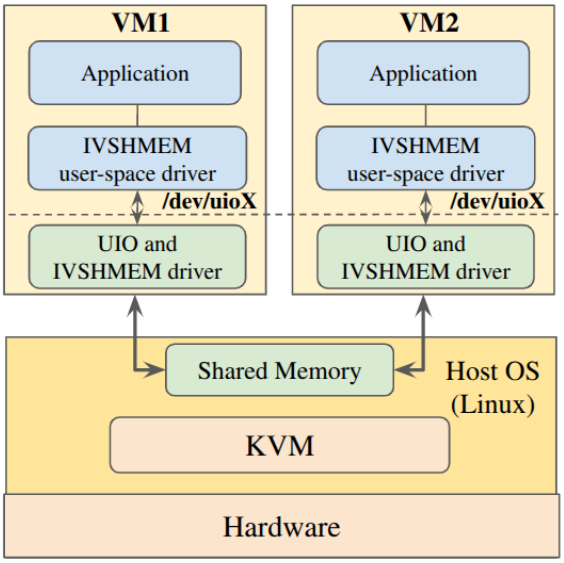

Figure 2: The overview of IVSHMEM architecture

2.3 Secure Communication Over Insecure Channels

The challenge of ensuring secure communication in IVSHMEM environments is analogous to securing communication over the Internet, where multiple parties exchange information over an inherently insecure channel. In network communications, protocols such as TLS rely on key exchange mechanisms, mutual authentication, and end-to-end encryption to safeguard data integrity and confidentiality[17, 19]. Similarly, secure multi-party communication techniques, such as Diffie-Hellman key exchange and advanced encryption standards, are employed to establish trust even when the channel is compromised[3, 14].

In the context of IVSHMEM, the situation is even more complex because multiple services must share the same restricted memory space as a communication channel. This necessitates designing a secure protocol that ensures confidentiality and integrity, similar to TLS or other network security protocols, while also accommodating the shared nature of the memory resource. Our research addresses these challenges by proposing a secure protocol that protect the data transmitted via IVSHMEM, while also mitigating the performance overhead typically associated with such security measures.

2.4 Recent Progress of IVSHMEM Communication

Recent research and developments in IVSHMEM communication have focused on balancing performance and security, with various approaches having distinct trade-offs.

The SIVSHM project introduces a segmentation approach to IVSHMEM, enhancing security by isolating shared memory regions among VMs. However, this strict isolation introduces notable overheads from reduced buffer size. [22].

Performance-centric solutions, such as XenLoop and MemPipe, have been proposed to optimize IVSHMEM by improving transparency and reducing latency[29][16][28][24][2][10]. These solutions integrate seamlessly with traditional socket-based network stacks, allowing applications to benefit from high-speed shared memory communication without explicit changes. However, these solutions primarily prioritize performance and transparency and lack of mechanisms for security threats[16].

Other approaches have leveraged hypervisor-managed policies and features, like Xen's grant tables, to enforce finer-grained security controls. Grant tables establish explicit, controlled memory-sharing agreements between VMs, restricting access to designated regions. However, this technique is tied to a specific hypervisor and does not provide a general-purpose driver interface [26].

Overall, while significant progress has been made in enhancing both the performance and security of IVSHMEM, current solutions optimize for one at the expense of the other. We aim to design a Secure IVSHMEM protocol that delivers security features while keeping performance overhead to a minimum.

3 Threat Model

In this section we define the assets to be protected, the adversary's capabilities, our trust assumptions, concrete threat scenarios with corresponding defenses, the security goals achieved, and known limitations.

3.1 Assets

- **Shared-Memory Contents:** All plaintext data exchanged via the IVSHMEM region (e.g., sensor readings, control commands).
- **VM Identities:** Certificates and private keys provisioned to each VM by the hypervisor CA.

3.2 Adversary Model

We consider an attacker with the following capabilities and goals:

Location: Co-resident on the same host, either in another VM or with limited host privileges.

Privileges in Guest: May be an unprivileged process or even gain root in one VM, and thus can open and attempt to `mmap()` the IVSHMEM region directly.

Goals:

(1) *Confidentiality breach:* Read plaintext data from another VM's IVSHMEM region.
(2) *Integrity breach:* Inject or tamper with messages in the shared region.
(3) *Authentication breach:* Impersonate a VM by forging or replaying handshake messages.

3.3 Trust Assumptions

- **Trusted Hypervisor:** Each VM trusts the hypervisor as the root of trust; although VMs do not inherently trust the IVSHMEM communication channel, they rely on the hypervisor acting as a Certificate Authority (CA) to issue, sign, and validate VM certificates.
- **Kernel-Module Enforcement:** All VMs in the system load and execute the same IVSHMEM enforcement kernel module, ensuring uniform, in-kernel access control and preventing any unauthorized memory mappings across the entire platform.

3.4 Security Goals

Under the above model and countermeasures, our protocol achieves:

(1) **Confidentiality:** No application in VM can read another's channel's plaintext data.
(2) **Integrity:** Any tampering with shared data is detected by authentication tags.
(3) **Mutual Authentication:** Only application in VMs with valid, hypervisor-signed certificates complete the handshake.

3.5 Limitations of Conventional Security Protocols

Conventional end-to-end security protocols such as TLS, IPsec, or DTLS are ill-suited to the IVSHMEM use-case for several reasons:

(1) **Performance Overhead:** Conventional TLS requires symmetric encryption and decryption on each record, which breaks IVSHMEM's zero-copy path and forces additional data copies and context switches[17][11]. In a high-throughput IVSHMEM environment—where direct page mappings sustain multiple gigabytes per second—this per-record crypto overhead introduces unacceptable latency and CPU load.
(2) **Limited Shared-Memory Capacity:** Unlike network channels, IVSHMEM regions are fixed and small (e.g., 1 MiB)[13][5]. To prevent a malicious VM or service from overwhelming the shared-memory resource, the hypervisor must strictly assign and enforce per-service channel quotas. Conventional socket-based protocols provide no mechanism for hypervisor-driven, size-limited region allocation or fine-grained resource control.
(3) **Inadequate Fit for End-to-End Schemes:** Conventional end-to-end protocols (e.g., IPsec, SSH, DTLS) assume a network stack with IP addresses, ports, and hostnames or DNS names to establish and authenticate channels. IVSHMEM operates entirely in-host via PCI BAR mappings without any network identifiers, so these protocols cannot provide true end-to-end security for shared-memory communication or integrate with hypervisor-managed channel assignment.

Conventional end-to-end protocols such as TLS, IPsec, and DTLS are ill-suited for IVSHMEM communication because they (1) impose per-record cryptographic overhead that breaks zero-copy performance[11, 20], (2) offer no mechanism to enforce fixed, size-limited shared-memory quotas, and (3) depend on network-layer identifiers (e.g., hostnames, IP addresses) while lacking support for hypervisor-driven channel assignment[19]. Securing IVSHMEM therefore requires a specialized protocol that leverages IVSHMEM's in-host, fixed-region semantics and hypervisor control, providing end-to-end protection with minimizing additional performance overhead.

4 Design Proposal

In this section, we present the Secure IVSHMEM design, which aims to provide dedicated, zero-copy shared-memory channels between VM services while preventing unauthorized access, spoofing, and impersonation. Our approach combines three complementary mechanisms: *service-based channel separation* to allocate isolated IVSHMEM regions per service pair, *granular kernel-module enforcement* to block any unauthorized `open`, `mmap`, or I/O operations, and a *hypervisor-mediated mutual-authentication handshake* to establish trust on channel setup. Together, these components ensure end-to-end security with negligible impact on IVSHMEM's high-performance communication.

4.1 Service-based Channel Separation

As illustrated in Figure 3, our design divides the IVSHMEM architecture into two primary sections: the **Control Section** and the **Data Section**. The Control Section is a fixed-size region where a trusted host stores dynamic configurations related to data allocation, while the Data Section is where the service in virtual machines (VMs) actually read and write data through their assigned channels. Importantly, only the trusted host has permission to modify data in the Control Section, and each VM is restricted to reading and writing only to its designated channels within the Data Section.

The **Data Section** comprises multiple channels, with each channel serving as a dedicated buffer space for a specific server and client service pair. For each pair, a dedicated channel is allocated, and the Control Section dynamically adjusts its size based on the activation of channels.

For example, consider a scenario where Service A in VM1 needs to send data to Service B in VM2. In this case, the trusted host allocates an initial channel with a buffer size of 512 KiB. The control information for this allocation is written into the Control Section, and only Service A and Service B are permitted to access the channel's buffer. Additionally, the size of the channel buffer can be adjusted based on the usage patterns between the services.

An exception to this rule is the first channel in the Data Section. This channel is of a fixed-size and is exclusively used for communication between the trusted host and the VMs, such as during the initial handshake when a VM sends data to the trusted host. All VMs have access to this channel.

The **Control Section** maintains all of the metadata needed for buffer allocation and channel management. It tracks the next free offset, the number of active channels, and protects updates with a lock. Per-channel metadata (service IDs, process IDs, buffer addresses and sizes) is stored in an internal array. Channels use this information to coordinate reads and writes to their assigned regions.

4.2 Granular Kernel Module Enforcement

To enhance the security of the IVSHMEM framework, we propose a granular access control mechanism that restricts access to the shared memory channels on a per-application basis. This mechanism is implemented via a dedicated kernel module that operates on top of the IVSHMEM device driver.

4.2.1 Kernel Module Integration. Our kernel module hooks all IVSHMEM-related system calls, including `open`, `read`, `write`, and `mmap`, as well as any I/O control operations targeting the IVSHMEM device. On each intercepted call, the module retrieves the caller's `service_id` and checks the Control Section's metadata to verify that this service identifier matches the channel being accessed. If the `service_id` does not correspond to that channel's assigned service, the module denies the operation. This enforcement ensures that only the authorized host or VM service can interact with its designated shared-memory region.

4.2.2 Channel-Specific Enforcement. Each channel within the Data Section is allocated to a specific pair of services (e.g., a server and a client). The kernel module uses the control section's metadata to determine channel assignments and enforces strict access control, permitting operations only on the designated channel buffers.

4.3 Hypervisor-Mediated Mutual-Authentication Handshake

A secure, hypervisor-mediated handshake is essential for IVSHMEM because it protects against spoofing and impersonation on both sides of the shared-memory channel. In our model, the trusted host (e.g., dom0 in Xen, SOS in ACRN[8], or the host in QEMU/KVM) cannot simply trust that any service presenting a request truly owns its claimed endpoint; similarly, a VM service cannot blindly accept control messages or credentials from the host without risk of impersonation. By embedding a mutual-authentication handshake into the Control Section—that is, having each party present and verify cryptographic credentials tied to its service id—the host first confirms that the requesting service is one of its pre-registered, trusted entities, and then the VM service validates that the host's response really comes from the genuine hypervisor authority. Only once both directions of identity proof succeed do we allocate a dedicated, zero-copy channel in the Data Section. This two-way validation thwarts malicious actors on either side and ensures end-to-end trust before any IVSHMEM communication occurs.

Our proposed protocol is different from the conventional internet based security protocol in that 1) The trusted host (hypervisor) is not merely a passive participant but is responsible for allocating finite resources and establishing the communication channel and 2) The hypervisor functions as a certification authority (CA)[12], validating service credentials and orchestrating the creation of dedicated secure channels between clients and servers.

The detailed handshake protocol steps are provided below, demonstrating the mutual authentication processes that lead to the establishment of a confidential IVSHMEM channel.

4.3.1 Protocol Steps (Fig. 4).

1. **Client Hello:**
   - **Purpose:** Initiate the handshake and propose communication parameters.
   - **Message Contents:** protocol version & extensions, supported cipher suites, client identity (service ID, PID, VM ID), nonce + timestamp for replay protection.
2. **Trusted Host Hello:**
   - **Purpose:** Acknowledge the client's request and provide trusted credentials.

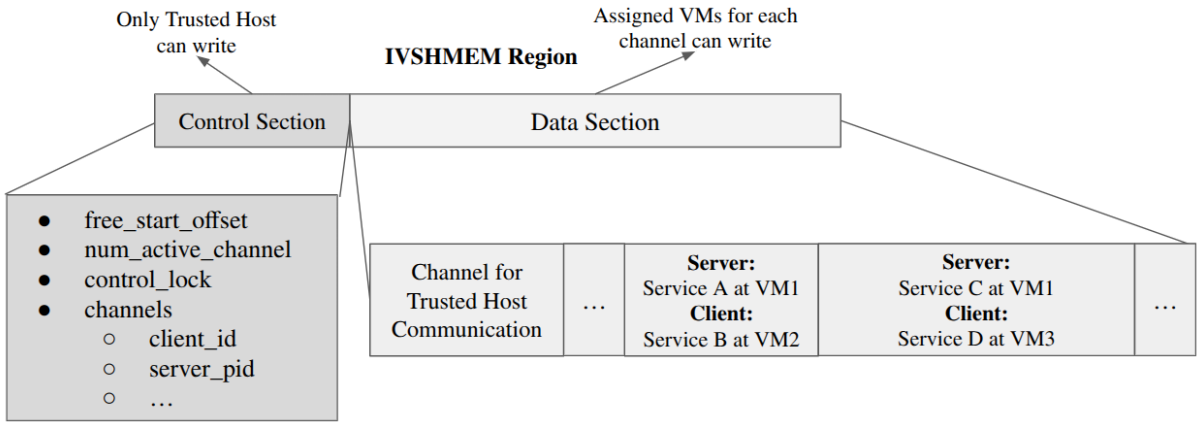

Figure 3: Service-based channel separation in Secure IVSHMEM: the trusted host controls metadata in the Control Section and is assigned its own channel in the Data Section for host–VM communication, while each service pair communicates over its dedicated Data Section channel.

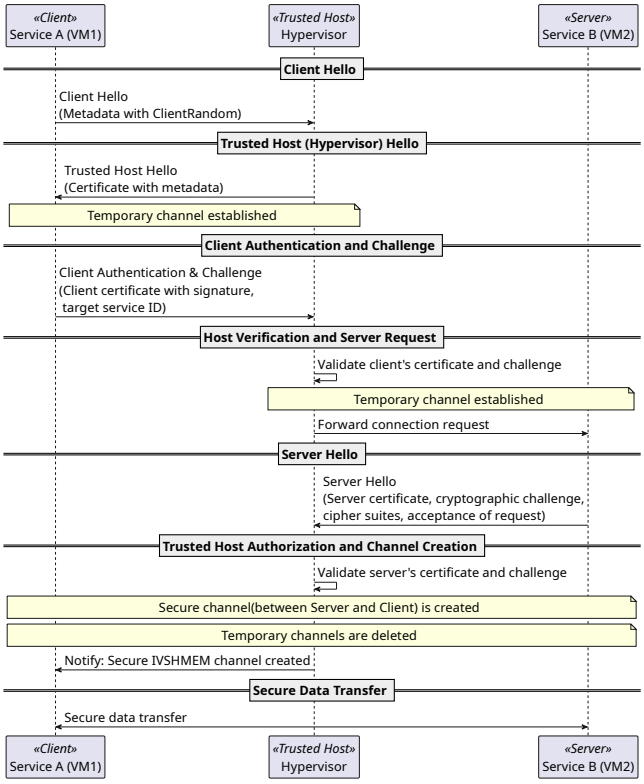

Figure 4: Secure IVSHMEM handshake flow: eight steps from Client Hello through Host authorization to establishment of the dedicated shared-memory channel

- **Message Contents:** host certificate (± selected ciphers);
- **Action:** A temporary secure channel is created between the client and the trusted host.

3. **Client Authentication and Challenge:**
   - **Purpose:** Enable explicit mutual authentication.
   - **Message Contents:** The client certificate, signature over nonce, target server's service ID.
4. **Host Verification and Server Request:**
   - **Purpose:** Verify the client's credentials and initiate communication with the server.
   - **Action:**
     - The trusted host validates the client's certificate and challenge.
     - Upon successful validation, the trusted host creates a temporary channel for the server and forwards a secure connection request to the server, including the nonce.
5. **Server Hello:**
   - **Purpose:** Server passes its credentials and decision to accept the client's request.
   - **Message Contents:** server certificate, signature challenge, supported ciphers, acceptance flag.
6. **Trusted Host Authorization and Channel Creation:**
   - **Purpose:** Establish a secure, dedicated IVSHMEM channel for data transfer between the server and client.
   - **Action:**
     - The trusted host verifies the server's certificate and the corresponding challenge.
     - The trusted host creates a communication channel for the server and client and notifies the client.
     - The trusted host deletes temporary channels previously established with both the client and the server.
7. **Secure Data Transfer:**
   - **Purpose:** Enable protected communication.
   - **Action:** The client and server commence secure data transfer over the authorized IVSHMEM channel.

8. **Session Management (Optional Enhancements):** Implement mechanisms (similar to TLS session tickets) for efficient session resumption without repeating the full handshake process.

4.4 Ring Buffer and Anticipation Timing Window

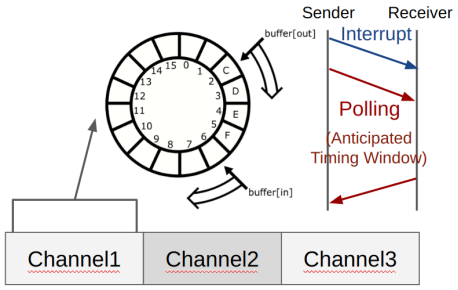

Figure 5: The IVSHMEM protocol is optimized using a Ring Buffer and an Anticipation Time Window mechanism.

To enhance performance in our IVSHMEM protocol design, we adopt two key mechanisms: a ring buffer and an anticipation timing window. Ring buffers are widely used in shared memory communication due to their low-overhead, lock-free design, which supports efficient producer-consumer patterns. This structure allows continuous data exchange without the need for frequent synchronization, thereby improving throughput and reducing contention.

In addition, we incorporate the anticipation timing window mechanism, originally proposed in [28], to mitigate the overhead associated with frequent interrupt handling. Rather than triggering an interrupt (IRQ) for every data transmission, which incurs significant system-wide context switching overhead, the receiver VM periodically polls the shared buffer for a predefined time window. This approach reduce interrupt cost while maintaining responsiveness, ultimately reducing communication latency and improving overall efficiency.

The performance benefits of these optimizations are evaluated in detail in Section 6.

5 Implementation

In this section, we implment Secure IVSHMEM through three components: a kernel module that hooks into the UIO PCI driver[25] to enforce per-channel access control, a user-space OpenSSL-based mutual-authentication handshake over the control page, and a BSD-socket-style library for zero-copy ring-buffer data transfers on authenticated channels.

5.1 Kernel-Module Integration via Dynamic Hooks

We build on top of the existing UIO PCI driver for IVSHMEM by inserting a lightweight kernel module that intercepts system calls(`mmap()`, `read()` and `write()`)to enforce per-channel access control.

Hook Implementation. Using a combination of kprobes and ftrace[4], our module attaches to the IVSHMEM driver's `mmap()` entry point. In the hook we:

- Extract the `vma` pointer from the CPU registers.
- Compute the requested mapping's channel ID and range.
- Look up the authorized-PID list stored in the IVSHMEM control section.
- If `current->pid` is not present or the channel is not marked `AUTHORIZED`, force-return `-EPERM` and skip the real handler.

Policy Management. An in-kernel hash table of `policy_entry` structs—keyed by channel_id—tracks which PIDs are permitted each channel. The hypervisor populates this table at boot, and upon handshake completion a simple ioctl marks the corresponding entry as `AUTHORIZED`.

Cleanup. On module unload or VM teardown, all probes are unregistered and the policy table cleared, restoring the original UIO driver behavior.

This dynamic-hook approach adds minimal overhead, preserves zero-copy data mappings for authorized clients, and requires no modification to the upstream IVSHMEM driver.

5.2 Handshake Implementation

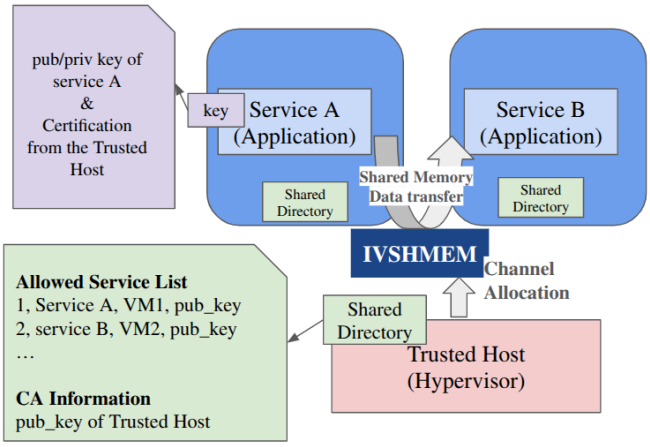

Figure 6: Credential management: the hypervisor publishes a CA certificate and per-service public keys in the Allowed Service List for Secure IVSHMEM

As illustrated in Figure 6, the mutual-authentication handshake is implemented entirely in user-land using OpenSSL[23] and leverages a IVSHMEM as an in-host transport. During the initial setup, the hypervisor generates a 4096-bit RSA CA key and self-signed certificate. It then builds an *allowed-service list* that maps each (service_ID, VM_ID) pair to its corresponding public key, and publishes that list together with the CA's public certificate to all VMs via a shared directory. Next, the hypervisor issues 2048-bit RSA key pairs for each registered service, retaining the public key in its allowed-service list and provisioning the private key to the guest VM. Each service in the VM uses this private key to generate its certificate when it performs the handshake. At runtime, host and VM

Scenario	Attack Flow	Countermeasure
Eavesdropping & Unauthorized Mapping	Rogue VM or process calls `open`/`mmap` on IVSHMEM and reads raw data from an unassigned channel before authentication.	Kernel module enforces per-channel access control, blocking any `open`, `mmap`, or I/O until the channel is marked `ESTABLISHED` by the handshake.
Replay & Man-in-the-Middle Attacks	Adversary captures or intercepts handshake messages (certificates, nonces) and replays or tampers with them to hijack or impersonate a service.	Mutual-authentication handshake with CA-signed certificates and nonce-based challenge prevents replay and ensures both endpoints verify peer identity.

Table 1: Key threat scenarios and corresponding Secure IVSHMEM defenses

exchange certificates over the IVSHMEM control channel, verify each peer's certificate against the CA and the allowed-service list, and send signed acknowledgments to confirm mutual credential validation. Once both sides have verified the signature, the VM mark the kernel module that the channel is `ESTABLISHED`, enabling zero-copy operations under the authenticated session.

5.3 Application Library Implementation

To simplify adoption of Secure IVSHMEM, we provide a lightweight user-land library with APIs analogous to BSD sockets:

- `ivshmem_listen(vm_id, service_id)` – being ready to accept connections from target on a dedicated IVSHMEM channel
- `ivshmem_connect(vm_id, service_id)` – initiate a connection to the target, performing the handshake over the control page
- `ivshmem_send(buf, len)` – copy `len` bytes from `buf` into the ring buffer slots of the established channel and ring the doorbell
- `ivshmem_receive(buf, len)` – poll the ring buffer for new data, copy up to `len` bytes into `buf`

Under the hood, `ivshmem_listen` and `ivshmem_connect` map the `(vm_id, service_id)` tuple to a hypervisor-assigned PCI BAR channel, then carry out the certificate exchange and mutual validation over the IVSHMEM control region.

For payload transfers, the library allocates a ring buffer within the IVSHMEM data section so that producers and consumers operate without blocking:

1. The sender writes into its next available slot in the ring buffer and triggers a doorbell interrupt. 2. The receiver, polling the doorbell and head pointer, copies the data into its local buffer and advances the consumer index. 3. A secondary doorbell notifies the sender that the slot is free.

By combining zero-copy ring buffers with doorbell interrupts followed by brief polling, this API achieves near-native IVSHMEM performance while enforcing end-to-end authentication. We evaluate its throughput in Section 6.

6 Measurements

The experiments were run on two Linux guest VMs, each using the 6.12.10-0-lts kernel and provisioned with 2 GiB of RAM and 4 vCPUs. The host is an x86_64 machine with an Intel® Core™ Ultra 7 155H (VT-x enabled, 400 MHz-4.8 GHz) and 32 GiB of DDR memory. To minimize interference, each VM was pinned to its own physical cores, and both the control channel and IVSHMEM devices were instantiated via QEMU's full-virtualized interfaces

6.1 Latency Overhead for Initial Handshake

We record the one-time handshake cost—from the initial `ivshmem_connect()` call through certificate exchange and verification until the first confirmation—and then measure steady-state data-plane round-trip latency for each 32-bit write (plus doorbell notification), comparing vanilla IVSHMEM to Secure IVSHMEM.

Operation	Vanilla (µs)	Secure (µs)
Initial Handshake	-	90
Round-Trip Transfer	8.1	8.4

Table 2: Round-trip latency for a single 32-bit integer. Secure IVSHMEM incurs a 90µs handshake cost, but per-message latency afterward is within 5 % of the vanilla baseline.

As shown in Table 2, although the initial handshake adds a modest 90µs one-time overhead, the steady-state data-plane latency (8.4 µs) is almost identical to vanilla IVSHMEM (8.1 µs). This shows that, although the initial handshake introduces some latency, our security mechanism adds negligible per-message overhead once the session is established.

6.2 Kernel Module Enforcement Overhead

For each experiment, a total of 32 GiB of random data was transferred to evaluate raw throughput and quantify the overhead introduced by granular access control via a kernel module.

Figure 7 compares the performance of IVSHMEM with and without access control, across message sizes ranging from 2^6 to 2^{15} bytes. For small messages (<= 2^8 B), the integration of kernel-module hooks incurs a throughput reduction of approximately 20–25%. However, this overhead diminishes rapidly with increasing message size. For transfers >= 1 KiB, the bandwidth of Secure IVSHMEM remains within 5% of the unmodified baseline, indicating that the additional hooking logic imposes minimal overhead at larger scales.

In terms of latency, the kernel-module integration introduces an overhead of up to 25–30% for small messages, but this impact also decreases as the message size grows. For messages ≥ 1 KiB, the

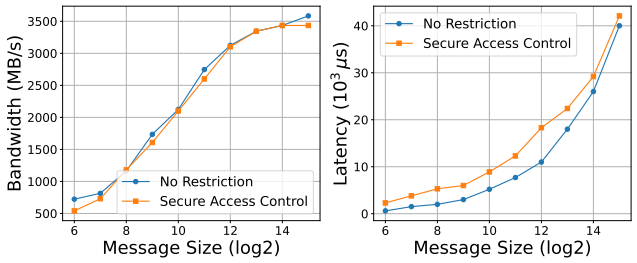

Figure 7: Bandwidth comparison of Secure IVSHMEM Protocol versus Vanilla IVSHMEM.

latency overhead drops to below 10%, demonstrating that the cost of access control becomes negligible for larger payloads.

6.3 Performance Optimization

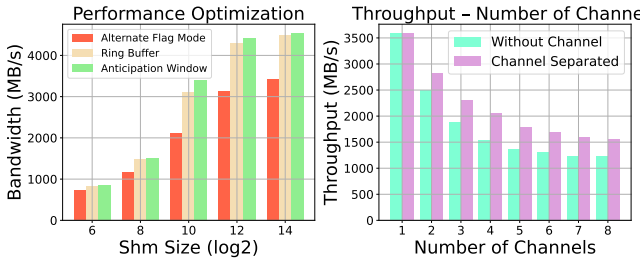

Figure 8: Performance improvements achieved through the use of a ring buffer and anticipation timing window. Channel separation further enhances bandwidth in multi-producer, multi-consumer scenarios

Figure 8 compares the performance of three IVSHMEM communication designs across increasing shared memory sizes: a basic message-passing scheme, a ring-buffer implementation, and a design enhanced by an anticipation timing window.

The results demonstrate substantial performance gains from both optimizations. Applying the ring-buffer architecture—which leverages batching and zero-copy techniques—reduces system call and memory copy overhead and increases bandwidth by more than 30% once the shared-memory size reaches $\geq 2^{10}$ bytes, with an improvement of approximately 46% at 2^{10} and sustained gains above 30% at larger sizes. The anticipation timing window further improves throughput by minimizing the cost of frequent interrupt handling, delivering performance boosts of at least 4% across all tested sizes. Together, these enhancements enable more efficient use of shared memory and higher communication bandwidth as the message size increases.

Additionally, one of the interesting things is that channel separation brings additional performance gains in multi-producer, multi-consumer scenarios. Figure 8 depicts throughput as a function of the number of concurrent producer/consumer pairs (separate IVSHMEM channels). With a single channel, both vanila and Secure IVSHMEM sustain $\sim$ 3.58 GiB/s. As the channel count increases to eight, per-channel throughput decreases to $\sim$ 1.56 GiB/s for vanilla and $\sim$ 1.23 GiB/s for the secure variant. This performance gain stems from reduced lock contention and parallel, lock-free processing across channels.

In summary, although our Secure IVSHMEM protocol incurs a one-time handshake latency when establishing each channel, it introduces negligible per-message overhead in both latency and bandwidth once the channel is established; moreover, channel separation combined with a ring-buffer and anticipation timing window delivers additional scalability and throughput gains in multi-producer/multi-consumer scenarios by reducing lock contention.

6.4 Security Validation Experiments

To validate that our kernel-module enforcement reliably blocks unauthorized access, we designed three attack scenarios reflecting realistic bypass attempts. Each scenario was exercised 30 times on our testbed with the Secure IVSHMEM module active:

Out-of-Bounds Data Access *Attack:* `mmap()` requests a data range that lies outside the limits recorded in the control-section metadata.

Control-Section Access Violation *Attack:* attempt to read via syscall from the read-only control section.

Impersonation Attack *Attack:* handshake request using an invalid or replayed credential (wrong service ID or stale nonce).

In all three cases, the kernel module returned `-EPERM` and no pages or credentials were granted. A valid data-section mapping (exactly matching the bounds in control metadata) succeeded and was cleanly unmapped.

Test Scenario	Attempts	Blocked (%)
Out-of-Bounds Data Access	30	30 (100%)
Control-Section Access Violation	30	30 (100%)
Impersonation Attack	30	30 (100%)

Table 3: Invalid Access Test Results: all invalid attempts were correctly blocked.

7 Discussion

Hypervisor Independence. Our Secure IVSHMEM protocol and its kernel-module integration are hypervisor-agnostic. Both the handshake mechanism and the IVSHMEM driver can be deployed on any virtualization platform that supports UIO and the IVSHMEM device, including ACRN and Xen.

Dynamic Channel Buffer Allocation. In our current prototype, each channel's buffer is allocated as a single contiguous region for simplicity. To reduce external fragmentation and improve memory utilization, a page-based or scatter/gather buffer allocation scheme could be adopted.

Key Exchange and Symmetric Encryption. While we focus here on authentication and integrity, confidentiality could be added via symmetric encryption. A TLS-style key-exchange (for example, ephemeral Diffie-Hellman over the control channel) would introduce only a modest one-time handshake delay and encryption overhead sacrificing zero-copy data-plane performance.

Transparency. Our protocol and kernel module require applications to link against the IVSHMEM-specific library and invoke its API, rather than using standard socket or networking calls. Achieving full transparency[24, 24, 28, 29]—so that unmodified applications could communicate over IVSHMEM as if it were TCP/IP or any other network transport—is beyond the scope of this work and is left for future exploration (e.g., via socket-API interposition or hypervisor-level redirection).

8 Conclusion

We have presented Secure IVSHMEM, a protocol that delivers end-to-end authentication and integrity for in-host shared-memory channels without sacrificing zero-copy performance. By treating the hypervisor as a trusted CA and implementing a hypervisor-mediated mutual-authentication handshake, our design prevents spoofing and impersonation attacks, while dynamic kernel-module hooks enforce fine-grained channel access control. Microbenchmarks show that the one-time handshake incurs less than 100 µs latency and that subsequent data transfers achieve near-native IVSHMEM throughput and $\leqslant$ 5% round-trip latency overhead. Thus, Secure IVSHMEM is well-suited for safety critical, high performance environments such as automotive systems, where untrusted services share memory in the same host.

References

[1] AppArmor. [n. d.]. *AppArmor Docs.* https://gitlab.com/apparmor/apparmor/-/wikis/Documentation

[2] François Diakhaté, Marc Perache, Raymond Namyst, and Herve Jourdren. 2008. Efficient shared memory message passing for inter-VM communications. In *European Conference on Parallel Processing*. Springer, 53–62.

[3] Whitfield Diffie and Martin E Hellman. 2022. New directions in cryptography. In *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*. 365–390.

[4] Mohamad Gebai and Michel R Dagenais. 2018. Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead. *ACM Computing Surveys (CSUR)* 51, 2 (2018), 1–33.

[5] Intel Corporation. [n. d.]. *Enabling Inter-VM Shared Memory Communication.* https://projectacrn.github.io/latest/developer-guides/hld/ivshmem-hld.html

[6] Lin Jiang, Feiyu Zhang, and Jiang Ming. 2024. Towards intelligent automobile cockpit via a new container architecture. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 205–219.

[7] Akber Kazmi. 2004. PCI Express and Non-Transparent Bridging Support High Availability. *Embedded Comping Design* (2004), 1–4.

[8] Hao Li, Xuefei Xu, Jinkui Ren, and Yaozu Dong. 2019. ACRN: a big little hypervisor for IoT development. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 31–44.

[9] Zonghong Li, Guoqi Xie, Wenhong Ma, Xiongren Xiao, Yong Xie, Wei Ren, and Wanli Chang. 2023. RTISM: Real-Time Inter-VM Communication Based on Shared Memory for Mixed-Criticality Flows. In *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 279–290.

[10] A Cameron Macdonell. 2011. Shared-memory optimizations for virtual machines. (2011).

[11] Steffen Müller, David Bermbach, Stefan Tai, and Frank Pallas. 2014. Benchmarking the performance impact of transport layer security in cloud database systems. In *2014 IEEE International Conference on Cloud Engineering*. IEEE, 27–36.

[12] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. 2006. vTPM: virtualizing the trusted platform module. In *Proc. 15th Conf. on USENIX Security Symposium*. 305–320.

[13] QEMU Project Developers. [n. d.]. *Device Specification for Inter-VM Shared Memory Device.* https://www.qemu.org/docs/master/specs/ivshmem-spec.html

[14] AU Rahman, SU Miah, and S Azad. 2014. Advanced encryption standard. *Practical Cryptography: Algorithms and Implementations Using C++* (2014), 91–126.

[15] Dominik Reinhardt and Gary Morgan. 2014. An embedded hypervisor for safety-relevant automotive E/E-systems. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*. IEEE, 189–198.

[16] Yi Ren, Ling Liu, Qi Zhang, Qingbo Wu, Jianbo Guan, Jinzhu Kong, Huadong Dai, and Lisong Shao. 2016. Shared-memory optimizations for inter-virtual-machine communication. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 1–42.

[17] Eric Rescorla. 2018. *The transport layer security (TLS) protocol version 1.3.* Technical Report.

[18] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.

[19] Ashutosh Satapathy, Jenila Livingston, et al. 2016. A Comprehensive Survey on SSL/TLS and their Vulnerabilities. *International Journal of Computer Applications* 153, 5 (2016), 31–38.

[20] Craig Shue, Youngsang Shin, Minaxi Gupta, and Jong Youl Choi. 2005. Analysis of IPSec overheads for VPN servers. In *1st IEEE ICNP Workshop on Secure Network Protocols, 2005.(NPSec)*. IEEE, 25–30.

[21] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, Dave Andersen, and Jay Lepreau. 1999. The Flask security architecture: System support for diverse security policies. In *8th USENIX Security Symposium (USENIX Security 99)*.

[22] Shesha Sreenivasamurthy and Ethan Miller. 2019. SIVSHM: Secure inter-vm shared memory. *arXiv preprint arXiv:1909.10377* (2019).

[23] John Viega, Matt Messier, and Pravir Chandra. 2002. *Network security with openSSL: cryptography for secure communications.* " O'Reilly Media, Inc.".

[24] Jian Wang, Kwame-Lante Wright, and Kartik Gopalan. 2008. XenLoop: a transparent high performance inter-vm network loopback. In *Proceedings of the 17th international symposium on High performance distributed computing*. 109–118.

[25] Hannes Weisbach, Björn Döbel, and Adam Lackorzynski. 2011. Generic user-level PCI drivers. In *Proceedings of the 13th Real-Time Linux Workshop. URL http://lwn.net/images/conf/rtlws-2011/proc/Doebel. pdf.*

[26] Xen Project. 2025. Xen Grant Tables. https://xenproject.org/developers/design-documents/grant-tables/. [Online; accessed 28-Apr-2025].

[27] Jiaming Zhang, Fengyun Li, Liu Yang, Yucong Chen, Hubin Yang, Qingguo Zhou, Yan Li, and Rui Zhou. 2023. The Optimization of IVSHMEM Based on Jailhouse. In *International Symposium on Advanced Parallel Processing Technologies*. Springer, 54–75.

[28] Qi Zhang and Ling Liu. 2016. Workload adaptive shared memory management for high performance network i/o in virtualized cloud. *IEEE Trans. Comput.* 65, 11 (2016), 3480–3494.

[29] Xiaolan Zhang, Suzanne McIntosh, Pankaj Rohatgi, and John Linwood Griffin. 2007. XenSocket: A high-throughput interdomain transport for virtual machines. In *Middleware 2007: ACM/IFIP/USENIX 8th International Middleware Conference, Newport Beach, CA, USA, November 26-30, 2007. Proceedings 8*. Springer, 184–203.