

IRMA: Reducing I/O Tax in Confidential Computing with RDMA

Paul Ferdinand Heinen
ETH Zurich

Supraja Sridhara
ETH Zurich

Valentin Ogier*
NetApp

Leonard Cohnen*
Cisco

Shweta Shinde
ETH Zurich

Abstract—Remote Direct Memory Access (RDMA) enables apps to efficiently transfer large volumes of data by letting RDMA-capable Network Interface Cards (NICs) directly access VM memory. Confidential computing hardware blocks all device accesses (including NICs) to confidential VM (CVM) memory, rendering RDMA impossible. This restriction forces developers to use significantly slower mechanisms (e.g., TLS), which imposes a substantial network I/O tax stemming from additional data copies, encryption overheads, and the loss of RDMA hardware offloads. We present IRMA, which preserves RDMA communication semantics while enforcing strict memory isolation, enabling RDMA apps to run in CVMs with minimal changes. Our evaluation on AMD SEV-SNP and Intel TDX demonstrates that IRMA achieves up to $17.23\times$ higher throughput and $46.70\times$ lower latency compared to TLS, substantially reducing the network I/O tax.

Index Terms—RDMA, confidential computing, NIC

1. Introduction

Confidential computing platforms (e.g., Intel TDX, AMD SEV-SNP, Arm CCA) enable users to run computation in isolated confidential virtual machines (CVMs) on untrusted cloud infrastructure [5], [10], [26]. These platforms are built to move existing VMs and apps to run inside CVMs with minimal changes. To protect the CVM memory, confidential computing hardware blocks all direct access from devices to the CVM memory. This design choice favors security but sacrifices performance. Traditional VMs use network interface cards (NICs) to connect to other hosts, VMs, and services over TCP/IP networks, which is slow and incurs significant overheads [23], [54]. To address these limitations, Remote Direct Memory Access (RDMA) provides a high-performance network protocol that enables low-latency, zero-copy communication between VMs using RDMA-capable NICs (RNICs). RDMA achieves this performance by letting RNICs directly access app memory, minimizing CPU involvement and enabling efficient data movement [12], [18], [21]. This direct access lets non-confidential VMs achieve high throughput (in tens to hundreds of Gb/s) and low latency (in μ s).

The fundamental abstraction that RDMA relies on, i.e., direct device access to app memory, is incompatible with the hardware-enforced isolation guarantees of confidential

computing. Consequently, CVMs cannot use RDMA and must fall back to TCP/IP-based communication secured by TLS. This fallback not only requires substantial app modification but also degrades the throughput and latency by two orders of magnitude. Thus, the increased cost of network communication in confidential computing introduces a substantial network I/O tax. Our experiments show that this tax arises from additional data copies, encryption and decryption overheads, and the loss of RDMA hardware offloads to the RNIC. Further, we report that this network I/O tax is particularly severe for distributed systems that are tightly coupled to RDMA-specific semantics, making simple rewrites impractical. Lastly, even with prior optimizations for encrypted TCP in CVMs [29], [30], these systems cannot match the latency and throughput of native RDMA.

Our goal is to address the high network I/O tax in confidential computing. Our insight is to preserve RDMA communication for performance while enforcing strict memory isolation for security. Our solution, IRMA, decouples RDMA semantics from the assumption of trusted RNIC hardware, treating both the network and devices as untrusted and protecting all the data sent over the RDMA network. It maintains compatibility with standard RDMA abstractions, including queue pairs, asynchronous completions, remote memory operations, and one-sided communication, while ensuring that CVM memory remains inaccessible to untrusted cloud infrastructure. IRMA reduces the network I/O tax imposed by confidential computing while preserving the performance and programmability benefits of RDMA apps.

On top of the RDMA channel itself, we identify that RDMA apps also rely on a security-critical out-of-band channel to exchange connection metadata and remote memory addresses. We show that even if the RDMA channel is protected, an attacker controlling the cloud infrastructure can exploit this out-of-band channel to manipulate remote memory addresses to compromise RDMA communication. IRMA addresses this by introducing a secure out-of-band channel, while requiring minimal app changes to preserve existing deployment models.

We implement IRMA (Isolated Remote Memory Access) and extensively evaluate it on both AMD SEV-SNP and Intel TDX CVMs connected to Mellanox ConnectX-5 RNICs [37]. Our microbenchmarks show that using TCP/TLS in CVMs degrades the throughput and latency by two orders of magnitude compared to RDMA. IRMA avoids this overhead by design and has five optimizations to further reduce the network I/O tax resulting

* Work done while at ETH Zurich.

in $17.23\times$ higher throughput and $46.70\times$ lower latency compared to TLS. We also evaluate IRMA using three real-world RDMA apps that execute out of the box on CVMs on both platforms. For one app that supports both TLS and RDMA backends, IRMA substantially outperforms the TLS-based deployment in CVMs, while preserving the RDMA performance and ensuring security. IRMA is open source at <https://github.com/IRMA-CC> [2].

We make the following contributions in this paper:

- We identify the substantial network I/O tax that RDMA apps pay when operating in confidential computing.
- IRMA preserves RDMA semantics while enforcing strict memory isolation from untrusted devices and hypervisor for fast and secure network communication.
- We analyze overhead sources and our optimizations reduce the network I/O tax in CVMs by two orders of magnitude.

2. Background & Potential Approaches

Virtual machines (VMs) typically use standard NICs for networking (e.g., using TCP/IP stack), where data must traverse through multiple privileged software layers (e.g., guest/VM operating system, cloud provider hypervisor, host operating system) on both the sender and receiver machines. These extra software layers degrade network performance.

2.1. Remote Direct Memory Access (RDMA)

RDMA lets VMs move data directly between app memory across the machines without involving privileged software [12], [18], [21]. For this, VM apps interact with RDMA-capable NICs (RNICs) using shared memory regions to issue *commands* to read or write data from remote memory, and exchange *data* chunks with remote apps.

RDMA communication is typically between two endpoints (apps) running on different machines. Apps use an RDMA library to communicate with the RNIC. This library provides standard APIs to create connection endpoints called queue pairs, set up memory regions that the RNIC can access, submit commands to the RNIC, and send and receive data. We identify that, to establish a connection, VM apps exchange connection information over a non-RDMA out-of-band channel (Figure 1, Step 1) that the app selects, as RDMA does not define this. Additionally, apps use the out-of-band channel to dynamically exchange information about remote memory regions during execution.

During connection establishment, each VM app creates a queue pair and uses it to issue commands to the RNIC. Based on the commands from the app (Figure 1, Steps 2–3), the RNIC can directly access app memory to perform RDMA operations (Figure 1, Step 4). Figure 1 (Steps 2–8) shows an example for an RDMA write operation.

2.2. RDMA with Confidential Computing

Confidential Computing platforms (e.g., Intel TDX, AMD SEV-SNP, Arm CCA) enable confidential VMs

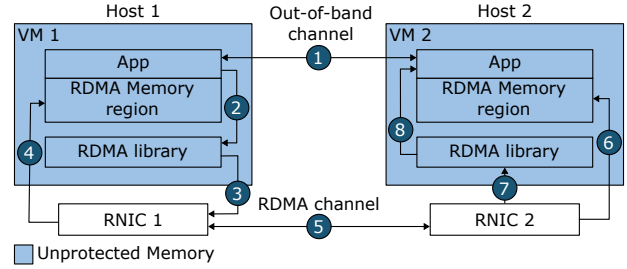


Figure 1: Unprotected RDMA: 1) App establishes connection using an out-of-band channel. 2)–3) App uses RDMA library to issue RDMA command to RNIC. 4) RNIC reads App memory region that it should send according to the submitted command. 5) RNIC sends data over RDMA network. 6) Remote RNIC places received data into remote App memory region. 7)–8) RNIC optionally notifies App of received data using RDMA library.

(CVMs) that are isolated from an untrusted cloud provider, including the hypervisor and all untrusted devices (e.g., RNICs) in a data center [5], [10], [26]. To protect CVMs, confidential computing hardware blocks all direct access from the untrusted RNICs to CVM memory [17]. Thus, RNICs cannot access RDMA data structures (e.g., connection endpoints) or shared memory regions, rendering RDMA-based app execution in CVMs infeasible. Three potential approaches can address this:

Replace RDMA with TLS. Since RDMA is not supported in CVMs, apps can switch to an alternate transport layer protocol, such as TCP. As TCP sends data in plaintext by default, we can protect the TCP connection using the well-established TLS protocol by modifying the apps. While this approach seems feasible at first glance, we find that it is not possible to replace all RDMA APIs with equivalent TCP APIs. For example, RNICs provide hardware support to notify apps of completed tasks (Figure 1, Steps 7–8). TCP NICs do not have hardware support to issue an analogous completion notification once a TCP operation is completed. So to support this feature, apps will need to be invasively modified to not rely on these completion notifications or build an intermediary that emulates the notifications. Moreover, the slow TCP/IP stack degrades RDMA performance, resulting in a high network I/O tax on CVMs [56].

RDMA over unprotected shared memory. If CVM apps use RDMA, they need to communicate with RNICs over unprotected shared memory regions that RNICs can access. Untrusted RNICs can eavesdrop, tamper, and replay RDMA data transferred by the CVMs over the shared memory regions. This unprotected memory is also accessible to the untrusted cloud provider (e.g., hypervisor, devices), compromising CVM security.

RDMA over protected shared memory. To secure RDMA communication, an obvious solution is to encrypt and integrity protect all RDMA communication that is written to unprotected memory using encrypted bounce buffers. As we show in Section 3, encrypted bounce buffers alone are insufficient to ensure end-to-end security of RDMA apps.

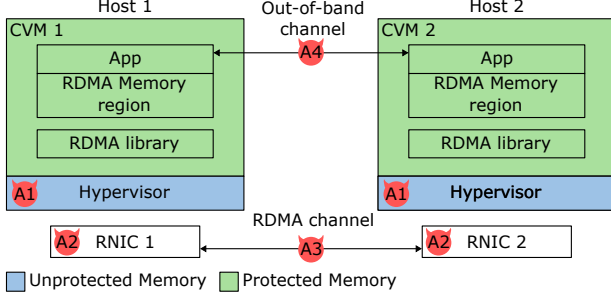


Figure 2: Attackers for RDMA in Confidential Computing: A1) Malicious Hypervisor, A2) Malicious RNIC, A3) Network attacker on RDMA channel, A4) Network attacker on out-of-band channel.

This gap motivates the need for a systematic solution to achieve RDMA over protected shared memory.

Thus, RDMA communication over protected shared memory provides the best balance between security and performance, preserving RDMA semantics while avoiding the substantial network I/O tax of TCP/TLS-based communication. Additionally, this keeps the RNIC hardware and drivers out of the TCB.

3. Security Challenges

In this section, we define our threat model and identify the attacks that RDMA apps face when they run in CVMs.

3.1. Threat Model

We consider a confidential computing threat model with a malicious cloud provider and four attackers:

Attacker A1. All cloud-provider-managed software (e.g., hypervisor) is untrusted (A1 in Figure 2).

Attacker A2. All RNIC hardware attached to CVMs is untrusted and managed by the cloud provider (A2 in Figure 2). For example, the cloud provider sets up the RNICs, manages their IDs, and configures routing rules.

Attackers A3 and A4. Cloud provider controls all networking hardware (including the RNIC), and all devices (e.g., PCIe devices). Thus, we operate under a strong network adversary who can snoop, record, and tamper with all network communication, including the RDMA network (A3 in Figure 2) and the out-of-band channel (A4 in Figure 2).

We trust the confidential computing hardware and firmware of the compute nodes that host the CVMs. The CVMs can obtain attestation reports to ascertain the trustworthiness of the confidential computing hardware they execute on. CVMs trust the apps, operating system, and device drivers (e.g., RNIC drivers) that execute in them. Furthermore, CVMs that want to communicate can trust each other after mutual attestation. We consider denial-of-service, side-channel, and microarchitectural attacks as out of scope for this work [19], [42], [57], [62].

Next, we outline security challenges in the above threat model when protecting RDMA and out-of-band channels.

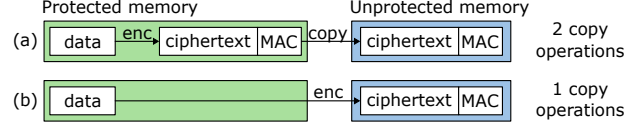


Figure 3: (a) Naive bounce buffering scheme. (b) Optimized bounce buffering scheme.

3.2. RDMA Channel

To use RDMA, a CVM must place all data that the RNIC reads or writes in unprotected shared memory (Section 2.2). Every attacker with access to this memory or to the network can therefore observe and modify the data.

Attack #1: All data sent over the RDMA channel is neither integrity protected nor confidential. Attackers A1–A3 can read and write the data undetected.

At the outset, protecting confidentiality and integrity in CVMs with untrusted RNICs might seem straightforward. We can use encrypted bounce buffers in unprotected shared memory to protect the confidentiality of RDMA data transferred between CVMs [16], [30], [35], [61].

A naive bounce buffering scheme (Figure 3a) first encrypts and integrity protects the data from CVM memory into another CVM memory location. Once the scheme calculates the ciphertext and MAC, it copies them from CVM memory to unprotected shared memory. This scheme results in two copy operations. An optimized version encrypts the data directly into unprotected shared memory, to avoid the additional copy (Figure 3b). However, we have to ensure that our optimized encryption scheme provides integrity protection so that the untrusted hypervisor cannot tamper with the data.

Attack #2: If we compute integrity tags after writing encrypted data to the unprotected shared memory region, the hypervisor (A1) and RNIC (A2) can tamper with the encrypted data and trick our system into accepting modified data before the integrity tags are computed (Figure 4a).

Even with confidentiality and integrity protection, apps remain vulnerable to network attackers replaying, reordering, or dropping encrypted data. Consider a local CVM app that writes data to a remote CVM app over RDMA. With encrypted bounce buffers, we could encrypt the data into encrypted chunks.

```

(c) c = encrypt(m);
    // compute mac using c
    update_mac(mac, c); // UNSAFE
(b) c = encrypt(m, state); // updates state
    // compute mac from state
    update_mac(mac, state); // SAFE

```

Figure 4: `c` and `mac` point to unprotected memory. `m` and `state` point to protected memory. (a) Unsafe encryption not used in IRMA. (b) Safe encryption used in IRMA.

Attack #3: An untrusted RNIC (A2) and an RDMA network attacker (A3) can capture encrypted chunks, and replay, reorder, or drop them. Such manipulation can force the remote CVM to read stale data, violating its security guarantees.

To prevent these attacks, we need to additionally provide replay, reorder, and drop protection for encrypted chunks.

3.3. Out-of-Band Channel

Before issuing RDMA commands, apps first exchange connection information and remote memory region addresses over a non-RDMA out-of-band channel (e.g., TCP sockets). During execution, VM apps use the out-of-band channel to dynamically exchange information about the addresses of remote memory regions. Because the out-of-band channel by default is unprotected (e.g., uses TCP/IP), a network attacker A4 can snoop, record, and replay all network communication.

Attack #4: A network attacker (A4) can modify remote addresses sent over the out-of-band channel. VM apps are not written for this threat model and have no mechanism to detect this tampering and simply assume that the addresses received were legitimate. As a consequence, VM apps will issue RDMA read and write requests to incorrect remote memory addresses, thus compromising the app.

This attack succeeds even if the RDMA channel is secure, because it targets the out-of-band channel instead of the RDMA channel. Apps can build this channel on any non-RDMA protocol, and an attacker can intercept and tamper with each of them. While static binary instrumentation can establish a baseline of benign run-time memory accesses to detect tampering in out-of-band channels, this approach remains insufficient because apps dynamically allocate shared memory regions with non-deterministic addresses across different executions (see Appendix A.1). Another solution would be to protect this channel with encryption (e.g., using TLS). Encryption, however, requires a mechanism to authenticate the remote app, and set up and manage keys for this channel. The challenge is to achieve this securely with minimal changes to the apps.

3.4. Trusted I/O for RDMA

Confidential computing CPUs support extensions (e.g., Intel TDX-Connect, AMD SEV-SNP Trusted I/O, Arm CCA Device Assignment) that enable trusted I/O, allowing trusted devices direct access to CVM memory [6], [11], [24]. To use these extensions, devices and drivers first need extensive hardware and software changes to be able to prove the device trustworthiness to the CVM. Then, CVMs can execute a trust establishment protocol with the device, after which the CVM adds the device to its TCB. Currently, there is no

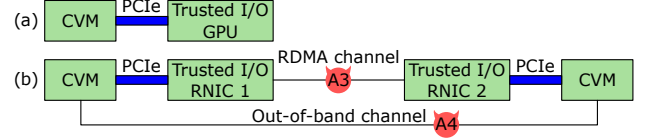


Figure 5: Machine setup for (a) trusted I/O GPU and (b) trusted I/O NICs. Thick blue connections are protected by TDISP. Black connections are unprotected.

RNIC hardware that supports the trusted I/O standard TDISP [44] that is publicly available or announced for upcoming release. However, even once they exist, securing RDMA apps will not be straightforward. Nevertheless, next, we discuss the applicability of IRMA to trusted I/O when it becomes available for RNICs.

Most trusted I/O devices (e.g., GPUs) are used by a local CVM and are only connected via PCIe to the host [45], [52], thus PCIe TDISP is sufficient to secure this path (Figure 5a). In contrast, trusted I/O RNICs have two connections: the physical PCIe connection to the local CVM host, and the RDMA network connection to the remote CVM host (Figure 5b). We observe that, even if RNIC hardware is modified for trusted I/O, it will only eliminate the need for encrypted bounce buffers by providing the RNICs direct access to CVM memory. However, apps will continue to use the out-of-band channel to obtain remote addresses, thus requiring out-of-band channel protection to ensure end-to-end app security. Furthermore, TDISP only provides security for the PCIe connection of the device and not the RDMA channel. Therefore, **Attacks #1, #3, and #4** presented in Section 3.2 and Section 3.3 can still be performed on the unprotected out-of-band channel and RDMA channel. To this end, IRMA protects the RDMA connection and the out-of-band channel in the trusted I/O setting with minimal changes (see Section 4.4).

4. IRMA Design

We designed IRMA to preserve RDMA semantics in CVMs while enforcing strict memory isolation from untrusted RNIC hardware. Next, we show how IRMA achieves this to bring down the network I/O tax on confidential computing and present optimizations for further reduction.

4.1. Preserving Transport Semantics

IRMA preserves the RDMA interface for the app while keeping the RNIC out of protected CVM memory.

Two views for RDMA channel. IRMA has to correctly relay all RDMA operations from the apps to the RNIC. Figure 6 shows how IRMA intercepts all communication between the app and the RNIC by creating two views of all RDMA data structures (e.g., queue pairs, memory regions, send/recv queues). IRMA creates one view inside CVM memory which the app operates on. The second view is in unprotected shared memory such that the RNIC can access it to perform

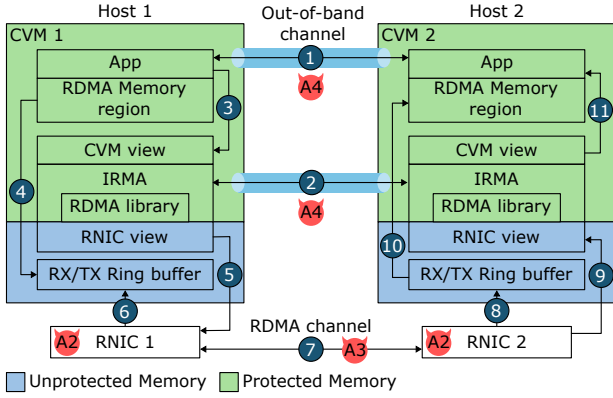


Figure 6: Protected RDMA in Confidential Computing using IRMA. 1) Connection establishment using protected out-of-band channel. 2) Exchange of security information using secondary protected channel. 3) IRMA intercepts RDMA command from app. 4) IRMA encrypts and integrity protects requested memory and places it in the TX ring buffer. 5) IRMA notifies RNIC to send encrypted data to remote. 6) RNIC reads encrypted data from TX ring buffer. 7) RNIC sends data over RDMA network to remote RNIC. 8) Remote RNIC places received encrypted data into RX ring buffer. 9) RNIC notifies IRMA that data was received. 10) IRMA decrypts data from RX ring buffer to protected app memory. 11) IRMA optionally notifies app of received data.

RDMA operations. IRMA translates all operations between the two views transparently to the app and RNICs.

RDMA lifecycle with IRMA. To manage the RDMA data structures, IRMA intercepts all RDMA API calls from the apps to create them (e.g., queue pairs, memory regions). For the CVM view, IRMA emulates these RDMA data structures in the protected memory of the CVM. For the RNIC view, IRMA invokes the RDMA library to create these data structures in unprotected RNIC-accessible shared memory. For example, when the app invokes the RDMA library to create a queue pair, IRMA intercepts the call and creates the queue pair in CVM memory. Then, IRMA invokes the RDMA library to create a corresponding queue pair accessible to the RNIC in unprotected memory. Additionally, IRMA creates an RX and TX ring buffer for each queue pair in unprotected shared memory which it uses to process incoming and outgoing encrypted chunks.

When the app issues commands to the RNIC using its CVM view of the data structures, IRMA intercepts the command (Figure 6, Step 3), encrypts the requested data into the TX buffer in unprotected shared memory (Step 4), and then issues commands to the RNIC (Step 5). Figure 6 (Steps 3–11) shows an example of an RDMA write.

4.2. Preserving Security in Confidential Computing

While building IRMA, we considered several security-sensitive design choices that we discuss next.

Out-of-band channel & connection establishment. To protect the data sent over RDMA and secure the out-of-band

channel IRMA first needs to establish a secure connection between the queue pairs of the two CVMs. IRMA sets up two secure channels between CVMs using attested TLS by first mutually attesting the two CVMs. It uses the first secure channel (Figure 6, Step 1) to wrap the out-of-band communication to prevent **Attack #4**. IRMA uses the second channel to exchange security information for the RDMA channel (Figure 6, Step 2). Concretely, IRMA establishes shared encryption keys, initialization vectors, and initial sequence numbers for each queue pair. Using this information, IRMA can securely encrypt and integrity protect the data sent over RDMA between the two CVMs.

Encrypted bounce buffers. To protect the RDMA channel, IRMA transparently enables optimized encrypted bounce buffers (Figure 3b) using the exchanged keys for all RDMA communication by intercepting RDMA API calls from apps. While encrypting the data, we need to ensure that encryption and integrity protection do not violate any memory allocation and alignment requirements of the apps and RNICs. To adhere to these requirements, IRMA splits all data that apps want to send over RDMA into fixed-size chunks. IRMA assigns each chunk a unique sequence number, adds padding where required, and encrypts and integrity protects it. These protections guard the chunk against **Attacks #1** and **#3**. For the remote node to process these encrypted chunks, it needs information about the remote address details. We include these in a metadata header, encrypt it separately, and prepend it to the encrypted chunk.

To transfer metadata and encrypted chunks between nodes, IRMA creates two ring buffers in unprotected memory for each queue pair: a transmit (TX) buffer for outgoing data, and a receive (RX) buffer for incoming data (Figure 6). IRMA writes outgoing metadata and data chunks to free TX buffer slots, and reads incoming metadata and data chunks from RX buffer slots. Using ring buffers in this way lets us efficiently reuse memory for RDMA transfers. However, we need to ensure that both the TX and RX buffers never overflow as this could lead to data loss. While this data loss does not compromise security, because we use sequence numbers to detect lost data, it can lead to performance degradation as apps will have to retransmit lost data. To prevent data loss, IRMA ensures flow control between the two endpoints by setting up a second control queue pair per queue pair. IRMA uses this control queue pair to exchange information about the number of free TX/RX slots between the two endpoints. This flow-control information lets IRMA safely reuse slots in the TX and RX buffers.

Encryption scheme. When using the IRMA optimized bounce buffering (Figure 3b), we might be prone to integrity attacks as presented by **Attack #2**. To prevent such attacks, we ensure that the encryption library we use always computes integrity tags and other cryptographic information from protected CVM memory only. Most encryption libraries are not designed for this threat model and do not differentiate between protected and unprotected memory. Table 1 lists the libraries we analyzed and whether each is secure under our threat model. We chose the Aegis encryption algorithm family, which integrates authentication into

the cipher’s internal state rather than treating encryption and authentication as separate steps. This integrated design lets Aegis only use information in protected memory to compute the ciphertext and authentication tags (Figure 4b) [58]. The selected encryption algorithm does not trade security for performance, but provides the same cryptographic guarantees as conventional encryption algorithms in a standard VM setting.

Metadata encryption. IRMA associates each encrypted data chunk with encrypted metadata that indicates to the remote node how to handle the data chunk. By encrypting the metadata separately from the data chunk, we can optimize the number of memory copies we perform when moving data between the two views (Section 4.3). However, it also opens up the possibility of an attacker mismatching metadata and data chunks. To protect against this, IRMA always uses sequential sequence numbers for metadata and their respective data chunks.

4.3. Optimizations

We present optimization schemes that preserve security while further reducing the network I/O tax.

Optimization #1: IRMA with different chunk sizes. If we choose a data chunk size that is larger than the typical transfer sizes of a given app, we need to pad the actual app data to the chosen chunk size. This excessive padding increases the amount of time IRMA takes to encrypt/decrypt and copy the chunks in and out of protected memory, degrading performance. Similarly, setting a chunk size too small compared to the typical data transfer sizes of an app, also degrades performance because of an unnecessarily high number of per-chunk operations that IRMA would perform. So, our first optimization adjusts the chunk size that IRMA uses for optimal performance. In Section 7.1 we set the chunk size based on the message size submitted by the app to achieve optimal performance.

Optimization #2: Pipelining. IRMA’s approach of chunking the data for encryption lets us efficiently pipeline encryption/decryption and RDMA operations to improve performance. With pipelining, IRMA can send one chunk over RDMA, while it encrypts the next chunk in parallel.

Optimization #3: Metadata. In IRMA we encrypt the metadata separately from the data chunk it is associated with. The metadata contains details that the remote node can use to decide how to process the next encrypted data chunk. Our optimization of encrypting the metadata chunk separately helps IRMA avoid unnecessary memory copies for large data chunks, improving performance. Concretely, IRMA first decrypts the metadata into a secure scratchpad memory. Using the metadata, IRMA can then decide if it can directly decrypt the data chunk into app memory. For example, if the metadata indicates the encrypted chunk contains no padding then IRMA does not need to process the chunk after decrypting it to remove padding. So, we can directly decrypt it into the app memory region, avoiding an extra data copy operation. Otherwise, IRMA first decrypts the chunk into a

TABLE 1: Secure authenticated encryption (AEAD) algorithms from libsodium [4] and libaegis [3].

AEAD algorithm	Library	Secure against integrity attacks		Used in IRMA
		Encryption	Decryption	
AES-256-GCM	libsodium	no	no	no
XChaCha20-Poly1305-IETF	libsodium	no	no	no
Aegis-1281	libaegis	yes	yes	yes
Aegis-128x4	libaegis	yes	yes	yes

secure scratchpad region, removes any padding, and then copies it into app memory.

Optimization #4: Polling vs. event-based management. In IRMA, we need a mechanism to wait for senders to write encrypted data chunks in the local RX buffers. Once these data chunks arrive in the RX buffer, we can decrypt them into the app’s protected memory regions. To implement a wait mechanism we can either leverage an RNIC feature that notifies IRMA once an encrypted chunk is written to the RX buffer, or choose to poll for new data. We find that the event-based notification mechanism increases the latency of processing the data and is sub-optimal. Thus, we choose to use polling in IRMA.

Optimization #5: CVM placement. Modern SoCs are built using modular building blocks which are made up of a small group of cores and caches that can communicate with low latency (e.g., AMD core complex blocks).¹ We observe that when a CVM executes on a single block, then all its cores access the same caches with low latency. Conversely, if the CVM is spread across cores belonging to several blocks, then inter-block cache accesses result in higher latencies. Therefore, we optimize IRMA latency by ensuring that whenever possible, the CVM executes on a single block of the processor.

4.4. IRMA with Trusted I/O

As discussed in Section 3.4, trusted I/O will not protect against the **Attacks #1, #3, and #4** presented in Section 3.2 and Section 3.3. IRMA is still necessary to protect the RDMA connection and out-of-band channel in the trusted I/O setting with minimal changes. The only change will be that instead of encrypting data into ring buffers in unprotected shared memory, IRMA writes the ciphertext into ring buffers in CVM memory. Because the trusted I/O RNIC has direct access to CVM memory, it can then access the ciphertext in the ring buffers without the need for bounce buffers. In summary, IRMA is necessary and sufficient for potential future trusted I/O support in RNICs.

5. Security Analysis

Attacks on the data plane. Attackers **A1–A3** can try to modify or forge data chunks stored in unprotected shared memory or being transferred over the untrusted network. IRMA encrypts and integrity protects these chunks, so the

1. AMD core complex blocks (CCX) in Zen 4 consist of 8 physical cores, up to 16 hyperthreads, and have dedicated L3 cache slices. Up to 12 CCX are connected through an interconnect.

decryption on the remote node will fail, detecting and stopping the attacks. Similarly, these attackers can try to record and replay, reorder, or drop encrypted packets. IRMA uses sequence numbers while encrypting metadata and data chunks that will detect these attacks. Further, these attackers can try to associate a data chunk with the incorrect metadata. IRMA prevents this by cryptographically binding each metadata to its corresponding data chunk using sequence numbers during encryption. Concretely, the sequence number assigned to a data chunk is always exactly one greater than the sequence number assigned to its metadata such that

$$\text{seq}(\text{data chunk}) = \text{seq}(\text{metadata}) + 1.$$

The attacker **A4** can try to tamper with the out-of-band channel but IRMA’s secure channel protection using attested TLS stops the attack.

Attacks on the control plane. Prior works have demonstrated network attacks that manipulate the RDMA control plane [38], [41], [43], [51]. They modify routing rules for the RNICs, change destination addresses of the remote app, or tamper with queue pair information to gain unauthorized access to host memory. IRMA protects the app view in CVM memory from any control-plane tampering. Concretely, with IRMA we encrypt all the control-plane information (e.g., remote app memory address in CVM memory) for a data chunk in the metadata sent with the encrypted data chunk. So, attackers **A1–A3** cannot tamper with the control-plane information in the metadata. However, these attacks can tamper with the control-plane information that affects the RNIC view transmitted in plaintext (e.g., the destination remote address of the RNIC-accessible RX ring buffer, and the queue pair information). Next, we show how IRMA protects against these attacks.

When transferring encrypted data over the RNICs, IRMA sets the remote address to the destination queue pair’s RX ring buffers in plaintext. An attacker-controlled RNIC **A2** can change this plaintext address to point to a different queue pair’s RX ring buffer, writing the encrypted data chunk to a different queue pair. The untrusted hypervisor **A1** can perform a similar attack and switch data chunks between RX buffers of different queue pairs. Furthermore, a network adversary **A3** can change routing rules to route encrypted data to an RX ring buffer on another node. IRMA uses per-queue-pair keys for encryption and therefore incorrectly placed encrypted chunks will fail to decrypt, stopping these attacks. The RNIC **A2** can change the remote address of the RX buffer to the queue pair’s TX buffer. By doing this, the RNIC can trick IRMA into transmitting the data chunk back to the node that originally sent it. Even though this data chunk is encrypted with the correct queue pair key, these chunks will not have the right sequence number (i.e., the sequence number expected for receiving data) in the original sender, stopping the attack.

During queue pair and out-of-band connection setup, the attacker can try to trick a victim app to connect to an attacker-controlled app. IRMA stops such attacks by mutually attesting the end points, before setting up the secure channel for the connections.

6. Implementation

We implement IRMA as a library (3612 LoC) and a manager (665 LoC). IRMA library interfaces with the popular RDMA library, libibverbs [31], while the manager assists with mutual authentication, secure channel setup, and exchanging cryptographic material (see Appendix C.1).

IRMA library. We build IRMA library as an implementation of the interface that libibverbs exposes to apps. Then, we modify 304 LoC in libibverbs to plug in IRMA library. Using this mechanism, we emulate 18 APIs to support all messages that are defined in the RDMA RFC (Figure 4 of RFC 5040 [40]). Appendix B.1 details each interface, its functionality, and how IRMA accommodates it securely.

Key setup. When apps establish a connection between two queue pairs, IRMA library opens a socket to create an attested TLS session. IRMA library uses this session to exchange keys, initial sequence numbers, and additional queue pair information. Once the information has been exchanged, IRMA library closes the TLS session.

Kernel support. To build support for IRMA’s ring buffers we need to allocate and manage unprotected memory as well as copy data back and forth between them. To do this, the RDMA driver needs to support allocating unprotected memory. One way this is typically done is using Linux SWIOTLB. The SWIOTLB is a Linux implementation of a bounce buffer that works with CVMs. To perform the bounce buffer operations, it requires the driver to explicitly transfer ownership of memory between device and processor using the DMA API. During ownership transfer, the SWIOTLB copies the data between the bounce buffer and the CVM’s memory. We observe that this mechanism is fundamentally incompatible with how the RDMA driver and RDMA library work. Apps allocate memory in user space and register it as RDMA memory regions using the RDMA driver, preventing the RDMA driver from performing ownership transfers. So, we change 192 LoC in the Linux 6.10 kernel to allocate unprotected memory using the DMA-BUF heap and 352 LoC in the RDMA driver to use these unprotected memory regions for IRMA’s ring buffers.

Platform choice. IRMA is compatible with CVMs built on AMD SEV-SNP, Intel TDX, and Arm CCA. Currently, Arm CCA is not publicly available in hardware so we compile and evaluate IRMA library and IRMA manager for the x86 architecture. We implement support for IRMA on AMD SEV-SNP [7] and Intel TDX [14].

7. Evaluation

We evaluate IRMA on AMD SEV-SNP and Intel TDX. **Experimental setup.** Our experimental setup for each platform consists of two host machines, Host A and Host B. Appendix C.2 lists the specifications of Host A and B for AMD SEV-SNP and Intel TDX. We conduct our experiments in CVMs hosted on Host A and Host B. We launch the CVMs using QEMU [49] provided by the latest AMD/Intel kernel repository [7], [14]. The CVMs execute a minimal Linux system using our modified Linux 6.10 kernel.

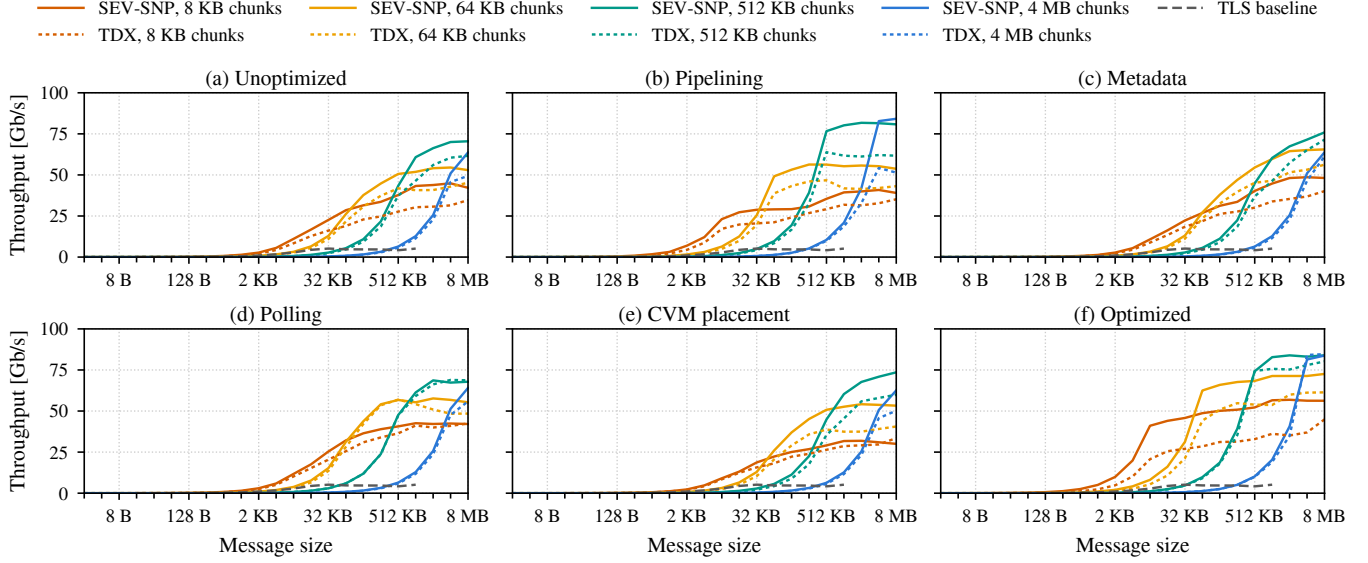


Figure 7: Average throughput for the different optimization configurations (Higher is better).

All changes to the kernel are platform-agnostic, so the kernel can be reused for both AMD SEV-SNP and Intel TDX without changes. Each host has a Mellanox ConnectX-5 RNIC. A 100 Gb/s InfiniBand link connects the two RNICs. We use standard QEMU vfiio device passthrough to attach a virtualized instance of the RNIC to the CVM. With this, KVM continues to manage and have access to the RNICs. We run the subnet manager, OpenSM, on Host A to route traffic between the RNICs. Besides the RNICs, each host is equipped with an Intel Ethernet X540 NIC.

Our evaluation is guided by five research questions:

RQ1: What are the effects of optimization on IRMA?

RQ2: How does IRMA compare to TLS, TCP, and unencrypted RDMA?

RQ3: How does IRMA scale with the number of RDMA connections?

RQ4: How does IRMA perform for existing RDMA apps?

RQ5: How much effort is required to secure the out-of-band channel of apps?

7.1. RQ1: Impact of Optimizations

Encryption algorithms. We determine the optimal AEAD encryption algorithm for IRMA by measuring the encryption and decryption cycles of 8 AEAD algorithms from libsodium [4] and libaegis [3] for multiple chunk sizes in a CVM running on Host A for AMD SEV-SNP and Intel TDX (Appendix C.4). We enable hardware acceleration for encryption and decryption. Aegis-128l performs best for chunk sizes ≤ 1 KB, while Aegis-128x4 is best for > 1 KB. IRMA therefore uses Aegis-128l for ≤ 1 KB and Aegis-128x4 for > 1 KB. Aegis-128x4 is fast for larger chunk sizes, because it encrypts 4 streams in parallel, denoted by the x4. Additionally, we find that using these encryption algorithms with our optimized bounce buffering (Figure 3b) is up to $2.58\times$ faster than naive bounce buffering.

Evaluation of optimizations. Each host runs one CVM, which executes perftest [32] using IRMA. Hardware vendors recommend perftest as the default suite for measuring RDMA latency and throughput [8], [25]. Using perftest, we benchmark throughput and latency of IRMA for different combinations of message and chunk size with different optimizations. We plot the average throughput of RDMA for each optimization. Figure 7a and Figure 8a show the average throughput and latency of unoptimized IRMA, which serves as the baseline for evaluating our optimizations.

Optimization #1: IRMA with different chunk sizes.

For small chunk sizes, throughput increases with message size up to a saturation point, beyond which further increases in message size yield no significant throughput gains (Figure 7a). This saturation is expected because, for larger message sizes the overhead of processing many small chunks reduces the throughput. For larger chunk sizes, the throughput of large message sizes is much higher. However, for smaller message sizes, large chunk sizes perform worse than small chunk sizes, because small messages are padded in large chunks, thus wasting bandwidth. Additionally, padding results in large chunk sizes having a higher latency for small message sizes (Figure 8a).

Optimization #2: Pipelining. Enabling pipelining of chunks improves the overall throughput of IRMA, because it lets us encrypt the next chunk while the previous one is being sent (Figure 7b). For large chunk sizes (512 KB and 4 MB), we observe throughput improvements of up to $1.81\times$ and $1.64\times$ for AMD SEV-SNP and $1.85\times$ and $1.94\times$ for Intel TDX. For smaller chunk sizes (8 KB and 64 KB), we observe even greater throughput improvements of up to $2.58\times$ and $2.00\times$ for AMD SEV-SNP and $2.16\times$ and $1.89\times$ for Intel TDX. Furthermore, there is a decrease of the latency for large message sizes of up to $1.22\times$ for AMD SEV-SNP and Intel TDX (Figure 8b). This drop in latency is because

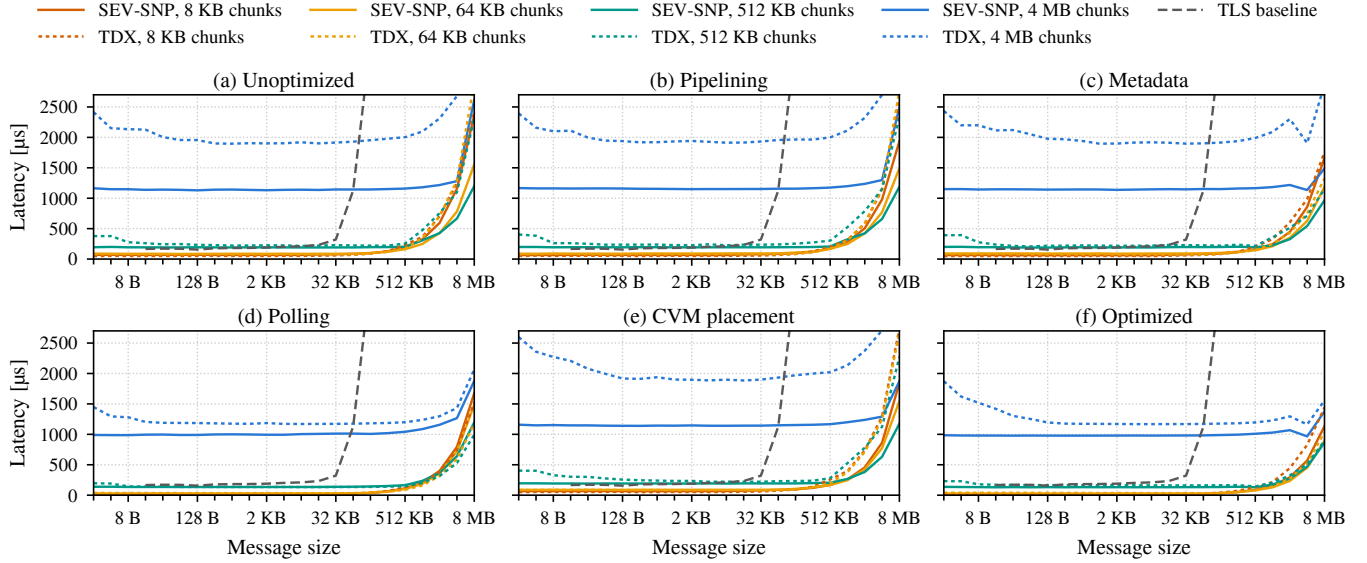


Figure 8: Average latency for the different optimization configurations (Lower is better).

IRMA splits large messages into multiple chunks, reducing the latency due to IRMA’s pipelined encryption and sending. For small messages, pipelining has no effect on latency, because only one chunk is sent per message.

Optimization #3: Metadata. Enabling efficient scattering of received data improves performance and latency (Figure 7c and Figure 8c). Throughput improves by up to $1.24\times$ (AMD SEV-SNP) and $1.26\times$ (Intel TDX), while latency reduces by up to $1.75\times$ (AMD SEV-SNP) and $2.18\times$ (Intel TDX). This effect does not apply for message sizes which are smaller than the chunk size because they contain padding and cannot benefit from efficient scattering.

Optimization #4: Polling vs. event-based management. Enabling polling has no significant effect on the throughput (Figure 7d). However, enabling polling decreases latency by up to $7.10\times$ (AMD SEV-SNP) and $5.15\times$ (Intel TDX) for 8 KB chunks and up to $1.39\times$ (AMD SEV-SNP) and $1.87\times$ (Intel TDX) for 4 MB chunks (Figure 8d). Smaller chunk sizes benefit more from polling, because event-based blocking has a higher relative impact on the overall chunk processing time. Since polling yields no throughput gains, deployments where CPU usage is a concern, or where latency is not critical, can disable it to avoid the overhead of busy-waiting and free those cycles for other workloads.

Optimization #5: CVM placement. Placing a CVM on the same CCX block on AMD SEV-SNP improves the overall latency by up to $1.40\times$ for large 4 MB chunks and up to $1.34\times$ for small 8 KB chunks (Figure 8e). Placing a CVM on the same CCX block has no influence on throughput for large chunk sizes and reduces the throughput for 8 KB chunks for large message sizes (Figure 7e). This throughput drop is not an issue, as apps sending larger message sizes should use a larger chunk size to maximize throughput. Intel TDX has no concept of core groups, so grouping CPUs has no significant effect on latency or throughput.

Optimized IRMA. We finalize IRMA by combining all the optimizations listed above (Figure 7f and Figure 8f).

Compared to unoptimized IRMA, optimized IRMA achieves up to $3.77\times$ higher throughput and reduces latency by up to $8.42\times$ for AMD SEV-SNP. On Intel TDX, throughput improves by up to $2.47\times$, while latency is reduced by up to $4.45\times$.

7.2. RQ2: Comparison to TLS, TCP, and Unencrypted RDMA

TLS. An alternate approach to establish a secure connection between two CVMs is to use a secure TLS connection. We benchmark multiple TLS configurations using sockperf [34], a network benchmarking tool for TCP and TLS by Mellanox. We compare TLS latency and throughput for 3 configurations using: (1) a dedicated Intel Ethernet X540 card, (2) the Mellanox ConnectX-5 NIC in Ethernet mode, and (3) the Mellanox ConnectX-5 NIC in InfiniBand mode, enabling TLS with the IPoIB driver [36]. The Intel Ethernet X540 is the best-performing configuration, achieving the lowest latency at comparable throughput to all other configurations (Appendix C.3). We select it as our TLS baseline, reaching up to 5.19 Gb/s. We compare the TLS throughput and latency with optimized IRMA in Figure 7 and Figure 8. Configured with the correct chunk size, optimized IRMA beats TLS for all message sizes.

Compared to TLS, optimized IRMA reaches up to $17.17\times$ higher throughput and up to $46.70\times$ lower latency on AMD SEV-SNP. On Intel TDX throughput is up to $17.23\times$ higher and latency up to $16.54\times$ lower compared to TLS.

TABLE 2: Relative throughput speedup of optimized IRMA vs. unoptimized IRMA for a different number of clients (Higher is better).

	1 Client	2 Clients	3 Clients	4 Clients
AMD SEV-SNP	1.20×	1.10×	1.00×	1.00×
Intel TDX	1.53×	1.39×	1.05×	1.07×

TCP. For completeness, we also compare the throughput and latency of optimized IRMA to unencrypted TCP. It should be noted that unencrypted TCP is not secure to use in our threat model. Configured with the correct chunk size, optimized IRMA beats TCP for all message sizes.

Compared to TCP, optimized IRMA reaches up to $9.43\times$ higher throughput and up to $42.10\times$ lower latency on AMD SEV-SNP. On Intel TDX throughput is up to $9.46\times$ higher and latency up to $14.90\times$ lower than TCP.

Unencrypted RDMA. We compare optimized IRMA with unencrypted RDMA. It should be noted that unencrypted RDMA is not secure to use in our threat model and does not work inside CVMs. We therefore measure unencrypted RDMA on normal VMs instead of CVMs.

Optimized IRMA on AMD SEV-SNP reaches up to $0.85\times$ in throughput and has up to $4.94\times$ higher latency compared to unencrypted RDMA. On Intel TDX, IRMA reaches up to $0.85\times$ in throughput and has up to $8.85\times$ higher latency compared to unencrypted RDMA.

7.3. RQ3: Scalability

We evaluate IRMA’s scalability by measuring the throughput for an increasing number of RDMA connections for optimized and unoptimized IRMA. We execute a server on Host A that accepts incoming RDMA connections from clients in CVMs on Host B. Because Host B has 4 CCX blocks on AMD SEV-SNP, we measure up to 4 clients. Table 2 shows that optimized IRMA improves throughput for 1 client by $1.20\times$ (AMD SEV-SNP) and $1.53\times$ (Intel TDX). For 2 clients the throughput improves by $1.10\times$ (AMD SEV-SNP) and $1.39\times$ (Intel TDX). For more clients, this effect is reduced, because the 100 Gb/s RDMA link is saturated, and all clients compete for bandwidth. Once the link is saturated, the bandwidth of the link becomes the bottleneck with each client being limited by its slice of the bandwidth.

We observe that the bandwidth of the link becomes the bottleneck for scalability, i.e., optimized IRMA does not impact scalability.

7.4. RQ4: Case Studies

Valkey. Valkey is a high-performance key-value store which supports both TLS and RDMA. Because TLS is an alterna-

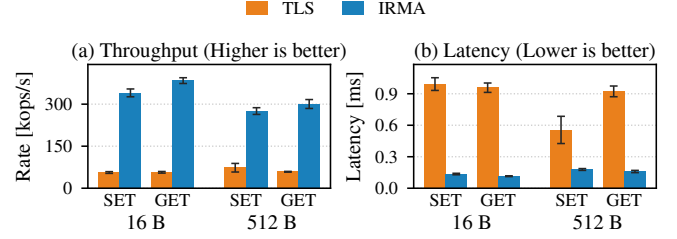


Figure 9: Performance of Valkey using IRMA and TLS.

tive secure network protocol for CVMs we compare Valkey running the TLS backend with Valkey running RDMA using IRMA. Our initial experiments of Valkey on AMD using an unencrypted VM showed that unencrypted RDMA does not yield a significant improvement compared to TLS. We assume this to be the case due to smaller cache sizes on AMD. Therefore, we run our evaluation only on Intel TDX. Figure 9 shows the ops/sec and latency for GET and SET operations. We evaluate message sizes of 16 B and 512 B and enable Valkey to batch up to 64 messages together. We select the optimal chunk sizes for each message size and set the chunk size to 3.6 KB and 34.75 KB respectively.

Using optimized IRMA, Valkey GET operations are up to $6.76\times$ faster and SET operations are up to $6.07\times$ faster compared to TLS. For latency, optimized IRMA is up to $8.36\times$ lower for GET operations and up to $7.34\times$ lower for SET operations compared to TLS.

GAM. GAM [13] is an efficient distributed in-memory platform that provides a directory-based cache coherence protocol over RDMA. We run GAM with optimized and unoptimized IRMA on 3 CVMs, with one CVM located on Host A and two CVMs located on Host B. GAM sends small messages of up to 512 B, so we configure IRMA chunk size to 512 B. We use the Distributed Hash Table (DHT) implemented in GAM to benchmark IRMA [13]. Table 3 shows the speedup of optimized over unoptimized IRMA for ops/sec and latency for read-most (95% reads), balanced (50% reads and 50% writes), and write-most (95% writes) YCSB traces [60].

Optimized IRMA achieves up to $1.23\times$ (AMD SEV-SNP) and $1.15\times$ (Intel TDX) higher ops/sec and up to $1.31\times$ (AMD SEV-SNP) and $1.22\times$ (Intel TDX) lower latency than unoptimized IRMA.

Hdrdmacp. Hdrdmacp [27] is a program that copies large files between two nodes using RDMA. We run the receiving node in a CVM on Host A, and the sending node in a CVM on Host B. We run hdrdmacp with default parameters for the buffer sizes (8 GB for the server, 1 GB for client) and transmit 10 GB files from the client to the server to saturate the buffers. Hdrdmacp lets users configure a slot size that determines the number of slots a buffer is split into. We benchmark the default slot size of 4 and additionally a slot

TABLE 3: Relative throughput speedup and latency reduction for optimized IRMA vs. unoptimized IRMA for GAM (Higher is better).

	Relative Throughput Speedup			Relative Read Latency Reduction			Relative Write Latency Reduction		
	95% read	50% read	5% read	95% read	50% read	5% read	95% read	50% read	5% read
AMD SEV-SNP	1.04×	1.23×	1.14×	1.08×	1.23×	1.18×	1.29×	1.31×	1.19×
Intel TDX	1.10×	1.11×	1.15×	1.11×	1.17×	1.14×	1.14×	1.22×	1.15×

size of 32. We configure IRMA with chunk sizes of 4 MB and 8 MB, to reduce the overhead of processing many small chunks, while still benefiting from pipelining. We observe that optimized IRMA reaches a throughput of up to 16.74 Gb/s (AMD SEV-SNP) and 14.18 Gb/s (Intel TDX). Table 4 shows the speedup from unoptimized to optimized IRMA for AMD SEV-SNP and Intel TDX.

Optimized IRMA increases throughput by up to 1.32× (AMD SEV-SNP) and 1.04× (Intel TDX) compared to unoptimized IRMA.

TABLE 4: Relative throughput speedup of optimized IRMA vs. unoptimized IRMA for hdrdmacp (Higher is better).

	4 MB Chunks		8 MB Chunks	
	4 Slots	32 Slots	4 Slots	32 Slots
AMD SEV-SNP	1.20×	1.16×	1.18×	1.32×
Intel TDX	1.02×	1.03×	1.01×	1.04×

7.5. RQ5: Adapting the Out-of-Band Channel

GAM and hdrdmacp use TCP to exchange data over the out-of-band channel. We adapt Valkey to use TCP to exchange connection data over the out-of-band channel. To secure the out-of-band channel, we use an attested TLS library that executes mutual attestation between the CVMs before setting up a TLS channel [55]. To secure the TLS channel for GAM, hdrdmacp, and Valkey we changed 378 LoC, 276 LoC, and 291 LoC respectively. It took under one hour for one researcher to make these changes.

8. Related Work

Next, we discuss prior works on RDMA, followed by the existing works on networking and devices for CVMs.

RDMA attacks. Prior works [38], [41], [46], [51], [53] have demonstrated a range of attacks on RDMA communication by network adversaries. While these works do not consider the confidential computing threat model like IRMA, they use a strong network adversary. These attacks fall into denial-of-service (DoS) and control-plane categories. DoS attacks are out of scope in our threat model. The control-plane attacks, analyzed in Section 5, do not affect IRMA’s protected app-view memory. Furthermore, IRMA’s encryption blocks any such attacks targeting its unprotected RNIC-view memory. In the confidential computing threat model, an untrusted cloud hypervisor can mount network-style attacks by manipulating data in the RNIC’s unprotected view. Our

analysis in Section 5 shows that IRMA’s encryption detects and prevents these attacks. Beyond prior RDMA attacks, IRMA identifies out-of-band communication as a new avenue for compromising the RDMA channel.

RDMA defenses. Prior works [15], [33], [47], [59] address RDMA attacks assuming trusted cloud provider software (e.g., hypervisors) and hardware (e.g., RNICs, processors). These solutions rely on trusted software on the hosts, cloud-provider-controlled network routing rules, and enhanced monitoring on trusted RNICs to bind RDMA queue pair endpoints and their respective memory regions, implement access control, and stop malicious accesses. However, we cannot use these mechanisms in the confidential computing setting where we do not trust the cloud provider’s infrastructure. Instead, IRMA uses mutual authentication between CVMs to securely bind RDMA queue pairs, and establishes secure channels for both the out-of-band and the RDMA data, preventing any tampering by the cloud provider.

Bounce buffers. Several prior works use encrypted bounce buffers to attach trusted devices to CVMs [29], [30]. To enable these bounce buffers the Linux kernel provides the SWIOTLB for CVMs [1], [61]. However, as shown in Section 6, these approaches are incompatible with RDMA which does not invoke the Linux kernel, not triggering bounce buffer operations. In IRMA, we design a kernel-independent mechanism for encrypting bounce buffers. More importantly, we show that just bounce buffers are not sufficient for securing RDMA. Intel SGX-based solutions for networking do not support RDMA [9], [28], [39], [48], [50]. To enable RDMA, IRMA library and IRMA manager can execute inside SGX enclaves. Running IRMA inside SGX needs further work on library OSe and kernel support.

Networking for confidential computing. To enable fast CVM networking, prior works propose optimized mechanisms to secure the TCP/IP stack with encryption [29], [30]. While these improve TCP performance for CVMs, they cannot be applied to RDMA as in IRMA. Adding Trusted I/O (TIO) support in next-generation RNICs is useful [20], [22], but one would still need IRMA as explained in Section 3.4.

9. Conclusion

We present IRMA, a design that reduces the network I/O tax in the confidential computing setting using RDMA. To this end, IRMA preserves RDMA communication semantics under strict memory isolation, untrusted RNICs, and network attackers. Our evaluation shows that IRMA significantly reduces the network I/O tax compared to TCP/TLS-based alternatives while maintaining compatibility and performance for existing RDMA apps.

Acknowledgments

We thank the anonymous reviewers for their feedback. We thank Ghassan Karame, Clément Thorens, and Philip Giersfeld for their helpful feedback, and Lejla Skelić for her feedback and for checking that the code repository submitted with the paper was properly anonymized.

References

- [1] DMA and swiotlb—The Linux Kernel Documentation. <https://docs.kernel.org/core-api/swiotlb.html>, 2026. Accessed: Feb. 3, 2026.
- [2] IRMA. <https://github.com/IRMA-CC>, 2026. Accessed: Oct. 7, 2026.
- [3] libaegis. <https://github.com/aegis-aead/libaegis>, 2026. Accessed: Jan. 31, 2026.
- [4] libsodium. <https://github.com/jedisct1/libsodium>, 2026. Accessed: Jan. 31, 2026.
- [5] Advanced Micro Devices, Inc. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. <https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf>, 2023. Accessed: Feb. 3, 2026.
- [6] Advanced Micro Devices, Inc. AMD Secure Encrypted Virtualization (SEV). <https://www.amd.com/en/developer/sev.html>, 2026. Accessed: Feb. 3, 2026.
- [7] Advanced Micro Devices, Inc. AMDSEV. <https://github.com/AMD-ESE/AMDSEV>, 2026. Accessed: Jan. 31, 2026.
- [8] Advanced Micro Devices, Inc. Multi-Node Network Configuration for AMD Instinct Accelerators. <https://instinct.docs.amd.com/projects/gpu-cluster-networking/en/latest/how-to/multi-node-config.html>, 2026. Accessed: Feb. 6, 2026.
- [9] Mansour Alharthi, Fan Sang, Dmitrii Kuvaitskii, Mona Vij, and Taesoo Kim. RAKIS: Secure Fast I/O Primitives Across Trust Boundaries on Intel SGX. In *ACM European Conference on Computer Systems (EuroSys)*, 2025.
- [10] Arm Limited. Arm Realm Management Extension (RME) System Architecture. <https://developer.arm.com/documentation/den0129/latest/>, 2026. Accessed: Feb. 3, 2026.
- [11] Arm Limited. Device Assignment and Memory Encryption Contexts. <https://developer.arm.com/documentation/den0125/400/Device-Assignment-and-Memory-Encryption-Contexts>, 2026. Accessed: Feb. 3, 2026.
- [12] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering Azure Storage with RDMA. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.
- [13] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient Distributed Memory Management with RDMA and Caching. *Proceedings of the VLDB Endowment (PVLDB)*, 2018.
- [14] Canonical, Ltd. Intel TDX. <https://github.com/canonical/tdx>, 2026. Accessed: May 19, 2026.
- [15] Hyunseok Chang and Sarit Mukherjee. Zeta: Transparent Zero-Trust Security Add-on for RDMA. In *IEEE Conference on Computer Communications (INFOCOM)*, 2024.
- [16] Gobikrishna Dhanuskodi, Sudeshna Guha, Vidhya Krishnan, Aruna Manjunatha, Rob Nertney, Michael O'Connor, and Phil Rogers. Creating the First Confidential GPUs. *Communications of the ACM*, 2024.
- [17] Erhu Feng, Dahu Feng, Dong Du, Yubin Xia, Wenbin Zheng, Siqi Zhao, and Haibo Chen. sIOPMP: Scalable and Efficient I/O Protection for TEEs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.
- [18] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When Cloud Storage Meets RDMA. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.
- [19] Stefan Gast, Hannes Weisstener, Robin Schröder, and Daniel Gruss. CounterSEVveillance: Performance-Counter Attacks on AMD SEV-SNP. In *Network and Distributed System Security Symposium (NDSS)*, 2025.
- [20] Dimitra Giantsidi, Julian Pritzi, Felix Gust, Antonios Katsarakis, Atsushi Koshiba, and Pramod Bhatotia. TNIC: A Trusted NIC Architecture: A Hardware-Network Substrate for Building High-Performance Trustworthy Distributed Systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.
- [21] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *ACM SIGCOMM Conference (SIGCOMM)*, 2016.
- [22] Taekyung Heo, Seunghyo Kang, Sanghyeon Lee, Soojin Hwang, Joongun Park, and Jaehyuk Huh. Supporting Trusted Virtual Machines with Hardware-Based Secure Remote Memory. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2024.
- [23] Jinbin Hu, Houqiang Shen, Xuchong Liu, and Jin Wang. RDMA Transports in Datacenter Networks: Survey. *IEEE Network*, 2024.
- [24] Intel Corporation. Intel TDX Connect Architecture Specification. <https://www.intel.com/content/www/us/en/content-details/773614/intel-tdx-connect-architecture-specification.html>, 2023. Accessed: Feb. 3, 2026.
- [25] Intel Corporation. Intel Omni-Path Performance Tuning. https://www.intel.com/content/dam/support/us/en/documents/network/omni-adp-tr/sb/Intel_OP_Performance_Tuning_UG_H93143_v1_0.pdf, 2026. Accessed: Feb. 6, 2026.
- [26] Intel Corporation. Intel Trust Domain Extensions (Intel TDX). <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html>, 2026. Accessed: Feb. 4, 2026.
- [27] Jefferson Science Associates, LLC for the U.S. Department of Energy. hrdmapc. <https://github.com/JeffersonLab/hrdmapc>, 2026. Accessed: Jan. 31, 2026.
- [28] Dmitrii Kuvaitskii, Dimitrios Stavrakakis, Kailun Qin, Cedric Xing, Pramod Bhatotia, and Mona Vij. Gramine-TDX: A Lightweight OS Kernel for Confidential VMs. In *ACM Conference on Computer and Communications Security (CCS)*, 2024.
- [29] Dingji Li, Zeyu Mi, Chenhui Ji, Yifan Tan, Binyu Zang, Haibing Guan, and Haibo Chen. Bifrost: Analysis and Optimization of Network I/O Tax in Confidential Virtual Machines. In *USENIX Annual Technical Conference (ATC)*, 2023.

- [30] Mengyuan Li, Shashvat Srivastava, and Mengjia Yan. Bridge the Future: High-Performance Networks in Confidential VMs without Trusted I/O Devices. <https://arxiv.org/abs/2403.03360>, 2024.
- [31] linux-rdma. libibverbs. <https://github.com/linux-rdma/rdma-core>, 2026. Accessed: Jan. 31, 2026.
- [32] linux-rdma. perftest. <https://github.com/linux-rdma/perftest>, 2026. Accessed: Feb. 5, 2026.
- [33] Hongyi Liu, Jiarong Xing, Yibo Huang, Danyang Zhuo, Srinivas Devadas, and Ang Chen. Remote Direct Memory Introspection. In *USENIX Security Symposium (USENIX Security)*, 2023.
- [34] Mellanox Technologies Ltd. sockperf. <https://github.com/Mellanox/sockperf>, 2026. Accessed: Feb. 1, 2026.
- [35] NVIDIA. Confidential Compute on NVIDIA Hopper H100. <https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitewater-v1.0.pdf>, 2024. Accessed: Feb. 3, 2026.
- [36] NVIDIA. IP over InfiniBand (iPoIB). [https://docs.nvidia.com/networking/display/mlnxofedv23070512/ip+over+infiniband+\(ipoib\)](https://docs.nvidia.com/networking/display/mlnxofedv23070512/ip+over+infiniband+(ipoib)), 2026. Accessed: Feb. 4, 2026.
- [37] NVIDIA. NVIDIA Mellanox ConnectX-5 Adapters. <https://www.nvidia.com/en-sg/networking/ethernet/connectx-5>, 2026. Accessed: Feb. 3, 2026.
- [38] J. Pinkerton and E. Deegan. Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMA) Security. Technical Report RFC 5042, Internet Engineering Task Force (IETF), 2007.
- [39] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. SafeBricks: Shielding Network Functions in the Cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.
- [40] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A Remote Direct Memory Access Protocol Specification. Technical Report RFC 5040, Internet Engineering Task Force (IETF), 2007.
- [41] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoeftler. ReDMArk: Bypassing RDMA Security Mechanisms. In *USENIX Security Symposium (USENIX Security)*, 2021.
- [42] Benedict Schlüter, Christoph Wech, and Shweta Shinde. Heracles: Chosen Plaintext Attack on AMD SEV-SNP. In *ACM Conference on Computer and Communications Security (CCS)*, 2025.
- [43] Anna Kornfeld Simpson, Adriana Szekeres, Jacob Nelson, and Irene Zhang. Securing RDMA for High-Performance Datacenter Storage Systems. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2020.
- [44] Richard Solomon. New PCIe TDISP Architecture Secures Device Interfaces with Virtual Servers. <https://www.chipestimate.com/New-PCIe-TDISP-Architecture-Secures-Device-Interfaces-with-Virtual-Servers/Synopsys/blogs/3759>, 2023. Accessed: Oct. 6, 2026.
- [45] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, Mark Kuhne, Fabio Aliberti, and Shweta Shinde. ACAI: Protecting Accelerator Execution with Arm Confidential Computing Architecture. In *USENIX Security Symposium (USENIX Security)*, 2024.
- [46] Konstantin Taranov, Benjamin Rothenberger, Daniele De Sensi, Adrian Perrig, and Torsten Hoeftler. NeVerMore: Exploiting RDMA Mistakes in NVMe-oF Storage Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.
- [47] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoeftler. sRDMA: Efficient NIC-Based Authentication and Encryption for Remote Direct Memory Access. In *USENIX Annual Technical Conference (ATC)*, 2020.
- [48] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. rkt-io: A Direct I/O Stack for Shielded Execution. In *ACM European Conference on Computer Systems (EuroSys)*, 2021.
- [49] The QEMU Project Developers. QEMU. <https://www.qemu.org>, 2025. Accessed: Jan. 31, 2026.
- [50] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. ShieldBox: Secure Middleboxes Using Shielded Execution. In *ACM Symposium on SDN Research (SOSR)*, 2018.
- [51] Shin-Yeh Tsai and Yiyang Zhang. A Double-Edged Sword: Security Threats and Opportunities in One-Sided Network Communication. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.
- [52] Chenxu Wang, Fengwei Zhang, Yunjie Deng, Kevin Leach, Jiannong Cao, Zhenyu Ning, Shoumeng Yan, and Zhengyu He. CAGE: Complementing Arm CCA with GPU Extensions. In *Network and Distributed System Security Symposium (NDSS)*, 2024.
- [53] Shicheng Wang, Menghao Zhang, Yuying Du, Ziteng Chen, Zhiliang Wang, Mingwei Xu, Renjie Xie, and Jiahai Yang. LoRDMA: A New Low-Rate DoS Attack in RDMA Networks. In *Network and Distributed System Security Symposium (NDSS)*, 2024.
- [54] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchun Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, Tao Ding, Kai Chen, and Chuanxiong Guo. SRNIC: A Scalable Architecture for RDMA NICs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.
- [55] Carsten Weinhold, Muhammad Usama Sardar, Ionuț Mihalcea, Yogesh Deshpande, Hannes Tschofenig, Yaron Sheffer, Thomas Fossati, and Michael Roitzsch. Separate but Together: Integrating Remote Attestation into TLS. In *USENIX Annual Technical Conference (ATC)*, 2025.
- [56] John Simon Wejin, Adewale Ajao Ajao, and Kennedy Okokpujie. Comparative Simulation of TCP and TLS Protocols in SDN Network. In *International Conference on Science, Engineering and Business for Driving Sustainable Development Goals (SEB4SDG)*, 2024.
- [57] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX. In *ACM Conference on Computer and Communications Security (CCS)*, 2024.
- [58] Hongjun Wu and Bart Preneel. AEGIS: A Fast Authenticated Encryption Algorithm. In *Selected Areas in Cryptography (SAC)*, 2013.
- [59] Jiarong Xing, Kuo-Feng Hsu, Yiming Qiu, Ziyang Yang, Hongyi Liu, and Ang Chen. Bedrock: Programmable Network Support for Secure RDMA Systems. In *USENIX Security Symposium (USENIX Security)*, 2022.
- [60] Yahoo! Inc. and YCSB contributors. YCSB. <https://github.com/brianfrankcooper/YCSB>, 2026. Accessed: Feb. 4, 2026.
- [61] Dongli Zhang. swiotlb: 64-bit DMA Buffer. <https://lwn.net/Articles/845096>, 2021. Accessed: Feb. 3, 2026.
- [62] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-Based Fault Injection Using Selective State Reset. In *USENIX Security Symposium (USENIX Security)*, 2024.

Appendix A. Analysis

A.1. Out-of-Band Channel Compatibility Analysis

We go into details of why certain approaches are infeasible for the out-of-band channel when ensuring app compatibility with IRMA. The out-of-band channel exchanges information about shared memory regions that apps can dynamically create depending on app-specific logic. Securing this out-of-band channel requires app-specific context to establish a ground truth. We can execute the app in a benign environment, and use the app-specific context to build a ground truth. This ground truth should capture which memory regions the app accesses and at what points during the benign execution. At run time, we can use this ground truth to detect any tampering. To build the ground truth, we can consider a mechanism that leverages static binary instrumentation and checks to ensure that the app only accesses expected memory regions at fixed points in its execution flow. We observe that apps create the shared memory regions dynamically, with addresses that might change between executions. So, static binary instrumentation that tracks this information to build a ground truth is not feasible to always guarantee security.

Appendix B. Design

B.1. Supporting RDMA APIs in IRMA

In the following, we explain the standard RDMA terms used by libibverbs. Table 5 details the changes IRMA library makes to the libibverbs API.

RDMA device context (CTXT). The RDMA device context is the user-space handle to an RNIC. It acts as the root object from which almost all other RDMA resources are created.

Protection domain (PD). A Protection Domain is a hardware-enforced isolation container, that controls which RDMA resources are allowed to interact with each other. The protection domain can be used to create associated Queue Pairs and register Memory Regions.

Queue pair (QP). A Queue Pair is the communication endpoint used for RDMA operations. It consists of a Send Queue (SQ) and Receive Queue (RQ).

Memory region (MR). Registering a Memory Region with a Protection Domain allows an RNIC to read and write data to this memory.

Work request (WR). A Work Request is a command that can be posted to the Send or Receive Queue of a Queue pair. Work Requests typically consist of an operation such as READ, WRITE, or SEND, and a list of data of registered Memory Regions on which to perform them.

Work completion (WC). A Work Completion signals the completion of a Work Request.

Completion queue (CQ). When a Work Request submitted to the Send or Receive Queue is completed, a Work Completion is added to the Completion Queue.

Appendix C. Implementation and Evaluation

C.1. IRMA Manager

When a CVM hosts multiple RDMA apps, each app has an independent IRMA library instance. If multiple instances listen for attested TLS sessions simultaneously, incoming requests cannot be easily differentiated. To solve this, we build IRMA manager with 665 LoC. Each CVM runs a single IRMA manager in protected memory. When an RDMA app starts, its IRMA library instance connects to the IRMA manager through a socket and registers its queue pairs. The IRMA manager opens a single socket to listen for incoming attested TLS connections. When a TLS connection is established, the IRMA manager routes the information to the correct IRMA library instance. After the connection is established, the IRMA library instance disconnects from the IRMA manager.

C.2. Platform Specifications

Table 6 lists the specifications of the AMD SEV-SNP and Intel TDX hosts.

TABLE 6: Specifications of the AMD SEV-SNP and Intel TDX hosts.

	AMD SEV-SNP	Intel TDX
CPU	AMD EPYC 9004	Intel Xeon Gold 6526Y
Clock	3.7 GHz	3.9 GHz
Cores	32 (4 CCX $\times$ 8)	32
L1 / L2 / L3	1 / 16 / 64 MiB	1.25 / 32 / 37.5 MiB
RAM	192 GB (A), 160 GB (B)	128 GB
Host kernel	6.11.0	6.8.0
OS	Ubuntu 24.04.1 LTS	Ubuntu 24.04.2 LTS

C.3. TLS Performance

Figure 10 compares the TLS throughput and latency for multiple different NIC configurations.

C.4. Encryption Performance per Chunk Size

Figure 11 compares the performance of multiple AEAD algorithms from libsodium [4] and libaegis [3] in AMD SEV-SNP and Intel TDX CVMs. We benchmark the average cycles it takes to encrypt and decrypt different chunk sizes.

Appendix D. LLM Usage Statement

LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality. LLMs were used specifically for development-stage code completion and grammar and spelling checks.

TABLE 5: Default and IRMA behavior for the libibverbs API.

Libibverbs API	Default behavior	IRMA behavior
ibv_open_device	Create CTXT for RNIC used by other verbs	Create CTXT for RNIC view and emulate CTXT for the CVM view. Replaces default API functionality with IRMA functionality in the emulated RDMA device context
ibv_close_device	Closes RDMA device context	Close RNIC view CTXT associated with emulated CVM view CTXT
ibv_alloc_pd	Allocates PD for a CTXT	Allocate PD for RNIC view and create emulated PD for CVM view
ibv_dealloc_pd	Frees PD for a CTXT	Free RNIC view PD associated with emulated CVM view PD
ibv_create_cq	Create a CQ for a CTXT	Create two CQs for RNIC view to separately track send and receive operations. Create emulated CQ for CVM view
ibv_destroy_cq	Destroy CQs for a CTXT	Destroy RNIC view CQs associated with emulated CVM view CQ
ibv_create_qp	Create a QP associated with a PD	Create a Data QP and a Control QP for RNIC view. The Data QP is used for sending the data. The Control QP is used to send acknowledgments. Allocate unprotected memory for RX and TX ring buffer. Register ring buffers with the underlying RNIC view. Create an emulated QP for the CVM view.
ibv_modify_qp	Modifies the attributes of a QP	Modify the attributes of the emulated CVM view QP. Modify selected attributes of the RNIC QPs.
ibv_query_qp	Returns the attributes and current values of a QP	Returns the attributes and current value of an emulated QP
ibv_destroy_qp	Destroy a QP associated with a PD	Destroy the QPs of the RNIC view associated with the emulated QP and PD. Deregister and free the RX and TX ring buffer memory in the RNIC view. Destroy the emulated CVM view QP.
ibv_reg_mr	Registers an MR associated with a PD	Register MR associated with a PD in the CVM view, but not with the underlying RNIC view.
ibv_dereg_mr	Unregisters an MR associated with a PD	Deregister MR associated with a PD from the CVM view.
ibv_post_send	Posts a list of WRs to the SQ of a QP	Split each WR into chunks and encrypt each chunk into the TX ring buffer. Post a WR of the encrypted chunk in the TX ring buffer to the SQ of the Data QP.
ibv_post_recv	Posts a list of WRs to the RQ of a QP	Save submitted WRs in IRMA and scatter data received through RX ring buffer according to the WRs.
ibv_create_comp_channel	Create a completion channel for a CTXT	Create an emulated completion channel for the CVM view.
ibv_req_notify_cq	Requests a WC on a CQ	Notifies IRMA to post WCs for an emulated CQ to the emulated CVM view completion channel
ibv_poll_cq	Polls WCs from a CQ	Poll the underlying CQ to check if chunked data arrived. If all chunks of a WR arrived, post a WC to the emulated CQ.
ibv_destroy_completion_channel	Destroy the completion channel for a CTXT	Destroy the CVM view emulated completion channel for a CTXT

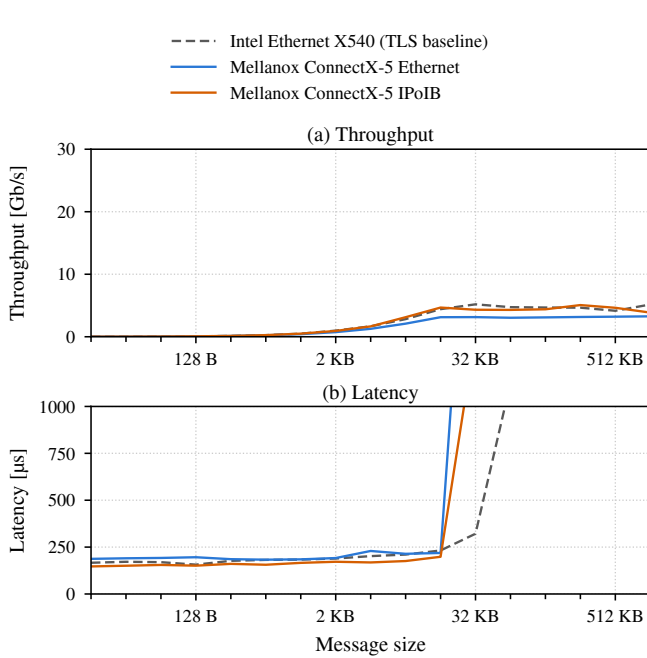


Figure 10: Mean TLS throughput (a) and latency (b) for different NIC configurations.

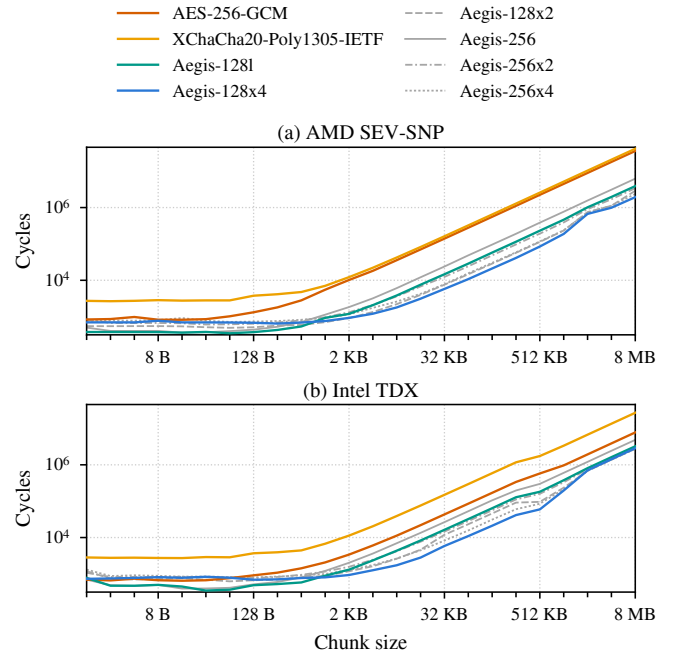


Figure 11: Mean cycles each AEAD algorithm requires to encrypt and decrypt different chunk sizes in AMD SEV-SNP and Intel TDX CVMs (Lower is better).