DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Fast and Secure Networking for Remote Direct Memory Access on Persistent Memory Leveraging Trusted Execution Environments

Philip Sändig

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Fast and Secure Networking for Remote Direct Memory Access on Persistent Memory Leveraging Trusted Execution Environments

Schnelle und sichere Netzwerk-Kommunikation für Remote Direct Memory Access auf Persistent Memory mithilfe von Trusted Execution Environments

Author:	Philip Sändig
Supervisor:	Prof. Pramod Bhatotia
Advisor:	Dimitrios Stavrakakis, Dimitra Giantsidi
Submission Date:	13/08/2021

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 13/08/2021 Philip Sändig

Acknowledgments

First of all, I would like to express my sincere thanks to my supervisor, Professor Pramod Bhatotia for offering me the thesis topic and for giving me the chance to participate in the anchor project.

I especially want to thank my advisors Dimitra and Dimitrios who always supported me when I needed their advice and helped me with many of my problems.

Lastly, I would like to thank my family and friends for supporting me throughout the time in which I worked for my thesis. A special thanks goes out to Johanna, Maxi, Simon and Simon for your support (not only) in our motivational Monday meetings.

Abstract

Since cloud environments gain popularity, new technologies for enhancing the security and performance properties of hosted services are developed. Therefore, in the last few years, research focused on using Trusted Execution Environments (TEEs) for protecting against attacks of a host. Furthermore, as Persistent Memory (PM) finds its way into data centers, the need for fast, low-latency userspace networking approaches like RDMA increases. However, TEEs are neither designed for protecting PM, nor for securing userspace networking.

In this thesis, we want to deliver answers to the question *How can we build a secure, high-performant network stack for use with PM under the assumption of an untrusted host?* Therefore, we will investigate two approaches: *(i)* an approach with one-sided RDMA and *(ii)* an approach with two-sided networking in general. We find that only the second approach lives up to our goals for performance, security, and generality. We, therefore, implement a network stack for userspace networking that guarantees confidentiality, integrity, and freshness. To reach these goals, we implement an encryption layer on top of a library providing us with fast Remote Procedure Calls (RPCs). We will show that, by making use of userspace networking, we get up to 60% better throughput compared to kernel TCP. The latency of our implementation only increases by at most $4\times$ when running inside a TEE.

Contents

1 Introduction

As there has been a massive growth in the use of cloud services in the last years that does not seem to cease [38, 48, 50], protection of data becomes an issue. The main problem is that cloud environments can neither be fully observed nor trusted by a client. As a solution for this, Trusted Execution Environments (TEEs) have recently gained popularity in systems research projects [2, 3, 4, 43, 44]. Since container environments that transparently use TEEs exist [2], there is plenty of research work on trying to leverage it for building secure systems in cloud environments [3, 4, 43, 44]. Another emerging technology promising low latency and big memory capacities is Persistent Memory (PM). It introduces new challenges for networking as well, as access latencies with PM are shorter than network roundtrip times [24]. The need for fast, low-latency networking resulted in the increasing popularity of userspace networking approaches like RDMA that can also be applied to PM [17]. Recent work especially focuses on making RDMA more efficient for use with PM [24, 25, 51]. There are also many approaches trying to secure RDMA [37, 42], but neither of them deals with RDMA on PM in untrusted environments. Hence, we aim to present a high-performant networking approach that works on untrusted host systems and can be used by secure PM architectures. This approach should result in a network stack that guarantees confidentiality, integrity, and freshness. While designing the network stack, we face three main challenges.

Firstly, TEEs are not trivially compatible with PM and Direct Memory Access (DMA) [41] approaches. The security guarantees of TEEs are limited to small main memory regions. Moreover, in TEEs, I/O devices, such as Network Interface Cards (NICs) do not have access to protected memory regions.

For the one-sided approach, this can be resolved by storing all data inside PM encrypted and authenticated. To solve this issue for the two-sided approach, we encrypt messages inside the enclave and pass encrypted messages to untrusted host memory where the NIC can access them. This approach was already successfully applied in existing work [3].

Secondly, system calls in TEEs induce high performance overheads [2]. Therefore, frequent system calls need to be avoided for reaching low latencies and high throughputs.

To solve this issue, we make use of userspace networking, as it reduces the number of system calls. The performance gain from userspace I/O has already been shown in several existing projects [3, 4, 43, 44].

Lastly, PM has a low latency compared to network roundtrip times [24]. Hence we need to reduce overheads to lower the average roundtrip latency and to minimize the number of roundtrips wherever possible.

The solution for this challenge is again userspace networking which grants us low-latency message processing. Furthermore, we provide asynchronous operations. This means, that a high number of messages can be enqueued without having received a response. We will show that we can thus reach a high number of roundtrips in a short time. Moreover, we provide the possibility to let our implementation run multi-threaded which also improves the performance. Lastly, we assert that with a two-sided approach, the number of network roundtrips can be reduced in comparison to one-sided approaches, if the PM host takes care of all PM accesses.

In this thesis, after having an introductory background part (§ 2), we will discuss further challenges for the two different secure networking approaches in detail (§ 3). Here, we will also present the two approaches: a one-sided approach for RDMA and a two-sided approach for userspace networking in general. After we see that the one-sided approach is not suited to match our security guarantees while having a high performance, we implement a two-sided network stack that protects against active attackers from the network and the host operating system (§ 4). This implementation is then benchmarked to prove that the design of our network stack lives up to our performance requirements (§ 5). In the end, we will introduce related and possible future work (§ 6, 7).

2 Background

2.1 Persistent Memory

Persistent Memory (PM), also called Non-volatile Main Memory (NVMM), is a recent storage technology aiming to provide low latency (100-1000 ns) while having high capacities. It thus closes the performance gap between SSDs and DRAM [49]. PM shares many similarities with DRAM. It is connected to the main memory bus, can be accessed with cache-line granularity, and is compatible with Direct Memory Access (DMA), meaning that also devices other than the CPU can access it [36].

Novel Programming Model. Because of its persistence property, one key difference to DRAM, a new programming model for PM has been introduced by the Storage Networking Industry Association [29]. An implementation of this model is Intel's Persistent Memory Development Kit (PMDK) [18], which is a collection of different libraries for the use of PM. Multiple abstractions can be used for accessing PM. Therefore, a couple of different libraries exist in PMDK [16]. One common approach supported by the Programming Model is to map PM into DRAM, the same way as with files on conventional background storage. This feature, called Direct Access (DAX), has the advantage that after the mapping is set up, PM access is possible without further interaction with the kernel. Another significant aspect of the Programming Model is that writes are not made persistent instantly. Instead, for ensuring better write-performance, usually, only the CPU cache is updated and changes need to be flushed to persistency. Write atomicity is only guaranteed for write sizes of up to 8 Bytes. This is especially important for systems that aim to ensure crash-consistency guarantees. [36]

Security issues. Having persistent main memory comes with the problem that on a system shutdown or crash, all data stored in PM is preserved. On the one hand, this is can be beneficial. On reboot, data stored on PM can be reused without the need for being reloaded. On the other hand, this implies that after a reboot, anyone could access any data inside PM. While for volatile memory, paging and other security mechanisms can protect memory regions from being accessed by unprivileged processes, this is not the case for PM anymore. [36]

2.2 Trusted Execution Environments

Since cloud computing becomes more and more popular, it is important to protect data in the cloud. As cloud servers are often not accessible directly, this leaves us with the question, what one could do to protect data even if the cloud environment is not trusted. Processor manufacturers answer that issue with Trusted Execution Environments (TEEs), for example Intel SGX [15], ARM TrustZone [46] and AMD SEV [1].

Intel SGX. In the case of Intel Software Guard Extensions (SGX) [15], so-called enclaves are provided as TEEs. These are environments that protect memory regions of a process from access by other processes. These memory regions are called Processor Reserved Memory (PRM). Inside this memory region, there is the Enclave Page Cache (EPC), which is the collection of pages of a process protected by the enclave. Because the EPC has a limited size, processes need to use it efficiently. A higher memory usage leads to massive performance overheads [2]. For running untrusted code from an enclave application, as for system calls that need to be executed by the host OS, the CPU needs to exit the enclave. When exiting and entering, program data, such as the CPU registers of the enclave process, need to be saved, encrypted, and stored. Therefore, system calls induce a high overhead and need to be avoided in favor of performance [2]. [7, 10]

Attestation. Another important concept for enclaves is attestation. After a remote attestation process, a client of a service running remotely, for instance at a cloud provider, can ensure that his code is running securely. This includes that he knows attributes of the currently running enclave and measurements of his code so that he knows that it is the expected code running unmodified inside an enclave. [7, 19, 32]

2.3 Userspace Networking

A modern approach for fast networking, often used in current research projects [3, 21, 26, 44], is to leverage userspace networking. The idea behind it is to use a userspace driver for more direct interaction with the Network Interface Card (NIC) [11]. Furthermore, by running the driver in userspace, fewer system calls and thus fewer transitions to the kernel space need to be made. We will introduce RDMA and DPDK as two examples for userspace networking libraries and then discuss the benefits of using a Remote Procedure Call (RPC) abstraction for networking.

2.3.1 Remote Direct Memory Access

The term Remote Direct Memory Access refers to the Direct Memory Access technique made accessible remotely. With DMA [41], I/O operations are offloaded to a DMA controller which provides instructions for the execution of the operation to a device. The major benefit resulting from DMA is that devices, such as NICs, can directly access the main memory without the intervention of the CPU. This makes so-called one-sided approaches for networking possible. With one-sided RDMA, only one CPU poses a read or write request that is sent over the network and triggers the respective operation at the machine we are communicating with [41]. This happens without any involvement of the other machine's CPU. Because RDMA does not work with the traditional kernel network stack, different protocols are needed for this purpose. Accordingly, two protocols exist [34]. *(i)* The Internet Wide Area RDMA Protocol (iWARP) and *(ii)* the RDMA over Converged Ethernet Protocol (RoCE) [28]. While iWARP is used for compatibility with IP networks, RoCE builds on the ethernet link layer [34].

Because of its high potential, there is a lot of research focusing on extending RDMA, for example, by security [37, 42]. Further, compatibility for RDMA with PM comes with challenges that are currently investigated in research and several projects [17, 39, 40]. As PM is an emerging technology, current CPUs and hardware configurations are optimized for DRAM. Consequently, finding optimizations that make remote persistent memory access faster is a hot research topic as well [25, 47, 51].

2.3.2 DPDK

The Data Plane Development Kit (DPDK) [14] is a project initially started by Intel for accelerating network packet processing using userspace NIC drivers. Therefore it can be seen and used as an alternative for kernel networking or RDMA. As well as RDMA it is used for fast userspace networking in current research, especially in combination with TEEs [3, 43, 44].

2.3.3 Remote Procedure Calls

Remote Procedure Calls (RPCs) are an abstraction that can be used in networking in distributed systems. The idea is to make networking transparent so that a caller of a procedure does not notice whether it is executed on a different machine. Using this abstraction is easy for a developer who does not want to take care of the networking implementation and is only interested in its performance. For providing efficient RPCs, userspace networking approaches can be employed. One example of a library that can be used on top of both, DPDK and RDMA, is eRPC [26]. By supporting both technologies, eRPC brings the benefit of providing portability between different systems with different NICs. Furthermore, it shows that this gain of portability does not cause any major performance overheads.

3 Design

3.1 Design Goals

Threat Model. As our system is designed for deployment in an untrusted cloud environment, we need to be prepared for the case that the host Operating System is compromised. Therefore, our threat model is similar to thread models from existing work in this area [2, 3, 44]. We assume that an attacker can act as a superuser on the host Operating System. Consequently, an attacker can read and manipulate any memory region outside the ones protected by a TEE. Moreover, every other information that is exchanged with the host OS, for example, through system calls, can be modified by an attacker. For the networking threat model, we make the same assumptions, because the potentially compromised host OS has control over the Network Interface Card and could act as Man in the Middle. As well as an active attacker on the network, an attacker having control over the host OS can also eavesdrop or change any message that is sent or received.

Information Security Goals. For secure network communication, we must ensure at least Confidentiality and Integrity. Additionally, we want to ensure Freshness for Replay protection. Since a compromised host OS could, for example, deny all system calls or overwrite all buffers in unprotected memory regions, we cannot protect against Denial of Service attacks. Availability, therefore, cannot be guaranteed under these conditions. Furthermore, we do not protect against side-channel attacks on Intel SGX [5, 22].

Performance Goals. Because the network stack should operate inside data center networks, we strive for fast, high-throughput network communication. As we can assume that we have access to modern high-performance Network Interface Cards supporting throughputs of multiple Gbit/s, we need to fully leverage them. Especially for big payload sizes, it is important to optimize for throughput. Our goal is for our implementation to be more performant than current TCP socket implementations inside a Trusted Execution Environment. Consequently, we will compare the throughput of our network stack with the socket networking benchmark iperf [20] in the Evaluation section (§ 5.3.2). Low networking latency is also desirable to take advantage of the low-latency accesses of Persistent Memory (PM). Networking became a bottleneck since PM was introduced in data center networks [24], so we need to ensure that *(i)* the roundtrip time of a single operation is short and *(ii)* the operation throughput is high. It is especially important for our implementation to avoid system calls since these are expensive when running inside Trusted Execution Environments [2].

Generality. Our network stack should make arbitrary operations on any software possible remotely. A developer should ideally be free to design the layout of the data stored in PM. The final goal is to provide operations that make access on a remote persistent memory region as transparent as possible in the ways of performance and API.

3.2 Design Challenges

Untrusted Host Operating System. Problem. Our threat model assumes a potentially compromised host OS. Hence we need to assume that an attacker can interfere in every interaction with the Operating System, for instance, at every syscall. The problem here is that the system call itself is executed by the Operating System. Therefore we need to make sure that private data passed inside syscalls is always encrypted and authenticated. Another problem caused by syscalls is that transitions from the enclave to the host OS are very expensive [2].

Approach. To be able to securely operate on an untrusted host OS, we need to rely on TEEs implemented in processors. In particular, we use SCONE [2], a secure container environment working with Intel SGX [15]. SCONE provides us with a transparent way of using SGX. We simply need to run our application inside a SCONE container to run it inside an enclave for protecting it against attacks from the host OS. For protecting private data passed to the host OS in system calls, SCONE introduces shields. If, for example, a File System Shield is enabled, the application can request SCONE to transparently encrypt and/or authenticate Files. A networking shield exists as well. However, it is only made for protecting sockets. As we investigate in the Evaluation section, userspace networking is preferable over sockets, and for this case, we can not use SCONE's networking shield. To solve the aforementioned performance problem induced by syscalls, SCONE buffers syscalls. This avoids frequent enclave exiting and reentering. Even though this makes syscalls more efficient especially for multiple threads, they still induce a high overhead. For this reason, we need to keep the number of syscalls as low as possible. Therefore, as described later in this section, we use userspace networking.

Secure networking. Problem. For networking, we can not simply assume a Dolev-Yao Attacker Model [8] which assumes that an active attacker can only manipulate messages from the network. Since we assume that the host is potentially compromised, we also have to take measures against attacks from the host. However, if we find a way for protecting our whole system from attacks from the host OS, the Dolev-Yao Model is suiting. In general, a secure connection for sending and receiving messages confidentially needs to be established. Therefore, we need an initial handshake for establishing a key between two communicating parties.

Approach. For protecting our implementation against attacks from the host, we use SCONE [2], as mentioned before. As our code will then run inside an enclave, we only need to protect the communication against attacks from the network. However, note that even though our code is protected by a TEE, a compromised host Operating System can still act as an attacker. Because the NIC can not access enclave memory, we need to consider it a part of the untrusted host. To make data accessible for the NIC at all, we need to copy all messages from the enclave memory to the untrusted host memory. In this case, the measures against network attackers are also effective against attackers on the host. That is because they have the same capabilities as active attackers in the network, namely eavesdropping, modifying, and dropping messages. Because the NIC is not trusted, approaches like using the NIC for encryption and authentication [37] do not work for us. That is why we need to encrypt messages and add a tag for checking integrity inside the enclave. This is done before data is moved to untrusted host memory from where the data is accessible by the NIC. An easy-to-use class of encryption algorithms that provides both, confidentiality and integrity, is Authenticated Encryption with Associated Data (AEAD) [27].

Bootstrapping Trust. Problem. When using TEEs, an important question arises: How can we be sure that it is our unmodified code that is running and that it is running inside a TEE? All our effort securing our application would be in vain if we did not have an answer to that. A host would always tell us that our application is running securely inside a TEE, no matter if it is compromised or not. Hence, we need formal cryptographically secured proof for that. Furthermore, in our case, before the actual network communication for transferring data can start, a secure key needs to be established between two communicating parties. We also need to make sure that this key is only accessible from within the enclave. Lastly, to make sure our security guarantees are not violated, we also need to recall our Trusted Computing Base (TCB), which consists of the code and the hardware components we have to trust.

Approach. For bootstrapping trust remotely, we can use a remote attestation mechanism which is provided, for example, by the Intel SGX implementation [19]. The idea of remote attestation is to take measurements with the help of a Trusted Platform Module (TPM) [45] on the attestor and report them to a verifier over the network. In detail, on an attestation request from the verifier, the attestor requests a Quote from the TPM, which is a fresh cryptographic signature of measurements. The verifier can check the attestation reply, consisting of the TPM quote and the list of measurements. This requires, that he knows the public key of the keypair the TPM used to sign the Quote, in advance. [12] describes, how a secure channel can then be linked to the attestation process. This tunnel can then be used for exchanging initial information. The aforementioned list of measurements can consist of, for example, the running code and the libraries needed. In general, we should make sure that the whole code we want to trust is in a state that we expect. [32]

In the attestation process provided by Intel SGX, enclave code and data, technical details about the enclave, and its current attributes are among the measurements. Instead of a TPM, a dedicated Quoting Enclave is used for generating the Quote. The SGX attestation implementation can also establish a secure channel. We can use it for exchanging a key later used by our implementation. [7, 19]

In the end, let us recall the software and hardware components that we need to trust. Of course, we need to trust the processor and the implementation of the enclave, in hardware and software. Furthermore, evaluating technical details about the enclave provided by the attestation response is crucial. For example, Intel recommends not to trust enclaves in debug mode [19]. Lastly, we need to trust the TCB of SCONE [2] and the libraries we are using. As described later in chapter 4, these will include OpenSSL [30], eRPC [26] and the library for userspace networking on the link layer (i.e. either DPDK [14] or RDMA [33]).

Fast networking with Persistent Memory. Problem. As PM finds its way into data centers, new challenges regarding fast networking are coming up [24]. The key problem is that a network roundtrip takes more time than writing data to PM [24, 36], which was not the case with SSDs. Therefore, the low latency of PM is only fully leveraged if networking is optimized for latency. Kernel networking tends to have high latency for small package sizes as expensive syscalls need to be issued for reads and writes. Especially in TEEs, where syscalls are expensive, traditional kernel networking implementations, such as TCP or UDP, lead to high latency and throughput overheads.

Approach. The general approach to avoid syscalls is to use userspace networking libraries, such as DPDK [14] or RDMA [33]. Lower latencies are achieved by using dedicated drivers in userspace. These only provide low-level access leading to bad portability. When we have a look at two different approaches in the following sections, we use higher-level libraries to solve that problem. The first approach is to use one-sided RDMA. This technology makes it possible to access memory regions on a server without the involvement of its CPU, which means that the NIC directly accesses the PM. This can be implemented with librpma [17] from PMDK [16]. In section 3.3 we will conclude that this approach is only compatible with our security goals under special circumstances and induces a high overhead. Hence, we focus on a second, two-sided approach with a Remote Procedure Call (RPC) abstraction providing us with the possibility to use either two-sided RDMA [33] or DPDK [14].

3.3 Design with one-sided RDMA on Persistent Memory

Because PM supports Direct Memory Access (DMA) [36, 41], the Network Interface Card (NIC) can access it. The word "one-sided" already implies that only one side, i.e. one CPU, of two communicating machines is involved. Therefore, only one party needs to perform a networking operation, so at the other party, work is offloaded to the NIC. Furthermore, RDMA is a userspace networking technology, so it fulfills our requirement to reduce the number of system calls. In the following, we will illustrate a Design approach and analyze the problems induced by the use of one-sided verbs in general.

3.3.1 Environment and Assumptions

We assume a scenario in which there is a central server offering access to data in a PM region registered for one-sided RDMA. The server code and sensitive data reside inside a TEE, while the PM region registered for RDMA is untrusted. Multiple clients can connect to the server and access the registered region. We assume that trust between the server and each client was already established through attestation and that each client shares a separate symmetric key for encryption with the server. To access the data provided by the server by its address, each client needs to be informed about the structuring of the data. This information can be exchanged in an initial handshake.

3.3.2 Problems with secure one-sided RDMA

The main issue with secure one-sided RDMA is that any adversary could potentially access all data in the shared memory region. However, as our threat model assumes an attacker having access to all memory regions, including the one provided for RDMA, we need to have security measures against these attacks anyway. Provided that we manage to protect against simple tampering and modification attacks, there are two more types of attacks that we additionally need to be aware of: *(i)* replay attacks where data is re-inserted to a fixed position at a later time and *(ii)* copy attacks where data inside the memory region is copied to a different location where it doesn't belong to. For protecting against replay attacks, we need to make sure that the entry is valid at the current *time*, while against copy attacks, we need to be sure that an entry is at the *space* it belongs to.

Another question that arises when using RDMA on PM is: How are changes written by a remote client made persistent? In librpma [17], a library for remote persistent memory access on PM by PMDK [18], this is solved by executing a read for accessing the remote host's PM region which implicitly forces flushing. Other solutions [39, 40] suggest a dedicated asynchronous flush operation. Both approaches need an additional networking roundtrip, which we want to avoid.

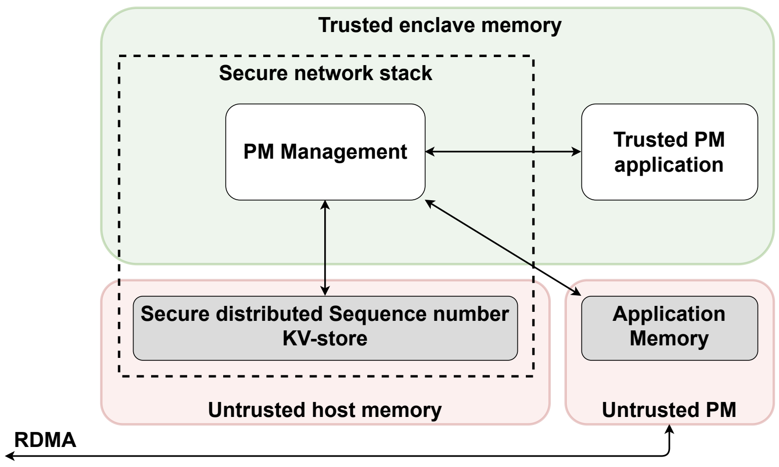

Figure 3.1: Design for one peer with one-sided networking. The PM Management is responsible for encryption and decryption. It also checks the sequence numbers. A local PM application can access the PM region via this Unit. Each peer needs this Management Unit and a copy of the distributed KV-store for sequence numbers in order to directly access the PM region

Initialization Vector	Sequence Number	Payload	Authentication Tag
12 B	8 B	Fixed size	16 B

Figure 3.2: Entry format for one-sided networking

3.3.3 Design Approach

The design of the one-sided networking approach is shown in figure 3.1. In this section, we describe, what the PM Management Unit's tasks would be with this kind of approach.

Protecting an entire PM region as a whole would be impractical because on every access all data would need to be sent, en-/decrypted, and integrity-checked at once. This is why the PM Management Unit must handle the data blockwise. For networking, it is beneficial to fit the block size to the network's Maximum Transmission Unit (MTU) to fill the packet size. Note, that this implies a big intervention for operations on the data as each operation needs to be aware of this structure and the data size. We can think of the storing of data as preparing whole encrypted and authenticated messages before they are sent. Therefore, an entry in the data structure would look like the one shown in Figure 3.2. We need a unique Initialization Vector (IV) for the encryption algorithm. The encrypted data can have an arbitrary but fixed size. As mentioned before, fitting the block size to the MTU is beneficial. We further need a Message Authentication Tag for providing integrity. As we are using an AEAD cipher [27], we can leverage the concept of Additional Authenticated Data (AAD) [9] to save memory. Therefore, the address or offset of a block, which has to be sent in plaintext for the NIC to know where it needs to access the data, can be included as AAD and thus authenticated. This ensures that, from the response, the client can check if the data was stored in the place where it was expected and accessed. Finally, we need a sequence number for providing freshness. However, there is a problem with that, which we illustrate in a short example. Therefore, let's assume we want to perform a read as a client. We issue an RDMA read call and get as a response a message according to the format in Figure 3.2. We successfully decrypt the message and verify the authentication tag, thus we know, the data was encrypted with the key only known by the server and the client. As we also include the address of the data during the decryption process as AAD, we can be sure that at some point in time the data we got was at the specified address. In the last step, we want to check the sequence number to make sure that the data is at the specified address *now*. But how do we know which sequence number is the correct one? We could have stored the sequence number in advance to know what sequence number we expect. But what if the data, and thus also the sequence number, has been changed once or multiple times by one-sided write operations? Even worse: What if the sequence numbers match, but the data was updated and now has a newer sequence number on the server? In this case, we would think the data was fresh, even though it is already outdated. This violates our freshness guarantee. The only solution for that problem is to notify all clients in case a data block is updated. This involves the server's CPU and causes one additional networking roundtrip per client per write

– something we want to avoid by all means. This approach is only beneficial with a very high read-to-write ratio, as in this case the communication overhead induced by two-sided write-operations only occurs rarely. Moreover, a low number of clients is beneficial because, in the case of a write, fewer sequence number updates need to be sent. Each client also needs a mapping from the address he wants to access to the sequence number he expects. As the EPC size is limited, we need a distributed key-value store in untrusted host memory or PM. Even though efficient solutions for this exist [3], it induces an overhead and adds a second networking channel, which we want to avoid for design reasons. However, we have seen that a one-sided approach can guarantee our security goals, even though only with a high effort.

3.3.4 Applicability

We want to investigate if there is a scenario where one-sided network communication is practical. Therefore, we consider the question: Does it make sense, in general, to access a remote persistent memory region as it was a local one? Consider, for example, that we have a tree data structure in the host's PM and clients accessing it. For a lookup, ideally $\mathcal{O}(\log n)$ accesses are needed. The problem is that, in general, these accesses are not serial, so each access needs a networking roundtrip. As discussed in section 3.1, we want to avoid networking roundtrips as they are far slower than PM accesses. So, in this case, it would be more efficient to let the server hosting the PM region access it for looking up an element in the tree data structure. An alternative solution would be to let the clients store internal information about the tree locally. However, if changes are applied to the data structure, all clients would need to be notified. They would have to update their information about the tree structure as well. In any case, this approach is a waste of resources. This simple example shows that the one-sided approach violates our aim for performance and generality.

To sum it up, the main problems about this approach are *(i)* that each client needs to keep track of sequence numbers and update them at each write inducing overheads and *(ii)* that we lose generality by having a fixed-size block structure in PM and we get significant overheads in case an operation includes multiple roundtrips. Therefore, we will not further investigate the one-sided approach, neither in the Implementation nor in the Evaluation section.

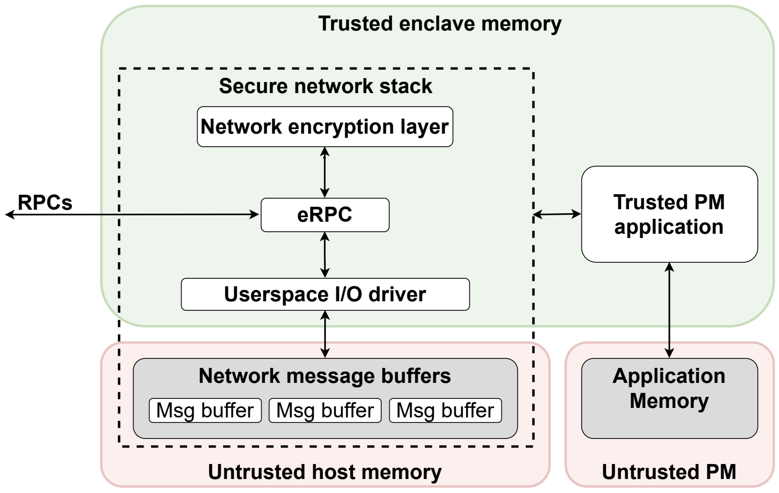

Figure 3.3: Design of a server for two-sided networking. Access to PM is made transparent by using RPCs. The network stack based on RPCs handles requests by fetching them from buffers in the untrusted host memory, decrypting them, and passing them to the PM application. After it has processed the request, a response is sent back over the network stack. It encrypts the response and passes it to untrusted host memory, from where it is sent to the client.

Initialization Vector	Sequence Number	Payload	Authentication Tag
12 B	8 B	Variably sized	16 B

Figure 3.4: Message format for two-sided communication

3.4 Design with two-sided networking

As we have investigated that a pure one-sided networking design is not fully compatible with our design goals, we make use of a two-sided approach. Luckily, there is a variety of different technologies working with this approach, such as kernel TCP, as well as DPDK [14] and a two-sided variant of RDMA. As we focus on userspace networking, we will only make use of the latter two technologies. In this section, we will illustrate our design approach and see that it meets our security and design requirements.

3.4.1 Problems with two-sided networking

The main disadvantage of two-sided networking compared to one-sided RDMA is that two CPUs need to be involved. Therefore, for each send-operation, there needs to be a corresponding receive-operation processing the message. In our case, we need to implement busy waiting to process requests effectively, which induces an overhead.

Another issue is that we need to communicate with the untrusted NIC from inside the enclave. As the NIC can not access buffers inside enclave memory, we need to write the encrypted messages which we want to send to buffers in untrusted host memory.

Further, the Application working on PM needs to be aware of the enclave and secure its PM accesses. As our two-sided variant of the network stack has no access to PM anymore, it does not need to deal with untrusted PM.

3.4.2 Design Approach

The basic design of our network stack at the server side is shown in figure 3.3. The whole networking code runs inside the enclave. We provide an encryption layer that encrypts all messages sent over the network with the network key exchanged in the attestation process. The encryption key is kept inside trusted enclave memory. It should differ from the key used to encrypt data inside PM. This ensures, that, in case a key between the server and a single client leaks, data kept in PM is still safe. Confidentiality and Integrity are provided by the AEAD encryption mechanism, so let us consider freshness. A client sends a request to a server with a unique sequence number. The server checks if the sequence number is matching. If it is, the server performs the requested operation and sends a response with the incremented sequence number back to the client. The client knows that the server's response matches its request if the sequence number is correct. Since networking sequence numbers are treated separately from the implementation of the application at the server, a user of our network stack can implement any method for guaranteeing freshness in his implementation. Hereby we have proved in an informal way that our two-sided implementation can guarantee our information security goals from section 3.1.

Below the encryption layer, we use eRPC [26], a library providing us with an RPC abstraction for accessing userspace drivers. This also ensures portability, as it can be used on top of DPDK as well as RDMA. We expose an asynchronous RPC interface to clients that can call functions registered on the server. A client sends a message, drains it and a callback is called on the arrival of the server response. This has the advantage, that multiple requests can be issued at once, improving average operation latency and operation throughput. We will also refer to this as request stacking. Function arguments need to be marshaled and unmarshaled, and return values need to be passed to our network stack in enclave memory. The memory regions with the arguments in plaintext need to reside in enclave memory as well. Then, we encrypt them and put them into message buffers residing in untrusted memory in the format shown in figure 3.4. Note, that this format is very similar to the data entry format we would have had with onesided RDMA, except for one difference: the payload size can be variable. As we had fixed-size blocks with the one-sided design, there could have been cases where resources would have been wasted for encrypting and sending such a block, while in this approach, there is no more overhead than necessary in any case. The message format in figure 3.4 results from our security goals. For keeping data confidential, we need encryption. Therefore, we need an IV and, because we use AEAD, we get a MAC for checking integrity. For guaranteeing freshness, we use unique sequence numbers. The lowest layer consists of a userspace driver communicating with the NIC and performing the networking operations. The driver itself runs inside the enclave, while it communicates with the NIC via untrusted host memory.

3.4.3 Applicability

As we provide RPCs, a user of the two-sided network stack can simply use it to transparently enqueue requests. To prove that this approach is indeed more beneficial than the one-sided one, we will consider the scenario described in section 3.3.4.

We consider the example of the tree data structure hosted at the server. Any operation on the data structure is performed by the server itself. A client only needs to enqueue a request that is then handled by the server who performs all accesses on the PM region. Only the response, which is everything the client needs, is sent back. This saves networking roundtrips compared to the one-sided approach.

To have a fair comparison, we also need to consider the case of simple reads and writes on PM. For a direct read, in case the client knows where to read from, a one-sided client would need to look up the sequence number for the corresponding block. After that, only one network roundtrip is required. The server's CPU is never involved. For the two-sided approach, a client would need to enqueue a request specifying the PM address. This causes the server to copy the contents of the requested PM region in a buffer and send the data. The overhead is only caused by the server's busy waiting for requests, the copying of the data in PM to the message buffer, and the freshness check at the server. Additionally, the data is decrypted and encrypted once more if different keys are used for networking than for storing. However, this is more secure than using a shared storage key for networking, as it needs to be the case for one-sided approaches. For writes with one-sided RDMA, the sequence number KV-stores of all clients need to be updated. Therefore, one message per client needs to be sent, causing a higher overhead than the two-sided variant.

To sum it up, in almost any case the two-sided networking variant causes lower overheads in practice. It is not only less complicated to implement, but also less complicated to use. Moreover, it provides users the flexibility to manage data in untrusted PM according to their needs and makes synchronization easier. By using different keys for storage and network, the two-sided variant is also more secure than a one-sided one can be. As using one-sided RDMA has no advantages under the given circumstances, we will only focus on and implement a two-sided network stack supporting both, RDMA and DPDK.

4 Implementation

In this chapter, we describe the implementation of a secure two-sided network stack that is aware of Trusted Execution Environments and leverages userspace networking. Therefore, we have a look at the tools used (§4.1), then we will look at some details of the implementation (§4.2, 4.3, 4.4) and provide short guides on the usage of the network stack.

4.1 Used tools

OpenSSL. For providing the aforementioned security guarantees, OpenSSL [30] is used as a library for cryptographic operations. As an encryption mechanism, the OpenSSL implementation of AES-128-GCM [9] is used. The cryptographically secure pseudo-random number generator provided by OpenSSL [31] serves for generating random initial sequence numbers and Initialization Vectors.

eRPC. A library that provides fast network communication with support for RDMA is eRPC [23, 26]. It provides an API for issuing Remote Procedure calls and can be used, for example, on top of DPDK and RDMA. The compatibility of eRPC with both RDMA and DPDK offers the chance to directly compare the performance of the network stack with RDMA against DPDK. eRPC implements reliable UDP-based communication and is designed for lossy networks. Therefore, requests are guaranteed to arrive at a server that can reply to incoming requests. Arguments for an RPC are transferred via DMA-capable Message Buffers residing in untrusted host memory. In these buffers, we place messages of the format described in section 4.2, which are encrypted and integrity-protected. [26]

We are using eRPC along with SCONE [2] and DPDK [14] or RDMA. As DPDK and eRPC are not trivially compatible with SCONE, we make use of adapted DPDK and eRPC versions from Avocado [3].

4.2 Message format

A message from the network stack includes four important fields. First of all, a 12-Byte Initialization Vector is used for secure encryption with AES128-GCM [9]. For guaranteeing freshness, an 8-Byte Sequence number is used. The next field is the encrypted message. Its size depends on the message size that a user wants to transmit. By using the Galois-Counter-Mode, which is an AES stream cipher mode [9], the ciphertext size is the same as the plaintext size. Lastly, the 16-Byte GMAC-Tag generated throughout the encryption process is appended. On decryption, the Tag is checked to make sure that the ciphertext remained unchanged during the transmission process.

4.3 Client Implementation

A user of the network stack can register types of requests and respective callbacks that are called when the response arrives. If a request is created, all arguments for the according RPC are added one by one, and then encrypted and authenticated as described in section 4.2. An alternative to this approach would have been to copy all arguments to a buffer and encrypt it. But since that causes copy overhead, we directly encrypt it and store it in a message buffer.

We also provide the option to move the data without encryption. This can be beneficial to avoid double encryption and decryption, in case the payload is already encrypted and authenticated. Note, that a shared key for data storage and network can lead to severe security vulnerabilities in case it leaks.

The message buffer with the encrypted payload resides in the untrusted host's hugepages memory. The buffers are provided by eRPC [26] and are DMA-capable. When finishing a request, the message can be drained and thus sent to the server.

We provide an asynchronous RPC interface, so a call to an issued request does not block until the response arrives. Instead, the client decides when he wants to process the response of the server. However, as eRPC does not provide a guarantee that a callback is called, we implement a queue where we store the requests for which a response is missing. This has the advantage, that we can guarantee that the response handler is called exactly once, even if no response arrives. Therefore, if, for example, memory needs to be deallocated on the arrival of the response, there are no memory leaks. Furthermore, we reduce memory usage and allocation overhead, if the size of the queue is constant since each allocated message buffer is bound to a fixed queue position. Lastly, we store the sequence number of the request in the queue, helping us to check if the response sequence number matches the one from the request.

Concluding, the workflow for using our network stack at a client is as follows:

1. Establish a connection.
2. Enqueue requests and specify respective response handlers.
3. Drain requests.
4. The response handler is called as soon as a response arrives and the user checks if responses arrived

4.4 Server Implementation

When a server is set up, request types with according handlers need to be registered. Then, a user-defined number of threads are spawned, each for one client. For benchmarks, this guarantees us a high performance. However, in real use cases, having a server thread handling more than one client might be more beneficial. When the threads are spawned, the server can then perform busy waiting and call the previously registered handlers on incoming requests by clients.

After the request of a client is decrypted and integrity-checked it is placed into a buffer in enclave memory. Next, the sequence number is checked. We start at a previously exchanged sequence number and increment it with every message sent. However, as eRPC works in lossy networks, messages may be dropped and we can not keep a strict message ordering for the sake of performance. Therefore, we accept all sequence numbers in a fixed range, based on the previously received sequence numbers. To avoid replays, we additionally need to check that in this range, every sequence number is unique. This way of sequence number handling implies a tradeoff between performance and security. Accepting 2^n sequence numbers is equivalent to using a sequence number that has n fewer bits.

After the sequence number has been checked, the registered handler is called with the respective arguments from the client. Inside this handler, a response for the client needs to be enqueued. This response is then encrypted and then needs to be drained so that it is sent.

To sum it up, the workflow for running a server is the following:

1. Register possible request types and handlers.
2. Perform busy waiting for connections and incoming requests.
3. As soon as a request arrived, the respective registered handler is called.
4. Enqueue and drain a response matching the request

5 Evaluation

In the evaluation, we want to find out, whether our implementation of the network stack lives up to our design goals from section 3.1.

First, we will test if the network stack's operations have low latency, especially for small payloads, and how latency is influenced by SCONE (§ 5.2). Further, we investigate how our approach for request stacking (see section 3.4.2) improves the average operation latency.

Second, we check the networking throughput performance in two experiments. At first, we will investigate the performance with different numbers of threads (§ 5.3.1). Finally, we compare our implementation with the network benchmarking tool iperf [20] to verify that our userspace networking approach is beneficial when working with TEEs (§ 5.3.2).

Lastly, we want to test the performance of our implementation under real workloads in an environment for which we designed our network stack. Therefore, we implement an encryption layer on top of the C++ standard library map which contains values in untrusted host memory. We test its performance with and without networking (§ 5.4). Furthermore, to prove that our implementation provides portability, we run this benchmark in two setups. One with DPDK (§ 5.4.1) and one with RDMA (§ 5.4.2).

5.1 Testbed

We evaluate the performance of our network stack using two setups. We perform all benchmarks (§ 5.2 - 5.4) on a pair of 2 SGX-capable servers with a NIC supported by DPDK, that both have the following specifications: CPU: Intel(R) Core(TM) i9-9900K with 8 physical, 16 logical cores. Memory: 64GiB. Caches: L1: 256 KiB instructions + 256 KiB data, L2: 2 MiB, L3: 16 MiB. NIC: Intel(R) XL710 40GbE.

The second setup is only used for the application benchmark with RDMA (§ 5.4.2). The two servers both have RDMA NICs, but they do not support SGX which is why we can not run any tests with SCONE on them. These servers have the following specifications: CPU: Intel(R) Core(TM) i9-10900K with 10 physical, 20 logical cores. Memory: 128 GiB. Caches: L1: 320 KiB instructions + 320 KiB data, L2: 2.5 MiB, L3: 20 MiB. NIC: Mellanox Technologies(R) MT27710 Family [ConnectX-4 Lx].

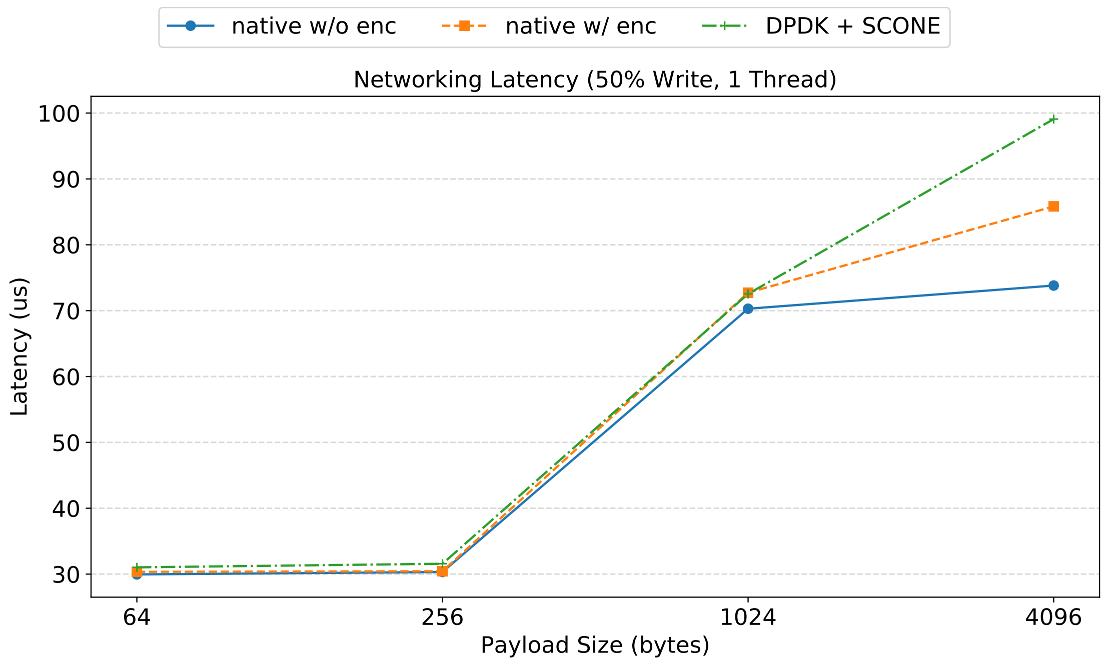

Figure 5.1: Single-Operation latency of our network stack with DPDK

5.2 Latency

We have two latency benchmarks. At first, we benchmark the latency of single operations. Subsequently, we observe, how the performance improves as soon as we send multiple requests at once with multiple threads.

5.2.1 Single Operation Latency

Setup. We measure latency by enqueueing single operations and waiting for the server response before enqueueing new requests. This way, we measure the latency of a single operation in case synchronous operations are required. As the network is not saturated in this scenario, there is hardly any congestion and it does not make a notable difference, with how many threads we are sending requests. That is why, for simplicity, we use one thread. We perform these operations with payload sizes from 64 to 4096 Bytes. Here, an operation is a transfer of a request and a response each of the same size. We compare native (i.e. unencrypted), encrypted networking, and encrypted networking inside SCONE. Note that in this scenario only the server runs inside SCONE. This is due to the fact that SCONE issues a system call for each time measurement. Since each operation latency time is measured separately at the client, for a SCONE client there would be a too high system call overhead to deliver sensible results.

Result. From the results in figure 5.1, we see that the encryption hardly causes any overheads, especially for low payload sizes. For higher sizes, the TEE, as well as the encryption, cause overheads. This is not surprising because encryption causes overhead proportional to the payload size. We notice that the latency abruptly increases from 256 to 1024 Bytes. This is because multiple packets need to be sent for one request with higher payload sizes, so the latency increases accordingly. Compared to the latency of PM, which is at about 100-1000 ns [49], the networking latency is causing a considerable overhead. However, consider that by enqueueing multiple requests at once, the average latency of a single operation is again reduced. We will show the performance benefit of it in subsection 5.2.2. Concluding, we can say that our network stack reaches our goals for low single-operation latency. Especially our userspace networking approach leads to a low latency overhead (at most 15%) inside TEEs compared to encrypted networking.

5.2.2 Latency with Request Stacking

Setup. For showing that asynchronous RPCs lead to low average latency and high operation throughput, we send payloads of sizes from 64 to 4096 Bytes. Again, each operation consists of a request and a response, each of the respective payload size. However, in this benchmark we don't wait for the response to arrive, instead, we send as many requests as fit inside the client request queue mentioned in sections 4.3 and 4.4 at once. In our case, these are 1024 requests. We use 4 client threads *(i)* to provide a realistic use case with multiple clients & high network load and *(ii)* to have data comparable with the throughput tests in section 5.3.

Result. From figure 5.2 we can derive that the average latency has indeed decreased compared to the latency for synchronous operations before. We even reach average latencies of under 1 ns for small value sizes if the client is running natively. Of course, we need to multiply the latency by the number of threads, to have a per-thread average latency for comparison with PM. We get no more than 4 µs, which means that we reached our goal of having comparable latency to PM. Furthermore, figure 5.2 shows that the latency with SCONE is at most $4\times$ higher than natively, which is still acceptable. If the client is running natively, the overhead is not more than $2\times$ as high. Another observation is, that if server and client both run in SCONE, for high payload sizes the performance is better than if only the server runs in SCONE. This behavior is subject to further investigation.

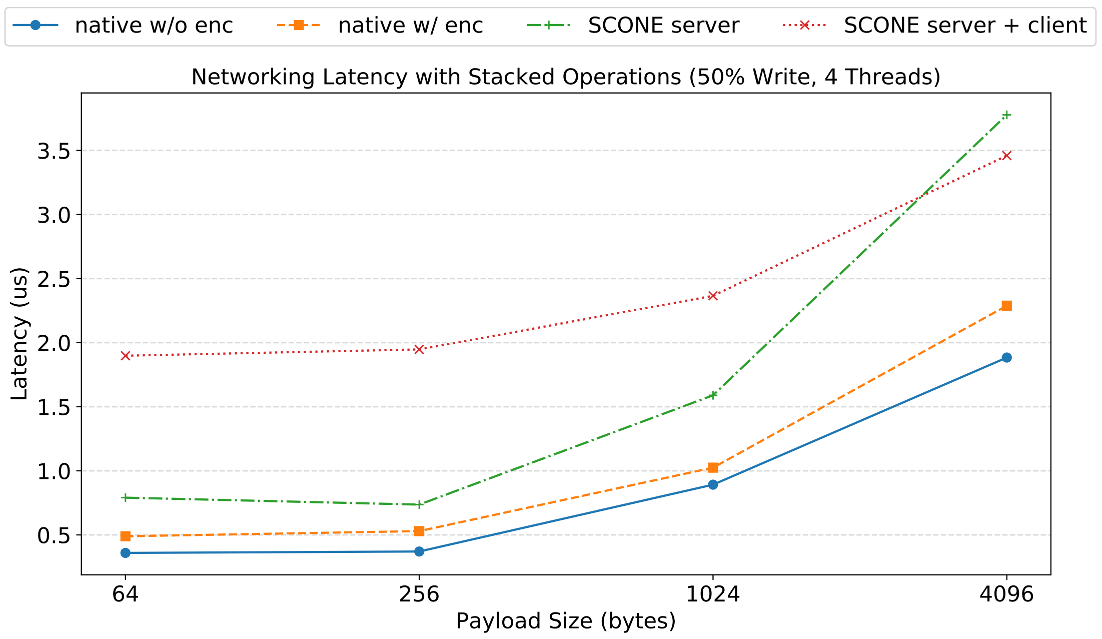

Figure 5.2: Latency with stacked requests

These two experiments show us that latency can be effectively reduced by using asynchronous RPCs.

5.3 Networking Throughput

The throughput of our implementation is tested in two benchmarks. At first, we look at the influence of multiple threads on networking. In the second benchmark, we simulate the behavior of iperf in our implementation for comparison with kernel networking approaches.

5.3.1 Varying Number of Threads

Setup. We want to consider the impact of using multiple threads on the networking throughput. Therefore, we keep the payload size fixed to 2048 and perform operations with a read/write ratio of 50%. This means, that we send and receive messages of equal size. As a result, we use the sum of the downlink and uplink throughput. The high payload size permits us to saturate the network with a sufficiently high number of threads. Again, we perform the tests without encryption, with encryption, and with encryption & a server running inside a TEE.

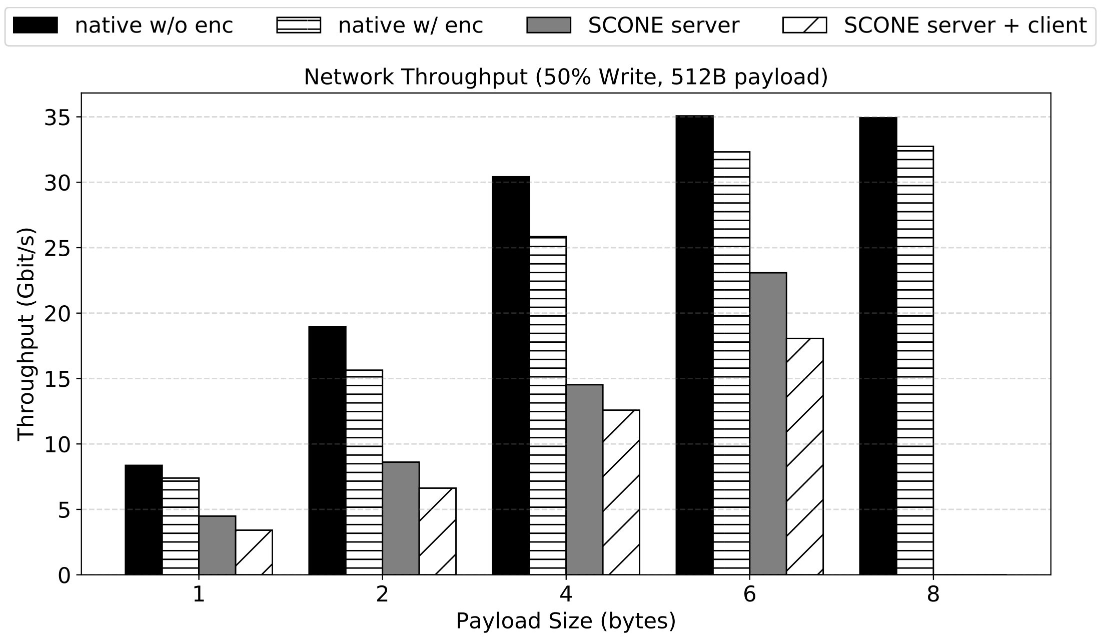

Figure 5.3: Throughput with varying number of threads

Results. From figure 5.3 we can see that, as expected, a higher number of threads leads to a higher throughput. We observe, that with a value size of 2048B and 6 threads the maximum throughput of our implementation can be reached in our testbed. The overhead caused by SCONE gets higher the more threads we use. This is also not surprising as there are more system calls and expensive operations inside the enclave if the workload is more intensive. Note that our implementation currently doesn't support 8 threads. SCONE only supports 8 threads and we observed that eRPC needs a background thread, so we can at most run 7 threads.

5.3.2 Bandwidth Throughput with SCONE and iperf

Setup. To make sure that our userspace networking approach is suited better than an approach that uses kernel TCP, we compare our network stack with iperf [20]. We need to simulate the behavior of the iperf benchmark. Therefore, we send requests of a certain size from the client to the server and at the server, we count how many of these arrive in a certain timespan. Note, that we still send responses without payload from the server to the client. These can be seen as equivalent to TCP ACK messages. The benchmark is run with different payload sizes and 4 threads. We use 4 processes for iperf to be able to use 4 ports in parallel for having a fair comparison.

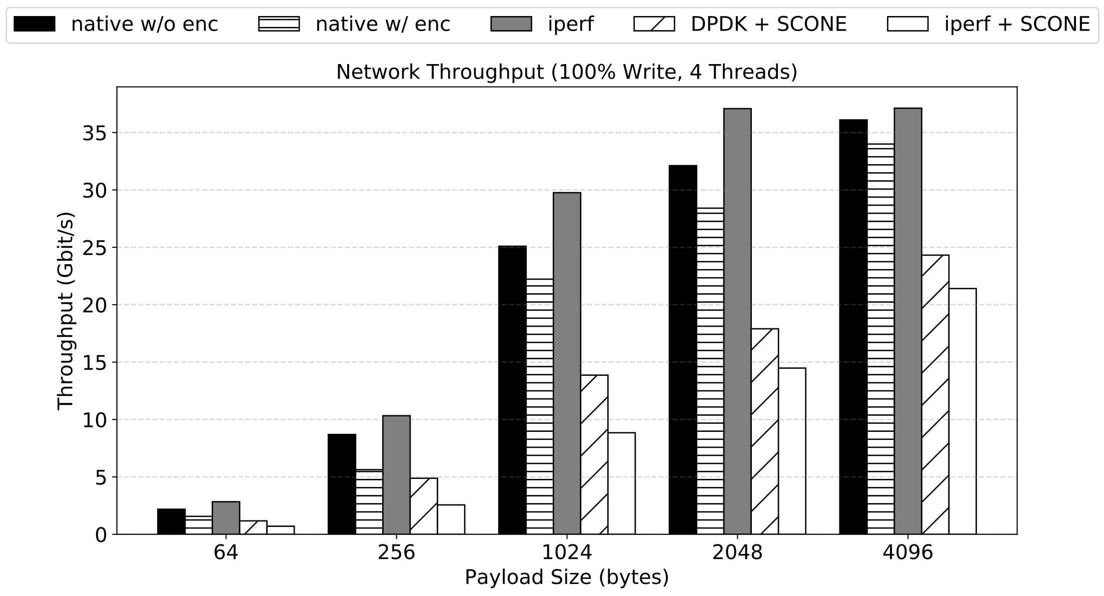

Figure 5.4: Bandwidth Throughput of our Network Stack compared with iperf

Results. From figure 5.4 we notice that iperf outperforms our implementation when not running inside the enclave. The throughput of iperf is at most 30% higher than the one of our network stack. However, our implementation was designed for running inside an enclave. In SCONE, the throughput of our implementation is higher than iperf's throughput for each payload size. This also confirms existing observations [3]. To sum it up, our choice for userspace networking is supported by this benchmark, as it shows that a kernel networking approach is less performant than our userspace network stack.

5.4 Operation Throughput with KV-store

5.4.1 DPDK Networking

Setup. For testing whether our implementation is suited for use in a real-world example, we use it with the C++ standard library's `std::map` implementation. On top of the `std::map`, we implement an encryption layer that encrypts values that are stored in untrusted host memory. Furthermore, we synchronize access to the map for being able to test with multiple threads. We simulate the higher latency of PM writes by adding overhead to the write operation. This implementation is tested natively without networking at first to have a baseline to compare with. Then, we test it with

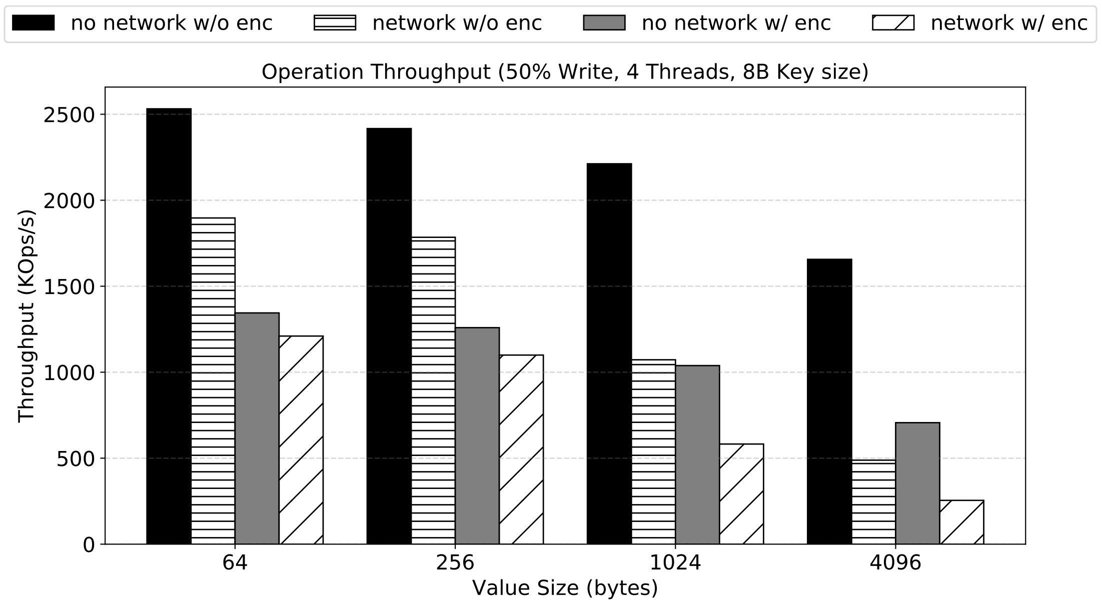

Figure 5.5: Operation Throughput with underlying KV-store with DPDK

networking. These tests are performed inside SCONE as well. We use value sizes from 64 to 4096 Bytes and 4 threads. This setup is tested with YCSB workloads [6] provided by PMDK [16]. The ratio of put operations to get operations amounts to 50%.

Results. From figure 5.5, we see that in the native case our implementation causes a low overhead, especially for small value sizes. The reason for the significantly lower throughput with bigger values is that, as already mentioned in section 5.2, multiple packets need to be sent for requests and responses. As the operation throughput equals the reciprocal of the average operation latency, its performance degrades the same way as the average latency increases. In this benchmark, the average latency per operation is between 0.5 µs and 2 µs in the native case.

Figure 5.6 shows the overhead caused by SCONE. Compared to native encrypted networking, the case in which only the server runs inside SCONE always causes less than 50% overhead. A remarkable characteristic of the case with a SCONE server and client is that the performance overhead does not decrease significantly with higher value sizes. The difference of the throughput with 64 Bytes value size to 4096 Bytes size is only 40 %. This can be explained by the fact that the networking latency is low in comparison to the time needed for the map operations. However, note that for value sizes of 4096 Bytes, the SCONE setup performs better than the setup without SCONE. We already observed this behavior in section 5.2. This issue needs to be further

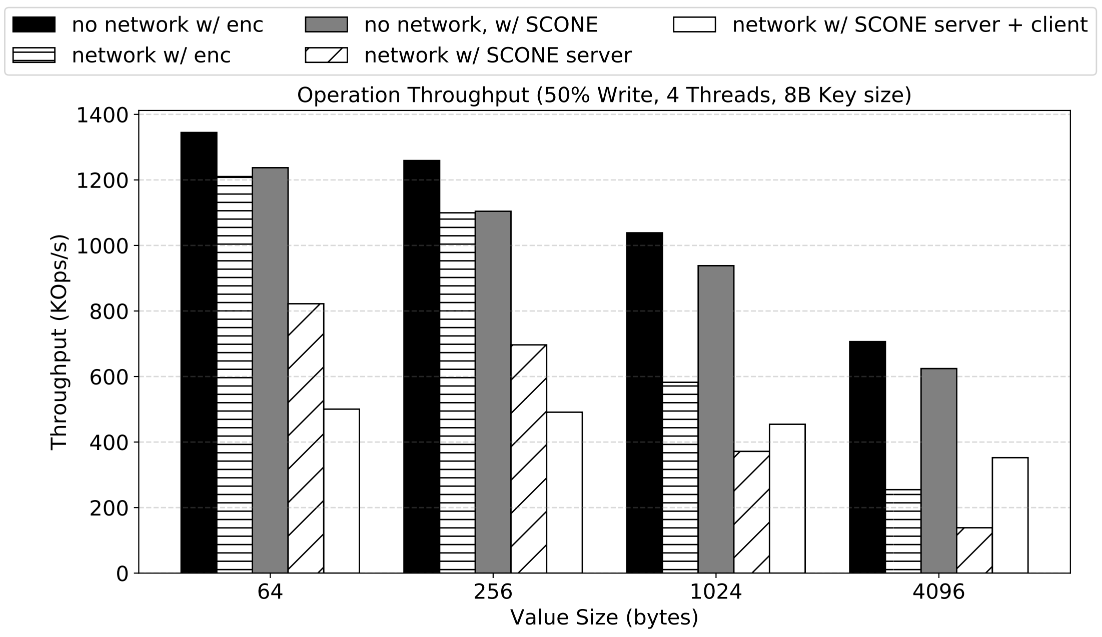

Figure 5.6: Operation Throughput with underlying KV-store with DPDK. Results with with SCONE

investigated.

All in all, the performance overheads caused by SCONE and our networking are low enough for us to claim that our implementation lives up to our performance goals.

5.4.2 RDMA Networking

Setup. To show that our implementation is portable, we perform the same benchmark as in the previous subsection with RDMA. Because we need to use a different setup that does not support SGX, we can not use a TEE here.

Results. The results shown in figure 5.7 are similar to the ones from the previous benchmark shown in figure 5.5. The relative performance loss for bigger value sizes is higher than with DPDK. This is most probably caused by the fact that the RDMA NIC only supports bandwidths of up to 10 Gbit/s, while the DPDK NIC reaches 40 Gbit/s.

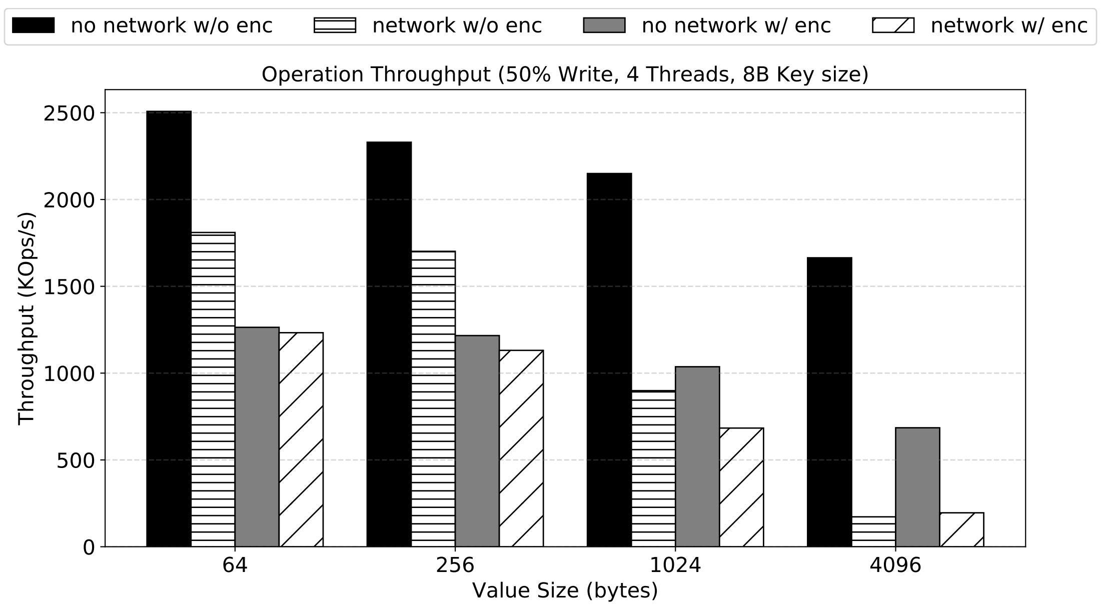

Figure 5.7: Operation Throughput with underlying KV-store with RDMA

6 Related Work

Securing RDMA. As secure RDMA is often required in datacenter networks, there was the need for work analyzing security issues of RDMA implementations [35, 37]. Also, a lot of papers that propose efficient and effective solutions for securing RDMA exist, such as sRDMA, which uses the NIC for fast encryption and authentication mechanisms without overheads caused for the CPU [42]. Note, that this approach can not be applied by our network stack, as we need to consider the NIC part of the untrusted host. Projects such as Avocado [3] consider the same threat model as we do and follow the same userspace networking approach. However, the Avocado network stack is specialized in key-value-store operations and does not use PM.

Userspace drivers in TEEs. There is plenty of work leveraging userspace drivers in TEEs for gaining performance by avoiding system calls [3, 4, 43, 44]. Each of these projects shares the same threat model of a potentially compromised host with our network stack. However, our network stack also considers the use of PM.

RDMA on PM A lot of work on extending RDMA for compatibility with PM exists, for example implementing a flush operation [39, 40]. To leverage the advantages of PM, new programming models have been introduced for RDMA on PM [13, 17].

One existing problem with RDMA on PM is that current RDMA implementations are optimized for DRAM. Recent work finds that caching optimizations for DRAM make RDMA on PM slower [25, 47]. [25] additionally suggests using Remote Procedure Calls for increasing write bandwidth which is something we considered in our network stack design. FileMR [51] eliminates overheads arising from incompatibilities of RDMA and Persistent Memory. Furthermore, it introduces a new address translation mechanism for improving the NIC's translation cache hit rate. However, many improvements mainly apply to one-sided RDMA which, as we found, is not suited for protecting against attacks in our threat model.

7 Future Work

Extending eRPC. During our work, we found that implementing a secure encryption layer on top of eRPC [26] introduces overheads at a few points because of duplicates. One problem is, for example, the request ID that should be kept confidential. In our implementation, we use a unique eRPC request ID for eRPC and add the encrypted request ID in an additional field. We could modify the eRPC message format so that only one ID is sent. This is only one example of a few possible optimizations that could be introduced when implementing a secure layer *inside* eRPC instead on top of it.

One-sided networking. We showed that one-sided networking in theory can not ensure our security guarantees while being performant, in general. However, future work could investigate special scenarios in which one-sided networking would be beneficial for our attacker model.

Benchmarks with PM- and TEE-aware systems. As soon as there are published implementations aware of systems using PM working in TEEs, our network stack can be used with these. Then, more realistic benchmarks can be performed to get more realistic data about the performance of our network stack and to find potential optimizations. Also, even though we tried to simulate PM by introducing artificial delays, benchmarks on real hardware PM would be beneficial to see whether our ideas work out in practice.

Different Threading Approach. Our approach is to have exactly one server thread per client thread. This is ideal for benchmarking, but may be inefficient in deployment, as a server thread is in a spinlock waiting for requests. Therefore, a new approach could allow handling multiple clients on one server thread. The actual overhead of this method would be an interesting question that could be answered with benchmarks on a respective implementation.

8 Conclusion

This thesis discusses two approaches for fast and secure networking in untrusted data center environments. We also consider the influence of the use of PM on networking, performance, and security aspects. In particular, we discuss solutions for three major problems: *(i)* The incompatibility of TEEs with PM and DMA, *(ii)* the performance overhead induced by TEEs and *(iii)* the low access latency of PM compared to networking roundtrip times. After discussing further challenges and our solutions, we introduce two possible designs for a network stack: *(i)* a one-sided variant without interaction of a host's CPU and *(ii)* a two-sided variant with userspace networking. We conclude that a one-sided design has too many disadvantages and causes too high overheads. Therefore, we only implement a two-sided network stack and provide an RPC interface. Further, we perform benchmarks with our implementation and conclude that our design for a two-sided networking approach performs well under the given conditions. In particular, we learn that our userspace networking approach is up to 60% faster than kernel TCP and that inside TEEs, the latency is at most $4\times$ higher. Furthermore, we see that our network stack can be used efficiently on top of a map implementation.

List of Figures

List of Tables

Bibliography

[1] *AMD Secure Encrypted Virtualization (SEV)*. AMD Incorporated. URL: `https://developer.amd.com/sev/` (visited on 07/24/2021).

[2] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. "SCONE: Secure Linux Containers with Intel SGX." In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 689–703. ISBN: 978-1-931971-33-1.

[3] M. Bailleu, D. Giantsidi, V. Gavrielatos, D. L. Quoc, V. Nagarajan, and P. Bhatotia. "Avocado: A Secure In-Memory Distributed Storage System." In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 65–79. ISBN: 978-1-939133-23-6.

[4] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. "SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution." In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 173–190. ISBN: 978-1-939133-09-0.

[5] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. "Software Grand Exposure: SGX Cache Attacks Are Practical." In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, Aug. 2017.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking Cloud Serving Systems with YCSB." In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. ISBN: 9781450300360. DOI: `10.1145/1807128.1807152`.

[7] V. Costan and S. Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086. 2016.

[8] D. Dolev and A. Yao. "On the security of public key protocols." In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208. DOI: `10.1109/TIT.1983.1056650`.

[9] M. Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. National Institute of Standards and Technology, Nov. 2007.

[10] D. Ehnes. "The Magic of Intel's SGX - A Tutorial on Programming a Secure Enclave." In: *Medium* (Nov. 21, 2018).

[11] P. Emmerich, M. Pudelko, S. Bauer, S. Huber, T. Zwickl, and G. Carle. "User Space Network Drivers." In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)*. Sept. 2019.

[12] K. Goldman, R. Perez, and R. Sailer. "Linking Remote Attestation to Secure Tunnel Endpoints." In: *Proceedings of the First ACM Workshop on Scalable Trusted Computing*. STC '06. Alexandria, Virginia, USA: Association for Computing Machinery, 2006, pp. 21–24. ISBN: 1595935487. DOI: `10.1145/1179474.1179481`.

[13] M. Honda, G. Lettieri, L. Eggert, and D. Santry. "PASTE: A Network Programming Interface for Non-Volatile Main Memory." In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 17–33. ISBN: 978-1-939133-01-4.

[14] Intel Corporation. *DPDK - Data Plane Development Kit*. DPDK Project. URL: `https://www.dpdk.org/` (visited on 07/30/2021).

[15] Intel Corporation. *Intel(R) Software Guard Extensions (Intel(R) SGX)*. Intel Corporation. URL: `https://www.intel.co.uk/content/www/uk/en/architecture-and-technology/software-guard-extensions.html` (visited on 07/24/2021).

[16] Intel Corporation. *Persistent Memory Development Kit*. URL: `https://pmem.io/pmdk/` (visited on 07/31/2021).

[17] Intel Corporation. *PMDK - Remote Persistent Memory Access*. URL: `https://pmem.io/rpma/` (visited on 07/30/2021).

[18] Intel Corporation. *pmem.io - Persistent Memory Programming*. URL: `https://pmem.io/` (visited on 07/30/2021).

[19] Intel Corporation. *Remote (Inter-Platform) Attestation*. Intel Corporation. Apr. 9, 2016. URL: `https://software.intel.com/content/www/us/en/develop/documentation/sgx-developer-guide/top/attestation/remote-interplatform-attestation.html` (visited on 07/30/2021).

[20] *iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. URL: `https://iperf.fr/` (visited on 07/24/2021).

[21] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems." In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 489–502. ISBN: 978-1-931971-09-6.

[22] S. P. Johnson. *Intel(R) SGX and Side-Channels*. Intel Corporation. Feb. 27, 2018. URL: `https://software.intel.com/content/www/us/en/develop/articles/intel-sgx-and-side-channels.html` (visited on 07/04/2021).

[23] A. Kalia. *eRPC: Efficient RPCs for datacenter networks*. URL: `https://github.com/erpc-io/eRPC` (visited on 06/28/2021).

[24] A. Kalia, D. Andersen, and M. Kaminsky. "Challenges and Solutions for Fast Remote Persistent Memory Access." In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 105–119. ISBN: 9781450381376. DOI: `10.1145/3419111.3421294`.

[25] A. Kalia, D. Andersen, and M. Kaminsky. "Challenges and Solutions for Fast Remote Persistent Memory Access." In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 105–119. ISBN: 9781450381376. DOI: `10.1145/3419111.3421294`.

[26] A. Kalia, M. Kaminsky, and D. Andersen. "Datacenter RPCs can be General and Fast." In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 1–16. ISBN: 978-1-931971-49-2.

[27] D. McGrew. *An Interface and Algorithms for Authenticated Encryption*. RFC 5116. Jan. 2008. DOI: `10.17487/RFC5116`.

[28] NVDIA Corporation. *RDMA over Converged Ethernet (RoCE)*. Features Overview and Configuration – Ethernet Network. NVDIA Corporation.

[29] *NVM Programming Model (NPM) Version 1.2*. Storage Networking Industry Association, June 19, 2017.

[30] OpenSSL Software Foundation. *OpenSSL - Cryptography and SSL/TLS Toolkit*. OpenSSL Software Foundation. URL: `https://www.openssl.org/` (visited on 06/28/2021).

[31] OpenSSL Software Foundation. *RAND - the OpenSSL random generator*. OpenSSL Software Foundation.

[32] B. Parno, J. M. McCune, and A. Perrig. “Bootstrapping Trust in Commodity Computers.” In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 414–429. DOI: 10.1109/SP.2010.32.

[33] *RDMA Core Userspace Libraries and Daemons*. Linux RDMA. URL: https://github.com/linux-rdma/rdma-core (visited on 07/30/2021).

[34] Red Hat Incorporated. *Configure Infiniband and RDMA Networks*. Red Hat Enterprise Linux – Networking Guide. Red Hat Incorporated.

[35] B. Rothenberger, K. Taranov, A. Perrig, and T. Hoefler. “ReDMArk: Bypassing RDMA Security Mechanisms.” In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 4277–4292. ISBN: 978-1-939133-24-3.

[36] A. Rudoff. “Persistent Memory Programming.” In: *;login:* (Summer 2017, Vol. 42, No. 2), pp. 34–40.

[37] A. K. Simpson, A. Szekeres, J. Nelson, and I. Zhang. “Securing RDMA for High-Performance Datacenter Storage Systems.” In: *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, July 2020.

[38] V. Sumina. “26 Cloud Computing Statistics, Facts & Trends for 2021.” In: (July 23, 2021).

[39] T. Talpey. *Remote Persistent Memory Extensions*. Mar. 20, 2019.

[40] T. Talpey and A. Bumgarner. *Nonvolatile Memory Programming TWG – Remote Persistent Memory*.

[41] A. Tanenbaum and H. Bos. “5. Input/Output.” In: *Modern Operating Systems*. Vol. 4. Pearson Education, Limited, Sept. 18, 2014, pp. 337–434.

[42] K. Taranov, B. Rothenberger, A. Perrig, and T. Hoefler. “sRDMA – Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access.” In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 691–704. ISBN: 978-1-939133-14-4.

[43] J. Thalheim, H. Unnibhavi, C. Priebe, P. Bhatotia, and P. Pietzuch. “Rkt-Io: A Direct I/O Stack for Shielded Execution.” In: *Proceedings of the Sixteenth European Conference on Computer Systems*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 490–506. ISBN: 9781450383349.

[44] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. “ShieldBox: Secure Middleboxes Using Shielded Execution.” In: *Proceedings of the Symposium on SDN Research*. SOSR ’18. Los Angeles, CA, USA: Association for Computing Machinery, 2018. ISBN: 9781450356640. DOI: 10.1145/3185467.3185469.

[45] *Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.59*. Trusted Computing Group. Nov. 2019.

[46] *TrustZone*. ARM Limited. URL: `https://developer.arm.com/ip-products/security-ip/trustzone` (visited on 07/24/2021).

[47] X. Wei, X. Xie, R. Chen, H. Chen, and B. Zang. "Characterizing and Optimizing Remote Persistent Memory with RDMA and NVM." In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 523–536. ISBN: 978-1-939133-23-6.

[48] K. Weise. "Amazon's quarterly profit jumps to $7.8 billion." In: *Ney York Times* (July 29, 2021).

[49] *What is Persistent Memory?* URL: `https://www.snia.org/education/what-is-persistent-memory` (visited on 07/30/2021).

[50] J. C. Wong. "Microsoft revenue exceeds $100bn boosted by cloud services." In: *The Guardian* (July 20, 2018).

[51] J. Yang, J. Izraelevitz, and S. Swanson. "FileMR: Rethinking RDMA Networking for Scalable Persistent Memory." In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 111–125. ISBN: 978-1-939133-13-7.