TYCHE: Shared memory communication

Maëlys Billon
maelys.billon@epfl.ch
École Polytechnique Fédérale de Lausanne

ABSTRACT

This report presents a lockless communication mechanism implemented in Rust for interprocess communication (IPC) for a single consumer single producer (SCSP) pattern and a ring buffer data structure. The SCSP pattern and lockless algorithm offer high-performance data transfer. The report discusses the design, implementation, and evaluation of the lockless SCSP communication mechanism, highlighting its benefits in reducing contention and improving scalability. Performance benchmarks demonstrate its efficiency. The report concludes by outlining potential extensions for accommodating multiple producers and consumers, enhancing the flexibility of the lockless SCSP mechanism in diverse applications.

Key words - *TYCHE, trusted execution environment, inter process communication, shared memory, ring buffer, single producer single consumer*

1 INTRODUCTION

Trusted Execution Environments (TEEs) are essential components in modern computing systems that provide secure and isolated execution environments for performing critical computations. To fully harness the benefits of TEEs, efficient and secure communication between different enclaves and confidential virtual machines or processes is vital. This necessitates the development of a robust and high-performance communication mechanisms for TEEs. The primary objective is to establish a seamless and efficient communication infrastructure that guarantees the confidentiality, integrity, and availability of transmitted data.

Shared memory communication emerges as a key mechanism for inter-process communication (IPC) involving TEEs. By allowing enclaves to share a common memory region, shared memory communication enables direct and low-latency data transfer. This approach eliminates the overhead of serialization and data copying between processes, resulting in faster and more efficient communication[5]. Moreover, the shared memory region can reside within the secure boundaries of the TEE, ensuring strong isolation and protection against external threats.

Efficient shared memory communication is vital to supporting the communication needs of complex software systems running within TEEs. As the number of TEE instances and the complexity of their interactions increase, the ability to handle multiple producers and consumers becomes crucial. Therefore, a communication library for TEEs should be designed and implemented to offer flexibility and scalability, accommodating diverse communication patterns and meeting the demands of real-world applications.

In our project, we focus on designing and implementing a Rust library for shared memory communication within TEEs. We want our library to be used safely by any user: we made sure that no memory threat or leak of any kind is possible when using it. Rust, a modern systems programming language, combines high-level abstractions with low-level control, making it well-suited for developing secure and efficient software components. Leveraging Rust's memory safety and concurrency features, our library aims to provide a reliable and efficient communication framework for TEEs.

The initial focus of our Rust library for shared memory communication centers on a single producer and single consumer scenario, utilizing a ring buffer data structure. This design choice facilitates efficient and ordered data transfer between enclaves, while minimizing contention and synchronization overhead. However, the library's flexibility allows for future enhancements to support more complex communication patterns, including multiple producers and consumers, or different synchronization mechanisms.

In addition to the design and implementation of the Rust library, this research report evaluates the performance characteristics of shared memory communication. We assess the throughput of our library under various workloads and system configurations. This performance analysis provides valuable insights into the practical implications of using shared memory communication for IPC within TEEs, aiding in identifying potential optimizations and trade-offs.

Furthermore, we discuss potential future directions and extensions of our Rust library for shared memory communication. As TEE-based applications evolve, there may be a need to support multiple TEE instances, advanced synchronization mechanisms, or additional security features. We emphasize the adaptability of our library, highlighting its potential for growth to meet evolving needs and requirements.

Through this research, we contribute to the field of communication for TEEs by providing a comprehensive and efficient solution for shared memory communication using Rust. Our work aims to enhance the capabilities and possibilities of inter-process communication within TEEs, promoting the development of secure, scalable, and efficient software systems.

By enabling seamless and efficient communication between enclaves, we strive to bolster the security, privacy, and collaboration capabilities of TEE-based applications.

2 BACKGROUND AND DEFINITIONS

2.1 Defintions

Definition 1 (inter-process communication) :

inter-process communication refers to the mechanisms and techniques used by processes or threads to exchange data, synchronize their actions, and collaborate in performing tasks. IPC enables communication and coordination between separate processes running on the same system or across different systems.

Definition 2 (Ring buffer) :

A ring buffer, also known as a circular buffer, is a data structure that represents a fixed-size queue where data elements are stored in a circular manner. It consists of a fixed-size buffer or array and two indices: a read index and a write index. The read index points to the next element to be read from the buffer, and the write index points to the next position where an element can be written. When the write index reaches the end of the buffer, it wraps around to the beginning, creating a circular behavior.

Definition 3 (Single producer-Single consumer) :

Single producer-single consumer refers to a communication pattern or scenario where there is only one consumer thread or process that reads data from a buffer, and one producer thread or process that writes data into the same buffer. In SPSC, the producer writes data to the buffer, and the consumer reads data from the buffer without any concurrent access from other producers or consumers.

2.2 Lockless Circular Buffer over Shared Memory

Lockless Circular Buffer over Shared Memory is a technique that offers efficient and low-latency communication between processes by utilizing a circular buffer data structure without the need for locks or synchronization mechanisms. The lockless design allows for concurrent read and write operations, reducing contention and maximizing throughput. The use of shared memory further enhances the performance by eliminating the overhead of data serialization and inter-process communication. This approach has been widely adopted in various domains where high-speed data transfer is crucial, such as real-time systems, high-performance computing, and distributed computing. [7]

2.3 Rust memory ordering

Our code use atomic operations (namely `load` and `store`) to ensure consistency between processes. In Rust[1], memory ordering is crucial to ensure proper synchronization and consistency when multiple threads or processes access shared data. The recommended memory ordering in Rust is "Sequentially Consistent" ordering (`SeqCst`), and provides the strongest consistency guarantees by enforcing a total order of all atomic operations, ensuring appearance in a global linear order. However, this ordering may introduce unnecessary synchronization overhead, especially in scenarios where strict consistency is not required. In our scenario we aimed to strike a balance between consistency and performance. We found that `SeqCst` ordering, with its strong consistency guarantees, imposed more overhead than necessary.

In our final implementation, we used `Acquire` load and `Release` store.

`Acquire` memory ordering guarantees that subsequent memory operations occur after the Acquire operation, which is essential for maintaining the desired order of data access between processes in TEEs. Paired with the `Release` ordering flag, `Acquire` ordering forms a "memory sandwich" that ensures synchronization with other CPUs, facilitating secure and efficient communication. In weakly ordered systems (such as ARM and Risc-V), additional CPU instructions, such as memory fences, are used to synchronize the `Acquire` operation with memory modifications by other CPUs, preventing memory reordering before the `Acquire` load. On strongly ordered CPUs (such as x86_64), these additional instructions and memory fences have no performance cost but maintain the desired ordering and synchronization guarantees.

Furthermore, in our project we leveraged this ordering with `Release` ordering to ensure that memory operations occur before the `Release` flag. This ordering is commonly used alongside `Acquire` ordering. In weakly ordered systems, memory fences are employed to ensure correct ordering and visibility for observing cores. An **Acquire** load guarantees the processing of all messages and fetches the correct value, even if other cores invalidate the loaded memory. A `Release` store must be atomic and invalidates other caches holding the value before modification. While `Release` ordering is weaker than Sequentially Consistent (`SeqCst`), in terms of consistency guarantees, it offers better performance, which is crucial for efficient communication within the TEEs. `Release` and `Acquire` ordering are often used together ensuring that specific operations occur after acquiring the lock but before its release. Strongly ordered CPUs invalidate all instances of a shared value in L1 caches before modification, guaranteeing an updated view for an `Acquire` load and instant invalidation of cache lines on other cores for a `Release` store.

3 ALGORITHM

We selected a single-producer single-consumer (SPSC) scenario for our inter-process communication (IPC) mechanism. This decision allows us to design and implement our algorithm with careful consideration, avoiding unnecessary complexity while ensuring flexibility and scalability for future endeavors.

3.1 Motivation

Different mechanisms for inter-process communication such as sockets and networking, remote procedure calls (RPC) or pipes and FIFOs are available. For TEEs, where the focus is on local and secure communication, the use of sockets and networking may introduce unnecessary overhead and security concerns. RPC frameworks often involve serialization and deserialization of function arguments and results, introducing computational and communication overhead that may not be suitable for high-performance TEE communication. Pipes and FIFOs are typically designed for communication between unrelated processes and may not be optimized for the SPSC pattern. They involve data going through the kernel, which is the opposite of what we want with TEEs.

User-level shared memory (Figure 1) thus seemed like the perfect candidate. This mechanism shows great performance by directly accessing shared memory regions, the overhead associated with serialization and deserialization or copying data between processes is minimized. This results in lower latency and higher throughput. Shared memory within TEEs provides enhanced data privacy and security. The shared memory region resides within the secure boundaries of the TEE, protecting the data from external threats. This ensures the confidentiality and integrity of sensitive information, making shared memory a secure option for inter-process communication in TEEs. In addition, shared memory offers a good trade off between simplicity and scalability. As the number of TEEs grows, shared memory enables direct communication channels between them, facilitating efficient collaboration and synchronization.

More specifically we decided to use a ring buffer data structure to implement the shared memory IPC mechanism. Ring buffers are efficient because they use a fixed-size buffer, eliminating the need for dynamic memory allocation or copying data between processes. The use of simple pointer updates and synchronization mechanisms results in low overhead for data transfer. The producer and consumer can operate independently, allowing for asynchronous communication between processes. However, ring buffers come with their limitations as they do not inherently support message boundaries. If messages of different sizes are stored in the buffer, additional synchronization or signaling mechanisms may be needed to indicate the size or presence of each message. This will be described in more details in section 5: Implementation for unfixed size messages.

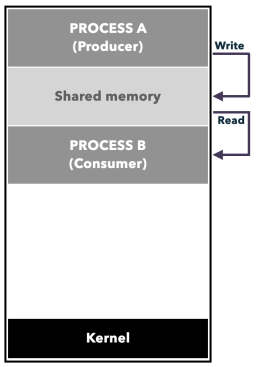

Figure 1: Shared memory: main idea

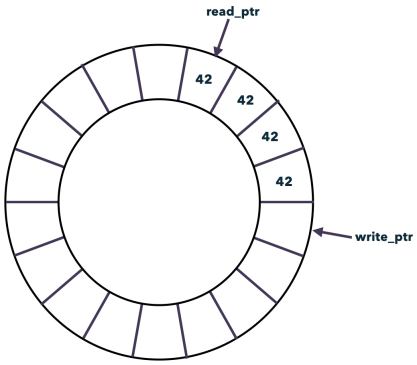

Figure 2: Ring buffer

3.2 Description

A ring buffer (Figure 2) is a specific implementation of shared memory IPC that allows for efficient and synchronized data transfer between two processes. It is a circular buffer where data is written by one process (producer) and read by another process (consumer). The structure of a ring buffer consists of a fixed-size memory region. This region is then split into same-sized slots. The buffer contains two pointers namely head and tail. The correct position for both writing and reading is determined by the head (reading), and the tail (writing). Those are updated when the data is produced or consumed.

The write pointer can only be updated by the producer and the read pointer can only be modified by the consumer. When a pointer reaches the end of the buffer, it wraps around to the beginning (hence the "ring" in ring buffer). The producer and consumer must coordinate and avoid reading from or

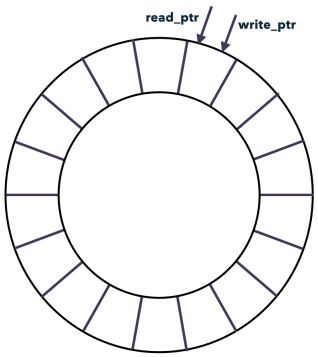

Figure 3: An empty ring buffer

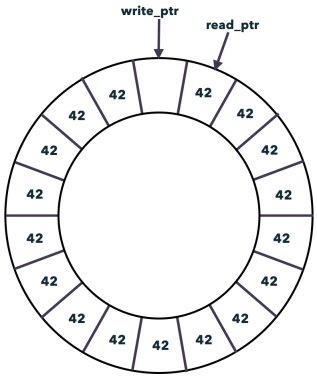

Figure 4: A full ring buffer

writing to the same slot simultaneously. Hence the buffer's state, such as whether it is full or empty, can be determined by comparing the positions of the read and write pointers.

In our implementation the buffer is said to be empty (Figure 3) when both pointers are equal (same index). The buffer is said to be full (Figure 4) when the write pointer is 1 element behind the read pointer in the ring (i.e. for a ring buffer of capacity N, if the read pointer is at 0 and the write one is at N-1). This relation can be represented by the operation

$$(\text{read\_idx} + \text{buffer\_size}) - \text{write\_idx} \equiv 1 \mod \text{buffer\_size}$$

In a few words, a producer needs to check if the buffer is full. If it is, the producer will not write anything but will return an error stating that the buffer is full. If not, the consumer will write the element to the buffer. A consumer will check if the buffer is empty. If it is, the consumer will not read anything and the read pointer will not be incremented. If it is not, the consumer will read the element on the buffer and its pointer will be incremented.

4 IMPLEMENTATION FOR FIXED-SIZE MESSAGES

The first goal of our library is to implement it for fixed size message (here we choose u32 values that are represented in 4 bytes). In that's way we can build strong foundations for our algorithm, and get familiar with Rust and its security rules. This library should be used without having to worry about memory security issues for the users. Our library creates a ring buffer to be used for SPSC. An user is able to create it starting from a memory space allocated for sharing (its address and its length).

4.1 Structure

Three structs enable us to implement this: RingBuffer, Consumer, Producer. Structs in Rust allow you to group multiple values of different types together. Each value in a struct is given a name, making it easier to understand the meaning of the values. Structs provide flexibility as you don't have to rely on the order of the data to specify or access the values of an instance.

```
pub struct RingBuffer {
    pub buffer: &'static mut [u32],
    read_idx: &'static AtomicUsize,
    write_idx: &'static AtomicUsize,
}
```

First, the RingBuffer struct represents the ring buffer. We characterize it with a mutable reference to a slice of `u32` values that has a static lifetime, representing the available slots of our ring buffer. Each slot can contain an `u32` which is the type of our messages. Our ring buffer also needs to have two pointers, one for the head (read) and one for the tail (write). In the context of SPSC scenario, these two values can be changed by two different processes, namely the consumer and the producer. It is important that those values are synchronized between each processes. An `&'static AtomicUsize` represents a mutable reference to an `AtomicUsize` value with a `'static` lifetime. This means the reference can be accessed and modified throughout the program's entire duration, enabling concurrent access to the underlying value with atomic operations, ensuring safe and synchronized access in multi-threaded scenarios.

Without the use of atomic operations, there is a risk of encountering data inconsistency issues. For example, a process could write data to the buffer and update the pointer indicating the availability of new data. However, without proper synchronization provided by atomic operations, other cores or threads may not yet see the updated data, leading to the possibility of reading outdated or incorrect values. Memory ordering

extends beyond the scope of a single memory address and aims to synchronize memory updates across different cores or threads.

The structure of the consumer and the producer are quite similar. Both take a mutable reference to a **RingBuffer**, and two `usize` which are locally stored value of the head and tail indices. Theses two values are important in order to increase the throughput of our library by decreasing the need to fetch the corresponding atomic values.

```
pub struct Producer {
    pub inner: &'static mut RingBuffer,
    local_read: usize,
    local_write: usize,
}

pub struct Consumer {
    pub inner: &'static mut RingBuffer,
    local_write: usize,
    local_read: usize,
}
```

4.2 Initialisation

The initialisation process of our ring buffer ensures proper alignment, non-null pointers, and correct initialization of the read and write pointers, as well as the buffer. This sets up the initial state for the ring buffer, allowing subsequent push and pull operations by the producer and consumer, respectively. The `new` function takes a pointer (*ptr*) and its length (*len*) as inputs, representing the allocated memory space. The `new` function calls the `init()` function, which returns a tuple containing initialized producer and consumer instances.

One important thing to consider is that a lot of functions used in the initialisation of our **RingBuffer** are `unsafe{}` functions. In Rust, the `unsafe` keyword is used to mark certain blocks, functions, or traits that contain operations or constructs that are not guaranteed to be memory-safe, data-race-free, or compliant with other safety guarantees provided by the Rust language. The *unsafe* keyword essentially allows one to bypass some of Rust's safety checks and take the responsibility of ensuring safety yourself. The Rust documentation [3] thoroughly documents such function, in such a way that the Safety check to perform are clearly stated (i.e., a pointer needs to be valid, to be a certain size, etc.). This documentation ensures the safety of our library.

The allocated shared memory space is divided in three parts: three raw pointers, namely *write_ptr*, *read_ptr* and *buffer_ptr*.

The first two pointers are of type `AtomicUsize`. They need to be non-null, correctly aligned and initialized to 0. In order to increase the throughput and reduce cache invalidation, we decided to put these two pointers whithin different cache lines (separate them to at least 128 bytes for the M1 chip architecture that we are using to run the test of our library). We aimed to have the second pointer as close as possible to the first while ensuring alignment and cache line spacing. When multiple threads or cores are accessing these variables concurrently, having them in separate cache lines can reduce cache contention and minimize the frequency of cache line invalidation caused by concurrent updates. Cache contention occurs when multiple threads or cores simultaneously attempt to access the same cache line. In such cases, cache coherence protocols may cause the cache line to be invalidated and reloaded, leading to additional latency and reduced throughput.

The last pointer is the one that will point to our buffer, it is raw pointer `u32`. Based on its value we then compute the size of our buffer (number of elements, `u32`, that we can push in it). This size is computed based on the length of our memory space, the space already used by our previous pointers (including the empty space for alignment and the change of cache line). Once we compute it and find the correct alignment for our *buffer_ptr* we use the `std::slice::from_raw_parts_mut()` [4] function. It will return a mutable slice from a pointer (*buffer_ptr*) and a length (*buffer_size*).

```
unsafe fn init(
    read_ptr: &'static AtomicUsize,
    write_ptr: &'static AtomicUsize,
    buffer_data: &'static mut [u32],
) -> (Producer, Consumer) {
    let rb: RingBuffer = RingBuffer {
        buffer: buffer_data,
        read_idx: read_ptr,
        write_idx: write_ptr,
    };
    let ring_buffer_ptr =
    Box::into_raw(Box::new(rb));
    (
        Producer { inner: &mut
    *ring_buffer_ptr, local_read: 0,
    local_write:0},
        Consumer { inner: &mut
    *ring_buffer_ptr, local_write:
    0,local_read:0},
    )
}
```

Ultimately, the initialization process creates a new **RingBuffer** and assigns its corresponding pointer to a consumer and producer. Both these struct should initialize their local read and write pointer to 0.

To give a more concrete example, let's state that the shared memory region has as a start address 0x02 and a length of 1024 bytes. The `read_ptr`'s address is 0x8 (since `AtomicUsize` is 8-align), the `write_ptr`'s address will be 0x90 (we want to change cache-line, therefore we add 128 bytes, there is no need to add bytes to align this `AtomicUsize` since 0x90 is a multiple of 8). The slice representing the buffer will have 109 `u32`'s slots, its start address will be 0x98.

4.3 Functions

Now let us focus on the function `push` and `pull`. These functions are defined for the **Producer** and **Consumer** processes, respectively.

Both of these functions follow the same workflow. In order to reduce the number of memory accesses and to increase the throughput, we limit the number of atomic operations that are unnecessary. When two threads are running at the same time, the processor will allocate some processing times for each, resulting in a block of push or pull instructions. It is thus, not necessary to load the read or write pointers at each instruction.

Let us take the `push()` function to illustrate it. The write pointer can only be modified by the producer, therefore, a producer can keep a local version of their pointer value, and increment it each time a push is successful. The producer, then, only need to store the updated value into the atomic without the need to load it first. Concerning the read pointer, a local copy is made in order to limit the number of memory accesses. As explained earlier, the atomic values do not need to be loaded with each function call, for example if we load the atomic *read_ptr* at time T and store its value N in the local variable *local_read*, a producer does not need to load it again until its local variable *local_write* is equal to N-1, meaning when the buffer is full based on these local variables. Our `push()` function takes a mutable reference *self* (the producer) as an argument along with the value to be pushed. It can be divided in three main steps:

(1) Check if the buffer is full (with respect to the local variables)
   (a) If it is not full, proceed to point (2).
   (b) If it is, then load the atomic value *read_ptr* into a variable *loaded_read*.
   (c) If the local value is the same as the atomic one, return an error indicating that the buffer is full.
   (d) If both values are different, update the local variable: `local_read = loaded_read`
(2) Push the value into the buffer.
(3) Increment the *local_write* variable (modulo the size of the buffer) and update the atomic value of *write_ptr*

The use of atomic operations in our implementation is crucial for ensuring proper synchronization and consistency in shared memory communication. Atomic operations provide specific memory ordering semantics that guarantee visibility and ordering of read and write operations across different cores or threads.

In our case, the atomic operations serve two important purposes. First, they ensure that any write operation performed before the write to the atomic variable becomes visible to any subsequent reader operation (release semantic). This guarantees that the data written to the shared memory buffer is properly synchronized and available for consumption by other processes.

Second, atomic operations ensure that any read operation performed before the read of the atomic variable behaves as if it happened after the read of the atomic (acquire semantic). This ensures that all necessary data updates are retrieved before reading from the shared memory buffer, preventing the possibility of reading incomplete or inconsistent data.

4.4 Throughput

In the throughput analysis for fixed-size messages, we evaluated the performance of our shared memory IPC mechanism in terms of data transfer rate. The throughput measurement provides insights into the efficiency and speed at which data can be exchanged between the producer and consumer. The throughput is highly variable based on the device machine used to do the benchmark. For these measurements we used a MacBook Pro with an M1 chip, and equipped with 8 CPUs. The benchmarking process was performed under optimal conditions, with no other concurrent activities running in the background.

We examined factors such as buffer size, number of messages sent or size of messages to understand their impact on the data transfer rate. The showcased results are messages of 4 bytes (`u32`). The memory space size varies between 1'000 and 1'000'000, the number of messages varies between 1'000 and 100'000'000. The example that allows us to calculate the throughput of our library is simple: We create a **RingBuffer** with a raw pointer and memory space of length *CAPACITY*, along with corresponding **Producer** and **Consumer** instances. We create two parallel threads, one for the producer to push (write) a message to the shared memory, and another, for the consumer to pull (read) the available messages. The producer can write the i+1th message if and only if the ith message has been correctly pushed (no error). If an error occurs, the producer needs to try to push it again. The consumer follows the same workflow.

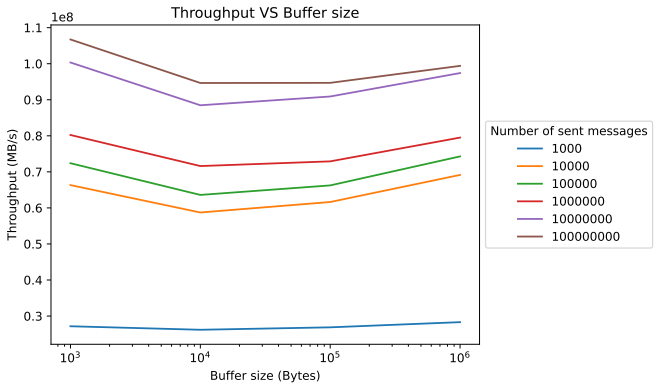

Figure 5: 4 bytes long messages' throughput

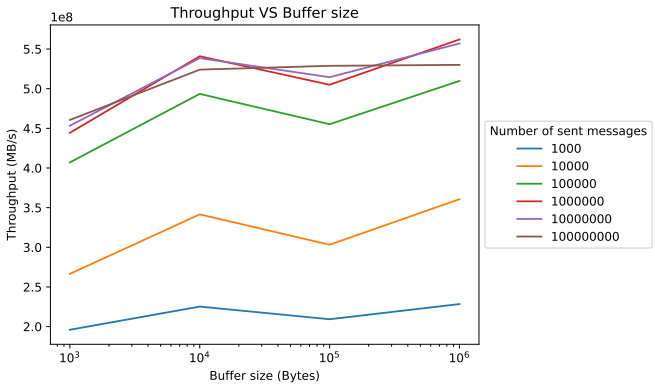

Figure 6: 16 bytes long messages' throughput

We start a timer before initiating both threads and stop it when both threads have completed. The throughput is then calculated as follows:

$$\frac{number\_of\_messages * 4}{elapsed\_time}$$

In this equation, the multiplicative factor 4 corresponds to the number of bytes needed to represent each message, considering they are `u32` messages.

During our benchmarking, the scenario with 100'000'000 messages sent was the only one which took a bit more than 1 seconds. Figure 5 shows that with this scenario our throughput is approximately 100MB/s. The throughput increases with the size of the memory allocated. This throughput was surprisingly low compared to what we anticipated. One reason for this result may be due to the size of the messages tested (4 bytes). indeed, this can result in poor cache use and less parallelism in the CPU compared to the use of bigger messages. This hypothesis was confirmed when we modify the library to write and read 16 bytes messages (`u128`), the throughput has significantly increased (up to 5 times), as we can see on Figure 6

The larger the size of the message is, the bigger the throughput. Since the throughput does not appear linear it appears that the size of both messages is not enough to compensate for the overhead caused by cache invalidation.

5 IMPLEMENTATION FOR UNFIXED SIZE MESSAGES

5.1 Modification

In our journey to implement a scalable library, we decided to implement our mechanisms for unfixed size messages. A producer can write any type of data to the buffer, and, more importantly, a consumer can read the entire data without missing any bytes of information.

To do so, we needed to modify our implementation and some part of our algorithm since the ring buffer data structure only supported slots of the same size. This is why our ring buffer is now sliced into single-byte slots. All types of messages and information can be expressed as a vector of bytes. To avoid adding too much overhead or losing too much memory, we decided to limit the size of the messages to a number that can be represented by 2 bytes.

Beside this, our initialization stayed the same. However, the write and read functions need to come up with a new workflow. The write method will now compute the size of the message being sent and verify not only if the buffer is full, but also if it can store the message. This would be done by another function that computes the available capacity of the buffer

(read_idx + buffer_size) - (write_idx+2) mod buffer_size

We add 2 to the write index to store the size of the message in the first two bytes following the pointer before storing the message's bytes. Then we need to compare the capacity with the size of the message. A full buffer and a buffer that does not have enough capacity result in different error messages, allowing the user to handle them according to their preference.

We then need to convert the length of the vector into a byte array (little endian) where each entry will be stored in a buffer slot. In a for loop, we will store every element of our message (represented as a vector of bytes) into our ring buffer and increment our local write index. At the end of the for loop, we store our local variable into the atomic ring buffer.

The read function also encounter some modification in this implementation. The consumer needs to read the first two bytes and convert it (little endian) into an integer in order to do a for loop to read every bytes that constitutes the message. In the for loop the local read index is incremented for each bytes that is read. At the end of the function we store our local variable into the atomic ring buffer.

5.2 Performance and possible modifications

During our evaluation of the unfixed size message implementation, we observed unexpected results in terms of throughput. The measured throughput was lower compared to the fixed size implementation. This outcome was anticipated for small size messages, as the additional overhead required by our implementation did not provide significant benefits. Surprisingly, even larger messages experienced low throughput. However, it is worth noting that the throughput still increased as the size of the messages grew, mirroring the behavior observed in the fixed size implementation. Additionally, we noticed that the buffer size did not have a significant impact on the throughput, similar to the fixed size implementation.

Despite our efforts, we were unable to pinpoint the exact bottleneck in our implementation. We suspect that the example used for benchmarking may have influenced the results. While our implementation functions as intended, further work is required to accurately assess the library's true throughput and identify any performance bottlenecks. This understanding is crucial for refining our implementation and enhancing its performance.

One possible modification that can be easily implemented into our library, depending of future constraints or requirements, can be a switch case on the message type. such that each type can be represented as an integer (i.e u32 -> 1; char -> 2...) to help the consumer interprets the bytes vector received without too much overhead.

We can also add intermediate functions in order to encode messages into fixed size ones. We can do it by using encoding algorithms or hashing.

6 CHOICES

6.1 Freedom library's users

Since the library is in its early stages, the users still have a lot of freedom regarding the use of our library. We enforce this with the use of multiple design choices for our implementation.

The library offers implementations for both fixed and unfixed size messages. This design choice provides flexibility to users, allowing them to choose the appropriate message size based on their specific communication requirements. By supporting both types, the library caters to a wider range of use cases and enables users to efficiently exchange data of varying size within the shared memory.

Our library will not crash if we cannot read or write any data from, or into the buffer. We decided to return an error with meaningful messages (i.e. "The local buffer is full, retry", "You currently do not have enough space to write the value" etc.). This enables users to decide subsequent actions, (do they want to retry to send the same message), and gives them more information on the buffer activities and its state. This will also help users understand our memory communication approach to better use it in order to achieve optimal performances.

By implementing support for fixed and unfixed size messages and providing meaningful error messages, the library strives to offer a user-friendly experience. Users can choose the message size that suits their needs, whether it's a predefined fixed size or a dynamically determined size. Additionally, the library's behavior, by returning informative error messages, ensures that users can easily handle different buffer states scenarios without encountering unexpected crashes or undefined behavior.

While these design choices enhance usability, trade-offs and considerations should be taken into account. Handling unfixed size messages introduce additional complexity, such as the need for synchronization and signaling mechanisms to indicate message boundaries or sizes. Furthermore, supporting both fixed and unfixed size messages may incur a slight overhead cost compared to a library that solely focuses on fixed-size messages. Nevertheless, these trade-offs are carefully weighed against the goal of providing flexibility and ease of use for users working with shared memory IPC in TEEs.

6.2 Security checks

While the library is designed to provide freedom of use to the users as seen above, it still needs to be secure as we are operating inside TEEs and want to prevent malicious communications between processes/enclaves. Our code, as well as our choice of shared memory, has been carefully designed with that perspective in mind. The library incorporates various safeguards to ensure data integrity and protection against potential vulnerabilities, mitigating the risk of data tampering, and other security threats.

The library is specifically designed to target Tyche[6], a trusted execution environment (TEE) platform. By leveraging Tyche's inherent memory isolation capabilities, the library creates a secure execution environment where shared memory communication takes place. The shared memory region is confined within the secure boundaries of the TEE, ensuring that only authorized processes within the TEE can access or modify the data. This isolation provides a strong defense against unauthorized access or tampering, establishing a robust security barrier. The library's design and implementation specifically address the SPSC scenario within the Tyche TEE environment.

Rust is often considered a secure language due to its focus on memory safety and strong compile-time guarantees. For example Rust's ownership and borrowing system ensures memory safety by enforcing strict rules on memory access. It eliminates undefined behavior, which can lead to security

vulnerabilities. It achieves this through various mechanisms, such as preventing null pointer dereferences, ensuring thread safety, and avoiding data races. Rust's ownership and borrowing system guarantees thread safety by preventing data races. It enforces strict rules to ensure that concurrent access to shared data is properly synchronized. Rust provides safe abstractions for concurrent and parallel programming. It offers built-in constructs like threads, channels, and synchronization primitives, which are designed to prevent data races and ensure thread safety. While Rust provides strong foundations for security, we still need to follow security best practices and apply secure coding techniques to ensure the overall security of their applications. This is even more important in our project since we interact with raw pointers, with unsafe functions. While doing so, we bypass some of the security limits Rust has.

As stated in section 4.2, Rust has great documentation about raw pointers, or even unsafe function to enable user of their language to code as safely as possible. This is the documentation that we used in order to make sure malicious scenarios result in an error in our library. Our library includes robust error handling and logging mechanisms to detect and report potential security incidents. By carefully handling errors and logging relevant information, the library facilitates the identification and investigation of security-related events, allowing for timely responses and mitigation of potential threats.

During the initialization phase, our library takes additional security measures to ensure the integrity and reliability of the shared memory. These measures include checking each pointer created during initialization to ensure that it is not null and that it is correctly aligned. The library verifies that each pointer created during initialization is not null. This check is crucial for preventing potential null pointer dereference vulnerabilities, which could lead to crashes or security exploits. By ensuring that pointers are valid and not null, the library avoids accessing uninitialized or invalid memory locations, enhancing the overall security and stability of the shared memory communication. In addition to null pointer checks, the library also validates the alignment of each pointer created during initialization. Alignment refers to the memory address at which data is stored, and it plays a significant role in the performance and security of memory operations. By enforcing correct alignment, the library avoids potential issues related to misaligned memory access, which can lead to data corruption, performance degradation, or even security vulnerabilities. Proper alignment ensures that memory operations are performed efficiently and correctly, reducing the risk of unintended behavior or security vulnerabilities.

As all secure code, the library is open[2]!

7 FUTURE WORKS

As the shared memory inter-process communication library is still in its early stages of development and deployment, it is essential to acknowledge that the precise needs and requirements of future users may not be fully apparent at this time. However, the library has been designed with flexibility in mind, allowing for potential modifications and enhancements to meet evolving demands. The following areas represent potential avenues for future improvements and adaptations:

Multi-Consumer Support: The current focus of the library is on facilitating communication between a single producer and a single consumer using a ring buffer. While this design suits many use cases, it is crucial to consider the possibility of future scenarios that require multiple consumers or producers. Implementing support for such configurations would enable greater scalability and accommodate more complex communication patterns.

Performance Optimization: Ongoing efforts can be directed toward optimizing the library's performance. This may involve fine-tuning the existing implementation, exploring advanced algorithms and data structures, and leveraging platform-specific features or hardware acceleration to enhance throughput and reduce latency.

64-bit arithmetic: Currently, the modulo arithmetic used limits the maximum number of elements the buffer can hold to N-1. By employing 64-bit arithmetic, it would be possible to increase this capacity and allow the buffer to be truly full with N elements. This would open up new possibilities for buffer utilization, offering greater flexibility and more efficient use of available memory space. However, it would require a revision of the buffer's data structure and associated operations to accommodate larger index and pointer values. Exploring this approach could lead to significant performance and scalability improvements for the library, especially in scenarios with higher workloads.

It is important to note that these potential future directions are preliminary and subject to change as the library evolves and user needs become clearer. The focus remains on maintaining a flexible and adaptable design that can accommodate a wide range of requirements in shared memory inter-process communication.

8 CONCLUSION

In conclusion, this report presens the design and implementation of a communication mechanism in Rust for inter-process communication (IPC) within Trusted Execution Environments (TEEs). The lockless mechanism, based on the single producer single consumer (SPSC) pattern and a ring buffer data structure, offers high-performance data transfer with reduced contention and improved scalability.

The primary objective of the project was to establish a seamless and efficient communication infrastructure that ensures the security principles of transmitted data within TEEs. By leveraging shared memory communication, the library enables direct and low-latency data transfer, eliminating the overhead of serialization and data copying.

The design choices made in the library prioritize flexibility and ease of use. The implementation caters to both fixed and unfixed message sizes, ensuring that the library is adaptable to various application requirements. Additionally, the library handles buffer full or empty scenarios by returning meaningful error messages instead of crashing, therefore enhancing usability and reliability.

Security considerations were paramount in the library's design and implementation. Measures were taken to ensure data integrity and protection against potential vulnerabilities. Memory isolation within the TEE environment and secure initialization procedures contributed to the overall security of the communication process.

Using performance benchmarks, the library's efficiency and throughput were evaluated, and demonstrated significant suitability for IPC within TEEs. The results validate the effectiveness of our chosen IPC mechanism in reducing contention and improving data transfer rates.

Looking forward, the library offers potential extensions to accommodate multiple producers and consumers, further enhancing its flexibility and scalability. This adaptability ensures that the library can meet the evolving communication needs of TEE-based applications as they grow in complexity.

By providing a comprehensive and efficient solution for shared memory communication using Rust, this research contributes to the field of communication for TEEs. The library's capabilities aim to enhance the security, and collaboration of TEE-based applications, promoting the development of secure, scalable, and efficient software systems.

Overall, this project advances the state of inter-process communication within TEEs, enabling seamless and efficient communication between enclaves while maintaining a strong focus on security and performance.

REFERENCES

[1] Explaining atomics in rust. https://cfsamsonbooks.gitbook.io/explaining-atomics-in-rust/.

[2] Ringbuffer library. https://github.com/MaelysBillon/ringbuffer.

[3] Rust pointer documentation. https:l//doc.rust-lang.org/std/primitive.pointer.html.

[4] Rust pointer documentation - from_raw_parts_mut. https://doc.rust-lang.org/beta/std/slice/fn.from_raw_parts_mut.html.

[5] Kishore Kumar Jagadeesha Aditya Venkataraman. Evaluation of inter-process communication mechanisms. https://pages.cs.wisc.edu/ adityav/Evaluation_of_Inter_Process_Communication_Mechanisms.pdf.

[6] Charly Castes, Adrien Ghosn, Neelu S. Kalani, Yuchen Qian, Marios Kogias, Mathias Payer, and Edouard Bugnion. Creating trust by abolishing hierarchies. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HOTOS '23)*, page 9, Providence, RI, USA, June 2023. ACM.

[7] Shubhadip Paul. Lockless circular buffer over shared memory (high speed data transfer via shared memory). In *2013 International Conference on Advanced Computing and Communication Systems*, pages 1–6, 2013.