XXX: A Capability-Based and Secure Shared Memory System for Efficient Microservices

XXX

Abstract

The design principle of service independence has propelled microservice to prominence as a cloud service architecture, with the great overhead on performance for frequently data sharing. Luckily, shared memory systems allow microservices to access the shared data directly, eliminating that overhead and achieving efficiency. However, the principle of service independence introduces unique and additional security requirements. The memory safety for shared memory is required to be guaranteed. The confidentiality of data in shared data needs guarantees as well. Additionally, systems must guarantee the isolation for shared memory to provent cascading failures. As microservices are developped independently, meeting these requirements becomes increasingly critical.

We propose XXX, the first shared memory system that uses data capabilities to meet all the security requirements for microservices. To achieve this, we restructure the guarantees of shared memory between compilation information and kernel ability by our capabilities designs that abstract from development analysis and function as kernel metadata for shared data. By enforcing capability protocols design, XXX kernel provide security guarantees and meet all the requirements for microservices. Furthermore, we design capability-based imp-replicas for shared data to support asynchronous communication in shared memory and implement additional capability-related components within XXX for distribution scenes. Our evaluation demonstrates that XXX reduces data copying by up to 67% and decreases end-to-end communication latency by 21%-72% in comparison to microservices on RPC. Additionally, when compared to existing shared memory systems, XXX maintains security with only an approximately 11% performance overhead.

1 Introduction

Microservices capitalize on the principle of service independence to acquire benefits such as flexibility, agility, reliability, and isolation, which have propelled microservices to prominence as a cloud service architectures. For example, like Netflix and Amazon [4, 10] are decomposing their monolithic applications into collections of small, independent microservices. Each microservice focues on specific functions and takes responsibility of its own data, and predetermined APIs are provided for invocations between microservices, which standardize format for shared data.

However, service independence also result in a great sacrifice since there are frequently data sharing across microservices. Shared memory system provides efficient and fast communication mechanism by accessing the shared data directly, which are quite beneficial for microservices. Nevertheless, shared memory introduces extra security problems for microservices and there are unique requirements for a shared memory system to support microservices. **(R1) Service Independence.** To support microservice, it is required to offer asynchronous communication and avoid depending mechanisms like mutexes, semaphores, or condition variables which grant microservices with independence in development, deployment, dynamic update and distributed migration. **(R2) Memory Safety.** For security, the system is required to avoid memory-related issues in shared memory which can lead to incorrectness, unintended behaviors and even crashes. The local memory for one microservice is guaranteed by its developer, shared memory introduces extra memory to be protected and the developers are lacking the global information for protection. **(R3) Confidentiality and Access Control.** For security, the system is also required to prevent unauthorized data access in shared memory, since microservices are inevitably sharing protected shared data with others. Shared memory leads to additional and unique risks at data leakage, requiring the system to provide efficient, distributed and fine-grained access control in shared memory. **(R4) Isolation Across Microservices.** For security, shared memory breaks the natural isolation across microservices, causing additional problems that failure of one microservice can impact others through shared memory. The system is reqiured to eliminate such impact and regrant microservices with isolation.

Unfortunately, there is no existing shared memory system that meets all the requirements for microservices. Systems with language-based protection provide safety guarantees with **compilation information** like compilers, runtime systems guarantees, and static analysis, They are generally facing the problems of breaking sesrvice independence*(R1)*, while confidentiality*(R3)* and isoation*(R4)* are also challenging for them. Systems with isolation-based protection gain guarantees from **kernel ability** like interceptions, access control and synchronization primitives, leading to dilemma between sesrvice independence*(R1)* and memory safety*(R2)* since most of synchronization primitives include depending mechanisms. Hardware design like CHERI [21] guarantee the security by **capability**, *an unforgeable token which grants its holder rights to access a specific memory region [22]*. They rely on specific hardware design, disabling distributed migration for microservices and therefore break service independence(**R1**).

Guarantees Based On		**Service Independence**	**Memory Safety**	**Confiden -tiality**	**Isolation**
Language	KaffeOS [3]	✗	✓	✓	✓
	J-kernel [20]	✗	✓	✓	✗
	RedLeaf [15]	✗	✓	✗	✓
Isolation	FlexOS [13]	✓	✗	✓	✓
	CubicleOS [18]	✓	✗	✓	✓
	Shimmy [1]	✗	✓	✗	✓
	Boost [9]	✗	✓	✓	✓
Hardware Capability	CHERI [21]	✗	✗	✗	✓
	CAP-VM [17]	✗	✓	✓	✓
	Capstone [22]	✗	✓	✓	✓
	XXX	✓	✓	✓	✓

Table 1: Comparison of XXX and existing shared memory systems.

The key reason why related works on shared memory do not meet all the requirements is that they offer all their security guarantees relying on either compilation information or kernel ability, making them in dilemma between service independence and security. Inspired by existing hardware capability designs [17, 21, 22], our key observation is that the standardized shared data across microservices have great potential to be managed through capabilities by restructuring the guarantees (i.e., access control and memory safety protocols) between compilation information and kernel ability. Security requirements needs to be guaranteed with not only compilation information and kernel ability but also the codesign of them. Only in this way can shared memory achieve security specifically for microservices scenes.

We proposed XXX, a shared memory system to support microservices with all requirements by our capability design, a kernel metadata for each shared data which grants XXX to manage shared memory in a efficient and secure manner. Capabilities are generated from compilation information with developers' declaration on shared data, which utilize kernel ability to enforce capability protocols and guarantee security. Addition kernel components related to capability are also introduced for assistance.

However, even with our capability for codesign, there are still challenges to utilize system-level capabilities across microservices. First, how to avoid memory-related issues dynamically happen in global scense with capabilities from each microservice locally and statically. To address this challenge, our capabilities apply affine system to limit one onwer and implement lifetime to simplify the memory model, which avoid majority of dynamic issues and ease the rest issues. Second, how to deal with distributed microservices with centralized kernel. For this challenge, we design capabilities as an independent and distributed metadata which can be transferred across machines and kernels. Additionally, we design capability proxy, a XXX components to establish host-to-host communication to manage capabilities and shared data in distributed machines. Third, how to provide asynchronous communications based on shared memory for microservices. Microservices are currently based on remote produce calls(RPC), a typical asynchronous mechanism, while shared memory provide synchronous mechanism. To solve this problem, we design implicit replicas model based on our capability design for XXX components to maintain, which can achieve the same results consistent with RPC with better performance during data race.

We implemented the core of XXX system and evaluated our design on DeathStarBench [11], a widely recognized benchmark for microservices with six popular services. We compared XXX to two notable shared memory systems and one classical RPC-based system. Our evaluation shows that:

- XXX is secure (*satisfy R2, R3 and R4*). For memory safety, XXX precisely detected vulnerable data access for the shared data, and efficiently protect shared memory from violations as shared memory systems baselines encountered faults; for data confidentiality, XXX detected all three typical simulation vulnerabilities while shared memory baseline fail to prevent two of them.
- XXX is efficient. XXX achieved up to 67% lower latency than the RPC-based baseline, with only 11%$\tilde{2}$3% overhead compared to shared meory baselines, while the baselines failed to prevent memory-related errors and data leakage in shared memory
- XXX maintians service independence (*satisfy R1*). XXX is able to take over shared data when faults or updates raised within certain microservices, while the most efficient shared memory baseline cascaded failures across microservices through shared memory.

Our major contribution is the system capability design specific for shared memory, the first work that achieves service independence, memory safety, data confidentiality and isolation for microservices on shared memory. Compared to existing relevant baselines, XXX achieved comparable memory safety, isolation as the RPC-based system, comparable efficiency as the shared memory systems, and strong more fine-grained confidentiality than costly access control. This makes XXX competent to handle unique requirements for microservices scenes, encouraging more microservices to utilize the efficiency of shared memory for better performance. Additionally, our imp-replicas model and capability proxy can enforce our capability protocols in distributed scenes to guarantee security for microservices on shared memory.

2 Background

2.1 Microservices with the Principle of Independence

Microservice is an outstanding services architecture on cloud based on the principle of service independence. This principle fosters flexibility and scalability, as services can be developed, deployed, and scaled independently. Moreover, microservices are also designed to prioritize resilience and memory safety. Each microservice is built by its own developer and deployed with strong isolation from other microservices. Communication between microservices is exclusively done through predetermined APIs, ensuring standardized interactions and data exchange. By applying appropriate exception handlers for remote invocations, all faults can be well isolated.

The principle of independence also introduces challenges for microservices. One of them is the dilemma between co-operation and data confidentiality. Each microservice has its own functions and privacy data. Different from monolithic services where all function modules are trusted, for one microservice, others are not malicious but considered to be vulnerable. When they share privacy data to others, the privacy data can be leaking.

Another significant challenge arises from the trade-off between memory safety and efficiency within microservices. Given the frequent data sharing inherent to microservices, the communication mechanisms for implementing APIs make an essential difference on performance. While remote procedure call (RPC) offers fine fault isolation, the replication of data it necessitates introduces inefficiencies. On the other hand, zero-copy approaches like shared memory and direct access to data within a single address space present their own set of challenges. These methods are intricately designed and can potentially compromise both isolation and memory safety.

2.2 Efficient Data Sharing Based on Zero-copy Approaches

For systems designed to support microservices, zero-copy [14] approaches are considered efficient as they aim to minimize redundant data replicas. In most cases, a microservice simply needs to read shared data from other microservices, and load balancers consistently allocate them to the same machines. Replicas are only inevitable when data races occur, consistency is compromised, or distribution requirements arise. Various approaches have been proposed to avoid redundant data replicas in order to achieve efficient data sharing. These approaches differ in their design principles and application contexts.

2.2.1 Direct Access Based on Language Protection

For monolithic applications, the compilation and deployment process combines all function modules within the same address space. This allows one module to pass pointers and references to another module, enabling direct access to shared data. Related works enhanced the ability of language protection. For KaffeOS [3], the JVM incorporates the concept of kernel mode and user mode, enhancing isolation and scheduling capabilities for Java. J-Kernel [20] introduces Java capability objects to track dataflow and ensure data confidentiality for shared data. Similarly, RedleafOS [15] leverages information from the Rust compiler, such as ownership, to enable both isolation and efficiency for data sharing.

However, such an approach is inherently unsuitable for microservices. Directly accessing shared data in microservices requires support from the runtime system or compiler. Without such support, data races and memory management issues can lead to incorrectness, inconsistency, and even system crashes. Additionally, deploying all microservices within a single address space is infeasible. Each microservice operates with its own set of libraries and language runtime, independent from other microservices.

2.2.2 Shared Memory Isolated by Operating System

In the context of multi-processes or threads, shared memory is common method to achieve efficiency. In this approach, the operating system manages the shared memory and maps it into user processes. Consequently, when one process moves data into the shared memory, all other processes can access it without the need for data replication. Various research efforts have focused on enhancing the capabilities of the operating system in terms of isolation and distribution. For instance, FlexOS and CubicleOS [13,18] enable flexible and automated configurations of hardware mechanisms, accommodating diverse isolation requirements. FaRM [2,6] leverages RDMA to enable shared memory functionality in distributed systems. Libraries such as Boost aim to encapsulate shared memory operations and provide interfaces that automatically maintain memory safety.

However, it is worth noting that modern shared memory systems are primarily not designed to address the requirements of independent microservices. Developers are typically required to agree upon protocols to ensure memory safety, which is often impractical for the independent development of microservices. Even shared memory libraries that offer automatic memory safety maintenance rely on techniques such as mutexes and semaphores, which introduce dependencies. Consequently, in the event of failures, fault isolation among microservices can be compromised.

2.3 Hardware Capability System

Hardware capabilities provide a viable solution for secure and efficient data sharing. One example is CHERI [21], which introduces its own hardware capability supported by specific registers and CPUs. Capabilities are assigned to each data, specifying how the data can be accessed and the associated permissions. One module can transfer the data capability to another, then the recipient module will be able to access the shared data directly. Related research has further enhanced CHERI's functionality and security. For instance, CAP-VM [17] has designed additional monitors for CHERI, allowing modules to revoke their capabilities, while Capstone [22] extends CHERI instructions to enhance security.

The design of hardware capability systems has provided valuable insights for efficient data sharing in microservices. Firstly, capabilities propose standardized metadata for shared data, enabling microservices to maintain independence and achieve efficiency. Furthermore, capabilities can contribute to data flow monitoring and protect data confidentiality. Additionally, capabilities facilitate the maintenance of memory safety in shared memory systems. As a result, a kernel-level capability, without requirement for specific hardware supports, would be well-suited for microservices.

3 Overview

Next we will describe the architecture overview, runtime workflow and threat model for XXX. XXX is a microkernel system designed for microservices to meet the requirements such as confidentiality, memory safety, isolation and efficiency. The core design philosophy is to management and maintenance shared data based on capabilities. Developers no longer need tracking the flow of private data or worry about dependences when accessing shared data. They only need to report the capability abstraction and follow the specifications of capability protocols, while XXX takes responsibility.

3.1 XXX Architecture

Figure 1 shows an example describing the relationship between XXX kernel and microservices. XXX allocates specific shared memory, named public memory, for each microservices to allocate shared data. Public will only be accessed when the predetermined APIs are invocated. Microservices are packaged and deplyed as containers running on the XXX system. Developers provide independent operating environments for each microservices like libraries. Microservices includes two pieces of memory, local memory and public memory.

Local memory is isolated from other components. There are microservice functions and other codes like dependent libraries allocating in local memory. They will provide functional interfaces to other microservices for data sharing and produce calls. Besides, the runtime data and private data are also allocated here. Developers are supposed to ensure memory safety for code and data in local memory, such as properly maintaining the stack and implementing fault handling when calling external APIs.

Public memory is shared with other containers through XXX kernel, which means other microservices may access this memory as well. There are only shared data allocating in public memory, which can be protected or public. Microservices can directly access their own public memory like reading and writing the data. The developers do not need to manage or maintain their public memory. They just needs to follow the specifications of capability protocols and XXX will maintain the data here.

XXX kernel is composed of capabilities-related components and common components. First, there are functional modules for capability namely capability protocols checker(CPC), capability exceptions handler(CEH) and capability remote proxy(CRP). The CPC will be performed when developers submit microservices for deployment. It will check the specifications in capability protocols and raise capability exceptions. The CEH is responsible for handling exceptions and is often triggered by CPC during deployment or system-call during runtime. For distributed shared data, CRP will be performed to manage and maintain them.

The capabilities heap and shared data heap are allocated here as well. Capabilities heap stores capability abstractions of all the microservices on this XXX kernel as their metadata. Shared data heap stores shared data whose owner microservices are failing, updating or remote. Besides, as a microkernel, XXX has memory management modules, network modules, file systems and system calls.

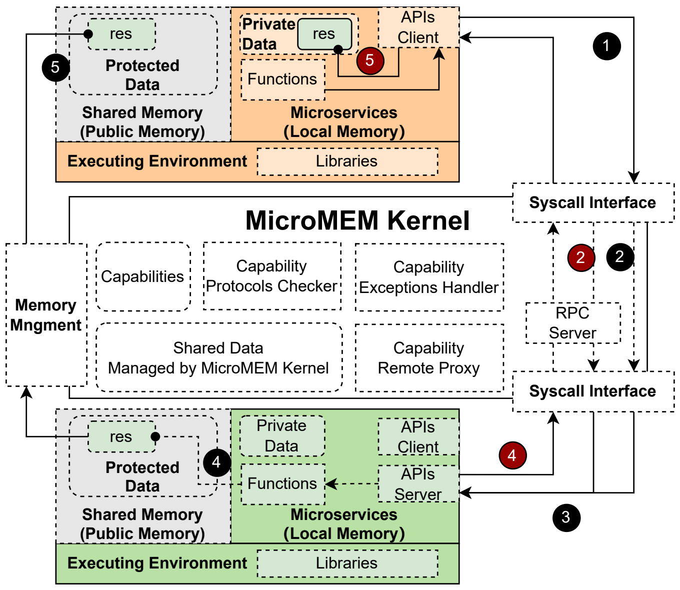


Figure 1: Trust model for memory safety.

3.2 Runtime Workflow

Figure 1 indicates the workflow for deployed microservices on the XXX system as well, describing the process of two microservices running on the same machine. In this example, microservice A calls the functions offered by microservice B to obtains certain return values as results. XXX supports both shared memory and rpc when microservices are calling each other. The shared memory is taken as default method as long as the capability protocols are followed.

The default workflow is shown as black nodes. Microservice A sends a request for microservice B to obtain shared data. First(❶), the APIs client will directly call system interface to inform XXX kernel of the invocation. Second(❷), then the kernel will check the records to see whether shared memory is enabled. Third(❸), the kernel will pass the request to the APIs server of microservice B. Next(❹), microservice B will accomplish the request and return to kernel. Final(❺), the result is allocated in public memory and kernel will remap the page to make the result accessible for microservice A. Microservice A will also receive the address of the result data. When microservice A finishes accessing the data, XXX will cancel the remapping and make the data inaccessible again.

When capability protocols are not followed, RPC will be enabled to support microservice functions. Red nodes describe the differences for this situation. First(❶), microservice A sends a request. Second(❷), XXX kernel will encapsulate the request into an RPC request and set up an RPC server for returning the result to microservice A. In special cases, when res is protected shared data, proxy will also take over the data copy in the kernel's shared data in step 5 and remap it to microservice 1. Third(❸), the RPC server will pass request to the APIs server of microservice B. Next(❹), microservice B accomplishes the request and copy the result back to the RPC server. Final(❺), microservice A receive the replica of result data in its local memory and the invocation is accomplished. Microservice A is supposed to deallocate the data and handle potential exceptions like type errors.

The details of the workflow, such as how capability-related components work, will be further elaborating in the next sections. Besides the workflows above, there will be exceptions which illustrate how XXX meets the requirements like data confidentiality and how XXX address the challenges such as data race.

3.3 Threat Model

Microservices run on cloud machines, which are assumed to be trusted because related works like TEE are considered to be able to protect microservices from cloud attacks. The microservices users, on the other hand, are regarded as malicious since attackers can pretend to be users and send requests to microservices. We assume that the attackers can take use of memory safety vulnerabilities to obtain private data, like SQL injection attacks and OpenSSL heartbleed [8]. For one microservice, other microservices are considered to be semi-trusted. On the one hand, they are not malicious to attack others or leak shared data by purpose. On the other hand, other developers cannot be assumed to be free from vulnerabilities, especially for microservices that often update to new versions dynamically. Consequently, it is necessary for microservices to additionally apply secure data sharing monitoring, which prevents shared data breach from other semi-trusted microservices and obtaining by attackers.

For data confidentiality, we segment the data into three distinct categories for microservice. **Private data.** Private data includes critical privacy and can not share with other microservices. **Protected data.** Protected data inevitably needs to be shared with other microservice. But they still carry secrets like user information and business data, and cannot be leaked to the outside attackers. The protected data is specific to microservices and requires additional monitoring. **Public data.** Public data available and accessible to anyone. It is not necessary to track its capability for security reasons.

For memory safety, the developer manages the local memory of its own microservice and XXX manages the shared memory for all the microservices. Therefore, for one microservice, others' local memory is considered as vulnerable and needs to be isolated, as shared memory protected by system capability of XXX is reliable. And for XXX, access from any microservices can be vulnerable but they do not provide fake or tampered capabilities.

4 XXX Design

Now we describe the design for the components of XXX kernel. XXX is a system that utilize capabilities to enable different microservices to benefit from the performance improvements offered by shared memory, while maintaining security for microservices such as memory safety, data confidentiality and isolation.

As illustrated in section 4.1, the **Capability Abstraction (CA)** is the key design of XXX, which describe the shape of capabilities, generates from source code with compilation information, serves as the metadata for a shared data and assist XXX kernel to maintian security. The **Capability Protocols** set up both static (described in section 4.1) and dynamic (described in section 4.2) specifications for capabilities. by following the capability protocols, XXX kernel can take advantages of compilation information on capabilities. When capability protocols are broken, there will arise **Capability Exceptions** and **Capability Handler** (illustrated in section 4.2) will deal with the exceptions, allowing microservices functions to continue. **Capability Proxy**, designed in section 4.3 will take over shared data and assist Capability Handler for certain exceptions like distribution or faults.

Additionally, to enable shared memory for microservices, there are very limited changes for microservices developers to adopt as section 4.4 indicates, including providing access control information for shared data, generating capabilities in compilation time and adopting XXX calls for invocations.

Preliminarily, we define it as src microservice when we refer to the arguments from caller microservices or the results from callee microservices. The user microservice is defined opposite to src microservice, referring to the other two cases. The src microservices who onws the shared data are defined as owner microservices.

4.1 Capability Abstraction and Static Protocols

This section describes what kind of compilation information of shared data is included in capability abstract, and what are the static protocols for them.

Capabilities are designed for each of the shared data and we generate capabilities from source codes in compilation time. And for non-shared data or executable codes, there is no capability for them. For source code, developers use variables to present the operation on data like allocating, accessing and sharing. Therefore, the shared data are defined as data relevant to variables in the predetermined APIs like arguments or return values (also denoted as results). There are four segments in the data structure of capability abstraction: ***Data, Lifetime, Permission*** and ***Accessibility***.

CA Segment	Content	Capability Protocols (Static)
Data	size, location, name, alias and owner	(**CPs1**): Convergence on Segment Data
Permission	{own, read-write, read-only}	(**CPs2.1**): Declaration following Demands (**CPs2.2**): Exclusive Ownership (**CPs2.3**): Descending inheritance
Lifetime	Inner lifetime& Outer lifetime	(**CPs3.1**): Safety on Inner Lifetime (**CPs3.2**): Convergence on Outer Lifetime
Accessibility	{Private, Protected, Public} exemption-list & white-list	(**CPs4.1**): No Private Data (**CPs4.2**): No Exp-Replicas for Protected Data (**CPs4.3**): No Unauthorized Accessing

Table 2: Brief summary for segments in Capability Abstraction and related specifications in static Capability Protocols.

Data, as a segment of CA, refers to the object item running in memory, including its size, location, name, alias and owner. There are three key points to elaborate the data segment, concerning owner, aliase and the data itself. First, for capabilities in user microservice, the owner can be omitted as unknown, which will be completed by XXX kernel before deployment. For capabilities in owner microservices, if the microservice owns the data, the owner should be marked as the microservice itself. If the microservice receives the data from another microservice, the owner can be either specified or omitted and completed by kernel. Second, The microservices will create new variables assigned with the same data, considering as aliases. For data segment in CA, all the alias of shared data should be included. Third, as introduced below in section 4.4, there will be exp-replicas and imp-replicas when sharing data. The former will be considered as another shared data with a new capability, while the later will be managed by Capability Proxy and share the same capability.

For static capability protocols, there is only one specification for data segment. (**CPs1**): Convergence on Segment Data. Meaning that the data segment is required to be analyzable, convergence on segment data demands no pointers or scattered data slices, which make it divergent to analyze all the alias.

Permission, as the second segment of CA, declares how one microservice is going to access the data. We designed three states for permission segment, named *own, read-write, read-only*. There are three elaborating points. First, the permission *own* includes *read-write* and all other permissions like allocation and release, while the permission *read-only* is the primary for accessing. Further design choices of permission and related protocols are discussed in section **??**. Second, for src microservices, permission segments demands how it should be accessed, while for user microservices, this declares how they will access. Third, there is unique permission for each alias of the shared data. For user microservices, the permission segment records the highest among all the aliases.

For static capability protocols, there are three specifications.

(**CPs2.1**): Declaration following Demands. The user microservices are not supposed to ask for permission higher than src microservices' regulations.
(**CPs2.2**): Exclusive Ownership. There should only be one microservice who declares to own the data.
(**CPs2.3**): Descending inheritance. When a user microservice passes the shared data to another as a src microservice, it can not demand permission higher than it used to declare.

Lifetime, as the third segment, indicates when one microservice accesses the shared data. First, for each alias of the shared data, its function scope implies its lifetime, which is the inner lifetime. The lifetime segment records outer lifetime, which belongs to either *limited* or *unlimited*. The former means lifetime for all aliases ends with invocation and the later means not.

For static capability protocols, there are three specifications.

(**CPs3.1**): Safety on Inner Lifetime. For aliases with permission of read-write, there is no intersections between its lifetime and others except owner.
(**CPs3.2**): Convergence on Outer Lifetime. For the user microservices, the lifetime cannot be unlimited.

Accessibility, as the last segment, describes authorities for shared data, including states, exemption-list and white-list. Accoring to threat model in section 3.3, there are three states for data confidentiality named *Private, Protected* and *Public* Owner microservices will set this segment and other microservices just needs to leave it omitted and take it as protected by fault. The two lists will be discussed later in section 4.4.

There are two specifications for accessibility.

(**CPs4.1**): No Protected Data. Protected data is not supposed to be shared with others.
(**CPs4.2**): No Exp-Replicas for Protected Data. Explicit replicas are considered as independent data with new capabilities and XXX cannot track data sharing across different capabilities.
(**CPs4.3**): No Unauthorized Accessing. When XXX detects possible dataflow of protected data to unauthorized microservices, it will be teminated to protect confidentiality.

4.2 Capability Exceptions and Dynamic Protocols

This section adds the description for dynamic capability protocols. When capability protocols are not followed, specific capability exceptions arise. To deal with the capability exceptions in a robust and reliable way, we introduce capability exception handlers.

Dynamic Capability Protocols, different from static protocols, cannot be examined during compilation time or deployment time. They might take place during runtime, and can never occur. While prohibiting all such possibilities in advance can lead to severe damage on system ability and performance, their possible ppearances also cause unacceptable problems like failure or incorrect results. To address such challenge, dynamical capability protocols are designed with a moderate overhead to ensure the robustness and correctness of microservices functions. Therefore, dynamic capability protocols always describe solutions for unexpected exceptions and result in handler, while static capability protocols usually introduce compulsory specifications for developers to follow.

(CPd1): Exclusive Modification. Similar to **(CPs3.1)**, this specification prevent intersections between a read-only microservice and other microservices with higher permissions. Specially, the owner microservices are except because there is another specification for him.

(CPd2): Limited Owner Access. Owner microservices are not allowed to write or release the shared data when the lifetimes of other microservices do not end.

(CPd3): Host-to-host for Distribution. When sharing data with microservices deployed on remote machines, exception is supposed to arise for XXX to start host-to-host communication to load shared data.

(CPd4): No Unauthorized Accessing. Unauthorized accessing can take place dynamically as well.

For most of the cases, it does not limit the microservices functions for developers to follow the static and dynamic specifications of capability protocols. Then XXX is going to enable shared memory for efficiency in a secure manner that meet all requirements for microservices. However, for cases like shared memory of small sizes or unpredictable data race, microservices developers are allowed to break the capability protocols with the assistance of capability exceptions handler, which provides security methods with overhead not heavier than existing RPC.

To provide robust and reliable solutions, the capability exceptions handler **(CEH)** is enabled with below functions:

Enable RPC for Exp-Replicas *(eRPC-E)*. The CEH can switch the workflow into the RPC one as shown in section 3.2.

Perform Interceptions *(pI)*. The CEH runs inside XXX kernel and can ignore invalid and unsafet operations.

Teminate Deployment *(tD)*. The CEH functions when developers submit microservices to deploy, and XXX can teminate invalid submissions.

Report Errors *(rE)*. The CEH is able to send errors reports to developers on behalf of XXX kernel.

Call Capability Proxy *(cCP-*)*. The CEH will call Capability Proxy, another capabilities-related component in XXX kernel, for assistance including manage imp-replicas **(cCP-I)**, start remote loading **(cCP-RL)**, and handle data **(cCP-hD)**.

As shown in table 3, the capability exceptions handler conducts to handle the violations from the exceptions of each capability protocols.

Exception	Exceptions Handler	Description for Violation
CPs1	eRPC-E	Capabilities are incomplete for SHM*.
CPs2.1	eRPC-E	Permissions do not match.
CPs2.2	eRPC-E or pI	Ownership is not clear.
CPs2.3	tD and rE	Capabilities is to be damaged.
CPs3.1	eRPC-E	Isolate vulnerabilities inside MS*.
CPs3.2	eRPC-E or cCP-I	Avoid data race between MSs*.
CPs4.1		Follow threat model.
CPs4.2	tD and rE	Ensure tracking of capabilities.
CPs4.3		Protect confidentiality for shared data.
CPd1:	cCP-I	Take imp-replicas to avoid data race.
CPd2:	cCP-hD	Take over shared data to avoid failure.
CPd3:	cCP-RL	Take remote communication for distribution.
CPd4:	pI and rE	Protect confidentiality dynamically.

Table 3: Brief summary for violations on Capability Protocols and relevant Capability Exceptions Handler. SHM refers to shared memory and MS refers to microservices.

Exceptions on *CPs1*: Enable RPC for Exp-Replicas. When the data segment is not analyzable, the capability is considered to be too weak for XXX, for instance, developers, compilers and other static analysis tools cannot guarantees that there is no unknown aliases. For public shared data, if such expection arises in src microservices, the CEH will enable RPC for all related microservices; if only arises in certain user microservice, the CEH will just enable RPC for that one.

Exceptions on *CPs2.1*: Enable RPC for Exp-Replicas. When user microservices demand higher permissions than owner microservices allow, the user microservices show potential risks against data consistance. The CEH will enable RPC for the user microservices and they will access the exp-replicas of the data.

Exceptions on *CPs2.2*: Enable RPC for Exp-Replicas, or Perform Interceptions. Only the owner microservice is supposed to declare the ownership of shared data. When user microservices declare ownership just for reading, the CEH will set flags and ignore their system call on releasing the data; when they acquire ownership for writing, the CEH will consider the extra calls introduced in section 4.4. If it does not belong to extra calls and the shared data is public, the CEH will enable RPC for user microservices. If the shared data is protected, the CEH will report errors and teminate when deploying.

Exceptions on *CPs2.3*: Teminate Deployment and Report Errors. Microservices are not supposed to generate data capability with permissions higher than what they have.

Exceptions on *CPs3.1*: Enable RPC for Exp-Replicas. XXX do not allow microservices with memroy safety risks within themselves to use shared memory. The CEH will enable RPC for them to access explicit data replicas.

Exceptions on *CPs3.2*: Enable RPC for Exp-Replicas, or Call Capability Proxy for Imp-Replicas. For cases that microservices pass shared data to another, the outer lifetime will be difficult to determine when deploying them. If the intersections between microservices with writing permissions and others are inevitable, the CEH will enable RPC for them. Otherwise, possible risks are allowed since ***CPd1*** takes responsibility for them.

Exceptions on *CPs4.1*: Teminate Deployment and Report Errors. Private data is not expected to share with others.

Exceptions on *CPs4.2*: Teminate Deployment and Report Errors. Generating exp-replicas lead to evasion on tracking capabilities, disabling XXX to guarantee data confidentiality. The CEH will teminate Deployment and reports errors to developers with detailed logs. To pass the check, the user microservices can take imp-replicas or the src microservices can update its exemption list.

Exceptions on *CPs4.3*: Teminate Deployment and Report Errors. When capabilities indicate that the dataflow of protected data includes unauthorized microservices, the CEH teminates Deployment and reports errors to developers. To pass the check, developers of owner microservices are reminded to provide desensitized data in addition. Besides, developers can also update the exemption lists. When unauthorized accessing is not determined, the CEH allows possible risks for another checking from ***CPd4***.

Exceptions on *CPd1*: Call Capability Proxy for Imp-Replicas. When such exceptions arise, data race is probably to take place. As introduced later in section 4.4, XXX prepares imp-replicas for modifiable data. Then the CEH will call Capability Proxy to provide imp-replicas for replacement.

Exceptions on *CPd2*: Call Capability Proxy to Handle Data. The arising of such expections implies the faults, dynamic updates, scalings or migratings are just underway for the owner microservices. To handle the impact, the CEH will call Capability Proxy to take over the shared data temporarily. Capability Proxy will serve as the owner microservices until all the user microservices ends up their lifetime as introduced later in section 4.3.

Exceptions on *CPd3*: Call Capability Proxy for Remote Loading. For shared data between microservices deployed in different machines, the CEH will response the exceptions and handler it with the help of capability proxy, who communicates with the capability proxy on the other machine to load the imp-replicas, enabling the user microservices for accessing shared data.

Exceptions on *CPd4*: Perform Interceptions and Report Errors. To protect confidentiality on protected shared data, the CEH will raise exceptions for the user microservices who invoke predetermined APIs to access unauthorized. Then the developers will receive reports to update their codes.

4.3 Capability Proxy

In this chapter we will introduce the design of capability proxy, a XXX component to manage shared data including allocating and releasing implicit replicas, handling and releasing the shared data whose owner is crashes, and conduct host-to-host communication with other capability proxies on distributed machines. Capability proxy will be called either by capability exceptions handler or by requests from other proxies. There is a heap in kernel prepared for the capability proxy to store and manage replicas.

4.3.1 Lifecycle of Implicit Replicas

The capability proxy maintains a queue for all the implicit replicas of one shared data, where there is always one ready replica, one writable replica and one readable replica. Capability proxy manages imp-replicas in a pipeline manner to provide a high-availability and high-reliability manner to deal with capability exceptions as figure 2 shows. At the same time, it will be consistent with asynchronous communication like RPC, which will be discussed later in section **??**.

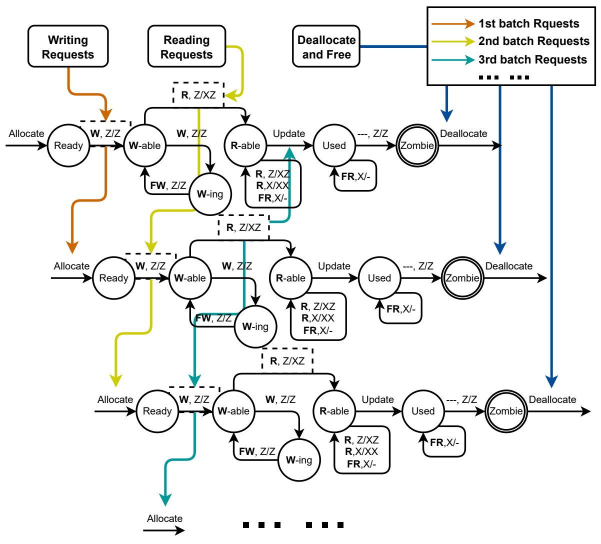


Figure 2: The lifecycle for an implicit replica. **W** in the figure means writing, **R** means reading and **F** refers to finish. **Z** serves as the initial bottom element of the pushdown automaton's stack by default.

For a modifiable shared data, the capability proxy will provide imp-replicas and mark as ready. Most of the cases, user microservices just access the shared data in order, since specifications of capability protocols efficiently prevent data race in advance. However, when exceptions on **CPs3.2** and **CPd1** arise, the capability proxy should allocate new imp-prelicas. Actually, as introduced in section 4.3.3, the proxy will merge some replicas for efficiency with the same function.

There are six states for an implicit replica, including **Ready, Writable, Writing, Readable, Used, Zombie**. Theoretically, the lifecycle of an imp-replica follows:

Firstly, when the ready replica is turned as writable replica, a new replica is copied and allocated as ready. If microservices continue to write the data, the new replica will be waiting as well.

Secondly, when one microservice requests to read the data, present writable replica is turened into readable replica, the ready replica is turened as writable data and the current readable replica is used up. This replica will remain writable until one microservice sends writing requests.

Thirdly, when new writing requests come, another new replica will be allocated copied from this replica. Then it turns to be writing and can not be accessing, while capability protocols support exclusive writing and there is another replica for reading. When the microservice finishes writing, this replica turns back to writable state.

Fourthly, when new reading requests come, this replica grows to be readable if it is writable now. The ready replica will serve as new writable replica and an update signal will be sent to the former readable replica. This replica will serve for the coming reading requests. There is a pushdown automaton stack (which is actually the lifetime in capability abstraction for user microservices).

Fifthly, when new writing requests come and the writable replica grows as readable replica, this repilca is approach the end of its lifecycle, growing up to be a used replica. The capability proxy will keep it until the last user microservices finish its lifetime.

4.3.2 How Capability Proxy Manages Imp-Replicas

To summarize the lifecycle of imp-replicas, there are three distinctive features. First, a replica grows from ready, writable, readable to used, which is irreversible. Second, only when the new request is different from the last one in writing and reading, will the new replicas be generated. Third, there will be always one replica of ready, writable and readable states.

However, experiments proved that it is not necessary to always keep a standardized lifecycle. As a matter of fact, the capability proxy usually takes two replicas in turn for efficiency and performance, because when there is no intersections between reading requests and writing requests, there will not be capability exceptions and capability proxy does not have to be called to perform imp-Replicas. As a result, sometimes the writable replica and readable replica is the same, with another ready replica waiting for capability exceptions from handler, making operation on imp-replicas a trival overhead.

4.3.3 Fuctions of Capability Proxy

Capability proxy will be called by the capability exceptions handler. Usually, the handler accquire for imp-replicas to deal with data race. Additionally, when the owner microservices of shared data are crashed, the proxy is called to manage the shared data temporarily to prevent failure cascading. The proxy will treat such kind of data like an imp-replica of the used state, keeping it until all the user microservices ends up their lifetime.

Capability proxy will also be called due to distribution issues. Then the proxy will generate an imp-replica through host-to-host communications with distributed machines, copying the capabilities and data. The XXX rejects writing through shared memory for distributed cases as discussed in section **??**. Therefore, the capability proxy manages those imp-replicas as readable state. When modification take place on the machine of the owner microservices, the proxy allocate a new imp-replica as readable and turns the current readable as used, waiting for the user microservices on this machine finish their lifetime to deallocate and free the zombie replicas.

4.4 Adopting to XXX

XXX is design with great compatibility for existing microservices developed on RPC. There are only minor adjustments for developers to adopt their microservices on XXX.

First, update the invocation method. Developers used to implement the predetermined APIs as RPC servers or RPC clients. For RPC servers, they will additionally implement the interface functions. For XXX, the predetermined APIs are implemented as system call, developers can just invoke by them and provide the kernel with only entry point of interface functions if they are APIs servers.

Second, clarify the shared data. For the arguments and return values in APIs, the developers are supposed to clarify them as hard copy or soft copy. For hard copy, XXX will provide explicit replicas(**exp-replicas**) which can not benefit from shared memory. Soft copy will be treated as implicit replicas(**imp-replicas**) and be allocated in shared memory. For soft copy, developers are supposed to additionally clarify their permissions as own, write-read or read-only. If the microservice is supposed to make local modification on the data, it should not be clarified as soft copy.

Third, submit the capabilities. There will be capability analyzer to utilize compiler and static analysis to generate most of the information. Developers are mainly supposed to add information on accessibility, which is related to confidentiality. Private data is not allowed to share and public data is not to be tracked. For protected data, if some microservices are trusted to hold exp-replicas, developers are supposed to add them to exemption list; if some microservices are trusted to have desensitized data or and to leakage data, developers are supposed to add them to white list; otherwise, developers are supposed to provide desensitized data and specific APIs.

With the minor adjustments above, developers can not only deploy their microservices on XXX to additionally benefit from performance improvements of shared memory, but also gain bonus of functionality.

Directly modify shared data. For current microservices based on RPC, a microservice is not able to modify the shared data from another one, while the owner microservices may provide APIs for user microservices to take such operations indirectly. With shared memory, XXX enables user microservices to modify the shared data directly. However, the microservices have to write exclusively and only shared data on the same machine is supported, in order to maintain a simple memory model. Complicated writing will be the future work for XXX.

Move shared data across microservices. With the capability, one microservice can move its shared data to another microservice rather than just hrad copy or soft copy. XXX will update the capabilities dynamically each time microservices move shared data. The two functionalities mentioned above are named as extra calls since they are not supported by RPC.

5 Evaluation

We have implemented a prototype core of XXX, performing basic functions designed for the capabilities-related kernel components like capability protocols checker, exception handlers and remote proxy. For evaluation, we prepared DeathStarBench [11], a widely recognized benchmark for microservices with independent functions based on RPC. Besides, we implemented our MiniBench with ∼1200 LoCs in total for breakdown performance and scalability, since we are supposed to illustrate how shared memory systems provide significant improvement for microservices performance and efficiency.

In this section, we present our evaluation results by testing XXX's security and performance. Specifically, our evaluation will answer the following questions:

- **Q1:** Can shared memory systems significantly provide microservices with good performance? How?
- **Q2:** How efficient is XXX compared to classical RPC as baseline?
- **Q3:** How secure is XXX compared to existing shared memory baseline and RPC baseline?
- **Q4:** How well does XXX maintain its performance as the workload increases?

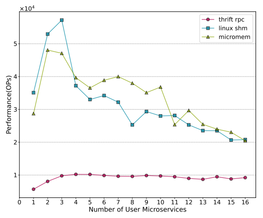
5.1 Evaluation Environment

DeathStarBench [11] is consisted of six classical microservices such as hotelReservation and socialNetwork. As a benchmark, DeathStarBench mainly focuses on the communication overhead based on RPC with minimal concerns on execution overhead, making it more suitable for related works on microservices schedule and load balancers [5, 7, 12, 16]. Therefore, we adopted DeathStarBench with ∼300 LoCs to enable it with shared memory and XXX.

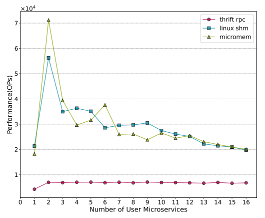
Besides, we implemented our ∼1200 LoCs MiniBench to explore the breakdown overhead within RPC and the scalability for XXX and RPC while the size of data increased and execution overhead increased. This MiniBench comprises a set of simple microservices applications based on two primary methods of data sharing, ❶ user microservices request shared data by calling owner microservice (REQ), and ❷ user microservices are called and receive shared data by broadcasting from owner microservices (BC).

In our MiniBench, microservices communicate with each other on both RPC (using Thrift [19]) and shared memory (using XXX). For sharing data by broadcasting method, to make it fair for RPC, we additionally supplied an **request window** to enable **asynchronous operations**, which significantly improve the performance for RPC.

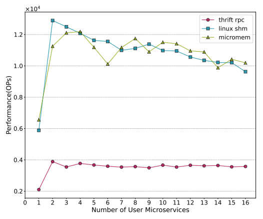
5.2 End-to-end Performance



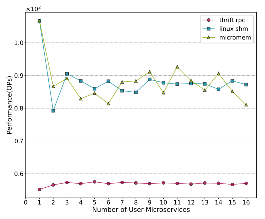
(a) *Data size: 128B*



(b) *Data size: 1KB*



(c) *Data size: 10KB*



(d) *Data size: 1MB*

Figure 3

To answer **Q1**, we first evaluated the end-to-end performance of XXX, Thrift RPC, and shared memory in both **REQ** and BC of our MiniBench. As shown in Figure 3 and Figure 4, the shared memory achieved the highest throughput because MiniBench provided with special cases where there is no data race and mutexes. XXX outperformed RPC in throughput by 2.1∼4.7 times because it enabled shared memory for microservices to access shared data directly.

To answer **Q2** and **Q4**, we collected the throughput of XXX with different numbers of user microservices for evaluating the scalability for XXX and baselines. As the number of user microservices increased from 1 to 16, for **REQ**, throughputs on shared memory and XXX are decreasing gradually by 55% compared to peak and 32% compared to average. For RPC, the throughputs are slightly increasing with more user microservices due to asynchronous operations' contribution. However, even with 16 user microservices, the throughputs of XXX is still 1.1 times better than RPC.

On the contrary, as shown in figure Figure 4, throughputs on shared memory and XXX for **BC** were increasing as more user microservices accessed the shared data. This was because the numbers of shared memory users did not increase the cost like copying shared data and serializing them.

We also collected the throughput of XXX with different data size. As shown in Figures 4(a), 4(b), 4(c) and 4(d), XXX's performance improvement compared to RPC was more and more obvious while the data size was larger. The four data volumes collected were typically presenting the data size for microservices scenes including command requests (128B), information exchanges (1KB, e.g. request user information), website data post (10KB) and streaming data (1MB).

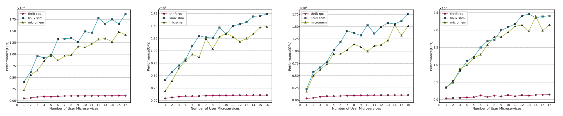

(a) *Data size: 128B* (b) *Data size: 1KB* (c) *Data size: 10KB* (d) *Data size: 1MB*

Figure 4

To explore the scalability and elaborate **Q2** , we conducted experiments to evaluate the breakdown of RPC. As figure 5 shows, the majority of overhead on RPC was caused by copying shared data especially when the data size was small, which was common across microservices. As the data size increased, the processing costs became the key overhead. XXX and shared memory allowed user microservices to accessing data at the same time, improving overall parallelism for better performance. Those accounted for the scalability resulted above.

5.3 Security Analysis

To answer **Q3**, we conducted this experiment to verify the security improvements of XXX compared to other shared memory systems and RPC, including memory safety of shared memory, confidentiality for data in shared memory and fault cascading through shared memory.

E1: Memory Safety of Shared Memory. According to the threat model defined in section 3.3, we generatedd three different types of memory-related errors for memory safety

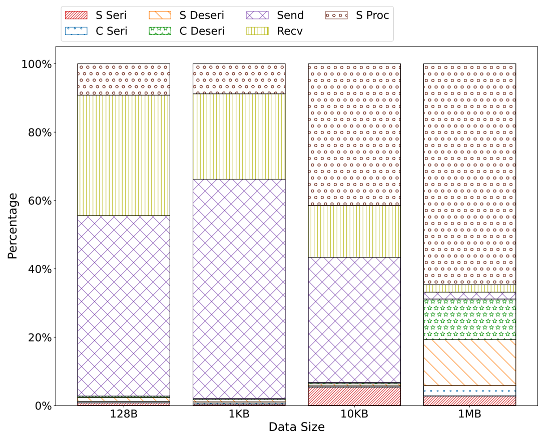
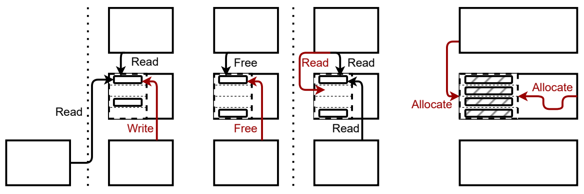


Figure 5: **Performance of BreakDown.** We broke RPC with seven steps, client serialization costs, client sending costs, server deserialization costs, server processing costs, server serialization costs, client receiving costs, and client deserialization costs.

as simulated possible vulnerabilities *(V1-V3)* for accessing shared data.



(a) *Data Race* (b) *Segmentation Fault* (c) *Memory Leakage*

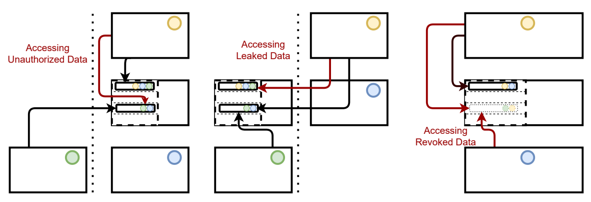
Figure 6: **Vulnerabilities on *E1*.** The red lines highlight the key operations to raise the vulnerabilities. Rectangles in the same column represent microservices deployed on the same machine. The rounded rectangles in the dotted boxes represent shared data. Data filled with hatches indicate no longer used.

V1: Data Race. This refers to a type of vulnerablity that occurs when multiple microservices concurrently access and modify the same shared data. To generate such vulnerablity, we started from an owner microservice who shared one data with another three users. Two of the users were deployed on the same machine as the owner and the other one is deployed remotely. Then we let three users repeat accessing and the data race raise.

V2: Segmentation Fault. This refers to vulnerabilities that lead to segmentation faults and crashes. Since microservices do not allow pointers in predetermined APIs, we generated such vulnerabilities by two common non-pointer-related memory violations, double-free and buffer overflow.

V3: Memory Leakage. This vulnerabilitiy happens when microservices fails to release memory it no longer needs, causing a gradual depletion of available resources for shared memory. We generated such vulnerabilities starting from an owner microservices who shared one data with another two user microservices. The two user microservices continuted to allocated new data in shared memory and did not release shared data since they do not own them.

E2: Confidentiality for Shared Data. Besides, we generated unintended data access as confidentiality vulnerabilities*(V4-V6)* for data in shared memory.



(a) *Unauthorized Data* (b) *Leaked Data* (c) *Revoked Data*

Figure 7: **Vulnerabilities on *E2*.** The meaning of the elements is the same as figure 6. The colored circles in the shared data represent its authorization to user microservices.

V4: Accessing Unauthorized Shared Data. This vulnerabilitiy raises when fine-grained access control is lacking. To generate such vulnerablity, we started from an owner microservice who shares data with three user microservices. Three user microservices were authorized with different permissions as picture 7(a) shown. When one of them was accessing unauthorized data, this vulnerabilitiy raise.

V5: Accessing Leaked Shared Data. This vulnerabilitiy raise when microservices pass shared data to another. We generated such vulnerablity by focusing one microservice who passed shared data to another unauthorized microservices as picture shown.

V6: Accessing Revoked Shared Data. This vulnerabilitiy raise when microservice failed to revoke. We generated such vulnerabilitiy by letting user microservices repeat accessing after the owner's revocation.

E3: Isolation across Shared Memory. Additionally, we generated one vulnerabilitiy for evaluation of isolation*(V7)*. We started from an onwer microservice who shares data to another microservice. When the other was reading, the owner microservice was terminated to test isolation.

After the generation, we ran the microservices to send requests related to data sharing. To record the experiment result, we monitored the vulnerabilities by checking the logs of user microservices and recording the crashes. For *V1* and *V2*, if the user microservices got invalid data or crashes occurred, it indicated that the system failed to protect memory safety; for *V3*, we made sure that the resources for shared memory was adquate and if systems failed to allocate memory, it failed; for *V4*, *V5* and *V6*, if the logs proved that unauthorized user microservices did got the data, it failed; for *V7*, it succeeded only when the crashes did not occur after the termination of the owner microservice. Shown as table 4, XXX can successfully prevents memory-related errors and protect data confidentiality while shared memory baseline failed for some vulnerabilities.

System		XXX		THRIFT PC		BOOSTSHM	
		✓	✗	✓	✗	✓	✗
V1	# of Errors	**100%**		**100%**		**100%**	
	was Prevented	21	0	21	0	21	0
V2	# of Errors	**100%**		**82.5%**		**45%**	
	was Prevented	40	0	33	7	18	22
V3	# of Errors	**100%**		**N/A**		**21.4%**	
	was Prevented	14	0	N/A	N/A	3	11
V4	# of Breaches	**100%**		**100%**		**30.8%**	
	was Prevented	13	0	13	0	4	9
V5	# of Breaches	**100%**		**23.1%**		**0%**	
	was Prevented	11	0	3	10	0	13
V6	# of Breaches	**100%**		**100%**		**13.3%**	
	was Prevented	15	0	15	0	2	15
V7	# of Tests	**100%**		**100%**		**20%**	
	was Isolated	5	0	5	0	1	4

Table 4: Security evaluation of XXX and baselines.

6 Conclusion

We have presented XXX, the first shared memory system work that support service independence, the design principle of microservices, to provide microservices with significant efficiency. It includes two fundamental design techniques: (1) a system capability abstraction specially designed for shared data of microservices, which restructuring shared memory guarantees between development analysis and runtime supporting, and (2) a collections of capabilities-related kernel components that allow capabilities to function as kernel metadata to protect memory safety and confidentiality for distributed scenes with faults. We have implemented a prototype of core for XXX with two benchmarks, showing that how shared memory improve microservices efficiency and how XXX meet all the requirements for microservices with a moderate overhead.

References

[1] Marcelo Abranches, Sepideh Goodarzy, Maziyar Nazari, Shivakant Mishra, and Eric Keller. Shimmy: Shared memory channels for high performance {Inter-Container} communication. In *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[2] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, 2018.

[3] Godmar Back, Wilson H Hsieh, and Jay Lepreau. Processes in {KaffeOS}: Isolation, resource management, and sharing in java. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, 2000.

[4] Netflix Technology Blog. Netflix conductor: A microservices orchestrator, December 2016.

[5] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[6] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[7] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.

[8] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[9] ethz. Boost.interprocess.

[10] Anandprasanna Gaitonde and Mohit Malik. Using api gateway as a single entry point for web applications and api microservices, oct 2019.

[11] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[12] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 152–166, 2021.

[13] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. Flexos: Towards flexible os isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 467–482, New York, NY, USA, 2022. Association for Computing Machinery.

[14] Hamid Reza Mohebbi, Omid Kashefi, and Mohsen Sharifi. Zivm: A zero-copy inter-vm communication mechanism for cloud computing. *Computer and Information Science*, 4(6):18, 2011.

[15] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39. USENIX Association, November 2020.

[16] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association, November 2020.

[17] Vasily A. Sartakov, Lluís Vilanova, David Eyers, Takahiro Shinagawa, and Peter Pietzuch. CAP-VMs: Capability-Based isolation and sharing in the cloud. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 597–612, Carlsbad, CA, July 2022. USENIX Association.

[18] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. Cubicleos: A library os with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 546–558, New York, NY, USA, 2021. Association for Computing Machinery.

[19] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift : Scalable cross-language services implementation.

[20] Thorsten Von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-kernel: A capability-based operating system for java. *Secure Internet programming: security issues for mobile and distributed objects*, pages 369–393, 1999.

[21] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis,

Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. *ACM SIGARCH Computer Architecture News*, 42(3):457–468, 2014.

[22] Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E. Carlson, and Prateek Saxena. Capstone: A capability-based foundation for trustless secure memory access. In *32st USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, August 2023. USENIX Association.