ASPLOS'24 Submission #XXX– Confidential Draft – Do Not Distribute!!

# Instructions for Submission to ASPLOS 2024

## Abstract

## 1. Background

### 1.1. Rust programming language

The Rust programming language combines the powerful expressiveness of high-level language with the excellent performance of C language and provides memory safety guarantees exceeding that of C language. Unlike other languages that rely on runtime overhead, Rust achieves this only through static analysis by the compiler, which means that its advantages lie in performance, safety, and expressiveness, but it will also encounter limitations where static analysis is not feasible.

The Rust memory model based on ownership mechanism and compilation analysis such as lifetime serves as the essential reason for this achievement. The ownership mechanism provides the permission relationship between variables and memory data, helping the compiler to implement reasonable access control in memory data according to the behavior of variables in the code.

In the Rust ownership mechanism, the relationship between a variable in the code and an object in memory can only be one of four types: ownership, read-only reference, variable reference, and irrelevant. At each moment, every object must have a valid owner, at most one valid mutable reference and several references. After the object owner rewrites the object, all references are invalid; after an object's mutable reference rewrites the object, all read-only references are invalid. Code that does not comply with these principles will fail to compile, in order to achieve the guarantee of the ownership mechanism in Rust.

Rust ownership not only ensures data security but also avoids data access conflicts that may arise in parallel situations. At the same time, the compiler eliminates the possibility of memory security leaks by taking over all pointer operations and strict boundary checks and initialization checks on this basis. Based on static syntax analysis, Rust can guarantee these characteristics without runtime overhead and achieve high efficiency.

The Lifetime mechanism appears during the syntax analysis of the compiler. Through static variable analysis, the compiler can determine the lifetime of obj, and then release the memory space occupied by variables whose life cycle has ended or provide support for ownership judgment.

The ownership mechanism and lifetime analysis provide important support for the Rust memory model. This article will also implement dynamic access control in the microservice scenario based on the analysis results of ownership and lifetime

### 1.2. Trusted Execution Environment

Trusted Execution Environment is an important calculation of modern cloud computing. In order to solve the problem of host operator system distrust and help users resist attacks from privileged states, hardware manufacturers provide TEE to ensure the confidentiality and integrity of specific processes.

TEE such as Intel SGX [**?**] can allow users to run the process that needs to be protected in a hardware space (called enclave) that the operator system cannot access. Once the initialization work is completed, the hardware will ensure that any external operator system and privileged users cannot access or rewrite the internal process. Different protected processes will allow in different compartments of the TEE that they cannot access or rewrite each other's data and instructions. TEE provides a strong hardware isolation guarantee for the protected program, but it cannot solve the crash of the enclave's internal process due to its own memory security. At the same time, a high level of isolation across enclaves also means high-cost communication across enclaves.

### 1.3. Capabilities-based system

capabilities is a design philosophy for access control. It is like a token with its own address and access list. In capabilities-based system, users' access to objects is realized based on capabilities. On the one hand, capabilities are the only metadata that can determine the address; on the other hand, capabilities describe what different methods the object has, and for each method, which users have the right to use it.

Generally speaking, capabilities are related to the characteristics of the object itself. For example, for a piece of memory, its capabilities include reading, writing, and execution, and for a piece of pipeline, its capabilities include send, receive, and flash. In addition, capabilities include some methods common to any type of object, such as transferring capabilities, withdrawing capabilities, generating a subset of capabilities and assigning them to other users.

## 2. Overview

In the microservice architecture, we make the following assumptions as our trust model:

i Cloud operating systems are malicious. It may access user memory, compromise sensitive data confidentiality, or even compromise the integrity of data and execution.

ii The hardware and its drivers are trusted such as GPU and TPU. Inside CPU, nevertheless, only the TEE is reliable since the host operator system is untrusted.

iii The network is secure and reliable. But other users on the network cannot be trusted. Other microservices may have vulnerabilities, which will increase the risk of data breaches. There may be malicious requests on the network that attempt to access data illegally.

iv The runtime with a small tcb designed by the article is trusted. That is to say, we guarantee that the runtime code is safe and reliable. In practice this is reasonable when the runtime is lightweight and accomplished in a safe language.

vThe compilation process is trusted. The source code is also trusted, but not available. This means that we assume that the process of getting an executable from source code is complete, and that the compiler itself is reliable.

### 2.1. Workflow

This section introduces the implementation of the microservice architecture with respect to two parts, the development period and the runtime period.

During the development period, the source code of developers is compiled into an executable file with capabilities lists. There are three steps in the development period: precompile, analyze capabilities, divide and recompile. In the first step, the compiler compiles the rust program, check whether the program conforms to the Rust language specification and analyze the life cycle and ownership of each variable. After ensuring that the program itself is legal, the capabilities analyzer in the second step will generate the capability lists of the application according to the life cycle of the variables in the first step and rust ownership and obtain a partition scheme. In the third step, the program will be equivalently divided according to the partition scheme in the previous step, and each partition part will be compiled to obtain a runtime executable file, and a corresponding capabilities file will be generated for each divided executable file according to the capabilities lists in the previous step.

After the development period, the source code will be compiled and analyzed to obtain executable files and accompanying capabilities. The executable file contains data and instructions and can be run directly on the runtime. And each executable file comes with a capabilities file, which marks their access control list and relative address.

At runtime, the VM will run the microservice according to the executable and the accompanying capabilities file. One or more vm will run on each machine, several microservice programs may run on each vm and each microservice program will run on one vm. When loading, the vm will check whether the new program can run without conflicts with existing programs according to the capabilities file, and check the dependencies it needs. At runtime, the vm will ensure that multiple microservices run on the same piece of address space without conflict. When two microservices need to communicate with each other, they do so by reference rather than by copy.

For the case of multiple machines, when two microservices need to communicate, they still do not directly use replication. Specifically, a microservice program will copy its data to the reserved memory of the other party's vm, and the other party's microservice will access the data by reference under the supervision of the other party's VMs. Such a situation will not bring additional burdens for issues such as data consistency. Specifically, for the problem of read failure, the capabilities obtained based on rust lifetime will avoid this situation; for the problem of read-write synchronization, this situation is equivalent to two processes communicating directly through copying.

Such a workflow provides standardized access control, which will improve the security of private data. Meanwhile, this method will not bring too much performance loss, in most cases, on the contrary, it will also bring performance improvement.

## 3. Protocol Description

### 3.1. Design Insight

### 3.2. Capability Analyzer

Capability is the core of to realize many functions. It provides important information to the runtime according to ownership and lifetime in rust programming language, and is used to realize microservice security. The capability analyzer is an important way to obtain capabilities. It runs at compile time and is the beginning of workflow. At the same time, the correctness and rationality of its analysis are also related to the safety and performance of runtime. This chapter will rigorously describe three parts in a formal language, the abstraction of code space and address space, the abstraction of rust concepts, and the implementation process of the capability analyzer.

First, according to the workflow, the capability analyzer needs to cooperate with the compiler to divide the source code into several independent loadable files, and generate a corresponding capability list for each loadable file. Since variables live statically in code and objects live dynamically in memory, or more specifically, in the process heap, we need to define a formal language to describe them.

We define the variable space, denoted by the symbol V. V represents a collection of all variables in a code segment. Correspondingly, for the objects in the runtime memory, we use the symbol O to represent the object space, which represents a collection composed of objects. Next, we can define the ownership relationship r of the elements in the V space to the elements in the O space in the rust programming language. According to rust syntax, r can be classified into four types: v owns o, v is a mutable reference of o, v is a read-only borrow of o, and v has nothing to do with o. We use the symbols r3, r2, r1, r0 to represent these four relations respectively. We do not need to consider the irrelevant case ie r0.

Rust syntax places constraints on the ownership relationship r. At the same time, for any object o belonging to O, there is a variable v belonging to V, so that there is one and only one of the three cases of v r1/r2/r3 o. At the same time, for any object o belonging to O, there is one and only one variable v belonging to V such that v r3 o. This variable v is also known as the owner of the object o. At the same time, for any variable v belonging to V, if there is an object o1 such that v r o1 exists, then there must not be an object o2 belonging to O and o2 is not equal to o1 such that v r o. That is to say, any variable v has one and only one object o associated with it.

In addition to ownership, rust syntax also has related constraints on the life cycle. For any object o belonging to O, its creation time is t1, and its recovery time is t2, then the closed interval [t1, t2] is defined as the life cycle Lt of object o. Here, creation is defined as the time when the compiler allocates the stack for the object, and recycling is defined as the time when the compiler reclaims the stack space. Both of them are determined by the real time at runtime and conform to the total order relationship. However, for the capability analyzer, the life cycle of static variables is more important.

For any variable v belongs to V, we define the time when v is initialized as t0, and record the time of each use as ti. Here we use the sequence [t0,t1,t2,...,tn] to describe the lifetime of variable v. Different from the life cycle of an object, in the declaration cycle of a variable, each moment does not have a total order relationship. Even if we assume that the program is sequential (it will be explained later that in the rust programming language, even in the case of non-sequential serialization, the nature to be demonstrated later still exists), due to the existence of branches, loops, etc., the time of variables is still only There can be a partial order relationship. We assume that for the object o belongs to O, the variable space has and only v r3 o, and the Lt of v=[t0,t1,...,tn], then the Lt of o can be inferred by [t0,tn].

Under Rust's constraints on ownership, variables with read-only references and writable references have strictly limited lifetimes. For any object o belonging to O, there exists a variable v0 belonging to V such that v0 r3 o. At the same time, there are several variables vi belonging to V such that v1 r2 o is XORed with v1 r1 o. Record V' as the set of these v, the active period of vi is Lti=[ti1,ti2,...,tin], where i=0,1,...,n-1, |V'|= n. Record interval [ti1,tin] as Lti. Then first, if vi, vj r2 o, then there is Lti intersection Ltj=empty set. Second, if vi r1 o, then any vj r2 o, there is Lti intersection [tjk,tjk+1]=empty set XOR Lti contained in [tjk,tjk+1] holds true for any meaningful k. Third, any vi i is not equal to 0, there is Lt0 intersection [tik,tik+1]=empty set XOR Lt0 belongs to [tik,tik+1] is established for any meaningful k. With the above definitions and constraints, we can start to define capability. Before that, we need to

Define user and object blocks. User space is defined as a division of the variable space V, denoted by the symbol U. The division here is defined as for U=u1,u2,...,un, where any ui intersection uj=empty set (i is not equal to j), generalized and ui=V (i=1,2,... ,n), obviously ui is a subset of V. Similarly, considering the set Oi=o|there is a v belonging to ui such that v r3 o, we define the set composed of all Oi or the generalized combination of its divisions as a memory block. Since any object o belongs to O, there is v such that v r3 o, and U is a division of V, so Oi is also a division of O.

Capability describes the relationship between user u and memory block Oi, and records the access method of memory block Oi. For a certain user ui, the capability is defined as a two-tuple (addr, relation), where address is the address of Oi, and relation is the relationship between the ui and Oi. Here the relationship of u to Oi is defined as follows. If any o belongs to Oi, there exists v belonging to u such that v r1 o, then u R3 Oi. If there exists o belonging to Oi and v belonging to u such that v r2 o, then u R2 Oi. If there is o belonging to Oi, there is v belonging to u such that v r2 o, and u R2 Oi is not established, then u R1 Oi. If u R3/R2/R1 Oi are not established, u R0 Oi is defaulted.

Here we define a total order relation > on R0, R1, R2, R3, that is R3>R2>R1>R0. The capability list of ui is the list of all Oj's capabilities, and R0 can be omitted. Each UI has a capability list.

The capability list itself has three important basic operations. 1. The capability list can generate a subset (O addr, relation'), where relation' represents the original relationship or a smaller relationship. 2.ui can pass a subset of ui capabilities to uj, so that uj updates its capability list. 3.The capability subset passed by ui can be withdrawn, even if there is a loop in the capabilities transfer.

Based on the above definition, the work of the capability analyzer can be described as, for a given user space U, based on the expressions of ownership and lifetime, the capability list of each user ui is obtained with necessary other information. In the capability tuple, the address is fixed, so the relation of u to Oi becomes the key to the analysis. Here, for the sake of safety during subsequent operation, we first need to further divide the memory block Oi.

Any ui corresponds to an Oi. According to the life cycle and ownership of each o in Oi, an Oi needs to be divided into four parts. According to the definition of Oi, Oi=o|the existence of v belongs to ui so that v r3 o, we divide the o in Oi into two sub-groups, Oia and Oib, according to whether the life cycle of v in v r3 o is within the life cycle of the ui code block set. For the o in Oia and Oib, we divide them into two subsets according to whether the life cycle of the variable v of v r2/r1 o is outside the life cycle of the ui code block. So we get a division of Oi Oia1, Oia2, Oib1, Oib2, denoted as Oi0, Oi1, Oi2, Oi3.

ui is an element in a given user space U, which is a partition of the set V, and its related algorithms will be described later. Here we only need to know that any u belongs to U, and u corresponds to a continuous code block. // and total sequence. Due to the characteristics of rust functional programming, we

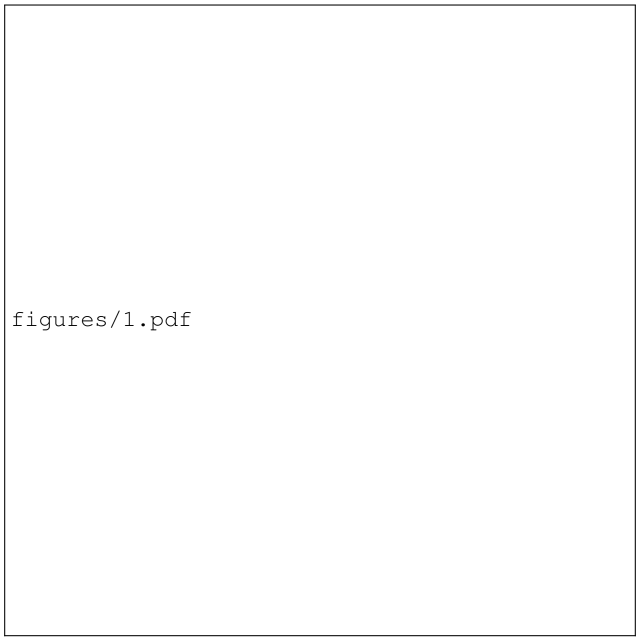

Figure 1: Normalized end-to-end latency of the seven microservice applications deployed on , Linux and RedLeaf.

can think that u corresponds to a well-defined function body, which consists of other parts such as input parameters, internal variables, return values, and constants. The union of lifecycles within a ui scope is the lifecycle of a code block. A variable life cycle [t0,t1,...,tn]///Definition of what is called the life cycle of variables is internal

### 3.3. VM Architecture

### 3.4. VM Management

## 4. Evaluation

Our evaluation aims to answer following questions:
- How is the overhead of our method compared to baselines?
- How do our method perform when scales of programs increase?
- How much do our method improve the security?
- How does our method perform when microservices are migrated?

### 4.1. End-to-end Performance

The evaluation of , Linux and RedLeaf are based on seven services. We give a brief introduction to the function and line of code (LoC) of the services in **?? ??**.

### 4.2. Scalability

### 4.3. Migration

### 4.4. Defense Against Attacks

### 4.5. Lessons Learned

**Security of related works.**

**Programming languages with ownerships.**
**Limitation of .**

## 5. Conclusion

| Field | Value |
|---|---|
| File format | PDF with numbered pages |
| Page limit | 11 pages, excluding references |
| Paper size | US Letter 8.5in × 11in |
| Top margin | 1in |
| Bottom margin | 1in |
| Left margin | 0.75in |
| Right margin | 0.75in |
| Column separation | 0.25in |
| Body | 2-column, single-spaced |
| Body font | 10pt |
| Abstract font | 10pt, italicized |
| Section heading font | 12pt, bold |
| Subsection heading font | 10pt, bold |
| Section spacing | ≥ 6pt |
| Caption font | 9 pt |
| Fonts in figures and tables | ≥ 8pt, preferably ≥ 9pt |
| References | 8pt, no page limit, list all author full names, include link to paper (preferably DOI), make citations clickable |
| Appendix | counts towards the page limit |

Table 1: Formatting guidelines for submission.

The ASPLOS'24 proceedings will be freely available via the ACM Digital Library for up to two weeks before the conference. Authors must consider any implications of this early disclosure of their work *before* submitting their papers.

## A. Appendix

### A.1. Evaluation Supplementary

Table 2: Brief introduction to benchmark services in our evaluation.

| Service | Description | LoC |
|---|---|---|
| ObserWard | A web fingerprint identification service. | 2686 |
| Rooster | A simple password manager. | 4463 |
| Polaris | A open-source music streaming application. | 7455 |
| Kafka | A event store and stream-processing platform. | 10804 |
| Spotify | A audio streaming and media service. | 12974 |
| Mail-auth | An email authentication and reporting library | 16115 |
| Wagyu | A library for cryptocurrency wallet service. | 48174 |