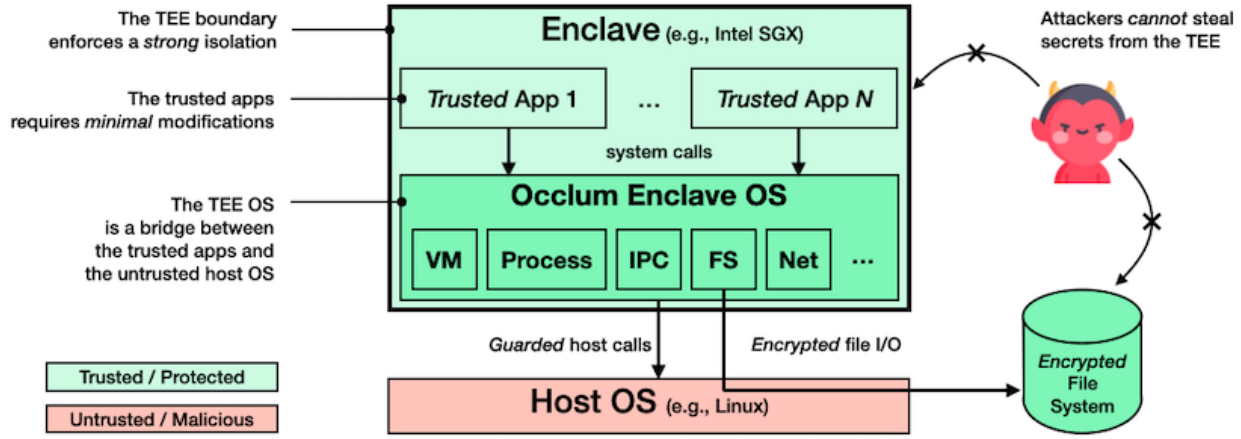
Occlum

Occlum Contributors

Nov 07, 2023

INTRODUCTION

Occlum is a memory-safe, multi-process library OS (LibOS) for Intel SGX. As a LibOS, it enables legacy applications to run on SGX with little or even no modifications of source code, thus protecting the confidentiality and integrity of user workloads transparently.



CHAPTER

ONE

TABLE OF CONTENTS

1.1 Quick Start

1.1.1 Environment

Supported HW

First, please make sure the baremetal or VM machine support SGX. Otherwise, users can only try SW simulation mode.

To have best user experience, SGX2 or SGX1 with FLC(Flexible Launch Control) feature are required.

Users can use command `cpuid` to detect if the HW satisfy Occlum requirements.

- SGX2, `cpuid | grep SGX2`
- FLC, `cpuid | grep SGX_LC`

Prerequisites

Kernel Version

To use SGX in-tree driver, Linux kernel 5.10+ is expected.

For example, in the Ubuntu 20.04 OS (Occlum default development OS), users could update the kernel with below command to get all Occlum required kernel features.

```
$ sudo apt install --install-recommends linux-generic-hwe-20.04
```

Start the Occlum dev container

To give Occlum a quick try, one can use the Occlum docker image by following below steps:

Step 1 is to be done on the host OS (Linux). Step 2-3 are to be done on the guest OS running inside the Docker container.

1. Run the Occlum docker container, which has Occlum and its demos preinstalled:

```
# 1. Create softlinks on host
mkdir -p /dev/sgx
ln -sf ../sgx_enclave /dev/sgx/enclave
ln -sf ../sgx_provision /dev/sgx/provision
```

(continues on next page)

(continued from previous page)

```
# 2. Create container in two methods:
# (1) With privileged mode
docker run -it --privileged -v /dev/sgx:/dev/sgx occlum/occlum:[version]-ubuntu20.04

# (1) With non-privileged mode
docker run -it --device /dev/sgx/enclave --device /dev/sgx/provision occlum/
↪occlum:[version]-ubuntu20.04
```

2. (Optional) Try the sample code of Intel SGX SDK to make sure that SGX is working

```
cd /opt/intel/sgxsdk/SampleCode/SampleEnclave && make && ./app
```

3. Check out Occlum's demos preinstalled at `/root/demos`. Or you can try to build and run your own SGX-protected applications using Occlum as shown in the demos.

1.1.2 Hello World

If you were to write an SGX Hello World project using some SGX SDK, the project would consist of hundreds of lines of code. And to do that, you have to spend a great deal of time to learn the APIs, the programming model, and the build system of the SGX SDK.

Thanks to Occlum, you can be freed from writing any extra SGX-aware code and only need to type some simple commands to protect your application with SGX transparently — in four easy steps.

Step 1. Compile the user program with the Occlum toolchain (e.g., `occlum-gcc`)

```
$ occlum-gcc -o hello_world hello_world.c
$ ./hello_world
Hello World
```

And Occlum can support `gcc` compile as well. The difference is that the binaries generated by `occlum-gcc` are musl-libc based, but are glibc based if compiled by `gcc`.

Note that the Occlum toolchain is not cross-compiling in the traditional sense: the binaries built by the Occlum toolchain is also runnable on Linux. This property makes it convenient to compile, debug, and test user programs intended for Occlum.

Step 2. Initialize a directory as the Occlum instance via `occlum init` or `occlum new`

```
$ mkdir occlum_instance && cd occlum_instance
$ occlum init
```

or

```
$ occlum new occlum_instance
```

The `occlum init` command creates the compile-time and run-time state of Occlum in the current working directory. The `occlum new` command does basically the same thing but in a new instance directory. Each Occlum instance directory should be used for a single instance of an application; multiple applications or different instances of a single application should use different Occlum instances.

Step 3. Generate a secure Occlum FS image and Occlum SGX enclave via `occlum build`

```
$ cp ../hello_world image/bin/
$ occlum build
```

The content of the `image` directory is initialized by the `occlum init` command. The structure of the `image` directory mimics that of an ordinary UNIX FS, containing directories like `/bin`, `/lib`, `/root`, `/tmp`, etc. After copying the user program `hello_world` into `image/bin/`, the `image` directory is packaged by the `occlum build` command to generate a secure Occlum FS image as well as the Occlum SGX enclave.

For platforms that don't support SGX, it is also possible to run Occlum in SGX simulation mode. To switch to the simulation mode, `occlum build` command must be given an extra argument or an environment variable as shown below:

```
$ occlum build --sgx-mode SIM
```

or

```
$ SGX_MODE=SIM occlum build
```

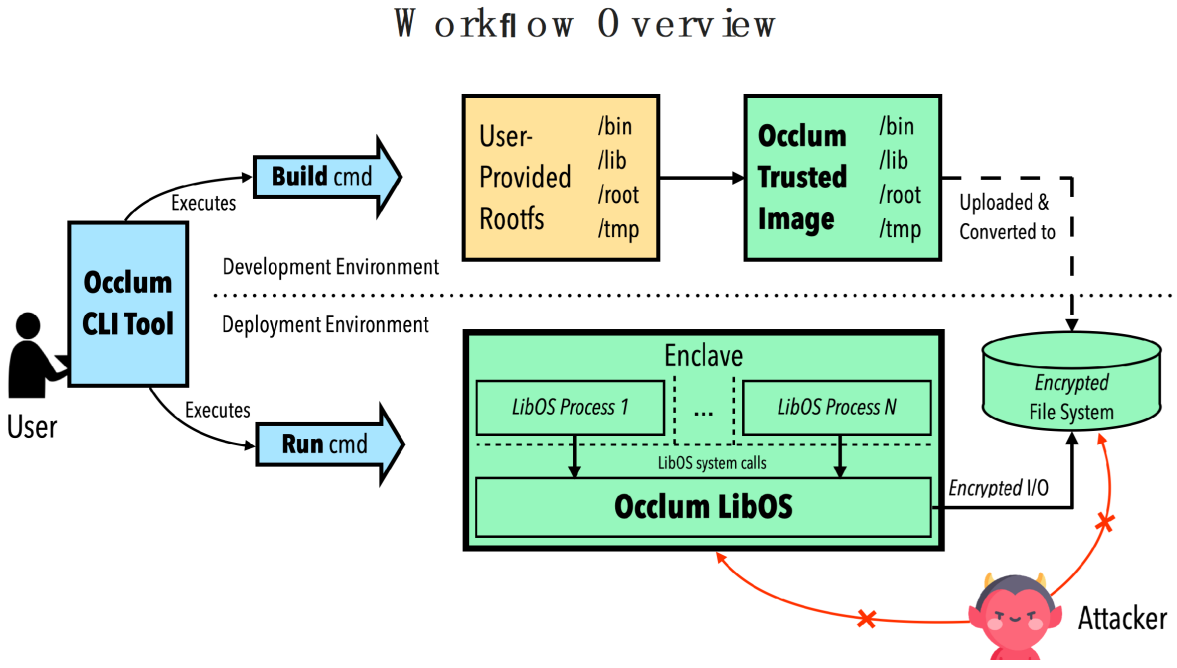
Step 4. Run the user program inside an SGX enclave via `occlum run`

```
$ occlum run /bin/hello_world
Hello World!
```

The `occlum run` command starts up an Occlum SGX enclave, which, behind the scene, verifies and loads the associated Occlum FS image, spawns a new LibOS process to execute `/bin/hello_world`, and eventually prints the message.

1.2 User Commands

1.2.1 Work Flow Overview



Workflow Overview

1.2.2 General Commands

Occlum provides easy user commands as below.

```
occlum new <path>
```

Create a new directory at and initialize as the Occlum instance.

```
occlum init
```

Initialize a directory as the Occlum instance.

```
occlum build [--sign-key <key_path>] [--sign-tool <tool_path>] [--image-key <key_path>]␣
↪[-f/--force]
```

Build and sign an Occlum SGX enclave (.so) and generate its associated secure FS image according to the user-provided image directory and Occlum.json config file. The whole building process is incremental: the building artifacts are built only when needed. To force rebuilding all artifacts, give the [-f/–force] flag.

```
occlum run [--cpus <num_of_cpus>] <program_name> <program_args>
```

Run the user program inside an SGX enclave.

```
occlum package [<package_name>.tar.gz]
```

Generate a minimal, self-contained package (.tar.gz) for the Occlum instance. The resulting package can then be copied to a deployment environment and unpacked as a runnable Occlum instance.

All runtime dependencies required by the Occlum instance, except Intel SGX driver and Intel SGX PSW, are included in the package.

If package_name is not specified, the directory name of Occlum instance will be used. In default only HW release mode package is supported. Debug or simulation mode package could be supported by adding "–debug" flag.

```
occlum gdb <program_name> <program_args>
```

Debug the program running inside an SGX enclave with GDB.

```
occlum mount [--sign-key <key_path>] [--sign-tool <tool_path>] [--image-key <key_path>]
↪<path>
```

Mount the secure FS image of the Occlum instance as a Linux FS at an existing . This makes it easy to access and manipulate Occlum's secure FS for debug purpose.

```
occlum gen-image-key <key_path>
```

Generate a file consists of a randomly generated 128-bit key for encryption of the FS image.

1.2.3 Container-like Commands

Occlum has added several new experimental commands, which provide a more container-like experience to users, as shown below:

```
occlum start
```

Start an Occlum instance, completing all the initialization including LibOS boots, Init FS and application root FS mount. A background service is started to listen which application is going to be executed.

```
occlum exec [cmd1] [args1]
```

Actually start executing the application.

```
occlum exec [cmd2] [args2]
occlum exec [cmd3] [args3]
```

If there are more executable application binaries in the Occlum instance entrypoint, users could start executing them in parallel.

```
occlum stop
```

Stop the Occlum instance including the background listening service.

1.3 Occlum Configuration

Occlum can be configured easily via a configuration file named `Occlum.json`, which is generated by the `occlum init` command in the Occlum instance directory. The user can modify `Occlum.json` to configure Occlum. The template of `Occlum.json` is shown below.

```
{
    // Resource limits
    "resource_limits": {
        // The total size of enclave memory available to LibOS processes
        "user_space_size": "256MB",
        // The heap size of LibOS kernel
        "kernel_space_heap_size": "32MB",
        // The stack size of LibOS kernel
        "kernel_space_stack_size": "1MB",
        // The max number of LibOS threads/processes
        "max_num_of_threads": 32
    },
    // Process
    "process": {
        // The stack size of the "main" thread
        "default_stack_size": "4MB",
        // The max size of memory allocated by brk syscall
        "default_heap_size": "16MB",
        // The max size of memory by mmap syscall (OBSOLETE. Users don't need to modify␣
↪this field. Keep it only for compatibility)
        "default_mmap_size": "32MB"
    },
```

(continues on next page)

(continued from previous page)

```
    // Entry points
    //
    // Entry points specify all valid path prefixes for <path> in `occlum run
    // <path> <args>`. This prevents outside attackers from executing arbitrary
    // commands inside an Occlum-powered enclave.
    "entry_points": [
        "/bin"
    ],
    // Environment variables
    //
    // This gives a list of environment variables for the "root"
    // process started by `occlum exec` command.
    "env": {
        // The default env vars given to each "root" LibOS process. As these env vars
        // are specified in this config file, they are considered trusted.
        "default": [
            "OCCLUM=yes"
        ],
        // The untrusted env vars that are captured by Occlum from the host environment
        // and passed to the "root" LibOS processes. These untrusted env vars can
        // override the trusted, default envs specified above.
        "untrusted": [
            "EXAMPLE"
        ]
    },
    // Enclave metadata
    "metadata": {
        // Enclave signature structure's ISVPRODID field
        "product_id": 0,
        // Enclave signature structure's ISVSVN field
        "version_number": 0,
        // Whether the enclave is debuggable through special SGX instructions.
        // For production enclave, it is IMPORTANT to set this value to false.
        "debuggable": true,
        // Whether to turn on PKU feature in Occlum
        // Occlum uses PKU for isolation between LibOS and userspace program,
        // It is useful for developers to detect potential bugs.
        //
        // "pkru" = 0: PKU feature must be disabled
        // "pkru" = 1: PKU feature must be enabled
        // "pkru" = 2: PKU feature is enabled if the platform supports it
        "pkru": 0
    },
    // Mount points and their file systems
    //
    // The default configuration is shown below.
    "mount": [
        {
            "target": "/",
            "type": "unionfs",
            "options": {
                "layers": [
```

(continues on next page)

(continued from previous page)

```
                    {
                        "target": "/",
                        "type": "sefs",
                        "source": "./build/mount/__ROOT",
                        "options": {
                            "MAC": ""
                        }
                    },
                    {
                        "target": "/",
                        "type": "sefs",
                        "source": "./run/mount/__ROOT"
                    }
                ]
            }
        },
        {
            "target": "/host",
            "type": "hostfs",
            "source": "."
        },
        {
            "target": "/proc",
            "type": "procfs"
        },
        {
            "target": "/dev",
            "type": "devfs"
        }
    ]
}
```

1.3.1 Runtime Resource Configuration for Occlum process

Occlum has enabled per process resource configuration via prlimit syscall and shell built-in command ulimit.

```
#! /usr/bin/bash
ulimit -a

# ulimit defined below will override configuration in Occlum.json
ulimit -Ss 10240 # stack size 10M
ulimit -Sd 40960 # heap size 40M
ulimit -Sv 102400 # virtual memory size 100M (including heap, stack, mmap size)

echo "ulimit result:"
ulimit -a

# Run applications with the new resource limits
...
```

For more info, please check demos/fish.

1.4 Build and Install

Generally, users don't need build the Occlum from source. They can directly use Occlum official docker image in Docker hub.

```
docker pull occlum/occlum:[version]-ubuntu20.04
```

1.4.1 Build from Source

To build Occlum from the latest source code, do the following steps in an Occlum Docker container (which can be prepared as shown in the last section):

1. Download the latest source code of Occlum

   ```
   mkdir occlum && cd occlum
   git clone https://github.com/occlum/occlum .
   ```

2. Prepare the submodules and tools required by Occlum.

   ```
   make submodule
   ```

3. Compile and test Occlum

   ```
   make

   # test musl based binary
   make test

   # test glibc based binary
   make test-glibc

   # stress test
   make test times=100
   ```

   For platforms that don't support SGX

   ```
   SGX_MODE=SIM make
   SGX_MODE=SIM make test
   ```

4. Install Occlum

   ```
   make install
   ```

   which will install the `occlum` command-line tool and other files at `/opt/occlum`.

If `release` build and install is required, just add **OCCLUM_RELEASE_BUILD=1** in front of every `make` command.The Occlum Dockerfile can be found at here. Use it to build the container directly or read it to see the dependencies of Occlum.

1.4.2 How to Build and Run Release-Mode Enclaves?

By default, the `occlum build` command builds and signs enclaves in debug mode. These SGX debug-mode enclaves are intended for development and testing purposes only. For production usage, the enclaves must be signed by a key acquired from Intel (a restriction that will be lifted in the future when Flexible Launch Control is ready) and run with SGX debug support disabled.

Occlum has built-in support for both building and running enclaves in release mode. To do that, modify `Occlum.json` [metadata]-[debuggable] field to `false`. And then run the commands below:

```
$ occlum build --sign-key <path_to/your_key.pem>
$ occlum run <prog_path> <prog_args>
```

Ultimately, whether an enclave is running in the release mode should be checked and judged by a trusted client through remotely attesting the enclave. See the remote attestation demo here.

1.5 Install Occlum with Popular Package Managers

Occlum can be easily installed with popular package managers like APT and RPM. This document walks you through the steps to install Occlum on popular Linux distributions like Ubuntu and CentOS using package managers.

1.5.1 Prerequisite

Install enable_RDFSBASE Kernel Module

If the Kernel version is before `v5.9`, please follow this README to install `enable_rdfsbase` kernel module.

1.5.2 Install Occlum with APT on Ubuntu 20.04

1. Install Prerequisite

```
apt update
DEBIAN_FRONTEND=noninteractive apt install -y --no-install-recommends ca-certificates␣
↪gnupg2 jq make gdb wget libfuse-dev libtool tzdata rsync
```

1. Install Intel® SGX Driver and Intel® SGX PSW

Please follow Intel SGX Installation Guide to install SGX driver and SGX PSW. SGX SDK is not required. Using PSW installer is recommanded.

To install PSW, follow the guide to add Intel® SGX repository to APT source. And then run:

```
apt-get update
apt-get install -y libsgx-dcap-ql libsgx-epid libsgx-urts libsgx-quote-ex libsgx-uae-
↪service libsgx-dcap-quote-verify-dev
```

After installing PSW, please make sure `aesm` service is in `active (running)` state by checking:

```
service aesmd status
```

1. Install Occlum

```
echo 'deb [arch=amd64] https://occlum.io/occlum-package-repos/debian bionic main' | tee /
↪etc/apt/sources.list.d/occlum.list
wget -qO - https://occlum.io/occlum-package-repos/debian/public.key | apt-key add -
apt-get update
apt-get install -y occlum
```

Occlum toolchains packages

Besides, users can choose to install the toolchain installer based on the application's language. Currently, Occlum supports only `musl-gcc`, `glibc`. Users can install each one on demand.

```
apt install -y occlum-toolchains-gcc
apt install -y occlum-toolchains-glibc
```

Occlum Runtime package

If users only expect to run the Occlum instance image, then `occlum-runtime` package is better choice for size reason.

```
apt install -y occlum-runtime
```

1.5.3 Version Compatability Matrix

When version is not specified, Occlum with the latest version will be installed. If a user would like to evaluate an older version, please make sure the corresponding Intel® SGX PSW is installed.

The matrix below shows the version compatability since Occlum `0.16.0`. Please check before installing or upgrading.

For more information about the packages, please checkout here.

1.6 Occlum-Compatible Executable Binaries

The `hello_world` demo is based on musl libc with recompiling. But Occlum actually can support both `musl libc` and `glibc` based executable binaries without recompiling if they meet below three principles.

1.6.1 No fork syscall

By design, Occlum doesn't support `fork` syscall. If there is `fork` syscall in the application, users have to assess if the `fork` could be replaced by `vfork + exec` or `posix spawn`. If yes, code modification and recompiling is inevitable.

1.6.2 libc version compatibility

No recompiling doesn't mean the original `libc` libraries can be directly used in Occlum. To run in Occlum TEE environment, customized `libc` libraries are provided in the Occlum development docker image.

Actually, the original `libc` libraries are to be replaced silently in `Occlum build` stage by `copy_bom` tool.

1.6.3 Compiled with PIE (Position-Independent-Executable)

Current Ubuntu:20.04 and Alpine:3.11 enable `PIE` in default.

1.7 How to Debug?

To debug an app running upon Occlum, one can harness Occlum's builtin support for GDB via `occlum gdb` command. More info can be found here.

Meanwhile, one can use `occlum mount` command to access and manipulate the secure filesystem for debug purpose.

If the cause of a problem does not seem to be the app but Occlum itself, then one can take a glimpse into the inner workings of Occlum by checking out its log. Occlum's log level can be adjusted through `OCCLUM_LOG_LEVEL` environment variable. It has six levels: `off`, `error`, `warn`, `debug`, `info`, and `trace`. The default value is `off`, i.e., showing no log messages at all. The most verbose level is `trace`.

1.8 Insight of Occlum Instance Generation

For every application to be running in Occlum (TEE env), all the running required files, libraries and binaries have to be put into Occlum file system. Here is the tree view of one Occlum instance.

```
./occlum_instance/
|-- Occlum.json
|-- build
|-- image       // Occlum root file system
|   |-- bin
|   |-- dev
|   |-- etc
|   |-- host
|   |-- lib
|   |-- lib64
|   |-- opt
|   |-- proc
|   |-- root
|   |-- sfs
|   |-- sys
|   `-- tmp
|-- initfs      // Occlum init file system
|   |-- bin
|   |-- dev
|   |-- etc
|   |-- lib
|   `-- proc
`-- run
```

1.8.1 File System from Occlum Perspective

Let's clarify some definitions users usually get confused.

- **host file system**

Host means the environment where users run Occlum build. Usually it is the Occlum official docker image. In this environment, users build and prepare all files to be running in Occlum.

- **Occlum init file system**

Occlum has a unique **Occlum -> init ->application** boot flow, please check boot_flow for detail. So the Occlum init file system is what the Occlum **init** process sees. In develop stage, it is in the path `occlum_instance/initfs`. Generally, a `occlum new` generates a default init file system, users don't need modify this part unless you know exactly what you are doing.

- **Occlum root file system**

It is the file system the real application sees. And it is also the place users need put all the running required files, libraries and binaries. In develop stage, it is in the path `occlum_instance/image`.

In summary, to generate Occlum instance, one important step is to copy application running required files from **host file system** to **Occlum root file system**.

Next, it is an example of using **copy_bom** to ease the Occlum root file system creation.

1.8.2 Redis in Occlum

There is a redis demo in github which is built from source. Actually, users could use the OS installed redis binary directly to generate a runnable Occlum Redis instance by following steps.

- Install the redis by **apt**.

```
apt install redis-server
```

- Create a **redis.yaml** to assist generate redis Occlum instance.

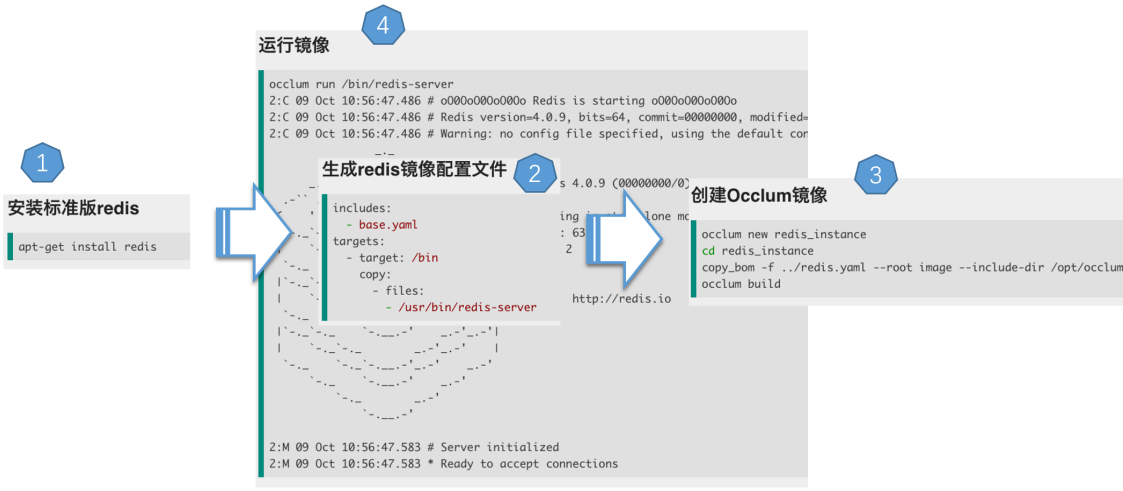
```
includes:
  - base.yaml
targets:
  - target: /bin
    copy:
      - files:
        - /usr/bin/redis-server
```

- Generate and build redis Occlum instance

```
occlum new redis_instance
cd redis_instance
copy_bom -f ../redis.yaml --root image --include-dir /opt/occlum/etc/template
occlum build
```

- Run the redis server in Occlum

```
occlum run /bin/redis-server
```



The whole flow is like below.

Very easy and straightforward, right?

Next let's explore the magic of **copy_bom**.

1.8.3 copy_bom Case Study

The `copy_bom` tool is designed to copy files described in a bom file to a given dest root directory. For details users could refer to page.

The most important and useful function of copy_bom is the automatic dependencies finding and copy. For the redis case, there are so many dependent libraries the `redis-server` required for running.

```
root@OCCLUM-DEV-HK:~# ldd /usr/bin/redis-server
        linux-vdso.so.1 (0x00007ffed057e000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fce4d57b000)
        libatomic.so.1 => /lib/x86_64-linux-gnu/libatomic.so.1 (0x00007fce4d571000)
        liblua5.1-cjson.so.0 => /lib/x86_64-linux-gnu/liblua5.1-cjson.so.0 (0x00007fce4d369000)
        liblua5.1-bitop.so.0 => /lib/x86_64-linux-gnu/liblua5.1-bitop.so.0 (0x00007fce4d166000)
        liblua5.1.so.0 => /lib/x86_64-linux-gnu/liblua5.1.so.0 (0x00007fce4d135000)
        libjemalloc.so.2 => /lib/x86_64-linux-gnu/libjemalloc.so.2 (0x00007fce4ce59000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fce4cd08000)
        librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007fce4ccfe000)
        libhiredis.so.0.14 => /lib/x86_64-linux-gnu/libhiredis.so.0.14 (0x00007fce4cceb000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fce4ccc8000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fce4cad6000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fce4d69d000)
        libstdc++.so.6 => /lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007fce4c8f4000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fce4c8d7000)
```

LDD for redis-server

All the dependent libraries above have to be copied to **occlum_instance**. But the **redis.yaml** showed above just has `redis-server`. How come it is running well in Occlum?

That is because the `copy_bom` would detect binaries or libraries defined in the yaml file, find all the dependencies and copy them to the corresponding path in **occlum_instance/image**. For this case, all required libraries would be in place after `copy_bom` operation.

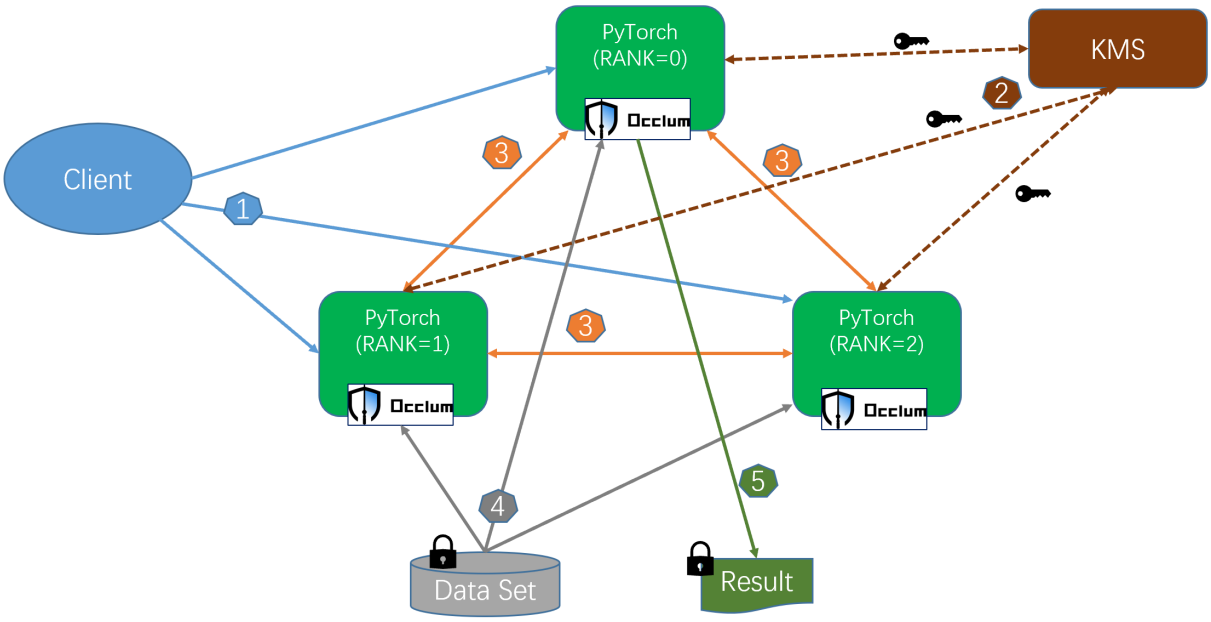
```
root@OCCLUM-DEV-HK:~/redis_instance# copy_bom -f redis.yaml --root image --include-dir /opt/occlum/etc/template
root@OCCLUM-DEV-HK:~/redis_instance# tree image/
image/
|-- bin
|   `-- redis-server
|-- dev
|-- etc
|-- host
|-- lib
|-- lib64
|   `-- ld-linux-x86-64.so.2
|-- opt
|   `-- occlum
|       `-- glibc
|           `-- lib
|               |-- libatomic.so.1
|               |-- libc.so.6
|               |-- libdl.so.2
|               |-- libgcc_s.so.1
|               |-- libhiredis.so.0.14
|               |-- libjemalloc.so.2
|               |-- liblua5.1-bitop.so.0
|               |-- liblua5.1-cjson.so.0
|               |-- liblua5.1.so.0
|               |-- libm.so.6
|               |-- libpthread.so.0
|               |-- librt.so.1
|               `-- libstdc++.so.6
|-- proc
|-- root
|-- sys
`-- tmp
```

Occlum redis tree

1.9 Distributed PyTorch

PyTorch is well supported in Occlum. Unmodified distributed PyTorch training (such as fasion-MNIST dataset training) could be enabled running in TEE by simple steps.

1.9.1 Overview



Flow

Above is the flow chart which could be break down as below.

1. Client starts the PyTorch Occlum instances, such as one master and two worker nodes.
2. Every PyTorch Occlum instance starts, get required keys from one remote attestation based KMS. The keys could be for data decryption, result encryption or Pytorch nodes TLS certs.
3. PyTorch nodes set up rendezvous.
4. Loading dataset, training started.
5. Encrypt and save result if necessary.

For step 2, users could use the Occlum init-ra AECS solution which has no invasion to the PyTorch application.

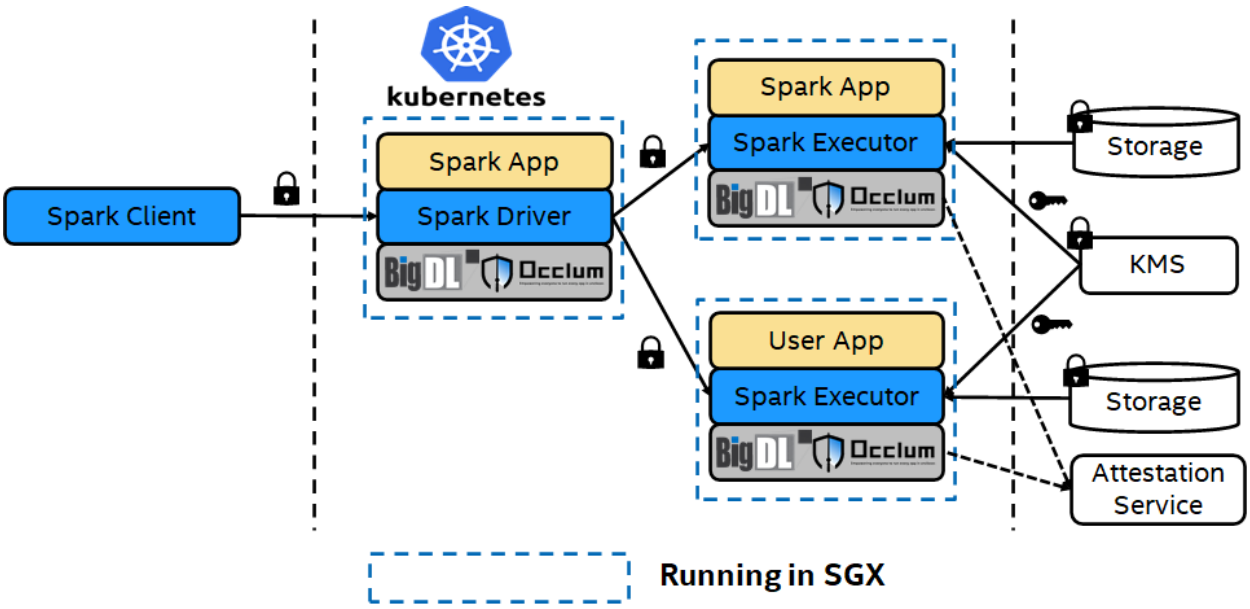
More details please refer to pytorch demo.

Note, the demo has not involved KMS and data encryption or decryption. Users could add them accordingly.

1.10 Secure Spark Data Analytics using BigDL PPML and Occlum

Protecting privacy and confidentiality is critical for large-scale data analysis and machine learning. Occlum collaborates with BigDL PPML, provides a Trusted Cluster Environment for secure Big Data & AI applications, even on untrusted cloud environment. The solution ensures end-to-end security enabled for the entire distributed Spark workflows.

1.10.1 Overall Architecture



Architecture

1.10.2 PPML Occlum Spark on Azure

Please check the link for all the details.

Also, there is a introduction document published in Azure Confidential Computing Blog, demonstrates the PPML Occlum solution using a sample analytics application built for the NYTaxi dataset.

1.10.3 PPML Occlum Spark on Aliyun (Chinese version)

It is a detail how-to and technical insight about deploying the PPML Occlum Spark solution on Aliyun. Please check the link for all the details.

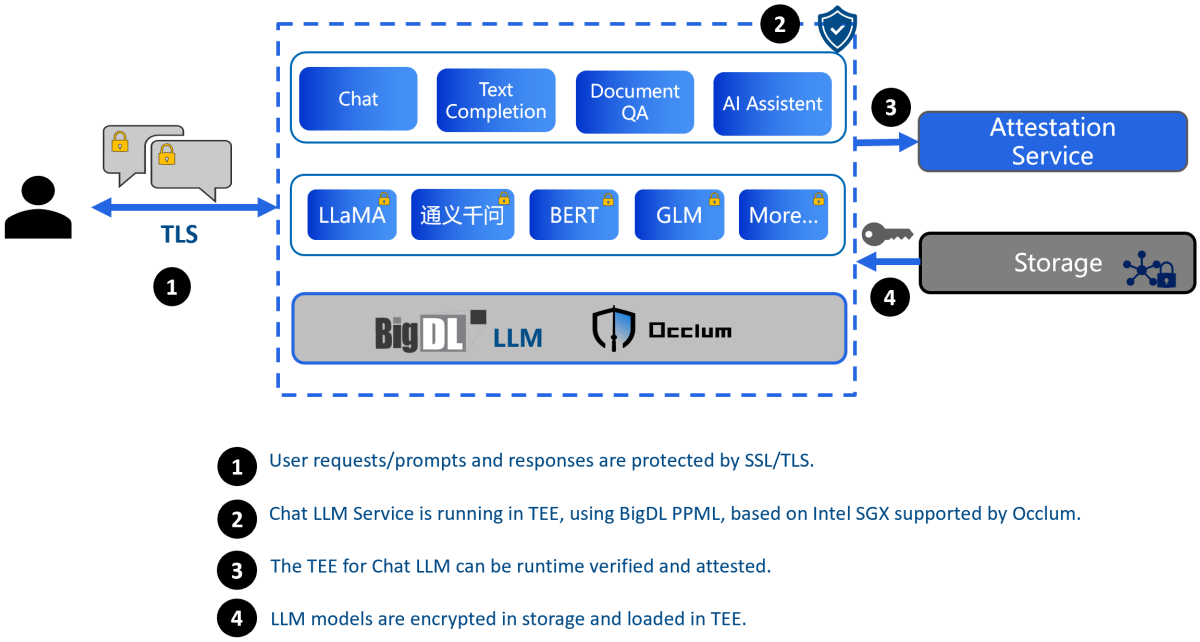
1.11 LLM Inference in TEE

LLM (Large Language Model) inference in TEE can protect the model, input prompt or output. The key challenges are:

1. the performance of LLM inference in TEE (CPU)
2. can LLM inference run in TEE?

With the significant LLM inference speed-up brought by BigDL-LLM, and the Occlum LibOS, now high-performance and efficient LLM inference in TEE could be realized.

1.11.1 Overview



LLM inference

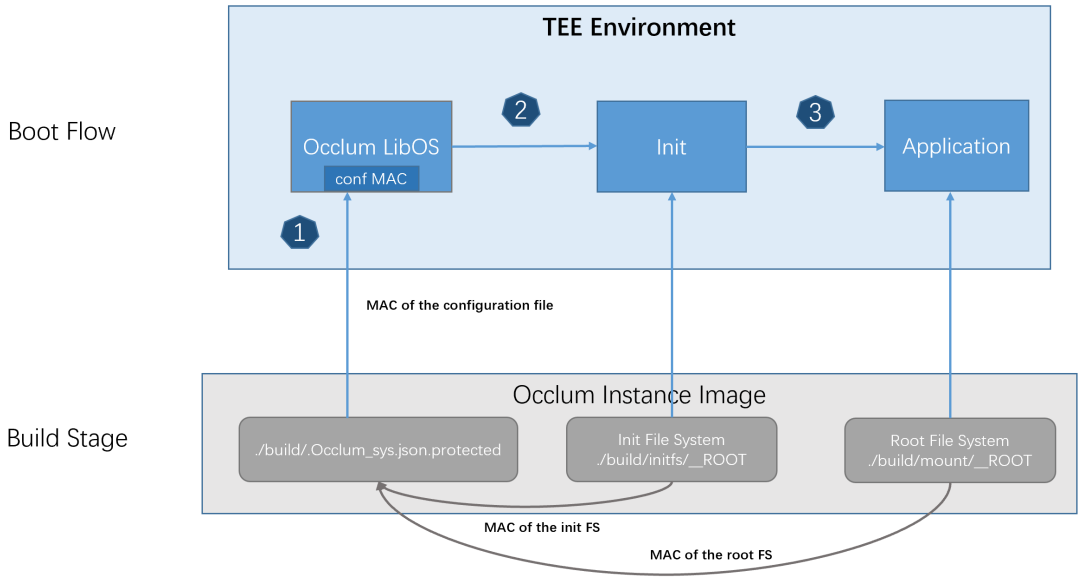
Above is the overview chart and flow description.

For step 3, users could use the Occlum init-ra AECS solution which has no invasion to the application.

More details please refer to LLM demo.

1.12 Boot Flow Overview

Occlum has a unique **Occlum -> init ->application** boot flow, plus integrity check stage by stage. Below is the high-level overview.



Boot_flow

1.12.1 Measurements

There are total three MACs which are all generated in building stage and verified in boot flow.

- MAC of the init FS, it indicates the measurement of init file system which will be mounted in **init** stage.
- MAC of the root FS, it indicates the measurement of application file system which will be mounted in **application** stage.
- MAC of the configuration file. Both above MACs are filled into the configuration file. Thus the MAC of the configuration file reflects the differences of the two file systems to some extent.

1.12.2 Boot flow

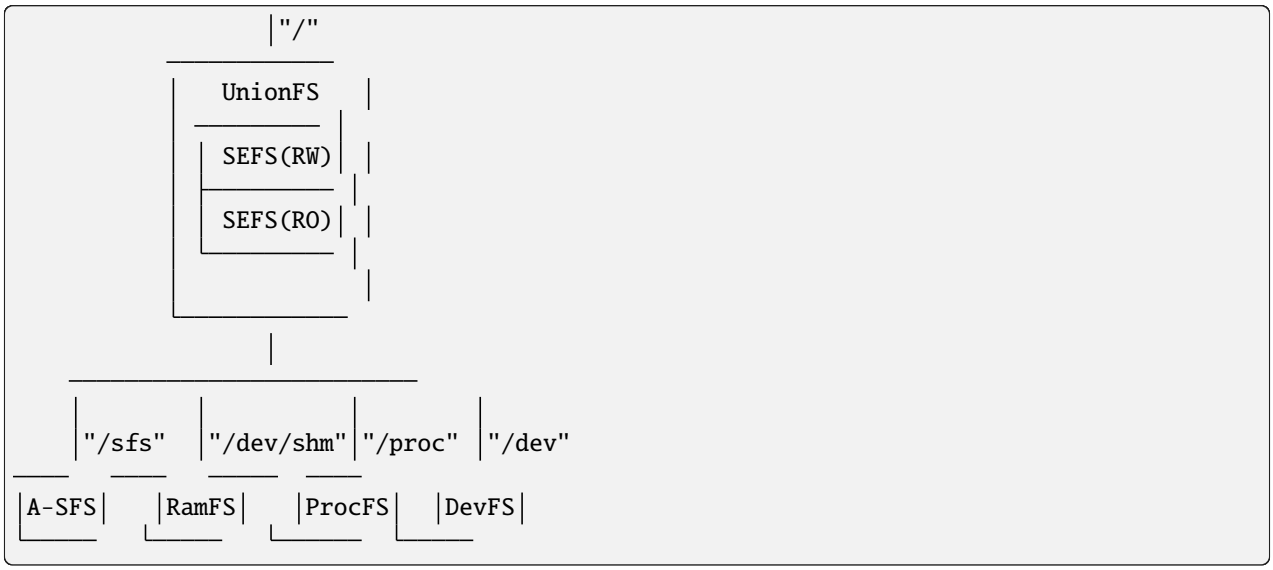
1. To user, the entry point is the command `occlum run`. The steps behind the command are PAL APIs `occlum_pal_init` and `occlum_pal_create_process`. It launches the Occlum LibOS kernel in the Enclave. The kernel then loads the configuration file `.Occlum_sys.json.protected`, doing integrity check with the MAC saved in the LibOS section **builtin_config** (marked as `conf_mac` in the picture). If pass, it uses the settings in the configuration file to do memory/process initialization. Then, it tries mount the init filesystem. It does integrity check again with init FS MAC in the `.Occlum_sys.json.protected`. If pass, the first user space process `init` got launched.
2. There is a default minimal `init` provided. In this `init`, it calls the Occlum added syscall `SYS_MOUNT_FS` to mount the application file system. The syscall implementation in the Occlum kernel does integrity check again with root FS MAC in the `.Occlum_sys.json.protected` to make sure expected application got launched.
3. At this stage, the real user application got launched.

Generally, all operation application required but not part of the application, such as remote attestation, could be put into **"init"**. This feature makes Occlum highly compatible to any remote attestation solution without involving application's change.

1.13 Occlum File System Overview

Occlum supports various file systems: e.g., read-only integrity-protected SEFS, writable encrypted SEFS, UnionFS, Async-SFS, untrusted HostFS, RamFS, and other pseudo filesystems.

Here is the default FS layout:



1.13.1 SEFS

The SEFS is a filesystem based on the Intel SGX Protected File (PFS), it protects the integrity and confidentiality of disk I/O data from the host.

Here is the hierarchy of SEFS:

There are two modes for SEFS:

1. Integrity-only mode
2. Encryption mode

Integrity-only mode

We modified the Intel SGX PFS to add this mode. It only protects the integrity of FS, which will generate the deterministic hash for the same FS image. So it is convenient to implement the remote attestation for the enclave with the same FS image.

The FS image (the `image` dir of occlum instance) provided by the user via the `copy_bom` tool will be transformed into a SEFS image in this mode by default. For the use of `copy_bom`, please refer to this.

Encryption mode

The integrity and confidentiality of the FS are both protected in this mode. There are two key-generation policies for this mode: the autokey generation policy and the user-provided key policy.

- The autokey generation policy

In this policy mode, the user is not required to provide the key. The key is automatically derived from the MRSIGNER of the enclave, the ProdID of the enclave, and the hardware info. So the same owner of two enclaves can share the FS data on the same machine. This policy is the default one for the encryption mode of SEFS.

- The user-provided policy

In this policy mode, the key should be provided by the user, which means the enclave owner should manage the key. This policy is more flexible for the user to control the data for sharing or isolation. The doc shows you how to use this policy mode.

1.13.2 UnionFS

As you can tell, we use the UnionFS consisting of SEFS as the rootfs of LibOS. To attest to the integrity of the user-provided FS image while having the ability to write data when running apps, we introduce a filesystem called UnionFS to satisfy this requirement.

UnionFS allows files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system.

We use two SEFSs to form the UnionFS. The lower layer is the read-only(RO) SEFS in integrity-only mode, and the upper is the writable(RW) SEFS in encryption mode. Generally speaking, the RO-SEFS is transformed by the `image` dir provided by the user while building the enclave, and the RW-SEFS is generated while the enclave is running.

Here is the configuration of rootfs, the first item is the lower layer RO-SEFS and the second item is the upper layer RW-SEFS. As you can tell, the RO-SEFS is at `./build/mount/__ROOT` and the RW-SEFS is at `./run/mount/__ROOT`.

```
  - target: /
    type: unionfs
    options:
```

(continues on next page)

(continued from previous page)

```
    layers:
      # The read-only layer which is generated in `occlum build`
      - target: /
        type: sefs
        source: ./build/mount/__ROOT
        options:
          MAC: ''
      # The read-write layer whose content is produced when running the LibOS
      - target: /
        type: sefs
        source: ./run/mount/__ROOT
```

1.13.3 Async-SFS

The Async-SFS is an asynchronous filesystem, which uses Rust asynchronous programming skills, making it fast and concurrent. It is mounted at `/sfs` by default. To achieve the high-performanced security, it uses the JinDisk as the underlying data storage and sends async I/O requests to it.

To accelerate block I/O, the page cache is introduced. It caches all the block I/O in the middle of Async-SFS and JinDisk. Thanks to the page cache and JinDisk, the result of the benchmark (e.g., FIO and Filebench) is significantly better than SEFS. If your App's performance is highly dependent on disk I/O, it is recommended to use Async-SFS.

Currently, there are some limitations of Async-SFS:

1. The maximum size of the file is 4GB.
2. The maximum size of FS is 16TB.

1.13.4 HostFS

The HostFS is used for convenient data exchange between the LibOS and the host OS. It simply wraps the untrusted host OS file to implement the functionalities of FS. So the data is straightforwardly transferred between LibOS and host OS without any protection or validation.

1.13.5 RamFS and other pseudo filesystems

The RamFS and other pseudo filesystems like ProcFS use the memory as the storage. So the data may lose if one terminates the enclave.

Please remember to enlarge the `kernel_space_heap_size` of Occlum.json if your app depends on RamFS.

1.13.6 Q & A

How to decrypt and view the rootfs?

One can use the `occlum mount` command to implements. Please refer to this doc for more information.

How to mount FS at runtime?

Please refer to this doc.

1.14 Mount Support

1.14.1 Mount command

The `occlum mount` command is designed to mount the secure FS image of the Occlum instance as a Linux FS at a specified path on Linux. This makes it easy to access and manipulate Occlum's secure FS for debug purpose.

Prerequisites

The command depends on Linux's Filesystem in Userspace (FUSE), which consists of two components: a kernel module and a userspace library. In most of Linux distributions, the FUSE kernel module is part of the Linux kernel by default, so it should not be a problem. The FUSE library can be installed via a package manager (e.g., `sudo apt-get install libfuse-dev` on Ubuntu). You do not need to install the package manually when using the Occlum Docker image as it has preinstalled the package.

One common pitfall when using FUSE with Docker is about privilege. A Docker container, by default, does not have the privilege to use FUSE. To get the privilege, the Docker container must be started with the following flags:

```
--cap-add SYS_ADMIN --device /dev/fuse
```

or

```
--privileged
```

For more info about enabling FUSE for Docker, see here.

How to use

To mount an Occlum's secure FS image successfully, three conditions must be satisfied:

1. The secure FS image exists and is not being used (e.g., via the `occlum run` or `occlum mount` command). This condition ensures that the secure FS will not be broken due to the concurrent access of different Occlum commands.
2. The three (optional) sign key, sign tool and image key arguments that are given to the `occlum mount` command must have the same values as those given to the `occlum build` command, which is how the image is created in the first place. This ensures that the secure FS can only be accessed by the owner of the enclave.
3. If the image key is not given to the `occlum build` command, the `occlum mount` command must be run on the same machine as the `occlum run` command that runs the current Occlum instance and writes to the image. This condition is due to the fact that the automatically drived encryption key of the secure FS is bound to the machine, i.e., the MRSIGNER key policy.

With the three conditions satisfied, the mount command is able to start a Linux FUSE FS server. Any I/O operations on the FUSE FS mounted at the specified path will be redirected by Linux kernel as I/O requests to the FUSE server. The FUSE server is backed by a special enclave, which can encrypt or decrypt the content of the secure FS image on demand.

Please note that if the `autokey_policy` field of the configurations of FS is set in Occlum.json, the mount command will not work. This is because the MRENCLAVE is used as an input to generate the encryption key, and the mount tool cannot mimic it.

The mount command **will not return** until the FUSE server is terminated in one of the two ways. The first one is to press ctrl+C. The second one is to use `umount` command. Both ways can terminate the server gracefully.

Step 1: Create an empty directory to serve as the mount point

```
mkdir <path>
```

Step 2: Mount the secure FS at the newly created mount point

```
occlum mount [--sign-key <key_path>] [--sign-tool <tool_path>] [--image-key <key_path>]
↪<path>
```

After mounting the secure FS successfully, you can access and manipulate the FS from the mount point as easy as regular Linux FS.

1.14.2 Mount and Unmount Filesystems at Runtime

Background

Users can specify the mount configuration in the `Occlum.json` file, then the file systems are mounted during the libOS startup phase. While this design provides a safe and simple way to access files, it is not as convenient as traditional Host OS. Apps are not allowed to mount and unmount file systems at runtime.

How to mount filesystems at runtime?

Apps running inside Occlum can mount some specific file systems via the mount() system call. This makes it flexible to mount and access files at runtime.

Currently, we only support to create a new mount with the trusted UnionFS consisting of SEFSs or the untrusted HostFS. The mount point is not allowed to be the root directory("/").

1. Mount trusted UnionFS consisting of SEFSs

Example code:

```
mount("unionfs", "<target_dir>", "unionfs", 0/* mountflags is ignored */,
      "lowerdir=<lower>,upperdir=<upper>,key=<128-bit-key>")
```

Mount options:

- The `lowerdir=<lower>` is a mandatory field, which describes the directory path of the RO SEFS on Host OS.
- The `upperdir=<upper>` is a mandatory field, which describes the directory path of the RW SEFS on Host OS.
- The `key=<128-bit-key>` is an optional field, which describes the 128bit key used to encrypt or decrypt the FS. Here is an example of the key: `key=c7-32-b3-ed-44-df-ec-7b-25-2d-9a-32-38-8d-58-61`. If this field is not provided, it will use the automatic key derived from the enclave sealing key.

2. Mount untrusted HostFS

Example code:

```
mount("hostfs", "<target_dir>", "hostfs", 0/* mountflags is ignored */,
      "dir=<host_dir>")
```

Mount options:

- The `dir=<host_dir>` is a mandatory field, which describes the directory path on Host OS.

How to unmount filesystems at runtime?

Apps running inside Occlum can unmount some specific file systems via the umount()/umount2() system calls. Note that root directory("/") is not allowed to unmount.

Example code:

```
umount("<target_dir>")
```

1.15 The Encrypted FS Image

Occlum has supported using an encrypted FS image, which is encrypted by a user-provided key, to run apps inside the enclave. The confidentiality and integrity of user's files and libraries are both protected with it.

1.15.1 How to use

To generate the encrypted FS image, user must give the `--image-key <key_path>` flag in the `occlum build` command (If the flag is not given, the secure FS image will be integrity protected only).

The <key_path> refers to a file consisting of a 128-bit key and the user can generate it via the `occlum gen-image-key <key_path>` command.

After generating the encrypted FS image, the init process is responsible for mounting the encrypted FS image as the rootfs for the user's Application. Usually the key should be acquired through RA or LA, please take the init_ra as an example to use this feature in real world.

1.16 Remote Attestation

Occlum provides wrapped library `libocclum_dcap` for `DCAP` remote attestion applications. This Occlum DCAP library is prebuilt as part of the Occlum toolchains in the Occlum Docker images.

The libraries are in the path `/opt/occlum/toolchains/dcap_lib`.

```
.
|-- glibc
|   |-- dcap_test
|   |-- libocclum_dcap.a
|   `-- libocclum_dcap.so
|-- inc
|   `-- occlum_dcap.h
`-- musl
    |-- dcap_test
    |-- libocclum_dcap.a
    `-- libocclum_dcap.so
```

Two versions (glibc and musl-libc), including static and dynamic libraries are provided to meet different scenarios. Unified header file `occlum_dcap.h` is provided as well in which defines the exported APIs for DCAP quote generation and verification.

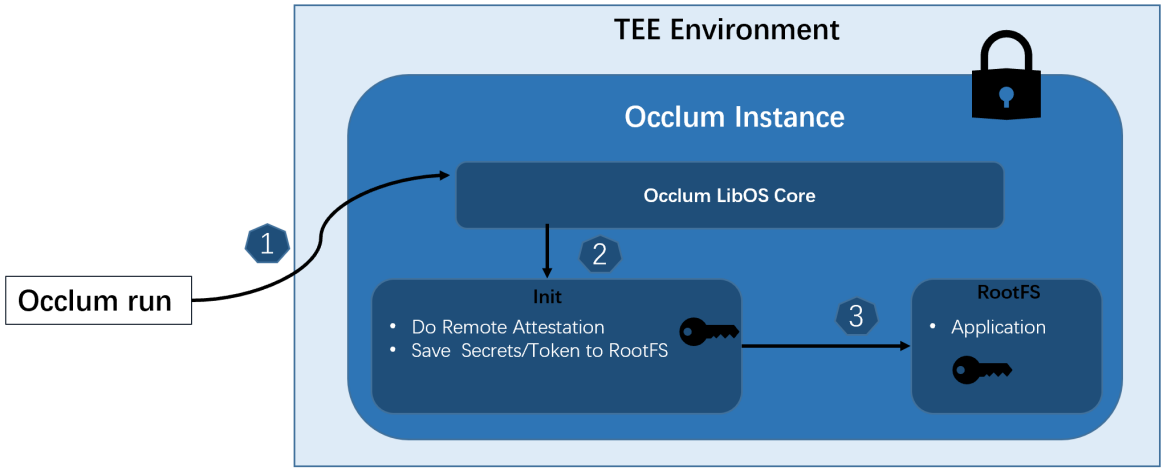
In short, applications can link to the prebuilt `libocclum_dcap.so` and use the APIs defined in `occlum_dcap.h` for their usage.

For details how to use the library, please refer to the demo.

The source code of the library is in the path.

1.16.1 Init RA Solution

Occlum also has a unique "Occlum -> init ->application" boot flow. Generally, all operation which is application required but not part of the application, such as remote attestation, could be put into `init` part. This feature makes Occlum highly compatible to any remote attestation solution without involving application's change.



init_ra_flow

This design off load the remote attestation burden from application. Two RA solutions are provided for reference. Details please refer to *doc*

1.16.2 Azure Attestation

To support Azure Attestation, there are some demos provided. Users could choose each one to their actual applications. Details please refer to the demo azure_attestation.

1.16.3 SGX KSS (Key Separation and Sharing feature) support

Starting from SGX2, there is a new Key Separation and Sharing feature which provides more flexibility. The new feature gives user a chance to fill in some meaningful information to the enclave either in the signing or running stage.

- Signning stage:

```
ISVFAMILYID, 16 bytes
ISVEXTPRODID, 16 bytes
```

- Running stage:

```
CONFIG ID, 64 bytes
CONFIG SVN, 16 bits
```

Occlum can support both above by either modifying the fields in `Occlum.json` (for `Signning stage`) or using Occlum run arguments `"--config-id"` or `"--config-svn"` (for `Running stage`).

Details please refer to the RFC.

1.16.4 References

- DCAP Quick Install Guide
- Intel(R) Software Guard Extensions Data Center Attestation Primitives

1.17 Init RA Solutions

There are two **Init-RA** solutions provided, **GRPC-RATLS** and **AECS**.

With these two solutions, two customized **init** are provided. Thus users don't need modify the **init** by themselves. Users only need fill in some fields in the template `init_ra_conf.json`

1.17.1 AECS Init-RA

AECS is a short name of **Attestation based Enclave Configuration Service**. Basically, part of its function is acting as a remote attestation based key management service.

Occlum provides a way to embed the AECS client function in Occlum Init process by simply running `occlum new occlum_instance --init-ra aecs` to initiate an Occlum instance.

Then, users can modify the template `init_ra_conf.json` in oclcum_instance accordingly.

```
{
    "kms_server": "localhost:19527",
    "kms_keys": [
        {
            "key": "demo_key",
            "path": "/etc/demo_key",
            "service": "service"
        }
    ],
    "ua_env_pccs_url": "",
    "ra_config": {
        "ua_ias_url": "https://api.trustedservices.intel.com/sgx/dev/attestation/v4",
        "ua_ias_spid": "",
        "ua_ias_apk_key": "",
        "ua_dcap_lib_path": "",
        "ua_dcap_pccs_url": "",
        "ua_uas_url": "",
        "ua_uas_app_key": "",
        "ua_uas_app_secret": "",
        "ua_policy_str_tee_platform": "",
        "ua_policy_hex_platform_hw_version": "",
        "ua_policy_hex_platform_sw_version": "",
        "ua_policy_hex_secure_flags": "",
        "ua_policy_hex_platform_measurement": "",
        "ua_policy_hex_boot_measurement": "",
        "ua_policy_str_tee_identity": "",
        "ua_policy_hex_ta_measurement": "",
        "ua_policy_hex_ta_dyn_measurement": "",
        "ua_policy_hex_signer": "",
        "ua_policy_hex_prod_id": "",
```

(continues on next page)

(continued from previous page)

```
        "ua_policy_str_min_isvsvn": "",
        "ua_policy_hex_user_data": "",
        "ua_policy_bool_debug_disabled": "",
        "ua_policy_hex_hash_or_pem_pubkey": "",
        "ua_policy_hex_nonce": "",
        "ua_policy_hex_spid": ""
    }
}
```

The json file specifies:

- The URL of the KMS server (AECS).

"kms_server" in the json file, which can be overwritten by environment value **OCCLUM_INIT_RA_KMS_SERVER** when running.

- The secrets users need aquire and where to put.

"kms_keys" part. It can define multiple keys to be aquired from KMS server (AECS), and the paths to save the keys. This part should align with the keys injected into KMS server (AECS).

- The PCCS URL.

"ua_env_pccs_url". It should be the same with the "pccs_url" in the file /etc/sgx_default_qcnl.conf. It also could be overwritten by environment value **UA_ENV_PCCS_URL** when running.

- The measurement of the KMS server (AECS) to be trusted.

"ra_config" part defines the information of the KMS server (AECS) to be trusted. Users could ignore this part if **"kms_server"** is guaranted to be trusted. Otherwise, some fields, ususally **ua_policy**_* measurements, are expected correspoding values – RA measurement values from the correct KMS server (AECS).

There is a demo init_aecs_client for reference.

1.17.2 GRPC-RATLS Init-RA

It is based on a GRPC-RATLS implementation.

Occlum provides a way to embed the AECS client function in Occlum Init process by simply running `occlum new occlum_instance --init-ra grpc_ratls` to initiate an Occlum instance.

Then, users can modify the template `init_ra_conf.json` in oclcum_instance accordingly.

```
{
    "kms_server": "localhost:50051",
    "kms_keys": [
        {
            "key": "demo_key",
            "path": "/etc/demo_key"
        }
    ],
    "ra_config": {
        "verify_mr_enclave" : "off",
        "verify_mr_signer" : "off",
        "verify_isv_prod_id" : "off",
        "verify_isv_svn" : "off",
        "verify_config_svn" : "off",
```

(continues on next page)

(continued from previous page)

```
        "verify_enclave_debuggable" : "off",
        "sgx_mrs": [
            {
                "mr_enclave" : "",
                "mr_signer" : "",
                "isv_prod_id" : 0,
                "isv_svn" : 0,
                "config_svn" : 0,
                "debuggable" : true
            }
        ]
    }
}
```

The json file specifies:

- The URL of the KMS server (GRPC RA Server).

"kms_server" in the json file, which can be overwritten by environment value **OCCLUM_INIT_RA_KMS_SERVER** when running.

- The secrets users need aquire and where to put.

"kms_keys" part. It can define multiple keys to be aquired from KMS server (GRPC RA Server), and the paths to save the keys. This part should align with the keys injected into KMS server (GRPC RA Server).

- The measurement of the KMS server (GRPC RA Server) to be trusted.

"ra_config" part defines the information of the KMS server (GRPC RA Server) to be trusted. Users could ignore this part if **"kms_server"** is guaranted to be trusted. Otherwise, some fields are expected correspoding values – RA measurement values from the correct KMS server (GRPC RA Server).

Details please refer to the demo init_ra_flow.

1.18 Demos

This part introduces sample projects that demonstrate how Occlum can be used to build and run user applications. Full source code could be found on github.

1.19 Benchmark Demos

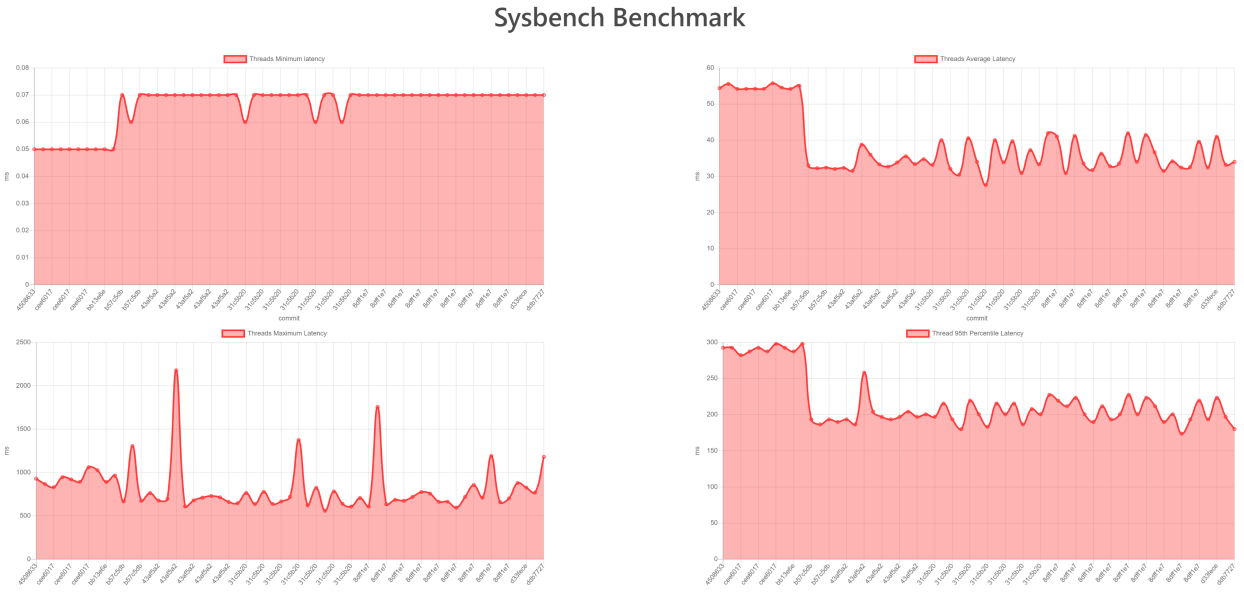
This set of demos shows how commonly used benchmarking tools can be run inside SGX enclaves with Occlum.

- filebench: A demo of Filebench, a file system and storage benchmark tool.
- fio: A demo of Flexible I/O Tester.
- iperf2: A demo of Iperf2, a tool for measuring Internet bandwidth performance.
- iperf3: A demo of Iperf3, a tool for measuring Internet bandwidth performance.
- sysbench: A demo of Sysbench, a scriptable multi-threaded benchmark tool for Linux.

1.19.1 Benchmarks Data

There are two enabled **benchmarks CI** for continuous benchmarking. It utilizes the `github-action-benchmark` to provide a chart view for visualized historical benchmarks data on the GitHub pages.

The CI runs periodically. For example, **sysbench** has the historical benchmarks chart as below.



sysbench_chart

CI and Data for Master branch

benchmarks CI

History Data

CI and Data for Dev branch

benchmarks CI

History Data

1.20 Tests for Occlum

1.20.1 Unit Tests

There is a set of unit tests in the source **test**. It includes almost all the syscall (Occlum supported) test cases. Every PR will run this unit tests to make sure of no failures introduced on basic functions.

Users could run the unit test manually as well.

```
// run all the unit test cases for musl-libc
# make test

// run all the unit test cases for glibc
```

(continues on next page)

(continued from previous page)

```
# make test-glibc

// run only specified test case, timerfd for example
# TESTS=timerfd make test

// run test cases for 100 times
# make test times=100

// run test without rebuilding Occlum, using binaries installed already
# OCCLUM_BIN_PATH=/opt/occlum/build/bin make test
```

1.20.2 Gvisor Tests

1.20.3 LTP Tests

Linux LTP is the most popular test suite for Linux. Occlum could also apply the LTP to verify the stability and compatibility to Linux app. For detail, please refer to **linux-ltp**.

1.21 Builtin Toolchains

Occlum provides multiple builtin toolchains or libraries in the Occlum development docker image to ease the porting or developing effort for users. Part of them are toolchains used to recompile or repackage the applications to make them runnable in Occlum TEE environment. The others are some frequently-used auxiliary libraries which can be directly used or linked by applications running in Occlum.

All the build scripts could be found on github.

1.21.1 Toolchains

Generally, Occlum supports both musl-libc and glibc based toolchains. It is up to the users to descide which one to be chosen.

- Less memory footprint, perhaps musl-libc based.
- More compatible to existed applications, maybe glibc based.

glibc

To support running glibc based application in Occlum, a customized glibc is provided in the Occlum development docker image.

The main changes on top of general glibc are as below:

- Redirect syscalls into Occlum
- Support posix_spawn syscall for Occlum
- Modify vdso to calling syscalls in Occlum

To users, all the glibc libraries to be used in Occlum need to be replaced by the ones in `/opt/occlum/glibc/lib/`.

Thus, Occlum can support `gcc` compilation with **PIE** enabled, which make it compatible to popular compile systems.

golang

To support compiling and running Golang in Occlum LibOS, a customized go is provided in the Occlum development docker image, path `/opt/occlum/toolchains/golang`. Every Golang to be executed in Occlum needs to be recompiled by **occlum-go** (a wrapper of go).

Currently Occlum supports two versions of Golang, 1.16 and 1.18 (default one in Occlum development docker image) which are both linked to musl-libc in default. But users can easily configure the **occlum-go** to use **gcc** like below.

```
# CC=gcc occlum-go build
```

java

There are three JAVA versions provided in the Occlum development docker image, path `/opt/occlum/toolchains/jvm`.

Both the unmodified openjdk 8/11 in the table are directly imported from Alpine:3.11.

The dragonwell-jdk11 for enclave is a musl-based JDK version compatible with the Alpine Linux and Occlum, and it's an open source project, see Alibaba Dragonwell.

By default Occlum uses Dragonwell JDK11 as the default JDK. Thus `occlum-java` and `occlum-javac` (provided in the Docker image path **/opt/occlum/toolchains/jvm/bin/**) use Dragonwell JDK11.

Besides the musl-libc based JDK (less memory footprint), glibc based JDK are also supported. Users are free to change to other JDK version by setting the `JAVA_HOME` to point to the installation directory of OpenJDK and copying it into Occlum instance.

musl-gcc

To support compiling and running musl-libc based application in Occlum, a customized musl is provided in the Occlum development docker image.

To users, all the musl-libc libraries to be used in Occlum need to be replaced by the ones in `/usr/local/occlum/x86_64-linux-musl/lib/`.

Moreover, wrapped **occlum-gcc**, **occlum-g++** and **occlum-ld** are provided as well to do the recompiling if necessary. Any applications generated by these wrapped tools, are expected to run successfully in Occlum.

rust

Occlum supports general glibc based rust tools such as **cargo** and **rustc**.

Wrapped **occlum-cargo** and **occlum-rustc** are also provided to do musl-libc based rust compilation, which can be found on the path `/opt/occlum/toolchains/rust/bin/` in the Occlum development docker image.

1.21.2 Auxiliary Libraries

Besides toolchains, several auxiliary libraries to be used in Occlum are provided as well to ease the development effort.

bash

To support running bash shell script in Occlum, a customized bash is provided in the Occlum development docker image. Both musl-libc and glibc versions are provided in the path `/opt/occlum/toolchains/bash`. Users can use it directly for application in Occlum, details could refer to demo bash.

busybox

To support running general CLI commands in Occlum, a prebuilt busybox is provided in the Occlum development docker image. Both musl-libc and glibc versions are provided in the path `/opt/occlum/toolchains/busybox`. Users can use it directly for application in Occlum, details could refer to demo bash.

DCAP library

Occlum provides wrapped library `libocclum_dcap` for `DCAP` remote attestion applications. Both musl-libc and glibc versions are provided in the path `/opt/occlum/toolchains/dcap_lib`

```
.
|-- glibc
|   |-- dcap_test
|   |-- libocclum_dcap.a
|   `-- libocclum_dcap.so
|-- inc
|   `-- occlum_dcap.h
`-- musl
    |-- dcap_test
    |-- libocclum_dcap.a
    `-- libocclum_dcap.so
```

Two versions (glibc and musl-libc), including static and dynamic libraries are provided to meet different scenarios. Unified header file `occlum_dcap.h` is provided as well in which defines the exported APIs for DCAP quote generation and verification.

In short, applications can link to the prebuilt `libocclum_dcap.so` and use the APIs defined in `occlum_dcap.h` for their usage.

For details how to use the library, please refer to the demo.

The source code of the library is in the path.

1.22 Copy Bom

The copy_bom tool is designed to copy files described in a bom file to a given dest root directory.

1.22.1 Bom file

Bom file is used to describe which files should be copied to the root directory(usually, the image directory). A bom file contains all files, directories that should be copied to the root directory. We also can define symbolic links and directories that should be created in a bom file. The meanings of each entry in the bom file can be seen in RFC: bom file. Also, a bom example with all optional entries can be seen in **example.yaml**.

```
# include other bom files
includes:
  - base.yaml
  - java-11-alibaba-dragonwell.yaml
# This excludes will only take effect when copy directories. We will exclude files or␣
↪dirs with following patterns.
excludes:
  - .git
  - .dockerignore
targets:
  # one target represents operations at the same destination
  - target: /
    # make directory in dest: mkdir -p $target/dirname
    mkdirs:
     - bin
     - proc
    # build a symlink: ln -s $src $target/linkname
    createlinks:
      - src: ../hello
        linkname: hello_softlink
    copy:
      # from represents the prefix of copydirs and files(to copy)
      # If there's no copydirs or files, copy the *ENTIRE from directory* to target: cp -
↪r $from/ $target
      - from: .
        # copy directory: cp -r $from/dirname $target
        dirs:
          - hello_c_demo
          - example_dirname
        # copy file: cp $from/filename $target
        files:
          - Makefile
          - name: Cargo.toml
            hash: DA665E483C11922D07239B1A04BEE0F0C7C1AB6D60AF041DDA7CE56D07AF723E
            autodep: false
            rename: Cargo.toml.backup
  - target: /bin
    mkdirs:
      - python-occlum
      - python-occlum/bin
  - target: /etc
```

(continues on next page)

(continued from previous page)

```
  copy:
    - dirs:
      # If there's a '/' as the postfix in directory name, copy the contents in
      # directories, not including the directory itself.
      # cp -r /etc/opt/ /etc
      - /etc/opt/
```

1.22.2 copy_bom

overview

copy_bom is the tool designed to create directories and symbolic links, copy all files and directories defined in a bom file to the root directory. Internally, copy_bom will use rsync to do the real file operations. copy_bom will copy each file and directory incrementally, i.e., only changed parts will be copied. The permission bits and modification times will be reserved. This is done by the -a option of rsync. copy_bom will not ensure the whole image directory as described in bom file (sync behavior) because it will not try to delete old files. To pursue a sync behavior, one can delete the old image directory and copy files again.

dependencies

copy_bom will analyze all dependencies(shared objects) of each ELF file. copy_bom will analyze dependencies for each user-defined file in files entry as well as files in user-defined directory in dirs entry. For user-defined elf file, it will report error and abort the program if we can't find the dependent shared objects. For files in user-defined directories, we will report warning if autodep fails. We analyze dependencies via the dynamic loader defined in the .interp section in elf files and automatically copy dependencies to the root directory. If there's no .interp section for an elf file, copy_bom will try to infer the loader if all other elf files have the same loader. Currently, copy_bom only copy dependencies with absolute paths. We support only one dependency pattern in the result of dynamic loader.

- name => path e.g., libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6All dependencies will be copied to the corresponding directory in root directory. For example, if root directory is image, then the dependency /lib64/ld-linux-x86-64.so.2 will be copied to image/lib64/ld-linux-x86-64.so.2. An entry named autodep with value false can be added to each file to avoid finding and copying dependencies automatically.

log

copy_bom uses the same log setting as occlum. One can set OCCLUM_LOG_LEVEL=trace to see all logs printed by copy_bom. To only view real file operations, OCCLUM_LOG_LEVEL=info is a proper level.

prepare and install

- **Prepare**

Since copy_bom relies on rsync to copy files. We need to install rsync at first. On ubuntu, this can be done by apt install rsync -y.

- **Install**.

copy_bom is part of the occlum tools which could be found on /opt/occlum/build/bin. The occlum tools are preinstalled in occlum development docker image or installed by apt install -y occlum.

basic usage

`copy_bom [FLAGS] [OPTIONS] --file <bom-file> --root <root-dir>`

- bom-file: The bom file which describes files we want to copy.
- root-dir: The destination root directory we want to copy files to. Usually the `image` directory for occlum.

options

- dry run mode: pass an option `--dry-run` to `copy_bom` will enable dry run mode. Dry run mode will output all file operations in log but does not do real operations. It is useful to check whether `copy_bom` performs as expected before doing real operations.

flags

- -i, –include-dir: This flag is used to indicate which directory to find included bom files. This flag can be set multiple times. If the `include-dir` is set as a relative path, it is a path relative to the current path where you run the `copy_bom` command.
- -h, –help: print help message

About the `occlum_elf_loader.config` file

This file is put in `/etc/template`. This file is used to define where to find occlum-specific loaders and occlum-specific libraries. If you want to find libraries in different paths, you should modify this config file.

The file content looks like as below:

```
/opt/occlum/glibc/lib/ld-linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu:/lib/x86_64-linux-
↪gnu
/lib/ld-musl-x86_64.so.1 /opt/occlum/toolchains/gcc/x86_64-linux-musl/lib
```

Each line in this file represents a loader. Since occlum supports both musl and glibc loader, there are two loaders in the config file now.Each line contains two parts, which is separated with a space. The first part is the path of occlum-specific loader. This loader is used to analyze dependencies of elf file.The second part in the line indicates where to find shared libraries. All paths should be separated by colons. The loader will first try to find libraries in the loader path, then will try to find libraries in user-provided path. This is done by set the `LD_LIBRARY_PATH` environmental variables. The order of paths matters, since we will find libraries in the order of given path.

1.22.3 known limitations

- The use of wildcard(like *) in files or directories is not supported. It may result in `copy_bom` behaving incorrectly. To achieve a similar purpose, directly add `/` after the directory name to copy all contents in the directory while not copying the directory itself.
- If we create symbolic link in bom file, it will always delete the old link and create a new one. It will change the modification time of the symbolic link.
- Environmental variables pointing to an empty value may fail to resolve.

1.22.4 Demos

Now all the Occlum demos are using **copy_bom** tool to generate Occlum file system. There is also a tutorial to give insight of Occlum instance generation using **copy_bom**.

1.23 Q&A

1.23.1 Does an Occlum instance directory correspond to only one binary, or does an Occlum instance directory contain multiple binaries? If yes, do they all belong to the security zone?

Occlum is a multiprocess LibOS, which means that a user could add multiple executables into one Occlum instance and all those applications are able to work together. Technically, all those applications are running in the same Enclave, so there are in the same security zone.

1.23.2 If there are two executables in an Occlum instance, is it possible to execute both of them by using occlum_pal_exec()?

The short answer is yes. But the user could only run one of them by occlum_pal_exec() at a time. If you want to run both of them at the same time, one of the application could spawn the other, or you could prepare a script to launch them in the script one by one.

1.23.3 Is there a way to share memory between an Enclave with Occlum instance?

No, Occlum does not support shared memory with other Occlum instances or enclaves.

1.23.4 How many CPU cores can be used inside Occlum/Enclave?

The host OS manages the CPU resource, and doing the scheduling. So it is totally controlled by host OS that how many cpus are running the applications inside Occlum.

1.23.5 Can Occlum support running network related applications?

Yes. Generally, the network related applications can run successfully in Occlum without modification. Just one note, besides the application itself, multiple files/directories may be required to be existed in Occlum image as well. For example,

- **`hostname`, `hosts` and `resolv.conf` files**

Generally, these files (in the host environment) are automatically parsed and transferred to Occlum LibOS for each `Occlum run` operation.

- **DNS related files**

Add below part in the bom file if required.

```
- target: /opt/occlum/glibc/lib
  copy:
    - files:
```

(continues on next page)

(continued from previous page)

```
        - /opt/occlum/glibc/lib/libnss_files.so.2
        - /opt/occlum/glibc/lib/libnss_dns.so.2
        - /opt/occlum/glibc/lib/libresolv.so.2
```

- **CA related files**

Add below part in the bom file if required.

```
- target: /etc
  copy:
    - dirs:
      - /etc/ssl
```

1.23.6 How to modify the default timezone in Occlum?

In Occlum, default timezone is Coordinated Universal Time (UTC). Users could do below to modify the timezone accordingly.

For example, **Asia/Shanghai** is expected timezone. Put below in the bom file and do copy_bom again, then rebuild the occlum instance.

- For glibc application:

```
- target: /opt/occlum/glibc/etc
  copy:
    - files:
      - name: /usr/share/zoneinfo/Asia/Shanghai
        rename: localtime
```

- For musl-libc application:

```
- target: /etc
  copy:
    - files:
      - name: /usr/share/zoneinfo/Asia/Shanghai
        rename: localtime
```

Above two could be verified by demo bash. Just adding above timezone parts into corresponding bom file, rebuild and try below command to make sure the timezone is as expected.

```
occlum run /bin/busybox date
```