Keystone Enclave Documentation

Release 0.1.1

Dayeol Lee, David Kohlbrenner, Dawn Song, Krste Asanovic

Dec 04, 2018

Getting Started:

CHAPTER 1

Getting Started

1.1 What is Keystone Enclave

Keystone Enclave is an open source secure enclave for RISC-V processors. You can migrate the Keystone enclave into arbitrary RISC-V processor, with a very small modification on hardware to plant the silicon root of trust.

Attention: We're actively adding more documents right now. Please post in Keystone forum or create a GitHub issue if you face any undocumented trouble.

Attention: The current version (0.X) of Keystone is not formally verified, nor matured, which means that it might have bugs or unknown security holes. We recommend you to use Keystone only for research purposes until it gets stablized. We appreciate any contribution for making Keystone better.

Since no hardware has been built for Keystone, we provide a few ways to deploy and test Keystone for building secure systems and developing enclave applications.

1.2 Testing Keystone on Various Platforms

1.2.1 Running Keystone with QEMU

QEMU is an open source machine emulator. The latest QEMU supports RISC-V ISA.

Keystone is tested in the latest RISC-V QEMU (GitHub). The upstream QEMU might not work because it has a bug in the PMP module (*See GitHub issue* <>. The fix will be upstreamed in the future.

Installing Dependencies

We tested Keystone with QEMU on CentOS and Ubuntu 16.04/18.04

Cent OS

```
sudo yum install autoconf automake autotools-dev bc bison build-essential \
curl expat expat-devel flex gawk gcc gcc-c++ git gperf libgmp-dev libmpc-dev \
libmpfr-dev libtool mpfr-devel texinfo tmux patchutils zlib1g-dev zlib-devel \
wget bzip2-devel lbzip2 patch
```

Ubuntu

```
sudo apt update
sudo apt install autoconf automake autotools-dev bc bison build-essential curl \
expat libexpat-dev1 flex gawk gcc git gperf libgmp-dev libmpc-dev libmpfr-dev \
libtool texinfo tmux patchutils zlib1g-dev wget bzip2 patch vim-common lbzip2 \
python pkg-config libglib2.0-dev libpixman-1-dev
```

Compile Sources

Clone the repository

```
git clone https://github.com/keystone-enclave/keystone
cd keystone
git submodule update --init --recursive
```

Install RISC-V GNU Toolchain

```
mkdir riscv
export RISCV=$(pwd)/riscv
export PATH=$PATH:$RISCV/bin
cd riscv-gnu-toolchain
./configure --prefix=$RISCV
make && make linux
cd ..
```

This step installs RISC-V GNU toolchain in the `keystone/riscv` directory.

To keep environment variables, add `export PATH=$PATH:<path/to/keystone>/riscv/bin` to your `.bashrc`. You can also manually run `source source.sh` to set the environment variables.

Create Disk Image using Busybear

See Busybear repo for more information.

```
cd busybear-linux
make
cd ..
```

Build RISC-V QEMU

You should apply patches before building the QEMU.

```
./scripts/apply-patch.sh
cd riscv-qemu
./configure --target-list=riscv64-softmmu,riscv32-softmmu
make
cd ..
```

Build Linux Kernel with Built-in Keystone Driver

```
cd riscv-linux
cp ../busybear-linux/conf/linux.config .config
make ARCH=riscv olddefconfig
make ARCH=riscv vmlinux
cd ..
```

Build Berkeley Bootloader (BBL) with Keystone Security Monitor

Make sure to add `--enable-sm` when you run `configure` so that the security monitor is included in the bbl.

```
cd riscv-pk
mkdir build
cd build
../configure \
    --enable-logo \
    --host=riscv64-unknown-elf \
    --with-payload=../../riscv-linux/vmlinux \
    --enable-sm
make
```

Build Root-of-Trust Boot ROM

```
cd bootrom
make
cd ..
```

Build Keystone SDK

Keystone SDK includes sample enclave programs and some useful libraries. To run sample programs, you should compile SDK library and apps, and copy all of them into the disk image. Following commands will compile the sdk, and copy sample binaries into the `busybear.bin` disk image.

```
cd sdk
make
make copy-tests
cd ..
```

Launch QEMU

Now, you're ready to run Keystone.

The following script will run QEMU, start executing from the emulated silicon root of trust. The root of trust then jumps to the SM, and the SM boots Linux!

```
./scripts/run-qemu.sh
```

Login as `root` with the password `busybear`.

You can exit QEMU by `ctrl-a``+``x`

Run Tests

You can run Keystone enclaves by using an untrusted host application. We already implemented a simple host `test-runner.riscv` for running tests. Following command will create and execute the enclave.

```
./test-runner.riscv <user elf> <runtime elf>
```

To run all tests, you could simply run

```
./test
```

1.2.2 Running Keystone with FireSim

FireSim is an FPGA-based cycle-accurate simulator for RISC-V processors. Using FireSim, you can test Keystone on open-source processors like RocketChip or BOOM.

Who needs it?

If you want to run your enclave application with Keystone, but you don't own any RISC-V processor, FireSim is the way to go. FireSim allows you to simulate the processors with reasonably high speed. You can actually boot Linux on the simulated processor and run real workloads. You can test functionality or measure the performance of Keystone enclaves. If you want to improve your enclave system by modifying hardware, you can freely modify the processor hardware, and deploy it to Amazon AWS FPGAs using FireSim.

Setting Up FireSim Manager Instance

Before we start, you have to create a FireSim manager instance. See FireSim Documentation to setup a manager instance. Be sure to use 1.4.0 or later version of FireSim.

Building Keystone Software

We have already packed every software required for running sample Keystone enclaves. Add a remote to the firesim-software by executing following commands:

```
cd firesim/sw/firesim-software
git remote add keystone https://github.com/keystone-enclave/firesim-software
git fetch keystone
```

Checkout `firesim-1.4.0-keystone` branch and update submodules.

```
git checkout firesim-1.4.0-keystone
git submodule sync --recursive
git submodule update --init --recursive
```

Build Boot Image

First, we need to build the Linux kernel with built-in Keystone module, and the Berkeley Bootloader (bbl) containing the Keystone security monitor. This command will compile both `riscv-pk` and `riscv-linux`, and create a bootable image. It also build `buildroot` to get a disk image.

```
./sw-manager.py build
```

Build Keystone SDK

Now, you're ready for launching Keystone. We provide sample enclaves with Keystone SDK, so let's build the enclaves and copy them to the disk image. Build the Keystone SDK by running following commands:

```
cd sdk
make
```

Next, we will copy the binaries into the disk image from the previous part. Open `Makefile` with any text editor, and change `DISK_IMAGE` parameters to `../images/br-disk.img`.

```
DISK_IMAGE = ../images/br-disk.img
```

Save the change, and run

```
make copy-tests
```

This command copies all of the test binaries and runtime into the disk image.

Launching Simulation

Use FireSim commands to launch the simulation. Go to the top-level FireSim directory and run:

```
cd <path/to/firesim>
source sourceme-f1-manager.sh
```

Choose hardware configuration in `deploy/config_runtime.ini`. See FireSim Single Node Simulation for more details.

Currently, Keystone works on a singlecore Rocket (e.g., `firesim-singlecore-no-nic-lbp`). Use this `runtime_config.ini` file:

```
# RUNTIME configuration for the FireSim Simulation Manager
# See docs/Advanced-Usage/Manager/Manager-Configuration-Files.rst for documentation␣
↪of all of these params.

[runfarm]
runfarmtag=mainrunfarm

f1_16xlarges=0
```

(continues on next page)

(continued from previous page)

```
m4_16xlarges=0
f1_2xlarges=1

runinstancemarket=ondemand
spotinterruptionbehavior=terminate
spotmaxprice=ondemand

[targetconfig]
topology=no_net_config
no_net_num_nodes=1
linklatency=6405
switchinglatency=10
netbandwidth=200
profileinterval=-1

# This references a section from config_hwconfigs.ini
# In homogeneous configurations, use this to set the hardware config deployed
# for all simulators
defaulthwconfig=firesim-singlecore-no-nic-lbp

[tracing]
enable=no
startcycle=0
endcycle=-1

[workload]
workloadname=linux-uniform.json
terminateoncompletion=no
```

Launch runfarm and test!

```
firesim launchrunfarm
firesim infrasetup
firesim boot
```

You can login to the f1 instance via `ssh` and attach to the simulated node using `screen` command. See FireSim Single Node Simulation for more details.

```
[On your manager instance]
ssh <f1 instance ip address>
```

```
[On the f1 instance]
screen -r fsim0
[Login via root/firesim]
```

Running Keystone Enclaves

The home directory must include SDK sample enclaves and the runtime.

```
[On the simulated node]
# ls
aes.riscv                fibonacci.eapp_riscv  test
attestation.eapp_riscv   long-nop.eapp_riscv   test-runner.riscv
c.eapp_riscv             loop.eapp_riscv       untrusted.eapp_riscv
```

(continues on next page)

(continued from previous page)

```
eyrie-rt                malloc.eapp_riscv
fib-bench.eapp_riscv    stack.eapp_riscv
```

Run `./test-runner.riscv` for testing each enclave.

```
./test-runner.riscv stack.eapp_riscv eyrie-rt
```

Run `./test` to run all enclaves sequentially.

```
./test
```

CHAPTER 2

A Guide to Keystone Components

The Keystone repository (https://github.com/keystone-enclave/keystone) consists of a number of sub-components as gitmodules or directories. This is a brief overview of them.

2.1 bootrom

Keystone bootrom, including trusted boot chain.

2.2 busybear-linux

Link: https://github.com/keystone-enclave/busybear-linux

Unmodified busybear Linux, supporting riscv. Our default untrusted Linux OS for testing.

2.3 docs

Contains read-the-docs formatted and hosted documentation, such as this article.

2.4 riscv-gnu-toolchain

Unmodified toolchain for building riscv targets. Required to build all other components.

2.5 riscv-linux

Link: https://github.com/keystone-enclave/riscv-linux

Linux kernel with riscv patches, updated to mainline semi-regularly. Only Keystone modification is the Keystone driver, in arch/riscv/drivers/.

2.6 riscv-pk

Link: https://github.com/keystone-enclave/riscv-pk

The proxy-kernel for machine-mode riscv. Trusted core component of Keystone, and includes the security monitor (in sm directory).

2.7 riscv-qemu

Qemu with riscv patches and minor modifications to better support PMP and Keystone needs for emulation. Our default testing platform.

2.8 sdk

Link: https://github.com/keystone-enclave/keystone-sdk

Tools, libraries, and tests for building applications on Keystone.

2.8.1 Runtime

Link: https://github.com/keystone-enclave/keystone-runtime

A submodule that implements the default minimal Keystone runtime running in S-mode for enclaves.

2.9 keystone-demo

Link: https://github.com/keystone-enclave/keystone-demo

A complete demo application using Keystone. Hosts an enclave that performs computation over data provided by a remote client using secure channels.

CHAPTER 3

FAQ

Q: What is the difference between the SM and runtime?

- A: The SM is part of the Keystone TCB, and is trusted by all components. The runtime is a (technically) optional part of the enclave itself. While the enclave app trusts it, it is not part of the trusted boot process and is not part of the Keystone TCB.

CHAPTER 4

SDK Overview

The Keystone SDK provides the tools required to build basic enclave hosting applications (*hosts*) and enclave applications (*eapps*).

The SDK consists of 4 main components, and the tests and examples.

- Host libraries (*lib/host*)
- Enclave Application libraries (*lib/app*)
- Edge libraries (*lib/edge*)
- Runtime (*runtime/*)

4.1 Host Libraries

The host libraries provide an interface for managing enclave applications via the *Keystone* class. Most of this library will work regardless of the runtime, but is tied directly to the kernel driver provided in the *riscv-linux* repository branches.

4.2 EApp Libraries

The eapp libraries provide both simple enclave tools (EXIT, etc) as well as some basic libc-style functions (malloc, string headers, etc).

4.3 Edge Libraries

The edge libraries provide features to both eapps and hosts for managing edge calls. Edge calls are function calls that cross the enclave-host boundary. Currently we only support calls from enclave->host. You can emulate host->enclave calls via polling on this interface. The edge libraries are used in many places, including the runtime and both host and eapp libraries.

4.4 Runtime

The runtime is the system level code that runs in the enclave. It handles the enclave entry point and basic system calls for the userland enclave, as well as all edgecall data transfers.

CHAPTER 5

Writing A Simple Application

The best way to see how to write a simple application is to look a the *untrusted* test eapp, and the *keystone-demo* repository.

A full system for using a Keystone enclave consists of possibly writing 3 things:

- Host (userspace, outside enclave, untrusted)
- Runtime (system level, inside enclave, trusted) - most users will not modify this
- Enclave app (userspace, inside enclave, trusted)

Most users will only need to write a simple Host, and use the default Keystone runtime. The bulk of the work is done in the enclave application, and in the glue that holds the components together.

NOTE: Right now all "edge" components (that is, anything that handles data and calls between enclave code and non-enclave code) is hand-written. A major next step for the SDK is to build a set of tools and compilers to do the majority of this code generation automatically.

5.1 Writing a host

Most host functionality is contained in the *Keystone* C++ class in the host library. To start an enclave application, first create one:

```
Keystone enclave;
Params params;

params.setEnclaveEntry(ENTRY_VADDR);
enclave.init(PATH_TO_EAPP_ELF, PATH_TO_RUNTIME, params);

edge_init(&enclave);
```

You can also set other enclave options via the params object, including stack size and shared memory size. *edge_init* is an edge wrapper function that registers edge call endpoints in the Keystone driver.

Then you start the enclave and transfer control with:

```
enclave.run()
```

At this point, control is transferred to the enclave runtime, and then application (if using the default runtime). Generally the first action after this would be to have the enclave send its attestation report to the host. Control will be transferred back to the host code either when the enclave exits, or when an edgecall is made.

5.2 Writing an eapp

An enclave application has a main equivalent:

```
void EAPP_ENTRY eapp_entry(){}
```

This is the entry point that the runtime will start at in the enclave application.

As with the host, the first action should be:

```
edge_init()
```

Which is another edge call wrapper function to register call points and setup buffers. Once complete, the first action should be to make an edgecall to the host to present the attestation report.

After this point, all functionality is up to the application developer. See the *keystone_demo* repository for an example application.

CHAPTER 6

Compiling Applications

Building a host and enclave application using the SDK is straight-forward. See Keystone-Demo as an example.

6.1 Toolchain

All compilation will need to be done using the riscv64- toolchain. This is provided by the riscv-gnu-toolchain submodule, and must be built first.

6.2 Libraries

Hosts and enclave applications will want to link against the edge library `libkeystone-edge.a`

Hosts will want to link against the host library `libkeystone-host.a`

Applications will want to link against the enclave app library `libkeystone-app.a`

6.3 Applications

Eapps need to be linked in a specific way. As this may change, please see the `app.lds` linker script in the sdk/tests directory to see the most up to date linking requirements.

CHAPTER 7

Edge Calls

In Keystone, as in other enclave systems, function calls that cross in or out of the enclave are called *edge calls*.

For example, if an enclave wishes to send a network packet, it must communicate the data to transmit to an untrusted host process via an edge call.

The current version of Keystone supports calls from enclave->host, referred to internally as *ocalls* (outbound calls, names under discussion).

If your application requires behavior similar to calls from host->enclave we suggest emulating these with a polling ocall. (Keystone-Demo uses this methodology with a `wait_for_message` ocall).

All ocall wrapping code currently passes data through shared memory regions. When referencing data in these regions only offsets into the region are used, never virtual address pointers.

7.1 Example ocall Lifecycle

Note: This is quite specific to the Keystone runtime and driver design.

Consider an example "print unsigned long" ocall, `print_value`. This call exports a value from the enclave to be printed to stdout by the host process.

The eapp calls `ocall_print_value(val);` which is a edge wrapper function.

`ocall_print_value(val)` uses the system-call-like interface to the runtime to run an ocall like `ocall(OCALL_PRINT_VALUE, &val, sizeof(unsigned long), 0, 0);` It passes a pointer to the value, the size of the argument, and any needed return buffer information. (None in this case)

The runtime then allocates an `edge_call_t` structure in the shared memory region, fills out the call type, copies the value into another part of the shared memory, and sets up the offset to the argument value. Note that edge calls do not use pointers, but instead offset values into the shared memory region.

Finally, the runtime exits the enclave with an `SBI_CALL`, passing a value indicating the enclave is not shutting down, but executing an ocall.

The Keystone kernel driver resumes execution, checks the exit status of the enclave, notes a pending ocall, and passes execution to the userspace host process.

The userspace host process consumes the `edge_call_t` and dispatches the registered ocall handler wrapper for `OCALL_PRINT_VALUE`. The wrapper generates a pointer to the argument value from the offset in the shared memory region, and calls `print_value` with the value as an argument.

`print_value` prints the given argument value.

On return, the host wrapper checks if any return values need to be copied into the shared memory region (none in this case.) Sets the `edge_call_t` return status to SUCCESS, and returns into the driver.

The driver re-enters the enclave runtime via an `SBI_CALL`.

The runtime checks if any return information needs to be copied from the shared region into return buffers (none in this case) and then resumes the enclave ocall wrapper code.

Finally, the ocall wrapper code passes any return values to the function that first called `ocall_print_value`.

CHAPTER 8

Attestation

Keystone support a simple attestation scheme using ed25519 signatures on hashes of the security monitor and enclave content.

See Keystone-Demo for the most up-to-date example of how the attestation flow works for applications.

8.1 Using Attestation

To use remote attestation in Keystone a remote client needs 3 things:

- A public key for the device root private key
- An expected hash for the enclave they wish to launch
- An expected hash for the security monitor
- A copy of the enclave report after launch

Once you have these items, the `sdk/lib/verifier` library provides the tools to validate the report.

8.2 Device Root Keys

For security guarantees to hold, the platform must support storage of a device root key accessible only to the bootloader/sm. Keystone cannot provide this support on its own.

For testing purposes we provide the ability to hardcode a device public/private key into the bootloader. This keypair is in `bootrom/test_dev_key.h`.

8.3 Generating Enclave Attestation Report

Once an enclave has started, it may request the SM to provide a signed enclave report and signed SM report:

SM Report contains:

- A hash of the SM
- The attestation public key

All signed by the device root key.

Enclave report contains:

- A hash of the enclave at initialization
- A data block from the enclave of up-to 2KB in size

All signed by the attestation public key.

The verifier, when provided with the device public key, expected SM hash, and expected enclave hash, will verify the signatures these reports.

8.4 Enclave Hashes

Currently generating the expected enclave hash value is a manual process. Keystone-Demo has an example of a way to do this

1. Modify enclave host application to emit the hash from given report.
2. Run modified host and application in qemu
3. Save emitted hash value on the remote client

Keystone will be adding support for a more streamlined hash generation process in the future.

8.5 Security Monitor Hashes

The Keystone repository will provide in the riscv-pk subproject an `sm_expected_hash.h` file that should contain the current expected hash value of the security monitor as a C array.

If modifications are made to the SM for testing, the hash can be obtained in the same way as the expected enclave hash.

CHAPTER 9

Keystone Demo

The Keystone Demo is an example of the current capabilities of the Keystone enclave framework. The demo consists of:

- A server application (server-eapp)
- An untrusted host (enclave-host)
- A “dummy client” for local testing (dummy-client)
- A remote client for demonstration of full remote attestation (trusted-client)

This demo shows how a remote client can request computation to be performed on an untrusted server using an enclave.

Note: The demo uses test keys and is not safe for production use.

9.1 Server eapp

The demo server enclave application, basic enclave features (attestation report generation, etc), a simple word-counting feature, and uses `libsodium` for establishing a secure channel.

The enclave first sends a copy of its attestation report, along with its eccdh public key to the client. Upon receving the client public key, it establishes a secure channel and asks the enclave-host to wait for messages.

Once it has received a message, it authenticates and decrypts the message.

If successful, it supports two message types:

1. word-count of arbitrarily sized block of text
2. exit

For word count, it performs a simple word counting calculation, and returns the result over the secure channel.

9.2 Enclave host

The host serves two functions: starting the enclave, and proxying network messages.

It can also use the 'dummy client' in which case it sends messages to the dummy client object, and not over the network. This is useful for testing in a single process in qemu.

9.3 Dummy Client

A simple, single file copy of the client that runs locally in the host. Useful for test only.

9.4 Trusted Client

A simple remote client that connects to the host, validates the enclave report, constructs a secure channel, and then can send messages to the host for computation.

CHAPTER 10

Building the Demo

Building the Keystone Demo application and testing it can be slightly complex on real hardware. Testing locally with qemu is straightforward.

10.1 Building dependencies

The demo relies on the full Keystone SDK, as well as `libsodium` for cryptographic support. Currently we require two separate builds of `libsodium`, one for the client, and one for the eapp. See the up-to-date build instructions in the subdirectory in the demo repository. We suggest keeping two copies of the `libsodium` repository to make modifications easier.

10.2 Building for qemu

With a successfull `libsodium` and SDK build, building for qemu is simple.

Set the required environement variables:

- `KEYSTONE_SDK_DIR`: should point to the base of the SDK repo
- `LIBSODIUM_DIR` : Should point to `libsodium/src/libsodium/` for the eapp targeted verion
- `LIBSODIUM_CLIENT_DIR` : Should point to `libsodium/src/libsodium/` for the client version

You can either build the regular version and remote trusted client with `make && make trusted_client.riscv` or modify the Makefile lines that are commented out containing refernces to the dummy client to build a single-file test.

Copy relevant binaries (enclave-host.riscv, enclave-host-dummy-client.riscv, server-eapp.eapp_riscv) to the sdk/bins/ dir, then run `make copy-tests` in the sdk directory. Running the qemu as normal should now have the enclave-host binaries available.

10.3 Building for HiFive Unleashed

First, you will need to get a working custom first-stage bootloader (FSBL) working on your board. This will require creating a new partition on your SD card as well as setting the MSEL2 dipswitch. See https://github.com/sifive/freedom-u540-c000-bootloader/issues/9#issuecomment-424162283 for details.

Using the keystone-hifive-unleashed repository, build all components.

Copy the new fsbl into the partition as created above, and set the MSEL2 switch. The board should now boot with a signed security monitor and device keys.

At this point, you can copy over the testing binaries built in the same way as for qemu and run them on the board.

This process can be quite long and tricky, if you run into problems please contact us.

CHAPTER 11

How to Contribute

Keystone is, and will be, fully open source. We hope to see many projects both under the Keystone name, and built on-top of it.

We welcome contributions at all levels, from platform specific bootloader modifications, to improved compilers and toolchains, to support for novel use-cases for end users.

11.1 Platform Specific Builds/Changes

Many SoCs and boards will require some amount of custom support for Keystone to function fully. If you are interested in ensuring Keystone operates correctly on your target there are several requirements:

- PMP support
- RISCV priv-1.10 ISA support
- Entropy sources
- >=3-run level support (U/S/M)

Where possible, changes that allow Keystone to operate on another platform should be cross-platform themselves, to avoid fragmentation.

See also the list of known areas to work on, we need better platform specific builds!

11.2 Known Issues

Check out the github isses pages for all Keystone subprojects to report problems or find places to contribute.

11.3 General Contributions

Submit pull-requests, with a good explanation, and following the coding guidelines.

CHAPTER 12

Future Objectives and Features for Keystone

This is a list of larger features or changes that Keystone may need. These are not bugs.

12.1 Entropy

Secure cryptography requires a dependable entropy source. All platforms that have such a hardware source should integrate it as a platform-specific SM feature.

Other platforms will need to use more intrusive entropy gathering strategies. See the Linux jitter entropy source, or the Welcome to the Entropics paper.

12.2 Multi-threading models

12.3 Formal Verification

12.4 Task/Message Queueing for the Keystone Runtime

12.5 Edge compiler and DSL/Toolchain

Similar to SGX, we need tools to help generate code for edgecalls between the enclave and host. Currently all such code is manually generated.

12.6 Misc

- Better ELF entry point detection and handling (more configurable)
- Better/more complex ELF loading for eapps

- Scheduling interfaces for closer runtime/os collaboration
- New shared-memory usage models