HIPv2

TDDE21 Advanced Project

Gustav Arneving, Mohammad Borhani, Shiwei Dong,
Simon Johansson, Victor Löfgren

December 5, 2022

Contents

1 Introduction **4**

1.1 HIP . . . 4

1.2 HIPv2 . . . 4

1.3 OpenSSL . . . 4

1.4 CORE . . . 4

1.5 Docker . . . 5

1.6 Review of previous groups . . . 6

1.7 Environment . . . 6

2 Milestones and Contributions **7**

2.1 HHIT . . . 7

2.1.1 Structure of HHIT . . . 7

2.1.2 Xoodyak . . . 7

2.1.3 Implement HDA and RAA . . . 7

2.1.4 Test . . . 8

2.2 Automatic Test and Docker Files . . . 9

2.2.1 Unittests . . . 9

2.2.2 Wireshark . . . 9

2.2.3 Docker Files . . . 9

2.2.4 CORE changes . . . 10

2.3 Debugging Nodes . . . 10

2.4 OpenSSL . . . 10

2.4.1 Our contribution . . . 11

3 Discussion and Further Work **11**

3.1 HHIT . . . 11

3.2 OpenSSL . . . 12

4 Practical Advice **13**

4.1 Installing using Docker . . . 13

4.2 Debugging and generating logs . . . 13

4.3 Installing OpenSSL 3 13

1 Introduction

1.1 HIP

The Host Identifier Protocol (HIP) is a new secure communication protocol designed to distinguish between an identifier and an address identifier[4]. The HIP identifier is a cryptographic address, more specifically, public cryptographic keys, used to indicate the sockets located at the application layer. Each host using the HIP protocol can establish an encrypted channel for secure communication through the IPsec protocol. The HIP protocol also solves the two Internet challenges of multi-homing and mobility by separating the identifier and locator.

The base exchange is depicted in Figure 1.

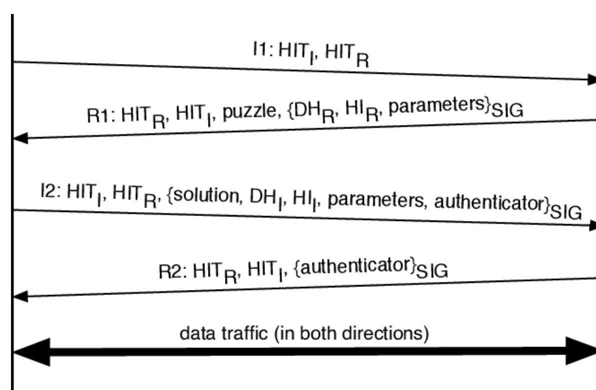


Figure 1: HIP BEX

1.2 HIPv2

Researchers proposed a second HIP version following the original HIP in 2015[1]. This version of the Internet draft is more practical and detailed than the previous version, focusing on the generation of cryptographic identifiers, the process of base exchange, and the generation and exchange of Diffie-Hellman keys.

1.3 OpenSSL

OpenSSL is a robust, commercial-grade, full-featured toolkit for general-purpose cryptography and secure communication which is licensed under an Apache-style license[2]. It can be used in commercial and non-commercial scenarios.

OpenSSL 3.0 is the latest LTS version, with OpenSSL 1.1.1 going end-of-life September 2023. Version 1.1.1 is currently used extensively in the project, and it is therefore of huge interest to us to upgrade to version 3.

1.4 CORE

The common open research emulator (CORE) is a real-time network emulator that allows rapid instantiation of hybrid topologies composed of both real hardware and virtual network nodes[1]. CORE uses

[1] `https://www.rfc-editor.org/rfc/rfc7401`
[2] `https://www.openssl.org`

FreeBSD network stack virtualization to extend physical networks for planning, testing, and development, without the need for expensive hardware deployments.

In this project, we chose to skip the CORE GUI and use the class properties defined in the CORE directly in the python program instead. The Python API in CORE is used to create each HIP node, build switches, and other components to form a customized isolated test environment.

1.5 Docker

An open platform for creating, distributing, and operating applications is called Docker. It can also be seen as a command-line program with a daemon that runs in the background and a collection of remote utilities that employ a methodical approach to address common software issues and streamline the user experience of installation and configuration, running, publicizing, and uninstalling applications. A Docker container is a software bucket that contains all the dependencies a piece of software needs to run on its own. Additionally, Docker's exclusion and safety features let you run various containers at once on a single host machine.

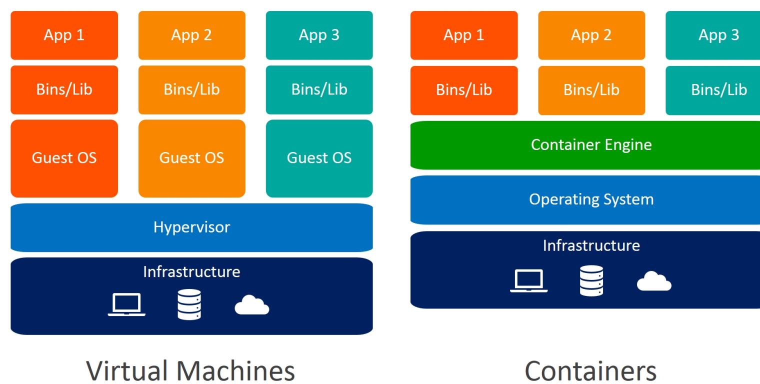


Figure 2: Docker compared to VMs

Docker allows users to create containers, which are instantiated images, and lightens the whole process, making it easier and more convenient for users and enabling multiple containers and images to be edited[3]. Since docker leverages the use of the container in contrast to virtual machine settings, the deletion of the guest os layer, as shown in Fig 2, can bring better performance when compared to virtual machine deployment using VirtualBox or Vmware. The full list of comparisons can be seen in Fig 3.

	Docker	Virtual Machines (VMs)
Boot-Time	Boots in a few seconds.	It takes a few minutes for VMs to boot.
Runs on	Dockers make use of the execution engine.	VMs make use of the hypervisor.
Memory Efficiency	No space is needed to virtualize, hence less memory.	Requires entire OS to be loaded before starting the surface, so less efficient.
Isolation	Prone to adversities as no provisions for isolation systems.	Interference possibility is minimum because of the efficient isolation mechanism.
Deployment	Deploying is easy as only a single image, containerized can be used across all platforms.	Deployment is comparatively lengthy as separate instances are responsible for execution.
Usage	Docker has a complex usage mechanism consisting of both third party and docker managed tools.	Tools are easy to use and simpler to work with.

Figure 3: Benefits of Docker [5]

[3]https://docs.docker.com

Client-server architecture is used by Docker. The Docker client communicates with the Docker daemon, which manages the creation, execution, and distribution of your Docker containers. User may connect a Docker client to a distant Docker daemon or execute the Docker client and daemon on the same machine. UNIX sockets, or a REST API are used by the Docker client and daemon for communication. The architecture of docker is shown in Figure 4 [3].

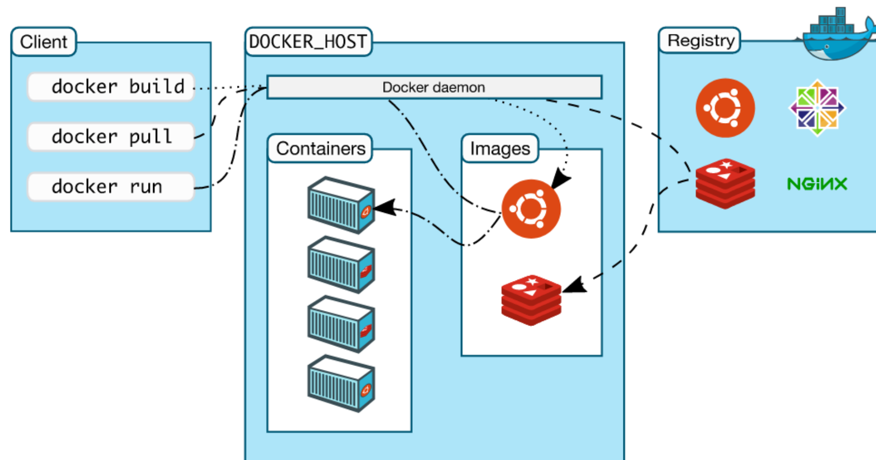


Figure 4: Architecture of Docker [3]

1.6 Review of previous groups

Before we implemented the HIP, four groups worked on the HIP enhancement from 2018 to 2021 respectively. We also reviewed and summarized the work of the previous groups before we officially started our work. The review shows that each group targets different enhancement directions.

The 2018 group modified the puzzles in BEX, improving them to more flexible I and J parameters and specifying new puzzle lengths and solution lengths. In addition, they added a section on ECDSA in the implementation generation configuration files and dynamized the HIT suite ID. They also upgraded the OpenSSL from v1.0.2 to v1.1.0 and recommended that successor groups upgrade to a higher version.

The 2019 group upgraded the HIT suite negotiations to make the protocol more flexible and supplement the start phase, but the complete HIT suite negotiations needed to have worked correctly. In addition, they corrected the ICMP parameters when the HIP versions did not match. In terms of testing, the group did not make significant progress, but they recommended that subsequent groups can use unittest for testing.

The 2020 group found other Internet drafts for HHIT and used them to improve HHIT's functionality. The use of ECDSA was extended to generate HI. in addition. They also used the unittest library to implement basic tests.

The 2021 group focused on implementing NAT Traversal and attempted to upgrade the OpenSSL version to v3.0.0. Although they were not completely successful in upgrading, they proposed an idea: to translate functions and data structures with the exact implications but different representations in the new version compared to the old version.

1.7 Environment

The OpenHIP runs in Ubuntu 20.04, with CORE installed to stimulate nodes. Since CORE can only run on Linux systems, the most common method is to use virtual box, install Ubuntu 20.04 with good compatibility, and then install CORE, OpenHIP, and OpenSSL according to the official instructions.

2 Milestones and Contributions

2.1 HHIT

Before the interim report, we carefully read the HIP and HHIT-related RFCs and Internet drafts and searched deeper according to the Internet draft chosen by the 2020 group. We found the Hierarchical Host Identity Tags (HHIT) Verification Architecture latest version `draft-kuptsov-hhit-05`[4].

The Host Identity Tag (HIT) from the Host Identity Protocol (HIP) provides a self-asserting Identity through a public key signing operation using the Host Identity's (HI) private key. And using HIT ensures that a device Identity has some degree of permanency.

The Hierarchical HIT (HHIT) is a small but important enhancement over the flat HIT space. By adding two levels of hierarchical administration control, the HHIT provides for device registration/ ownership, thereby enhancing the trust framework for HITs.

2.1.1 Structure of HHIT

Based on the newest definition, A HHIT is built from the following fields:

- 28 bit IANA prefix: 2001:20::/28
- 4 bit HITSuiteID: specifies the hash algorithms.
- 32 bit Hierarchy ID (HID)
- 64 bit ORCHID hash: Hash_function specified in HITSuiteID

The illustration is below.

2.1.2 Xoodyak

After a long time of inspection, we develop the code when HIT suite ID == 6, with a new sponge function called Xoodyak. This implementation works in both flat HIT and HHIT. Most changes are in the c files `hitgen.c` and `hit_utils.c`.

The Xoodyak sponge function is a candidate in the NIST Lightweight Cryptography (LWC) Standardization process[2].

The hash algorithm is shown below.

$\textsc{Cyclist}(\epsilon, \epsilon, \epsilon)$ {initialization in hash mode}
$\textsc{Absorb}(x)$ {absorb string x}
$h \leftarrow \textsc{Squeeze}(n)$ {get n bytes of output}

2.1.3 Implement HDA and RAA

The Hierarchy ID (HID) provides the structure to organize HITs into administrative domains.

[4] `https://datatracker.ietf.org/doc/draft-kuptsov-hhit/`

HIDs are further divided into 2 fields:

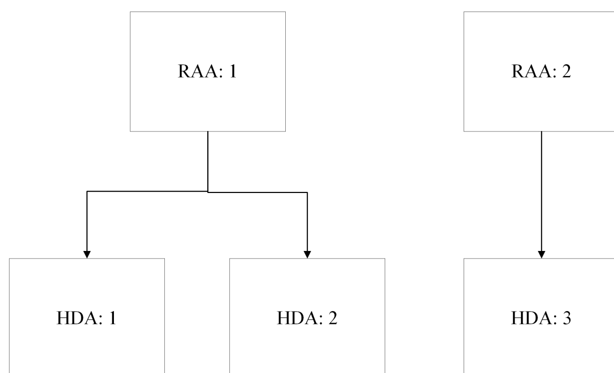
- 16 bit Registered Assigning Authority (RAA)
- 16 bit Hierarchical HIT Domain Authority (HDA)

An RAA is a business or organization that manages a registry of HDAs. (65,536 RAAs)

An HDA may be an ISP or any third party that takes on the business to provide RVS and other needed services for HIP-enabled devices. (65,536 HDAs per RAA)

Since no group had previously implemented HDA and RAA, this was the first time our group had implemented them in this direction. Since we did not have an official mandate to create an official HDA and RAA, we manually assigned HDA and RAA to each test node by hardcoding a static hierarchy in OpenHIP. We created three HDAs (1, 2, 3) and two RAAs (1, 2), where HDAs 1 and 2 belong to RAA1 and HDA3 belongs to RAA2.

The illustration is below.



2.1.4 Test

The previous group implemented unit tests to test OpenHIP. The advantage of this approach is that we can quickly create all kinds of scenarios for comprehensive testing. However, the disadvantage is that we can only see whether the test passed (the output result is OK or Error), which prevents us from getting more information from the output result. Therefore, based on Unittest, we carefully study the code and usage of the Unittest library and break down its execution process into several steps to output all kinds of information when the HIP protocol is executed. In the test for HHIT, we can choose to output HHIT with different HIT suite IDs and nodes using different HDA and RAA to confirm that our implementation is valid.

```
HIT for n0: 2001:31:1:1:3e09:af50:ff1:914c
HIT for n1: 2001:31:2:3:3e95:876:cc6f:89d5
```

Additionally, it is significant to test xoodyak, which is a novel hash function. Keccak team established a repository called The eXtended Keccak Code Package (XKCP) which includes xoodyak function. We use self-tests in this repository to launch the test.

```
shiweidong@shiweis-mbp XKCP % ./bin/generic64/UnitTests --Xoodyak
* Xoodyak: 32-bit optimized implementation
    - OK
```

2.2 Automatic Test and Docker Files

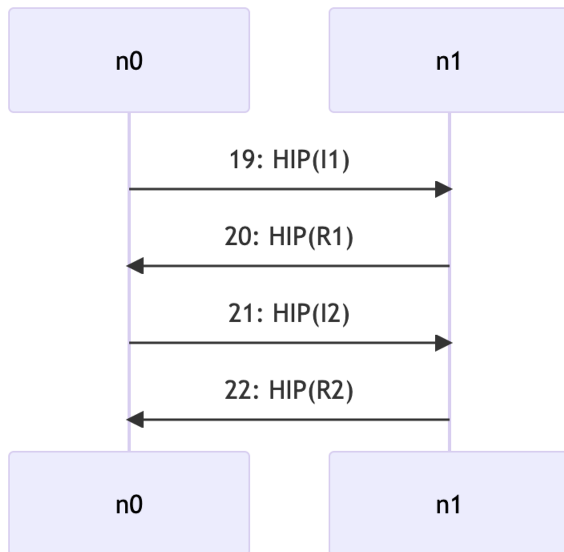
2.2.1 Unittests

In addition to the previously mentioned problem of not being able to get the internal information, we found that the previous group hardcoded the HIT suite ID, forcing the parameters to be HIT suite ID == 5 when `conf=True` whatever the parameters are, and then setting the flag to 5 in the python test file to pass the test. But fixing the parameters will cause hipnodes which use other HIT suite IDs to get errors at runtime. So we extended the suite list in `hipnode.py` and pass in the parameters in the test to achieve different test conditions, enabling any suite id to be tested.

2.2.2 Wireshark

In order to better obtain the internal information of the nodes using the HIP protocol while communicating, the group use Wireshark to capture packets. Wireshark is an open-source tool commonly used for network troubleshooting analysis. The new process is as follows. First, start Wireshark while generating hipnode, listen to two interfaces of each node: hip0 and eth0, and create pcap files respectively. Then, at the end of the base exchange of the two nodes, the two packets of each node are combined and de-duplicated to form a merged packet, and the pcap file is automatically translated into an intuitive diagram.

The following diagram shows the successful establishment of the connection between the two nodes.



Also, if the communication fails, we can get more specific information on the error message and locate exactly which phase failed.

2.2.3 Docker Files

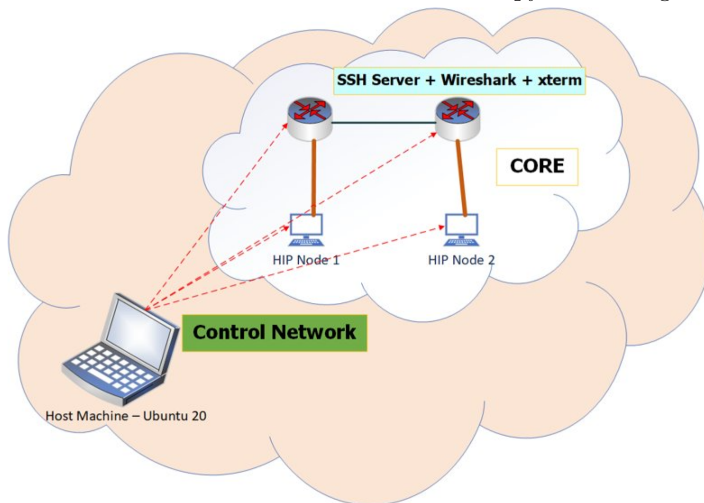
We tried to make a working docker file, including CORE for simulating hipnode and OpenHIP for communication. Since OpenHIP was first written in 2012, some dependencies were outdated, so the latest version of CORE didn't fit and we needed to find an older version of CORE, which is v7.5.2. Since the GitHub page for CORE does not provide a direct link to download the old version, we spent a lot of time finding and compiling the old version of CORE, and finally, we built a docker file where both CORE and OpenHIP work. The main issue that we faced was that the docker container needs to run using the privileged flag because of internal commands used in CORE.

The size comparison of the docker file shows that the image of DockerFile needs approximately 950MB while the VM image needs about 4.8 GB.

See Section 4.1 for instructions on how to get the project up and running using Docker.

2.2.4 CORE changes

In the base configuration, the CORE runs the node in an isolated environment. Hence, there exists no way to control the CORE nodes (e.g., HIP nodes) directly from the host machine. However, for better visibility and debugging goals, we sometimes need to control each machine as a way of investigating the error. We add the below items to the CORE python configuration, as depicted in the below illustration.



- We add ControlNet to the CORE so as to make a network connection between host nodes and CORE nodes
- Each Core Node has a built-in SSH server that is accessible from the host machine
- Since each CORE node can be controlled via SSH access, Wireshark and xterm can be installed on each node for testing purposes
- You can see more details about these changes by considering CORE changes video.
- The python scripts used in the video can be accessed by Test-switch.py and Test-casecore.py

2.3 Debugging Nodes

One of the biggest issues during the whole project was that it proved hard to debug the code when running multiple nodes at once. Therefore we spent quite some time creating a method for easily debugging each node. Using the VS Code editor it is easy to start debugging, the file `openhip/test/README.md` contains information on how to start a debugging session.

2.4 OpenSSL

For OpenSSL, debugging is a huge undertaking. The previous groups have tried but made no significant progress. Group 2018 began with 1.0.2 and attempted to upgrade to 1.1.0 but didn't finish. Group 2019 ignored the previous work, but upgraded v1.0.2 to v1.1.0 successfully. Group 2021 made a version using OpenSSL 3.0.0 that compiles, but it isn't tested.

Our original plan was to install OpenSSL 3 on Ubuntu 20.04, compile and run OpenSSL 3.0.5 and the existing OpenSSL 1.1.0, and compare the differences in output between them. Also, to get a better understanding of OpenSSL in OpenHIP, we also plan to document all functions or data structures from the OpenSSL library used in OpenHIP. However, we encountered many unanticipated difficulties during installation, compilation and running, so we finally changed our goal to mainly debug OpenSSL 3.0.7.

We initially tried to map out every part of the code that needed changes, without trying to compile or run it. This effort proved rather meaningless, because after running it and seeing what exactly the program did, it was a lot easier to understand.

We began our development by trying to install OpenSSL 3, which proved rather cumbersome. See Section 4.3 for brief instructions on how to do it.

2.4.1 Our contribution

We based our work off of `OpenSSL-3.0.0`-branch, which contained many changes, seeing as it tried to convert from OpenSSL 1 to 3. Although most of it were useful to some degree, nothing seemed to work completely. We therefore began by running the program, seeing how far it got, and fixed the issue. This process was repeated until we ran out of time, at which point the base exchange worked, but the messages afterwards weren't encrypted/decrypted properly.

We chose to get one single path working, i.e. one alternative for every option. This means that the combination of RSA, HIT_SUITE=1, HIP_CIPHER=2, DH_GROUP=10 and ESP_TRANSFORM=1 almost works in the OpenSSL 3 version. What the former numbers correspond to can be found in HIPv2 RFC[5]. This strategy was in stark contrast to the previous group, who instead tried to fix every path without having the ability to test it.

Testing our code was a high priority for us. To facilitate this, we made it possible to choose what algorithms the program supported via a configuration file. The hit_suite list and esp_transform were already there, so we just simply added the rest of them in the same fashion. We also made it possible to automatically change this configuration file in the Python test script. This meant that we could turn on only those algorithms that we focused on, as previously mentioned, and basically ignore the rest. In this way, we made it easier to incrementally update the program, instead of tackling everything at once.

Another issue we ran into was that the program simply crashed when we ran it. It was initially hard to find the location of the error, but we developed the ability to run a debugger on each node. This proved tremendously helpful, since when the program crashed, we could see instantly where in the code it happened. However, the reasons for crashing were often unclear, and we spent most of the time just figuring out what was wrong.

3 Discussion and Further Work

In this section, we analyze both HHIT and OpenSSL, mainly based on our existing work, and give feasible suggestions.

3.1 HHIT

At the moment we can manufacture HHIT correctly, but during the session there's something wrong with configuration files.

Currently, we use hardcode to build HDA and RAA, perhaps later groups could define it in a more flexible way (e.g. automatic assignment based on user requirements).

[5]`https://www.rfc-editor.org/rfc/rfc7401.html`

3.2 OpenSSL

Even though we didn't manage to implement a fully working example, we still believe our contributions will be valuable. We hope that our example of almost working OpenSSL 3 code can be of help to the next group and can serve as inspiration for how to implement the other alternatives. In some cases, the different options will look rather similar, and in others, less so. We also believe that our added configuration and debugging support will make developing and testing a lot easier than before.

4 Practical Advice

This section contains some instructions for using Docker and installing OpenSSL 3.

Some actual advice is also in its place: get the program up and running as fast as possible, instead of spending time in the beginning just looking at the code. As soon as it runs, it will be a lot easier to tell what's going on, where things go wrong and what to work on it.

4.1 Installing using Docker

Docker seems to work best on Ubuntu, but you can try to get it to work on Windows as well if you want. Usage of the Dockerfile is simple. Prior to executing the below commands, you need to install docker in Ubuntu using the Docker documentation. For a video demonstration, you can see Docker Video. We should note that in the video presentation, the docker installation commands have been skipped. Hence, you need to consider the below commands before running the Docker container.

```
# Clone the OpenHIP project
git clone https://bitbucket.org/openhip/openhip.git
cd openhip/

# Switch to branch containing Docker
# NOTE: In the Dockerfile the checked out branch of OpenHIP needs to be specified
git switch docker-xoodyak-hhit-openssl3
cd docker/

docker build -t core .
# NOTE execute the below in-line command
docker run -itd --cap-add=NET_ADMIN
--privileged --device=/dev/net/tun --name=coree
--cap-add=SYS_ADMIN -p 5900:5900 -p 8080:8080 core
```

4.2 Debugging and generating logs

As mentioned previously, debugging, diagrams and log files have proven very useful during our development process, and we would recommend further developers to, as early as possible, get familiar with the debugger and the terminal logs, packet logs, and diagram logs to aid during the development process. For using the debugger, we refer you to the readme file in the tests directory of OpenHIP, alternatively, see this video. To generate terminal logs and diagrams, see this video where we demonstrate how to run the test and where the log files are located. To properly visualize the diagram logs, a tool for rendering `.mmd` files are needed, during our project we used the VS Code extension `tomoyukim.vscode-mermaid-editor`.

4.3 Installing OpenSSL 3

We initially had some trouble installing OpenSSL 3, therefore we provide a short guide here. Replace the specific version number with the version you want to install. You don't need to type sudo when running inside Docker.

```
# Install OpenSSL 3
sudo apt install -y build-essential checkinstall bridge-utils net-tools psmisc zlib1g-dev
cd /usr/local/src/
wget https://www.openssl.org/source/openssl-3.0.7.tar.gz
```

```
sudo tar -xvf openssl-3.0.7.tar.gz
cd ./openssl-3.0.7/
./Configure
make
sudo make install
export LD_LIBRARY_PATH="/usr/local/lib64"

# Compile OpenHIP with OpenSSL 3
cd ~/openhip/ # Where you cloned OpenHIP to
./bootstrap
./configure
make
sudo make install
```

References

[1] Jeff Ahrenholz et al. "CORE: A real-time network emulator". In: Nov. 2008, pp. 1–7. ISBN: 978-1-4244-2676-8. DOI: 10.1109/MILCOM.2008.4753614.

[2] Joan Daemen et al. "Xoodyak, a lightweight cryptographic scheme". In: (2020).

[3] *Docker Documentation*. Website. 2022. URL: https://docs.docker.com/.

[4] Pekka Nikander, Andrei Gurtov, and Thomas R. Henderson. "Host Identity Protocol (HIP): Connectivity, Mobility, Multi-Homing, Security, and Privacy over IPv4 and IPv6 Networks". In: *IEEE Communications Surveys & Tutorials* 12.2 (2010), pp. 186–204. DOI: 10.1109/SURV.2010.021110.00070.

[5] *VMs vs. Containers*. Website. 2021. URL: https://www.backblaze.com/blog/vm-vs-containers/.