Improving the Open Source HIPv2 implementation

TDDE21 - Secure Distributed and Embedded Systems

Erik Delbom, Axel Hammarberg, Hampus Holm, Martin Kaller

December 16, 2024

Contents

1 Introduction and Background

This report describes the work done on the OpenHIP project in the course TDDE21 - Advanced Project: Secure Distributed and Embedded Systems, at Linköping University. OpenHIP is an open-source project that aims to implement an open-source version of the Host Identity Protocol Version 2.

1.1 Host Identity Protocol

Host Identity Protocol (HIP) is a secure network protocol that aims to solve a number of issues with the current network architecture. One issue is that an IP address is used both as identifier and location, which inhibits the mobility of devices. HIP solves this by using a Host Identity (HI) for identification and the IP address for location only. This makes it possible to maintain a connection while the IP address changes, improving mobility. The identification is implemented by an asymmetric key encryption and the communication between hosts is performed with a symmetric key encryption.

1.2 OpenSSL

The OpenSSL library is a programming library consisting of cryptographic algorithms to meet the requirements of many internet standards. OpenSSL aims to comply with standards such as the Federal Information Processing Standards (FIPS), established by the National Institute of Standards and Technology (NIST) and the Canadian Centre for Cyber Security (CCCS), as well as comply with select Request for Comments (RFCs) established by the Internet Engineering Task Force (IETF) [1, 2].

OpenHIP complies with HIPv2 and uses OpenSSL for cryptography: including generating Host Idenities and Host Idendity Tags, signing and authenticating messages, exchanging keys during the HIP base exchange, and symmetric encryption for communication after the base exchange. Some algorithms used are RSA, EdDSA, and SHA among others, and their usage is specified in Host Identity Protocol (HIP) version 2 [14, 6].

1.3 HIT

To identify a host, a Host Identity Tag (HIT) RFC-7401 [14] is used. It is a type of Overlay Routable Cryptographic Hash Identifiers Version 2 (ORCHIDv2) RFC-7343 [9]. The format of the HIT is the following:

IPv6 prefix	Suite ID	HASH
28 bits	4 bits	96

The HIT is the HIP prefix of 2001:20 and the Suite ID is a reference to the cryptographic algorithm used to make the hash. To produce the hash the IPv6 prefix, Suite ID and public key is used as input to a hash function. For the cryptography algorithms ECDSA and EdDSA the curve id is added to the input. The format for the hash input is therefore as follows:

IPv6 prefix	Suite ID	Public key
28 bits	4 bits	n bits

or

IPv6 prefix	Suite ID	Curve label	Public key
28 bits	4 bits	16 bits	n bits

1.4 HHIT

Hierarchical Host Identity tags (HHIT) [13] is a type of ORCHIDv2, similar to HIT. However, ORCHIDv2 has been extended [13] to include the optional info field. The HHIT has the following format:

IPv6 prefix	HID	HHSI	HASH
28 bits	28 bits	8 bits	64 bits

A new prefix of 2001:30 is introduced to differentiate A HHIT from a flat HIT. The optional info field is populated with a 28 bit Hierarchy ID (HID) which consists of 14 bits of Registered Assigning Authority (RAA) and 14 bits of HHIT Domain Authority (HDA). An 8-bit HHIT Suite ID (HHSI), similar to the Suite ID of HIT. The hash input differs from HIT, as it has the following format:

IPv6 prefix	HID	HHSI	Curve label	Public key
28 bits	28 bits	8 bits	16 bits	n bits

Note that the curve label and public key are the input when EdDSA is used, which is the only valid cryptographic algorithm for HHIT [13].

1.5 CORE

Common Open Research Emulator(CORE) is an open-source tool for creating virtual networks developed by U.S. Naval Research Group [8]. In this project, it is used for testing and debugging HIP. It can be used by running the script called `debug.py` that sets up a network and emulates it. A graphical interface (GUI) can also be used to set up a network for testing.

1.6 PyHIP

PyHIP is an open-source Python implementation [15] of HIPv2, designed to run on routers. The following sections describe the tools and libraries specifically used for the PyHIP implementation in this project.

1.6.1 Mininet

Mininet [11] is a network emulation tool used to create and test virtual networks on a single computer. It supports realistic simulations of hosts, switches, and links, making it useful for developing and testing network configurations and applications.

1.6.2 PyCryptodome package

PyCryptodome [5] is a Python library for cryptographic operations, offering features like encryption, decryption, hashing, and digital signatures. It supports algorithms such as AES, RSA, and SHA. It is a popular choice for secure data processing and is often used as a replacement for the older PyCrypto library.

1.6.3 Cryptography package

The Cryptography library is a Python package for encryption, decryption, hashing, and digital signatures. It supports algorithms like AES, RSA, and SHA-256. It provides both high-level and low-level APIs for cryptographic operations. Its reliability and modern standards make it a popular choice for secure data handling.

1.6.4 CProfile

cProfile is a built-in Python module used for performance profiling. It helps developers analyze the execution time of Python programs by providing detailed statistics on function calls, including the time spent in each function and how often they are called. This allows for identifying bottlenecks and optimizing code for better performance. cProfile is easy to use and can be integrated directly into Python scripts or run from the command line to analyze performance.

1.7 GitLab group

GitLab groups are used to organize multiple repositories in one place. A group lets several members be owners, so they can manage and work on the same set of repositories without needing to fork them all the time. This makes it easier to keep all the source code together and simplifies collaboration and project management for teams.

In this project, a GitLab group is used to organize all the repositories, ensuring that all related code is stored in one location for better coordination and accessibility.

2 Milestones

The following milestones outline the key objectives to be completed in this project. These milestones guide the development process, focusing on critical tasks such as upgrading OpenSSL, implementing new standards, and optimizing performance.

1. Complete migration of OpenSSL from 1.1.1 to 3.0.
2. Verify implementation of Hierarchical Host Identity Tags defined in RFC-9374 [13].
3. Merge the branches of HHIT and OpenSSL.
4. Performance improvement of PyHIP
5. Diet-ESP
6. Latest crypto support

2.1 OpenSSL

At the start of the project, OpenSSL was in process of a major version migration. This year the work on migrating from OpenSSL 1.1.1 to OpenSSL 3.0.X continued, with the goal of completing the full migration. This milestone was divided into the following sub tasks: verifying the base exchange implementation from previous years, identifying and fixing the issues causing the ping test to fail, and making sure all other tests pass.

2.2 HHIT

Heirichal Host Identity tags (HHIT) [13] is an extension of Host Identity Tags (HIT) [14], which builds upon the Overlay Routable Cryptographic Hash Identifiers Version 2 (ORCHIDv2) [9].

The goal was to verify the implementation of HHIT according to RFC-9374 [13].

3 Previous Work

The milestones for this year will be the continuation of work done by previous groups. This section will summarize the work done previously that is relevant to this year's milestones.

3.1 OpenSSL

The migration work was started 2021. The 2021 group attempted migrate the whole project using the OpenSSL migration guide (REF). They managed to achieve a version that compiled, but did not work.

The 2022 group continued this work and managed to implement an almost working base exchange using OpenSSL 3.

The 2023 group continued further and completed the base exchange implementation (for some chiphers), and started on identifiying issues with the ping test.

3.2 HHIT

The 2020 group started the HHIT implementation following the draft `draft-moskowitz-hip-hierarchical-hit` [12]. At this stage, no HHIT prefix was allocated and the HID was 32 bytes long.

The 2022 group continued the implementation of HHIT, now following the latest (at the time) draft of `draft-kuptsov-hhit-05` [7]. This group misinterpreted the HDA property of HID, namley that a HDA is unique to only one RAA.

The 2023 group updated the HHIT generation to follow RFC-9374 [13] instead of the draft used by previous years. While the format of the Orchid was correct, the RFC was not properly followed. The misinterpretation of HID remained from last year.

3.3 PyHIP

The 2023 group started investigating the slow performance of PyHIP and assumed that PyCryptoDome was the bottleneck and started migration to Cryptography. They abandoned the attempt due to lack of time.

4 Contributions

This section describes the specific contributions our group has made to the OpenHip projects. It is divided into sections for the different milestones that were worked towards.

4.1 Approach

Our approach towards completing the milestones involved spending a lot of time investigating and verifying the implementation of the previous years. A lot of effort was needed in order to properly understand both HIP and the given implementation. The behavior of the code was compared to the specifications in the various RFC:s to make sure everything worked predictably. Changes were made incremental, often by first writing smaller test programs and then integrating the code into the OpenHIP code base.

4.2 OpenSSL

Our group finalized the migration from Openssl 1.1.1 to Openssl 3.0.X mainly by fixing errors in the implementation of both the control plane and data plane. Additionally, the usage of the OpenSSL library was altered in some cases to simplify functions and to remove redundant dependencies. The following specific changes / fixes were made in order to get hip communication working with Openssl 3:

- Fixed incorrect OpenSSL usage in hip_esp_encrypt and hip_esp_decrypt.
- Fixed support for HMAC-SHA256 in hip_esp.
- Fixed segfaults in EdDSA hitgen implementation.
- Fixed double free in parsing of R1 package using other HI signature algorithms than RSA.
- Fixed validate_signature by changing from Digest and Verify to DigestVerify. Also changed from Digest and Sign to DigestSign, removed unique treatment of cSHAKE128 to use the functions for all algorithms.
- Fixed mixups in usages of hit suite variables.

These fixes means both the base exchange and communication via ESP [6] work for some cryptographic algorithms. Specifically, the base exchange works when using either RSA, or EdDSA with curves EdDSA25519 and EdDSA448. The ESP layer works using AES-128 or AES-256, using sha-1 or sha-256 for authentication.

4.3 HHIT

In the implementation done by the previous group there was no option to pass RAA and HDA to the HIT generation function. Instead, they used a flag called `-drip` which read allowed the user to input a HDA. The HDA was then used in a mapping function to retrieve a RAA. This implementation was not correct since a RAA does not depend on a HDA. To fix this we have removed the `-drip` flag and instead added the flag `-hhit` which accepts a RAA and HDA. If no RAA or HDA is provided the HIT generation function defaults to the experimental values 16376 and 1026 for RAA and HDA respectively.

There were a few incorrect arguments passed to the ORCHID hash function in the previous implementation. One was the that the context ID for a regular context ID was passed. We fixed this by passing the context ID for HHIT instead. The other incorrect argument was the length of a string. This was previously set to 8 bits, but since an empty string is passed this was changed to 0 bits instead.

The test of HHIT from previous year's group only tested if the HHIT was routeable, but not if the HHIT was correctly generated. We therefore created a test script `hhit_test.py` that compares that the HHIT generated from `hitgen` is equal to a HHIT generated by a Python script from the DRIP project, which matches RFC-9374 [13].

After our changes the HIT and HHIT generation now follows RFC-7401 [14] and RFC-9374 [13] respectively.

4.3.1 `HOST_ID` error in RFC-9374

In RFC-9374 [13], section 3.4.1.1, we have identified what appears to be an error. The `HOST_ID` is described as consisting of the EdDSA curve for the first two bytes, followed by two bytes of NULL padding, and then the public key. However, in RFC-7401 [14], section 5.2.9, the `HOST_ID` is described as using the ECC curve for the first two bytes, followed directly by the public key, without any NULL padding. The inclusion of the two-byte NULL padding would make the HHIT incompatible with HIPv2, as specified in RFC-7401 [14].

We believe this difference is an unintentional mistake by the authors of RFC-9374 [13]. The figure in RFC-7401 [14] closely resembles the one in RFC-9374 [13], but the slash (/) at the end of the first row and the beginning of the second row indicates a continuous field rather than empty padding.

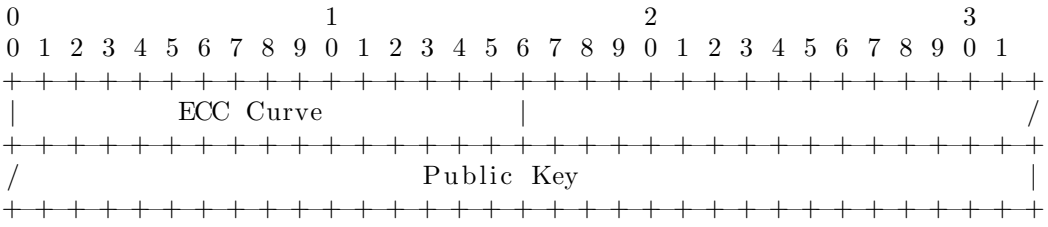

Figure 1: Figure from RFC-7401

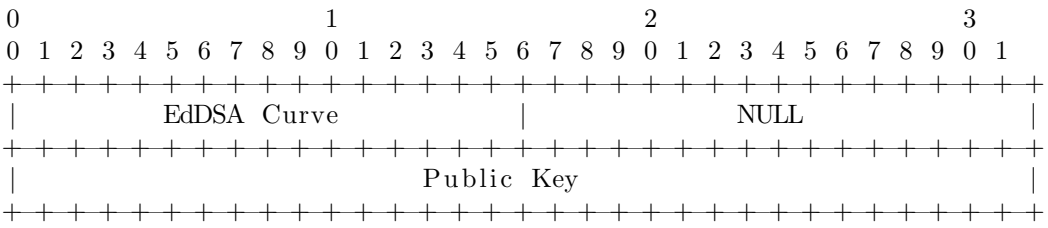

Figure 2: Figure from RFC-9374

4.4 Pyhip

The primary objective of PyHIP was to identify performance bottlenecks. Initial concerns revolved around the performance of the PyCryptoDome library, compared to more modern libraries like Cryptography. Consequently, the initial approach was to replace PyCryptoDome with Cryptography, as PyCryptoDome was suspected to be a performance bottleneck. Martin Christensson had previously attempted to migrate portions of the asymmetric cryptography to the Cryptography library [4]. His modifications reportedly doubled throughput. However, our experiments did not reproduce these results. The discrepancies might be attributed to hardware differences, as we could not ascertain the hardware setup he used for testing and thus could not directly compare it to ours.

Given this, we concluded that profiling PyHIP would be the most effective method to pinpoint the code segments most impacting performance.

4.4.1 Code restructuring

Before implementing any modifications to PyHIP, we restructured the code to address a significant issue: code duplication across four locations in the repository. This duplication made development cumbersome and error-prone, as any change required copying updates across all instances. This increased the risk of inconsistencies and bugs if updates were not consistently applied.

To resolve this, we moved all duplicated code into a shared directory named `hiplib`, located in the parent folder. A script was created to generate symlinks at the locations where the code was duplicated, allowing the existing structure to remain functional without requiring changes to the code itself. This restructuring simplified development and provided better oversight. Additionally, we organized PyHIP within the mentioned GitLab group for improved project management.

4.4.2 Profiling with CProfile

For profiling Python code, CProfile is a suitable tool. It allows profiling of entire functions, provided they exit successfully. The main functionality of PyHIP resides in three threads, each responsible for processing packets from specific sockets:

- HIP packet processing
- IPsec packet processing
- Ether packet processing

Some modification was done to `switchd.py` in `router1`, which wraps the three threads with CProfile. In the main repository of PyHIP, the three threads were just killed and not exited properly, and since CProfile needs to exit properly to report results, code was added to make sure the threads exit on Keyboard interrupt in the main thread.

To profile the threads, the Mininet environment was started as described in the README file. Profiling was enabled only on `router1`, while `router2` remained unprofiled. The profiling procedure involved starting the two routers, and doing an iperf between them:

```
h4 iperf -s -w 100000K &

h1 iperf -c h4 -f m -b 10000M -l 1000 -w 500000K
```

4.4.3 Results

The profiling was conducted for the three threads both before and after applying Martin Christensson's changes. Detailed results are included in the appendix. From the profiling data, it is evident that asymmetric cryptography is used only in the HIP packet processing thread. The impact of the changes was minimal, as the cumulative time spent on asymmetric cryptography accounted for less than 1% of the total time in HIP packet processing.

version	hlib.py:161(process_hip_packet)	asymmetric.py (sum of all)	normalized asymmetric.py
no changes	5.618	0.021	0.00373798504
asymmetric	7.327	0.04	0.00545926027

We investigated the results of profiling and looked into which of the threads spend the most time.

thread	cumulative time
process_hip_packet	5.618
process_ip_sec_packet	10.384
process_l2_frame	45.386

As we can see in the table, most of the time is spent in the thread responsible for ether packet processing, therefore finding slowdowns in there would result in the most performance gain.

Out of the 45 seconds of cumulative time spent, 20 of those were spent on the digest.py, which uses the PyCryptoDome library. Therefore we determined to try and migrate this code to the Cryptography libary.

version	process_ip_sec_packet	digest	normalized digest
no changes	10.384	6.042	0.5818567026
migrated	7.809	3.301	0.4227173774

With the migrated `digest.py`, we see a a big decrease on the cumulative time spent on digest, however no improvement in throughput. So far we have discovered that the speed of the cryptography has no clear relation to the throughput, which might show that there is a deeper issue with the performance of the PyHIP.

All the results of the profiling can be found in the repository [1].

4.5 Maintainability

Last year's group highlighted the importance of addressing growing technical debt. While this remains an ongoing challenge, we expand the focus to the broader concept of maintainability, which we have actively improved.

4.5.1 Repositories

Last year's group used a repository for OpenHIP hosted under an individual student account [3]. As a first step, we forked this repository. To prevent the need for creating new forks each year, we established a GitLab group, `openhip-tdde21`, on LIU's GitLab. This group includes all students for the current year as well as the examiner/supervisor. As a result, future groups will not need to fork the repository but can instead request access directly from the examiner.

4.5.2 Getting started

One issue that has been lacking in previous years is clear guidance for getting started with the project. At the beginning of this year, there were no instructions on how to properly set up the project. Last year's group tried to address this by saving VM files for the following year (us), but due to their large size, these files were lost and never received. To avoid this issue, we are not leaving behind large artifacts. Instead, we are focusing on providing comprehensive getting-started instructions that cover everything the next group of students will need to begin working on the project. Most of these instructions will be included in the `README` files within the appropriate repositories.

4.5.3 Formatting

Last year's group decided to format the entire OpenHIP codebase. However, this had the unintended consequence of effectively erasing the git blame history, making it challenging to track and compare changes. The situation was further complicated by the fact that formatting was applied only to one of the two active development branches, and no documentation or configuration files outlining the formatting rules were provided, leaving us unable to reproduce the same formatting on the unformatted branch.

[1] `https://gitlab.liu.se/openhip-tdde21/pyhip/-/blob/6117798e227c44ee6ab1cfd5e65e5b7cdc83e77f/router1/router1_profiling_no_changes.txt`

While strict formatting guidelines can enhance code readability, applying them to an actively developed feature branch is not best practice. It makes it difficult to compare differences between the two branches, as the resulting merge conflicts are mostly related to formatting changes, rather than actual code modifications.

4.5.4 Branches

Last year's group left the Git repository in a less-than-ideal state, particularly with regard to the branching structure. The two subgroups each maintained their own branches. Additionally, no merge of these changes was made, leaving us with two branches that hadn't shared a common commit in two years, making it very time-consuming to merge.

There are also indications that Git was not used properly, as the core migration from last year was copy-pasted between the two branches rather than being cherry-picked or merged, which would have been the better practice. As a result, the tooling and testing in each branch became increasingly divergent.

To improve this, we let the branch containing the openssl3 migration become our main branch and treated the branch containing HHIT as the feature branch. After rebasing the HHIT changes onto the main branch, we successfully merged the HHIT branch into openssl3 and unified the codebase again.

5 Future Work

In this chapter, we present what future work exists on all milestones covered in this report.

5.1 HIP specification

The current RFC specifications can be difficult to follow due to the fragmentation and numerous versions that exist. For example, RFC 9374 (DRIP Entity Tag (DET) for Unmanned Aircraft System Remote ID (UAS RID)) makes amendments to which HHIT Suites are to be supported for DET functionality. In the future, creating a document summarizing the algorithms that should be supported and under what category (Base Exchange, ESP) can be useful to provide a clearer overview.

5.2 OpenSSL

Currently, only RSA and EdDSA are supported in the base exchange. However, there still exists non-working code (that was incorrectly ported to OpenSSL 3) implementing several other algoritms (DSA, Xoodyac, EcDSA). For all of these, a decision should be made if they should be removed from the project, or fixed so that they can be used as well. In any case, OpenHIP should not run and crash when given these algorithms.

5.3 HHIT

There is currently no future work on HHIT, however since HHIT for HIP has not been finalized, there might be additional changes made in the future. The current implementation follows RFC-9374 [13].

5.4 PyHIP

To continue the work with PyHIP, one could continue migrating to Cryptography and determine which functions have an impact on performance. After our migration of digest.py, we can see that digest.py and symmetric.py are the two cryptography methods which make up the majority of the packet processing time in both ipsec and ether threads. We noticed that digest.py creates a new hmac context every time it is used, which makes up about half the time of the migrated digest.py. The Cryptography package supports reuisng the hmac context, which could potentially half the cumulative time of digest.

However, while these optimizations might look good on paper, we have not found any evidence that it will lead to a higher throughput. A more thorough investigations into the inner workings of PyHIP is necessary to determine what is the bottleneck, we suggest investigating the synchronization between threads and the communication between the two routers running PyHIP, and find if any unecessary delays exist.

5.5 Diet-ESP

Currently OpenHIP uses ESP (Encapsulated Security Payload) as a transport format to send messages. This is used to allow secure communication. There exists a format call Diet-Esp, that better optimizes the size of messages. This could be a valuable addition to hip. Currently, there exists a draft implementation in Python, available at `https://github.com/mglt/pyesp/blob/master/examples/draft-diet-esp.py`. Also, a master thesis by Migault et al. discusses implementations of the format [10].

5.6 Release

We did not create a release package for OpenHIP, as no process exists for creating this in the code base, even though it is currently in a quite stable version.

Future work could include creating a release process for OpenHIP. Although time would also need to be spent making sure OpenHIP works on actual hardware, as we have only been running it in virtual machines and virtual networks. Theoretically there should be no issues, but is still untested.

6 Discussion

6.1 Future guidelines for development

We have collected a few guidelines we deem appropriate for development of openhip. These are to reduce the amount of technical debt created and improve the maintainability for a longer term.

6.1.1 Main and feature branch

One branch should always be considered the main branch, and it should remain in a working state at all times. Therefore, committing directly to the main branch is generally inappropriate, except for small, non-functional changes like documentation updates.

Instead, developers should create a feature branch from the main branch, implement the necessary changes, and, once the feature is in a stable state, merge it back into the main branch.

In cases where the main branch has changed after the feature branch was created, such as in smaller subgroups, it is appropriate to rebase the feature branch onto the main branch before merging. It is also good practice to regularly update the feature branch with the latest changes from the main branch.

If code needs to be shared between subgroups working on separate feature branches (e.g., updating the core emulator to a new version), this should also be done on a feature branch. Once the changes are finalized, merge them into the main branch, and then rebase the feature branches of the subgroups. In previous years, this was often handled by copy-pasting code with minor modifications, which makes the git history difficult to track and follow.

6.1.2 Keeping documentation up-to-date

While the codebase as a large remains undocumented (no student has been able to grasp the entirety of the codebase), if changes are made that invalidates the documentation, it is important to update it. This is especially true for documentation that the next year's group will encounter early, such as getting started instructions.

If your work is left in an not working state, it is even more important to document what is missing, what bugs are known and what is verified to be working correctly.

6.2 Getting started

To facilitate for future groups we have compiled a few steps for getting started with the project.

6.2.1 Virtual machine

For future groups we recommend to begin with creating an environment for development that can be shared among the members. This is preferably done using a virtual machine running Ubuntu 22.04 for x86-64 architecture. This will also make it possible to change system configurations that might be needed, without affecting the host operating system. To share a virtual machine instance, it can be exported to a .ova file which can be imported in a virtualization software, for example Virtual Box.

6.2.2 CORE and OpenHIP

When you have created a functioning virtual machine instance, clone the GitLab repo and install CORE and OpenHIP. Make sure that the system is using OpenSSL 3.0.

1. Install the prerequisites listed in the documentation.

2. Initiate the Git submodules.

```
git submodule update --init
```

3. Install CORE. From the `core/` directory run:

```
./setup.sh
source  /.bashrc
inv install
```

4. Install OpenHIP. From the `openhip` directory run:

```
./bootstrap.sh
./configure
make
sudo make install
```

Before starting any development, verify that you have a functioning setup of CORE and OpenHIP. This can be done by running the script `openhip/test/debug.py`. If the ping is successful, everything should be working fine. To run the test use:

```
sudo su
source /opt/core/venv/bin/activate
cd test
core-cleanup && python3 debug.py
```

6.2.3 PyHIP

To setup PyHIP clone the PyHIP repository from the GitLab group. We recommend to create an outer directory where you clone the repository inside it, as PyHIP will create directories a step up from its own directory. From the `pyhip/` directory run :

```
cd hip-vpls
sudo bash deploy.sh
```

Before running PyHIP all routers must contain the hiplib code. To do this run the `create-hiplib-symlink.sh` script. It will create symbolic links to the code in the hiplib directory in all router directories. This was the best solution to avoid the need to write replicated code in all router directories.

To run PyHIP with mininet use the following commands:

```
cd hip-vpls
sudo python3 hipls-mn.py
```

It is now possible to ping nodes and do a performance test. To measure the performance first open the routers in separate terminals:

```
mininet> xterm r1 r2
```

To start each router run the following command in each router's terminal. <x> is replaced by the routers number.

```
cd router<x>
sudo python3 switchd.py
```

Start the performance test with `iperf` by using the following two commands in the mininet terminal:

```
h4 iperf -s -w 100000K &
h1 iperf -c h4 -f m -b 10000M -l 1000 -w 500000K
```

7 Conclusion

The primary milestones of this year are completed. We have successfully migrated OpenHIP from OpenSSL 1.1.1 to OpenSSL 3.0.0. The HHIT generation is now correct and merged with the OpenSSL migration branch to create one stable main branch. OpenHIP is now in the most complete state it has been over the last few years. Good luck next year.

References

[1] The OpenSSL Project Authors. "OpenSSL FIPS 140-2 Security Policy". In: (2024). URL: https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp4282.pdf.

[2] The OpenSSL Project Authors. *ossl-guide-libcrypto-introduction*. [Accessed 08-12-2024]. 2024. URL: https://docs.openssl.org/master/man7/ossl-guide-libcrypto-introduction/.

[3] Gustaf Lindgren Bodemar. *OpenHIP*. Accessed: 2024-12-12. 2024. URL: https://gitlab.liu.se/gusbo010/openhip.

[4] Martin Christensson. *Hip-vpls*. GitHub repository. 2023. URL: https://github.com/MartinChristensson756/Hip-vpls.

[5] PyCryptodome Developers. *PyCryptodome: A self-contained Python package of low-level cryptographic primitives*. https://github.com/Legrandin/pycryptodome. Accessed: 2024-12-12. 2024.

[6] Petri Jokela, Robert Moskowitz, and Jan Melen. *Using the Encapsulating Security Payload (ESP) Transport Format with the Host Identity Protocol (HIP)*. RFC 7402. Apr. 2015. DOI: 10.17487/RFC7402. URL: https://www.rfc-editor.org/info/rfc7402.

[7] Dmitriy Kuptsov, Andrei Gurtov, and Dacheng Zhang. *Hierarchical Host Identity Tags (HHIT) Verification Architecture*. Internet-Draft draft-kuptsov-hhit-05. Work in Progress. Internet Engineering Task Force, Mar. 2011. 8 pp. URL: https://datatracker.ietf.org/doc/draft-kuptsov-hhit/05/.

[8] US Naval Research Laboratory. *Common Open Resource Emulator (CORE)*. [Accessed 09-12-2024]. 2024. URL: https://www.nrl.navy.mil/Our-Work/Areas-of-Research/Information-Technology/NCS/CORE/.

[9] Julien Laganier and Francis Dupont. *An IPv6 Prefix for Overlay Routable Cryptographic Hash Identifiers Version 2 (ORCHIDv2)*. RFC 7343. Sept. 2014. DOI: 10.17487/RFC7343. URL: https://www.rfc-editor.org/info/rfc7343.

[10] Daniel Migault, Tobias Guggemos, Sylvain Killian, Maryline Laurent, Guy Pujolle, and jean-philippe Wary. "Diet-ESP: IP layer security for IoT". In: *Journal of Computer Security* 25 (Apr. 2017), pp. 1–31. DOI: 10.3233/JCS-16857.

[11] Mininet. *Mininet: An Instant Virtual Network on your Laptop (or other PC)*. https://github.com/mininet/mininet. Accessed: 2024-12-12. 2024.

[12] Robert Moskowitz, Stuart W. Card, and Adam Wiethuechter. *Hierarchical HITs for HIPv2*. Internet-Draft draft-moskowitz-hip-hierarchical-hit-00. Work in Progress. Internet Engineering Task Force, Sept. 2019. 9 pp. URL: https://datatracker.ietf.org/doc/draft-moskowitz-hip-hierarchical-hit/00/.

[13] Robert Moskowitz, Stuart W. Card, Adam Wiethuechter, and Andrei Gurtov. *DRIP Entity Tag (DET) for Unmanned Aircraft System Remote ID (UAS RID)*. RFC 9374. Mar. 2023. DOI: 10.17487/RFC9374. URL: https://www.rfc-editor.org/info/rfc9374.

[14] Robert Moskowitz, Tobias Heer, Petri Jokela, and Thomas R. Henderson. *Host Identity Protocol Version 2 (HIPv2)*. RFC 7401. Apr. 2015. DOI: 10.17487/RFC7401. URL: https://www.rfc-editor.org/info/rfc7401.

[15] StrangeBit.io. *HIP VPLS (Virtual Private LAN Service)*. https://github.com/strangebit-io/hip-vpls. Accessed: 2024-12-12. 2024.