Improving the Open Source HIPv2 Implementation

TDDE21 – Secure Distributed and Embedded Systems
Linköping University

Project Report 2025

Tobias Rosengren (tobro174)
Mabest Amin (mabam091)
Casper Nerf Kanefall (casne582)
Abuzar Sohail (abuso062)

December 17, 2025

Contents

1 Introduction and Background

This report describes the 2025 continuation of the multi–year effort to improve the open–source implementation of the Host Identity Protocol Version 2 (HIPv2). The work was carried out within the course TDDE21 at Linköping University. The project builds directly upon the codebase and results delivered by the 2024 group, focusing on both correctness and performance of the implementation.

HIP was designed to decouple host identity from network location, enabling stronger cryptographic identities for endpoints and supporting use cases such as mobility and multihoming without binding long-term identity to an IP address [8, 10]. In practice, this makes HIP an interesting research and experimentation platform for secure routing, machine-to-machine deployments, and emerging identity-driven networking scenarios.

This year, the project group was divided into two subgroups:

- **Core**: focused on the C-based OpenHIP implementation, OpenSSL integration, and Wireshark dissector.
- **PyHIP**: focused on the Python-based HIP router implementation, profiling, performance analysis, and experimental improvements.

The following subsections summarize essential background concepts.

1.1 Host Identity Protocol (HIP)

HIP introduces a separation between host identity and network location. Instead of using IP addresses as both identifier and locator, hosts are identified using asymmetric public keys (Host Identities), while IP addresses function purely as routable locators [8,10]. During the HIP base exchange, public-key signatures authenticate peers, after which symmetric cryptography protects subsequent data traffic.

1.2 Host Identity Tag (HIT) and ORCHIDv2

A Host Identity Tag (HIT) is a 128-bit identifier derived from a host's public key using the ORCHIDv2 format [5,8]. The HIT embeds a fixed IPv6 prefix, a Suite ID referencing the cryptographic algorithm, and a hash over the key material. The resulting value is used as the host identifier throughout the HIP protocol.

1.3 HHIT

Hierarchical HITs (HHITs) extend the HIT format by introducing structured fields for hierarchical allocation, such as Registered Assigning Authority (RAA) and HHIT Domain Authority (HDA). These fields support applications such as remotely identifiable tags, and HHITs follow RFC 9374, which builds upon the ORCHIDv2 structure [7].

1.4 CORE Network Emulator

CORE is a network emulation environment used to test OpenHIP in controlled virtual networks. It enables rapid iteration during development and provides reproducible debugging setups through scripts such as `debug.py`. CORE has been widely used for lightweight, scriptable network experiments in research and teaching contexts [2].

1.5 OpenSSL

OpenSSL is a powerful, enterprise-grade toolkit designed for general-purpose cryptography and secure communications. It provides a command-line utility that allows users to perform a wide range of cryptographic tasks, including generating keys and certificates [1]. In addition, OpenSSL includes a collection of providers that deliver implementations for an extensive set of cryptographic algorithms and secure network communication.Our process started with validating the existing test cases in OpenSSL 3.0, followed by adapting and converting them for compatibility with OpenSSL 3.5.

1.6 PyHIP

PyHIP is a Python-based implementation of the Host Identity Protocol version 2 (HIPv2) intended primarily for experimentation and education. It implements HIP control-plane functionality in user space and is commonly deployed in virtualized network environments such as Mininet, where multiple HIP routers and hosts can be instantiated on a single machine.

In PyHIP, each router runs a software forwarding and control stack that handles HIP base exchanges, HIP control messages, and ESP-protected traffic. Packet processing is largely implemented in Python and relies on raw sockets for capturing and injecting packets. Cryptographic functionality is provided through Python cryptography libraries, while inter-component coordination is handled using threads and shared data structures.

Because PyHIP prioritizes flexibility and readability over raw performance, it provides a useful platform for studying protocol behavior, experimenting with new features, and validating changes before they are introduced into lower-level implementations. At the same time, its reliance on Python introduces inherent performance limitations that make it particularly suitable for research on architectural trade-offs and optimization strategies.

1.6.1 Mininet

Mininet provides lightweight network emulation for Linux, enabling virtual routers and hosts on a single machine. Its process-based virtualization and scripting enable rapid prototyping of protocol experiments and reproducible performance tests [6].

1.6.2 PyCryptodome & Cryptography

These libraries implement cryptographic operations in Python. PyCryptodome offers low-level primitives, while the `cryptography` package provides modern APIs and can offer better performance and maintainability for selected operations.

1.6.3 cProfile

`cProfile` is a built-in Python tool used for profiling execution time across function calls, and was used to identify bottlenecks in PyHIP's packet-processing threads.

1.7 GitLab Group Structure

The project repositories are maintained in a shared GitLab group to ensure continuity across academic years, reduce fragmentation, and simplify collaboration.

1.8 Wireshark

Wireshark is a widely used protocol analysis tool that captures packets and presents decoded protocol fields to the user. The availability of custom dissectors is essential for validating experimental or evolving protocol implementations such as HIPv2 [13].

1.9 Cryptographic techniques and modern suites

HIPv2 typically combines asymmetric and symmetric cryptography. Public-key signatures authenticate endpoints during control-plane exchanges, while hash functions support identifier derivation and message binding. After authentication, symmetric cryptography is used for efficient protection of data traffic, commonly via IPsec ESP [4].

1.9.1 ECDSA/SHA-384

A common modernization is migrating from RSA/SHA-256 to ECDSA/SHA-384. ECDSA provides comparable security with smaller keys and signatures than RSA, reducing protocol and storage overhead [12]. Using SHA-384 raises the strength of the hash component within the signature suite [11].

1.9.2 AES-256-GCM for ESP (ES-256-GCM)

AES-256-GCM is an authenticated-encryption (AEAD) mode that provides confidentiality and integrity together, and is widely used with ESP [3, 4]. Correct security depends on ensuring nonce/IV uniqueness per key when using GCM-based ESP [3].

2 Milestones

2.1 OpenSSL

2.1.1 Migration from OpenSSL 3.0 to 3.5+:

The primary milestone was completing the migration to OpenSSL 3.5+, ensuring that HIP base exchange and ESP traffic remained functional. This required verifying compatibility with RSA, ECDSA, and EdDSA suites.

2.1.2 Algorithm Validation:

A milestone was set to confirm that all supported algorithms (RSA, ECDSA, EdDSA Curve25519/448) could be generated and validated correctly. This ensured interoperability with HIP's identity and authentication mechanisms.

2.1.3 Provider Initialization:

Another milestone focused on integrating OpenSSL's modular provider architecture into HIP daemons. This step guaranteed that cryptographic services were loaded consistently and reported accurate metadata.

2.2 Wireshark Disector for HIP v2

A wireshark dissector for the first HIP version exist as defined by rfc 5201 [9]. Changes to this dissector need to be made to match the HIP version 2 as defined by rfc 7401 [8].

- Identify defferences between HIP v1 [9] and HIP v2 [8]
- Build and test the original wireshark dissector on HIP version 2 implimantation.
- Impliment a simple test dissector.
- Update the values of changed packet parameters.
- Add new packet types to be recognized.
- Change packet alignment to match HIP v2 [8].
- Test new wireshark dissector on HIP v2 implimentation.

2.3 PyHIP

The PyHIP subgroup focused on improving end-to-end throughput in the HIP implementation during 2025. To understand where performance limitations originated, the group concentrated on identifying bottlenecks through systematic benchmarking and detailed profiling. The main milestones were:

- Investigate and improve throughput by migrating from threading to multiprocessing in switchd.py
- Implement and validate the AES-256-GCM cryptographic primitive, including integration into the existing HIP codebase and extension of test coverage in `hiplib/tests`.
- Migrate the deployed identity suite from RSA/SHA-256 to ECDSA/SHA-384, including regeneration of keys, HITs, and firewall rules across all routers.
- Update documentation.

3 Previous Work

The 2024 group delivered a stable baseline that our 2025 effort relied on. Their report, "Improving the Open Source HIPv2 Implementation," documents how the codebase reached its current state. The highlights below capture the elements that directly influenced this year's PyHIP focus.

3.1 OpenSSL Migration

The migration to OpenSSL began in 2021, when the project was first compiled against version 3.0 but remained non-functional. In 2022, partial base exchange functionality was introduced, though stability issues persisted.

By 2023, the base exchange was completed for RSA and EdDSA algorithms, and work began on resolving problems with the ping test. In 2024, the migration to OpenSSL 3.0.X was finalized, with critical fixes applied to ESP encryption and decryption, HMAC-SHA256 support, EdDSA segfaults, and signature validation. These improvements ensured reliable HIP base exchange and ESP traffic, providing the foundation for the current upgrade to OpenSSL 3.5+.

3.2 HHIT Evolution

HHIT work started in 2020 from early drafts and eventually aligned with RFC 9374. The 2024 group corrected the remaining misunderstandings around HDA/RAA handling, updated the ORCHID parameters to match the final specification, and supplied a Python-based regression test that compares generated HHITs against known-good DRIP tooling. Their implementation gave us a consistent reference when regenerating identities for the ECDSA migration.

3.3 PyHIP Foundations

PyHIP was restructured substantially last year. Duplicate router directories were consolidated into a shared `hiplib` module with symlinks, making it feasible to evolve the code once instead of four times. The team profiled the three packet-processing threads using `cProfile`, identified `digest.py` and `symmetric.py` as the heaviest consumers and began migrating selected primitives to the `cryptography` package.

4 Contributions (2025)

4.1 PyHIP Subgroup

4.1.1 Approach.

The work began by restoring the Mininet lab environment from 2024 and converting the throughput experiment into a repeatable and automated benchmark. The resulting script, `analysis/throughput_test.py`, boots a four-router topology, waits for the HIP base exchange to complete, and then runs scripted `iperf3` traffic between selected host pairs.

4.1.2 Instructions

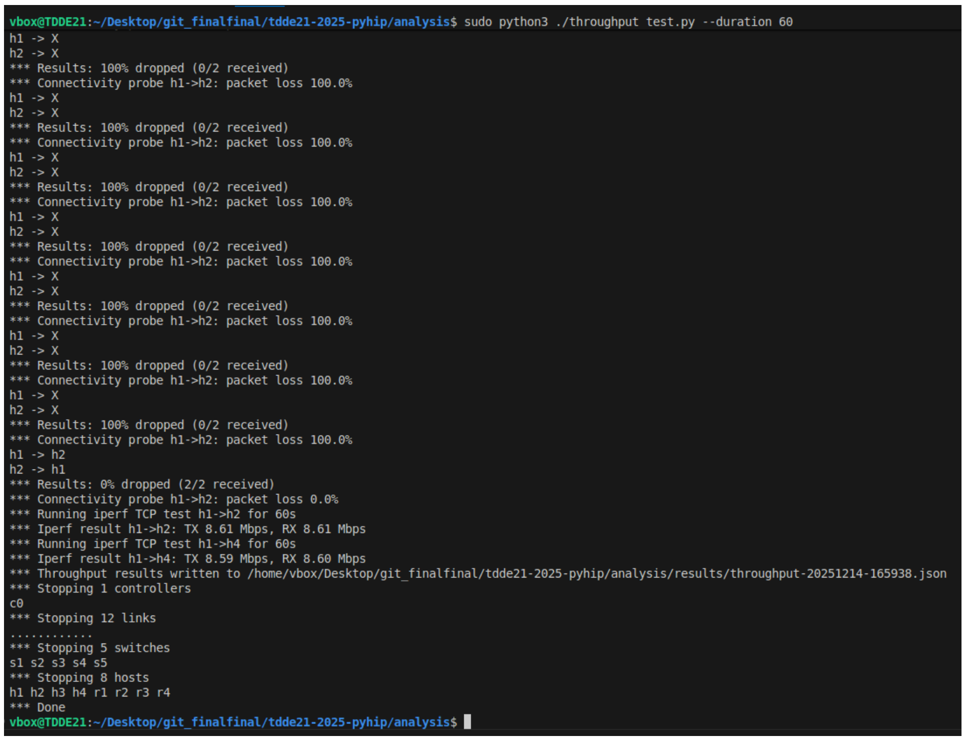
Because the script manipulates namespaces and raw sockets, it must be run with root privileges:

```
cd analysis
sudo python3 throughput_test.py --duration 15
```

Once connectivity is detected, the script launches each `iperf` flow and prints a live summary, e.g. `*** Iperf result h1->h2: TX 11.70 Mbps, RX 11.72 Mbps`.

Each run generates a JSON artifact under `analysis/results/throughput-<timestamp>.json` containing the per-flow throughput values, the `switchd` PID map (for later profiling), and tails of every `router*/hipls.log` file to correlate performance drops with HIP or IPSec events. The CLI supports several flags for repeatable experimentation:

- `--pairs` selects any host pairs (default `h1:h2 h1:h4`).
- `--duration` sets the per-pair `iperf` runtime in seconds (default 10).
- `--l4` switches between TCP and UDP (UDP adds `-u -b 10M`).
- `--ready-pair` / `--ready-timeout` control the BEX readiness probe.
- `--output` chooses where the JSON report is written.
- `--switchd-log-dir` points to the directory that captures `switchd` stdout/stderr.
- `--log-level` forwards the desired verbosity to Mininet.



```
vbox@TDDE21:~/Desktop/git_finalfinal/tdde21-2025-pyhip/analysis$ sudo python3 ./throughput test.py --duration 60
h1 -> X
h2 -> X
*** Results: 100% dropped (0/2 received)
*** Connectivity probe h1->h2: packet loss 100.0%
h1 -> X
h2 -> X
*** Results: 100% dropped (0/2 received)
*** Connectivity probe h1->h2: packet loss 100.0%
h1 -> X
h2 -> X
*** Results: 100% dropped (0/2 received)
*** Connectivity probe h1->h2: packet loss 100.0%
h1 -> X
h2 -> X
*** Results: 100% dropped (0/2 received)
*** Connectivity probe h1->h2: packet loss 100.0%
h1 -> X
h2 -> X
*** Results: 100% dropped (0/2 received)
*** Connectivity probe h1->h2: packet loss 100.0%
h1 -> X
h2 -> X
*** Results: 100% dropped (0/2 received)
*** Connectivity probe h1->h2: packet loss 100.0%
h1 -> X
h2 -> X
*** Results: 100% dropped (0/2 received)
*** Connectivity probe h1->h2: packet loss 100.0%
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
*** Connectivity probe h1->h2: packet loss 0.0%
*** Running iperf TCP test h1->h2 for 60s
*** Iperf result h1->h2: TX 8.61 Mbps, RX 8.61 Mbps
*** Running iperf TCP test h1->h4 for 60s
*** Iperf result h1->h4: TX 8.59 Mbps, RX 8.60 Mbps
*** Throughput results written to /home/vbox/Desktop/git_finalfinal/tdde21-2025-pyhip/analysis/results/throughput-20251214-165938.json
*** Stopping 1 controllers
c0
*** Stopping 12 links
............
*** Stopping 5 switches
s1 s2 s3 s4 s5
*** Stopping 8 hosts
h1 h2 h3 h4 r1 r2 r3 r4
*** Done
vbox@TDDE21:~/Desktop/git_finalfinal/tdde21-2025-pyhip/analysis$ 
```

Figure 1: Example run of throughout test

4.1.3 Throughput Test Examples

Each run waited for the HIP base exchange to converge (typically eight probe cycles) and then drove either TCP or UDP traffic between selected hosts. All switchd instances negotiated the same ECDSA/SHA-384/AES-256-GCM profile as defined in `router1/hiplib/config/config.py`. Table 1 summarises the per-flow transmit/receive rates recorded in the JSON artifacts under `analysis/results/`.

Run ID	JSON File	L4	Duration (s)	Pair	TX (Mbps)	RX (Mbps)
T1	throughput-20251214-211430	TCP	10	$h_1 \to h_2$	19.49	19.50
				$h_1 \to h_4$	24.25	24.51
T2	throughput-20251214-211542	TCP	30	$h_1 \to h_2$	24.31	24.43
T3	throughput-20251214-211654	UDP	10	$h_1 \to h_2$	10.49	10.49
T4	throughput-20251214-211956	UDP	20	$h_1 \to h_4$	10.49	10.49

Table 1: Measured throughput for varied transports, durations, and host pairs.

Across all cases TCP maintained symmetric throughput between 19.5–24.5 Mbps, with longer durations helping the $h_1 \to h_2$ flow exit slow-start.

UDP tests are bounded by the harness' `-b 10M` setting and, as expected, stayed pinned at 10.5 Mbps with negligible loss across both hop counts.

4.1.4 Profiling

Profiling was initially planned as a means of identifying fine-grained bottlenecks inside `switchd.py`. In practice, combining Python profiling tools with Mininet, raw sockets, and multiple interacting router processes proved difficult. Profiling runs were unstable or incomplete, and no reliable profiling data could be extracted within the project timeframe. Consequently, decisions were not based on profiling results.

4.1.5 Multiprocessing experiments

Several approaches were explored to reduce Python interpreter overhead by introducing multiprocessing. The first approach attempted to run separate HIP processes with independent HIPLib instances. This design failed during the HIP base exchange due to state desynchronization between processes.

A second approach attempted to share a single HIPLib instance across multiple processes. This revealed limitations in Python raw socket handling across process boundaries, resulting in kernel errors (`ENXIO`, errno 6). Finally, queue-based designs were tested to reduce lock contention between execution contexts, but these did not lead to measurable throughput improvements.

None of the multiprocessing variants were stable enough to integrate into the main codebase.

4.1.6 Performance benchmarking

All performance results in this report are based on the working Mininet and `iperf3` throughput-testing setup. While no throughput improvements were achieved during the project, the benchmarking script provides a decent baseline for future optimization efforts.

4.1.7 Cryptographic changes.

The cryptographic work focused on adding support for AES-256-GCM and migrating the identity suite to ECDSA/SHA-384. AES-256-GCM was integrated alongside existing symmetric ciphers to provide modern authenticated encryption.

All router identities were regenerated using ECDSA/SHA-384, and configuration files, HITs, and firewall rules were updated consistently across the topology.

4.2 OpenSSL 3.5 Implementation:

This test suite confirms that OpenSSL 3.5+ is fully integrated and stable within the OpenHIP environment. The validation focused on ensuring that all baseline cryptographic algorithms, configuration files, and provider mechanisms function correctly after the migration from OpenSSL 3.0. The results demonstrate that HIP base exchange and ESP traffic are supported end-to-end with RSA, ECDSA, and EdDSA suites.

4.2.1 Test Environment Initialization

The suite successfully initialized the testing environment and directories, confirming readiness for execution. This step ensured that the OpenHIP framework could consistently reproduce test conditions and validate integration across multiple components.

Algorithm Compatibility Validation

The test suite verified that all supported algorithms could be generated and validated without errors. Specifically:

- **RSA:** Key generation and validation passed.
- **ECDSA:** Baseline ECDSA keys generated successfully.
- **Curve25519:** Keys generated and validated correctly.
- **Curve448:** Keys generated and validated correctly.

This confirms that OpenSSL 3.5 provides full support for modern elliptic curve cryptography, ensuring interoperability with HIP's identity and authentication mechanisms.

4.2.2 Binary Linkage Confirmation

All binaries were confirmed to link against OpenSSL 3.5+ libraries. This prevents runtime inconsistencies and ensures that HIP components consistently use the correct cryptographic backend.

4.2.3 Configuration File Compatibility

Existing configuration files were validated as compatible with OpenSSL 3.0. This ensures that legacy setups remain functional and that administrators can migrate without rewriting configuration suites. RSA and ECDSA suites were confirmed to load correctly under the `DEFAULT` configuration profile.

4.2.4 Key Generation Tests

The suite validated successful generation of:

- **ECDSA keys**
- **EdDSA Curve25519 and Curve448 keys**
- **RSA keys**

All keys were generated and validated without errors, confirming that OpenSSL 3.0 supports HIP's identity requirements across both traditional and modern cryptographic suites.

4.2.5 Provider Loading and Daemon Integration

The HIP help daemon was tested to ensure correct loading of OpenSSL providers. The daemon reported accurate metadata and confirmed provider availability. This validates OpenSSL's modular provider architecture within HIP, ensuring that cryptographic services are accessible to system daemons.

4.2.6 Version Compliance and Enforcement

The suite confirmed that OpenSSL 3.0 is correctly enforced and compliant with expected standards. Version detection via `pkg-config` reported OpenSSL 3.6.0, confirming proper system integration and alignment with the intended library version.

4.2.7 Script and Enforcement Checks:

The existence of required scripts was verified, and provider enforcement policies were confirmed. This ensures that HIP does not bypass OpenSSL's provider framework and that all cryptographic operations remain compliant with the enforced configuration.

4.2.8 Upgrade Validation Summary:

All tests passed, confirming:

- Successful upgrade to OpenSSL 3.5+.
- Full algorithm validation for RSA, ECDSA, and EdDSA curves.
- Correct provider integration and enforcement.
- Reliable version detection and binary linkage.

As shown in Figure 2, all tests passed successfully, confirming that OpenHIP is now fully interoperable with OpenSSL 3.5+ and provides a secure, standards-compliant cryptographic foundation for HIPv2.

4.2.9 OpenSSL Integration Details:

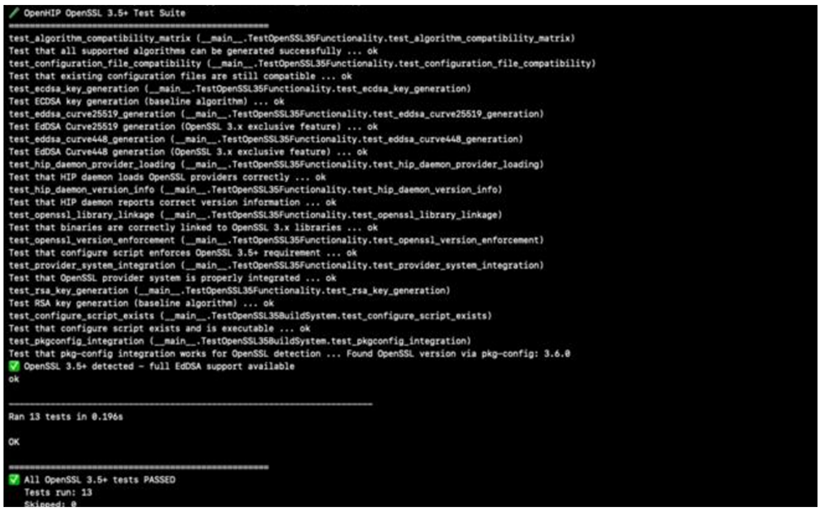
Because the OpenSSL integration required changes across build automation, daemon startup, and test coverage, several steps were introduced to ensure stability and compliance. These adjustments were implemented directly in the OpenHIP codebase and validated through automated tests:

- Added `pkg-config` checks for `openssl >= 3.5.0` in `configure.ac`, rejecting LibreSSL.
- Initialized OpenSSL 3.x providers (default and legacy) during HIP daemon startup in `hip_main.c`.
- Updated `build.sh` on macOS to auto-install `openssl@3` via Homebrew and run `bootstrap → configure → make`.
- Ensured `hitgen` generates EdDSA keys (Curve25519/448) and emits correct XML fields, verified through tests.
- Add `setup_openhip_ubuntu.sh`) to align with the original setup and reduce confusion.
- Verified that `hip` and `hitgen` link to `libssl.3` and `libcrypto.3` via automated checks.
- Confirmed RSA, ECDSA, and EdDSA (Curve25519/448) key generation and XML correctness in `test_openssl35.py`.
- Validated provider behavior and daemon startup messages through test coverage in `test_openssl35.py`.

These steps ensured that OpenHIP's OpenSSL integration was stable, portable, and fully validated across environments.

4.2.10 OpenHIP OpenSSL 3.5+ Test Suite

To validate OpenSSL 3.5+ integration with OpenHIP, a dedicated test suite was executed covering algorithms, configuration, key generation, provider loading, and version enforcement. All 13 tests passed, confirming correct linkage, provider integration, and full support for RSA, ECDSA, and EdDSA suites.



```
✏️ OpenHIP OpenSSL 3.5+ Test Suite
==================================================
test_algorithm_compatibility_matrix (__main__.TestOpenSSL35Functionality.test_algorithm_compatibility_matrix)
Test that all supported algorithms can be generated successfully ... ok
test_configuration_file_compatibility (__main__.TestOpenSSL35Functionality.test_configuration_file_compatibility)
Test that existing configuration files are still compatible ... ok
test_ecdsa_key_generation (__main__.TestOpenSSL35Functionality.test_ecdsa_key_generation)
Test ECDSA key generation (baseline algorithm) ... ok
test_eddsa_curve25519_generation (__main__.TestOpenSSL35Functionality.test_eddsa_curve25519_generation)
Test EdDSA Curve25519 generation (OpenSSL 3.x exclusive feature) ... ok
test_eddsa_curve448_generation (__main__.TestOpenSSL35Functionality.test_eddsa_curve448_generation)
Test EdDSA Curve448 generation (OpenSSL 3.x exclusive feature) ... ok
test_hip_daemon_provider_loading (__main__.TestOpenSSL35Functionality.test_hip_daemon_provider_loading)
Test that HIP daemon loads OpenSSL providers correctly ... ok
test_hip_daemon_version_info (__main__.TestOpenSSL35Functionality.test_hip_daemon_version_info)
Test that HIP daemon reports correct version information ... ok
test_openssl_library_linkage (__main__.TestOpenSSL35Functionality.test_openssl_library_linkage)
Test that binaries are correctly linked to OpenSSL 3.x libraries ... ok
test_openssl_version_enforcement (__main__.TestOpenSSL35Functionality.test_openssl_version_enforcement)
Test that configure script enforces OpenSSL 3.5+ requirement ... ok
test_provider_system_integration (__main__.TestOpenSSL35Functionality.test_provider_system_integration)
Test that OpenSSL provider system is properly integrated ... ok
test_rsa_key_generation (__main__.TestOpenSSL35Functionality.test_rsa_key_generation)
Test RSA key generation (baseline algorithm) ... ok
test_configure_script_exists (__main__.TestOpenSSL35BuildSystem.test_configure_script_exists)
Test that configure script exists and is executable ... ok
test_pkgconfig_integration (__main__.TestOpenSSL35BuildSystem.test_pkgconfig_integration)
Test that pkg-config integration works for OpenSSL detection ... Found OpenSSL version via pkg-config: 3.6.0
✅ OpenSSL 3.5+ detected - full EdDSA support available
ok

----------------------------------------------------------------------
Ran 13 tests in 0.196s

OK

==================================================
✅ All OpenSSL 3.5+ tests PASSED
   Tests run: 13
   Skipped: 0
```

Figure 2: "OpenHIP OpenSSL 3.5+ test suite output"

4.3 Wireshark Dissector for HIP v2

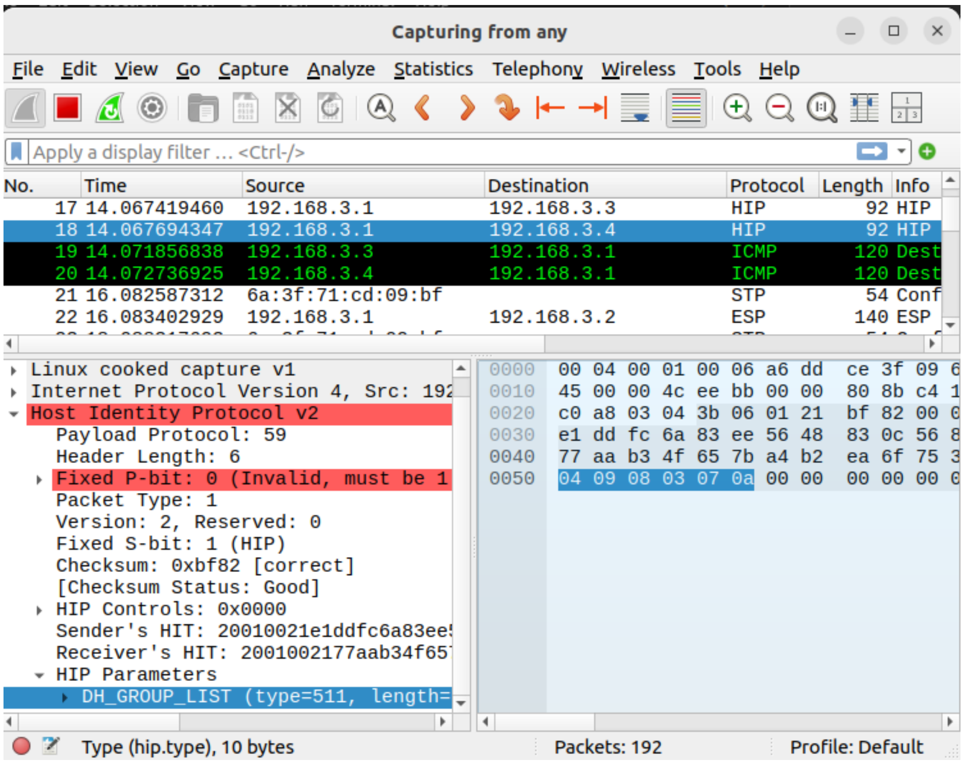
4.3.1 Approach

The work began with reading the rfc documentation for HIP v2 [8] and identifying the changes compared to the original HIP rfc [9]. Using the Minnet simulation the HIP v1 Wireshark dissector was run on the routers to observe the traffic. As expected many packet types were unidentifiable by the dissector and not able to be interpreted.

For the new wireshark dissector we first downloaded the raw source files from Git and the necessary libraries needed for building. We then compiled the entire project to a run file and verified it worked as expected. The first step of implementing a custom dissector was to write a simple test dissector *foo.c* that simply should detect and report custom packets sent. After compiling wireshark again and fixing all errors two simple python scripts were created to simulate packets sent over the network, *server.py* and *client.py*. After verifying and testing the dissector work on the HIP v2 dissector started.

4.3.2 Dissector Experimentation

The first dissector was built only had basic functionality to test the Minnet routers could access the custom Wireshark implementation. After confirming the routers had access to the custom Wireshark the source code for Wireshark Dissector HIP v1 was imported and small changes were made for testing. The larger dissector file however lead to very long compile times, often several minutes. After experimentation it was discovered to be much quicker to include the HIP dissector inside the Wireshark dissector source folder instead of including it as a plugin. A marginal time difference was also discovered between using ninja or make as build system. Ninja was consistently faster to compile Wireshark.



Capturing from any
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.  Time                Source              Destination     Protocol  Length Info
  17 14.067419460   192.168.3.1         192.168.3.3      HIP          92 HIP
  18 14.067694347   192.168.3.1         192.168.3.4      HIP          92 HIP
  19 14.071856838   192.168.3.3         192.168.3.1      ICMP        120 Dest
  20 14.072736925   192.168.3.4         192.168.3.1      ICMP        120 Dest
  21 16.082587312   6a:3f:71:cd:09:bf                    STP          54 Conf
  22 16.083402929   192.168.3.1         192.168.3.2      ESP         140 ESP
```

```
▸ Linux cooked capture v1
▸ Internet Protocol Version 4, Src: 192
▾ Host Identity Protocol v2
    Payload Protocol: 59
    Header Length: 6
  ▸ Fixed P-bit: 0 (Invalid, must be 1
    Packet Type: 1
    Version: 2, Reserved: 0
    Fixed S-bit: 1 (HIP)
    Checksum: 0xbf82 [correct]
    [Checksum Status: Good]
  ▸ HIP Controls: 0x0000
    Sender's HIT: 20010021e1ddfc6a83ee
    Receiver's HIT: 2001002177aab34f65
  ▾ HIP Parameters
    ▸ DH_GROUP_LIST (type=511, length=
```

```
0000  00 04 00 01 00 06 a6 dd  ce 3f 09 6
0010  45 00 00 4c ee bb 00 00  80 8b c4 1
0020  c0 a8 03 04 3b 06 01 21  bf 82 00 0
0030  e1 dd fc 6a 83 ee 56 48  83 0c 56 8
0040  77 aa b3 4f 65 7b a4 b2  ea 6f 75 3
0050  04 09 08 03 07 0a 00 00  00 00 00 0
```

Type (hip.type), 10 bytes   Packets: 192   Profile: Default

Figure 3: Wireshark example with new DH-Group-LIST, P-bit check and alignment check

4.3.3 Updating the HIP Dissector

New parameters added, HIT-SUITE-LIST, TRANSPORT-FORMAT-LIST, DH-GROUP-LIST. HMAC and HMAC-2 changed name to HIT-MAC and HIT-MAC-2. R1-COUNTER changed value to 129. Support detection of new cryptography algorithms, Elliptic Curve RSA and Elliptic Curve Diffie-Hellman.

In HIP v1 the padding of packets were optional thus the length parameter of the the packets included the length of the padding. In HIP v2 padding to an alignment of 8 bytes are mandatory, therefore the length parameter no longer need to include the padding. The Wireshark v2 dissector were modified to enforce the packets are aligned to 8 bytes. Otherwise Wireshark alert the user. Another new rule that was changed to be hard enforced was the P-bit must be set in the parameter headers, it will now alert if it does not.

4.3.4 Testing HIP v2 Dissector

For all the testing of the HIP v2 dissector the HIP v1 dissector was excluded from the custom Wireshark build. The new dissector was tested on Minnet. All the packet types could be identified. When a packet not following the new alignment standard was sent the dissector alerted for malformed packet.

5 Future Work

5.1 OpenSSL

- Expand algorithm support beyond RSA and EdDSA, deciding whether to remove or repair incomplete suites such as DSA and EcDSA.
- Refine provider integration with further testing to ensure stability and security across different platforms.
- Improve performance by extending continuous integration pipelines with automated regression and throughput tests.
- Maintain ongoing alignment with evolving RFCs and cryptographic standards to ensure interoperability and compliance.

5.2 Wireshark

- Ensure that Wireshark detect what HIP version is used and utilize the correct dissector.
- Alternatively integrate HIP v1 and v2 dissector into one program and inform the user if a packet follow v1.
- Improve Tshark utility, starting and switching between the Wireshark GUI for each router is very inconvenient.
- Add enforcement of more specific rules for the packet types.

5.3 PyHIP

- Use the existing throughput-testing framework as a baseline for evaluating future changes.
- Investigate architectural changes that reduce reliance on Python raw sockets, maybe by offloading performance-critical paths to native code.

- Revisit execution models for `switchd.py`, informed by documented multiprocessing failure modes from this project.
- Successfully do profiling, to find potential bottlenecks
- Extend HIP message handling to support for example UPDATE exchanges and rekeying as specified in RFC 7401 [8].

6 Discussion

6.1 PyHIP

The PyHIP results seem to suggest that throughput is currently constrained by architectural aspects of the Python implementation rather than by individual cryptographic primitives. The restored Mininet lab and the scripted `analysis/throughput_test.py` benchmark provide reproducible evidence of a stable but limited throughput ceiling.

Although profiling was attempted, no usable results were obtained due to implementation issues and the interaction between profiling tools, Mininet, and raw sockets. As a result, performance analysis was based on end-to-end throughput measurements rather than detailed function-level timing.

Multiple multiprocessing-based designs for `switchd.py` were explored in an attempt to overcome Python interpreter limitations. These efforts exposed fundamental challenges, including state desynchronization during the HIP base exchange, raw socket failures across process boundaries, and ineffective queue-based designs. Together, these results indicate that a naive transition from threading to multiprocessing is insufficient to improve throughput.

In contrast, the cryptographic changes were successful and improved the robustness and longevity of the PyHIP implementation. Adding AES-256-GCM and migrating to ECDSA/SHA-384 align the system with modern security recommendations while maintaining compatibility with the existing routing and benchmarking setup.

Overall, the 2025 PyHIP work prioritised correctness, reproducibility, and understanding of system limitations. While no throughput gains were achieved, the project leaves behind a reliable measurement framework, documented failure modes for multiprocessing, and a more future-proof cryptographic configuration that future groups can build upon.

6.2 OpenSSL

The move to OpenSSL 3.5+ in OpenHIP brings up several important points for future work. One key issue is making the system run reliably across different platforms. The migration has been tested on Linux and macOS, but keeping builds consistent everywhere is still a challenge. Using container tools or continuous integration pipelines could make the process more reliable and reduce the need for manual setup.

Another point is the stability of OpenSSL's provider system. Providers make it possible to load cryptographic services in a flexible way, but they also add complexity. It will be important to study how stable and secure this model is over time, especially as new algorithms are added and older ones are phased out.

The future of cryptographic algorithms also needs attention. RSA is still supported, but its long-term use is becoming less secure. A gradual shift toward EdDSA and other elliptic-curve algorithms will help OpenHIP stay in line with modern standards.

Automation is another area for improvement. Adding cryptographic regression tests to continuous integration pipelines would make the system more reliable and help catch problems early. This would also make it easier for future student groups to reproduce results and validate changes.

Finally, security and compliance with standards remain ongoing priorities. As RFCs and regulations evolve, OpenHIP must keep up to date. Regular monitoring of cryptographic standards and proactive updates will be necessary to ensure interoperability and trustworthiness.

6.3 Wireshark Dissector

While there are many significant changes between HIP v1 and v2 it reflect some modest between the new and old Wireshark dissector. During development several problems were encountered. The first big challenge was very long build times, often exceeding ten minutes. This leads to slow iterations on code and a simple typo cold and did cause long delays when waiting to rebuild. There dissector had been added as plugin and for the simple this has proven relatively fast. The HIP v2 dissector had many more dependencies and much more closely with the rest of the Wireshark source code. Probably because of this large parts of the entire Wireshark code had to also be rebuilt. When testing placing the dissector inside the main dissector folder yielded much better build time. Depending on what parts of the code changed the build time could be as low as a few minutes. It should be noted however that a VM with more RAM will give more time savings as this was the biggest bottleneck when building.

HIP v2 added several new rules that according to the specification should be enforced. While this is not strictly necessary for the dissector to be aware of the decision to alert the user if either the alignment was of or P-bit not set were implemented because this signify a major problem with the HIP structure. Checks for other rules such as, all the required parameters are included in their respective packets, could be implemented. However those rules are more specific for each packet type and are not a general for all HIP packets.

7 Conclusion

7.1 PyHIP

While throughput remains constrained by architectural aspects of the Python implementation, the project delivers clearer experimental infrastructure, documented design limitations, and a more future-proof cryptographic configuration. These results provide a solid foundation for future attemps to pursue deeper architectural refactoring, performance improvements or cryptographic additions.

7.2 OpenSSL

The OpenSSL migration effort has reached a stable and validated endpoint. All 13 tests in the OpenSSL 3.5+ suite passed successfully, confirming full support for RSA, ECDSA, and EdDSA algorithms, correct provider integration, and reliable version detection. The HIP daemon and system components are now fully interoperable with OpenSSL 3.5+. This achievement secures the cryptographic integrity of OpenHIP and lays the groundwork for future upgrades, including post-quantum readiness and continuous performance monitoring.

7.3 Wireshark Dissector

Using Wireshark to inspect the HIP packets facilitate easier development of future HIP versions and error searching. While it remain somewhat cumbersome to run the Wireshark GUI for each it is now functional for the new HIP v2 feature. All the milestones for the Wireshark HIP v2

References

[1] The OpenSSL Project Authors. Openssl guide: An introduction to libcrypto. OpenSSL Documentation. `https://docs.openssl.org/master/man7/ossl-guide-libcrypto-introduction/`, September 2024.

[2] CORE Developers. CORE network emulator documentation. `https://coreemu.github.io/core/`, 2024.

[3] S. Housley. The use of galois/counter mode (gcm) in ipsec esp. RFC 4106. `https://www.rfc-editor.org/info/rfc4106`, June 2005.

[4] S. Kent. Ip encapsulating security payload (esp). RFC 4303, Internet Engineering Task Force (IETF), December 2005. `https://datatracker.ietf.org/doc/html/rfc4303`.

[5] A. Laganier and F. Dupont. An ipv6 prefix for overlay routable cryptographic hash identifiers version 2 (orchidv2). RFC 7343. https://www.rfc-editor.org/info/rfc7343, September 2014.

[6] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, New York, NY, USA, 2010. Association for Computing Machinery. https://doi.org/10.1145/1868447.1868466.

[7] Robert Moskowitz et al. Hierarchical host identity tags (hhit). RFC 9374. https://www.rfc-editor.org/info/rfc9374, August 2023.

[8] Robert Moskowitz, Tobias Heer, Petri Laari, and Thomas R. Henderson. Host Identity Protocol Version 2 (HIPv2). RFC 7401. https://www.rfc-editor.org/info/rfc7401, April 2015.

[9] Robert Moskowitz, Petri Laari, Tom Henderson, and Pekka Nikander. Host Identity Protocol. RFC 5201. https://www.rfc-editor.org/info/rfc5201, April 2008.

[10] Robert Moskowitz and Pekka Nikander. Host identity protocol (hip) architecture. RFC 4423. https://www.rfc-editor.org/info/rfc4423, March 2006.

[11] National Institute of Standards and Technology. FIPS 180-4: Secure hash standard (shs). Federal Information Processing Standards Publication. https://csrc.nist.gov/pubs/fips/180-4/upd1/final, August 2015.

[12] National Institute of Standards and Technology. FIPS 186-5: Digital signature standard (dss). Federal Information Processing Standards Publication. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf, February 2023.

[13] The Wireshark Foundation. Wireshark documentation: User's guide and developer's guide. https://www.wireshark.org/docs/, 2025. Accessed 2025-12-14.