James Madison University
JMU Scholarly Commons

Masters Theses, 2020-current

The Graduate School

5-11-2023

Enabling security analysis and education of the Ethereum platform: A network traffic dissection tool

Joshua Mason Kemp
James Madison University

Follow this and additional works at: https://commons.lib.jmu.edu/masters202029

Part of the Computer and Systems Architecture Commons, and the Digital Communications and Networking Commons

Recommended Citation

Kemp, Joshua Mason, "Enabling security analysis and education of the Ethereum platform: A network traffic dissection tool" (2023). *Masters Theses, 2020-current*. 214.
https://commons.lib.jmu.edu/masters202029/214

This Thesis is brought to you for free and open access by the The Graduate School at JMU Scholarly Commons. It has been accepted for inclusion in Masters Theses, 2020-current by an authorized administrator of JMU Scholarly Commons. For more information, please contact dc_admin@jmu.edu.

Enabling Security Analysis and Education of the Ethereum Platform:
A Network Traffic Dissection Tool

Joshua Mason Kemp

A thesis submitted to the Graduate Faculty of

JAMES MADISON UNIVERSITY

In

Partial Fulfillment of the Requirements

for the degree of

Master of Science

Department of Computer Science

May 2023

FACULTY COMMITTEE:

Committee Chairs: Dr. Emil Salib and Dr. Mohamed Aboutabl

Committee Members/ Readers:

Dr. M. Heydari

Dr. Brett Tjaden

Acknowledgments

I would like to express my sincere gratitude to the following people who have provided invaluable support and guidance throughout my journey:

First and foremost, I would like to thank my thesis advisors Dr. Emil Salib and Dr. Mohamed Aboutabl, for their unwavering support and guidance throughout the research process. Their insightful, feedback and constructive criticism were instrumental in shaping this thesis.

I would also like to thank my family and friends for their unconditional love, unwavering support, and patience throughout this journey. Their encouragement has been a constant source of motivation for me.

I would like to extend my gratitude to the faculty and staff of the Computer Science Department at James Madison University, for their assistance and support throughout all my studies.

Thank you all for your support, encouragement, and guidance throughout this journey.

Table of Contents

List of Figures

List of Tables

Abstract

Ethereum, the decentralized global software platform powered by blockchain technology known for its native cryptocurrency, Ether (ETH), provides a technology stack for building apps, holding assets, transacting, and communicating without control by a central authority. At the core of Ethereum's network is a suite of purpose-built protocols known as DEVP2P, which provides the underlying nodes in an Ethereum network the ability to discover, authenticate and communicate confidentiality. This document discusses the creation of a new Wireshark dissector for DEVP2P's discovery protocols, DiscoveryV4 and DiscoveryV5, and a dissector for RLPx, an extensible TCP transport protocol for a range of Ethereum node capabilities. Network packet dissectors like Wireshark are commonly used to educate, develop, and analyze underlying network traffic. In support of creating the dissector, a custom private Ethereum docker network was also created, facilitating the communication amongst Go Ethereum execution clients and allowing the Wireshark dissector to capture live network data. Lastly, the dissector is used to understand the differences between DiscoveryV4 and DiscoveryV5, along with stepping through the network packets of RLPx to track a transaction executed on the network.

Keywords: Ethereum, Dissector, DEVP2P, DiscoveryV4, DiscoveryV5, RLPx, RLP, ECIES, ECDH, ECDSA, Wireshark, Python, Lua, Go

1. Introduction

Ethereum, launched in 2015 as a toolkit to build decentralized applications, transact and communicate without a controlled central authority while also providing a framework for Ethereum nodes to facilitate communication [1]. Ethereum's native cryptocurrency, Ether (ETH), is positioned second in terms of market cap under Bitcoin with $146.8 billion in circulation, over 71 million wallets holding a balance, and handling $11.6 trillion in just 2021 [2]. As Ethereum grows in popularity, acquiring the interest of the masses, it is becoming imperative that the inner workings of the Ethereum network are examined and understood on a deeper level particularly transaction flows and the algorithms used to secure them.

1.1 Ethereum Network Background

An Ethereum node is simply a computer connected to the Ethereum network, running the specific tools required to communicate amongst other nodes. Ethereum nodes help maintain the decentralized network by validating transactions within data blocks, referred to as their consensus mechanism. The communication amongst these nodes resides on the Ethereum network, composed of a custom-built network protocol suite known as DEVP2P. DEVP2P provides a mechanism for nodes to discover one another throughout the network, authenticate with each other, and communicate amongst themselves over a secure channel with a wide range of node-specific capabilities. These capabilities can be implementation dependent or be known as Ethereum capabilities to support state management and synchronization with SNAP, block propagation, and transactions with ETH. DEVP2P started along with the Ethereum project to provide a set of protocols that can serve any networked application under the Ethereum umbrella. This means DEVP2P served most if not all, the network communication under the hood among decentralized applications (dAPPs), handling $11.6 trillion in transactions in just 2021 alone.

However, on September 6th, 2022, the Ethereum Merge took place, transitioning from a proof-of-work to a proof-of-stake consensus algorithm while integrating the existing execution layer with a new consensus layer [3]. Each layer has specific jobs and networks broken down into two different types of clients. Execution clients utilize the previous existing DEVP2P execution-layer network stack, gossiping transactions and requiring encrypted communication amongst authenticated peers. Consensus clients thus utilize the new consensus-layer network, utilizing a different p2p network stack known as LIBP2P, used for gossiping beacon blocks throughout the p2p network [4]. Together execution clients and consensus clients make up an Ethereum Mainnet node, where both DEVP2P and LIBP2P exist together, requiring their own methods for discovery and communication protocols. Due to "the merge", the Ethereum network has become increasingly more complex. The need to understand the intercommunication between Ethereum nodes increased significantly to understand the exact use of DEVP2P post-merge or to analyze the security and performance of underlying algorithms and protocols.

As the Ethereum network grows in its usage and importance throughout the world while also growing in complexity, a tool must exist to aid with understanding and analyzing the inner workings of the underlying protocols utilized throughout the Ethereum network. DEVP2P provides a wide array of capabilities related to a peer-to-peer networking schema, with two major components, (a) discovery and (b) authenticated and encrypted communication. Discovery is facilitated by two somewhat unrelated protocols, DiscoveryV4 and DiscoveryV5 [15]. DiscoveryV4 is the original protocol "version," where messages are sent in the clear with little to no authentication of peers. DiscoveryV5 was meant to be the successor to DiscoveryV4 to make it more secure and faster, but implementations of DiscoveryV5 were only completed in the GO Ethereum version of the Ethereum execution client. This implementation was purely experimental to test DiscoveryV5; its use never saw daylight amongst other implementations except in consensus clients in place of the standard LIBP2P discovery mechanism. The other major component of DEVP2P is RLPx, the TCP-based transport protocol used for authenticated and confidential communication among Ethereum nodes after peers discover one another.

1.2 Problem Statement

Analyzing DEVP2P provides a vehicle for conceptualizing the inner workings of the Elliptic Curve Integrated Encryption Scheme (ECIES) as it pertains to RLPx, understanding the security and performance differences between discoveryV4 and discoveryV5, the two UDP-based discovery mechanisms, and lastly, tracing the usage of RLPx concerning block propagation, chain synchronization, state management, and transaction processing. Traditionally, a network traffic dissector tool provides a window into the communications amongst networked assets such as nodes. Dissectors are commonly used for debugging, protocol analysis, security and scalability analysis, and, lastly, for educational purposes. The best-known and most utilized tool for dissecting network packets is Wireshark. It intercepts network traffic via the kernel in a non-intrusive manner and provides a live view of the frames and packets flowing through a link. It allows us to identify protocols, decode data, follow streams and conversations, calculate statistics, and more [5].

Currently, two known packet dissectors exist for Ethereum's DEVP2P protocol suite; one was built off of Wireshark's plugin engine using the programming language LUA and one compiled with Wireshark source code using the programming language C. These packet dissectors come with limitations, created around five years ago, only supporting the encryption-less DiscoveryV4, minus the newer packet types released in EIP-868 in October 2019. Ethereum Improvement Proposals (EIPs) describe standards for the Ethereum platform, including core protocol specifications, client APIs, and contract standards [6]. The LUA dissector was built by BCSEC organization, also known as Blockchain Security org, this group has since been disbanded but was known as a security group aiming "to elevate the security of the entire blockchain ecosystem" [7]. The second packet dissector built with C, was created by PegaSys, now known as ConsenSys, a large corporate player in the Ethereum and blockchain market whose "mission is to build blockchain solutions ready for production in business environments" [8]. Both do not dissect the newer packet types in DiscoveryV4, while also not having support for DiscoveryV5 and RLPx and any of its sub-protocols such as ETH and SNAP.

Both projects have been abandoned, citing reasons for complexity and pushing the open-source community to finish the job. The reason for this complexity will be touched on a great deal throughout this report. As stated by PegaSys in August of 2018, the process of dissecting RLPx is "somewhat complicated, as TCP connections are encrypted with an AES symmetric key derived per-session via ECIES (Elliptic Curve Integrated Encryption Scheme)". This means the dissector must have "access to the private key of the local node, it would not be enough to decrypt communications, as the encryption key factors in our private key, the public key of the node, and a randomly generated ephemeral key [5]."

With this, we propose a new tool, a network packet dissector, explicitly used for dissection and analysis of Ethereum's DEVP2P protocols found in execution clients, including their UDP-based DiscoveryV4, DiscoveryV5, and RLPx, including its sub-protocol capability messages ETH and SNAP. After successfully dissecting, deciphering, and decrypting the contents of the network payloads among Ethereum nodes on the network, we will then utilize the dissector tool to prove its value as a dissector for the community and educators while also providing a deeper analysis of the security and performance differences between DiscoveryV4 and DiscoveryV5 while also looking at RLPx, transactions, and block propagation. This Wireshark dissector for DEVP2P can provide network-level security and performance analysis for the Ethereum community and educators.

1.3 Contributions

Throughout this document, many contributions will be made in order to meet the goal of creating a Wireshark dissector for DEVP2P, as found on execution clients like Go Ethereum (GETH), specifically the protocols DiscoveryV4, DiscoveryV5 and RLPx including the ETH and SNAP sub-protocols. Then using this created dissector to analyze live network traffic between nodes/peers of an Ethereum network. Each of these will be explained throughout this document in detail, the main contributions are completely new. The contributions for this thesis are as follows. Please see the Appendix 7.1 to locate these contributions.

- discovery.lua and rlpx.lua - The main interface between Wireshark and the dissectors, facilitating the packet capture data and sending it off to PYDEVP2P

- PYDEVP2P - The backend to the dissectors, Python-based with very minimal 3rd party dependencies, provides most of the deciphering, decryption, and packet layout tooling required for Wireshark display.
- Go Ethereum Docker Images and Network
- Go Ethereum Source Code Modifications
- Lunatic Python Modifications

1.4 Organization

As many packet types are captured within the Ethereum network, the organization of this document will step through a scenario for each protocol and its use case from a structured and controlled test network. This scenario, in the next chapter, Chapter 2, will provide a vehicle for understanding at a high level how an Ethereum node is used by users/accounts. This scenario will briefly describe the Docker development network used throughout this document and visualize multiple nodes connecting, followed by a transaction between two nodes.

Then, some background information on the inner workings of the Ethereum network will be discussed in Chapter 3, followed by the use of existing dissectors to understand better discovery followed and their shortcomings.

Then, in Chapter 4, the steps are taken to create the dissector for each protocol within DEVP2P. We will provide the work that went into dissecting and displaying the packet information for each packet type within Wireshark while also explaining the message contents and use of the packet as related to the scenario outlined. This will provide a method to understand the flow of DEVP2P packets during each stage of communication while understanding the primary purpose for each contribution, including the Docker network, LUA Dissector plugins, and the PYDEVP2P dissection library. Starting with the discovery phase, showing the use of DiscoveryV4 and DiscoveryV5 on GETH clients, then unraveling RLPx packets, including ETH and SNAP messages found as part of the RLPx transport protocol.

After discussing the creation of the dissector and understanding the flow of DEVP2P packets amongst nodes on an Ethereum network, performance, and security characteristics will be analyzed for both DiscoveryV4 and DiscoveryV5, in Chapter 5. Lastly, the actual transaction that took place during the scenario will be analyzed with the dissector, understanding on a message-by-message approach what took place on the network level to fulfill the transaction amongst two nodes.

2. Scenario

It is important to understand just how an Ethereum node is used, whether it is connecting to the network via a Bootnode, connecting and communicating with peers, and of course validating and transacting amongst one another. This chapter will step through each process at a high level from connecting a node to our own private/development network, with a network of 3 nodes and 1 bootnode (See Figure 1). In this scenario each node will be connected to the same chain, where proof-of-work is the underlying consensus algorithm amongst nodes.

2.1 Custom Network Description

To start, a custom private development will be used, using the ETH cryptocurrency and running GETH clients on each node. Each node will be running on a docker container, custom built to support certain network connectivity operations and mainly to spin up several very quickly. The details of the creation of this docker network will be explained in the "Creating the Dissector" Chapter 4. Each node will communicate and exist on the Chain ID: "12345", which again is a unique identifier for a given network, for similarly configured peers to connect.

As stated before, there will be one BootNode, which is a specially designated node that acts as the initial point of contact for new nodes attempting to join the network. When a new node wants to join the network, it contacts the bootnode to get a list of active nodes that can provide it with additional information about the network. The bootnode is typically a highly available and reliable node that is maintained by the Ethereum development team or a trusted third-party service. It plays a critical role in ensuring the stability and security of the network by helping to distribute new nodes across the network and preventing the formation of isolated clusters. There will also exist 3 other nodes on the network, each running the latest version of GETH as of this writing, and each node will be a miner on the network. Lastly, for each docker container to reside in their own subnet, a custom bridge router docker container is set up, to facilitate the connectivity amongst the node docker containers. Below, Figure 1 depicts the network topology, followed by networking details in Table 1 used throughout the scenario, dissection and analysis chapters.

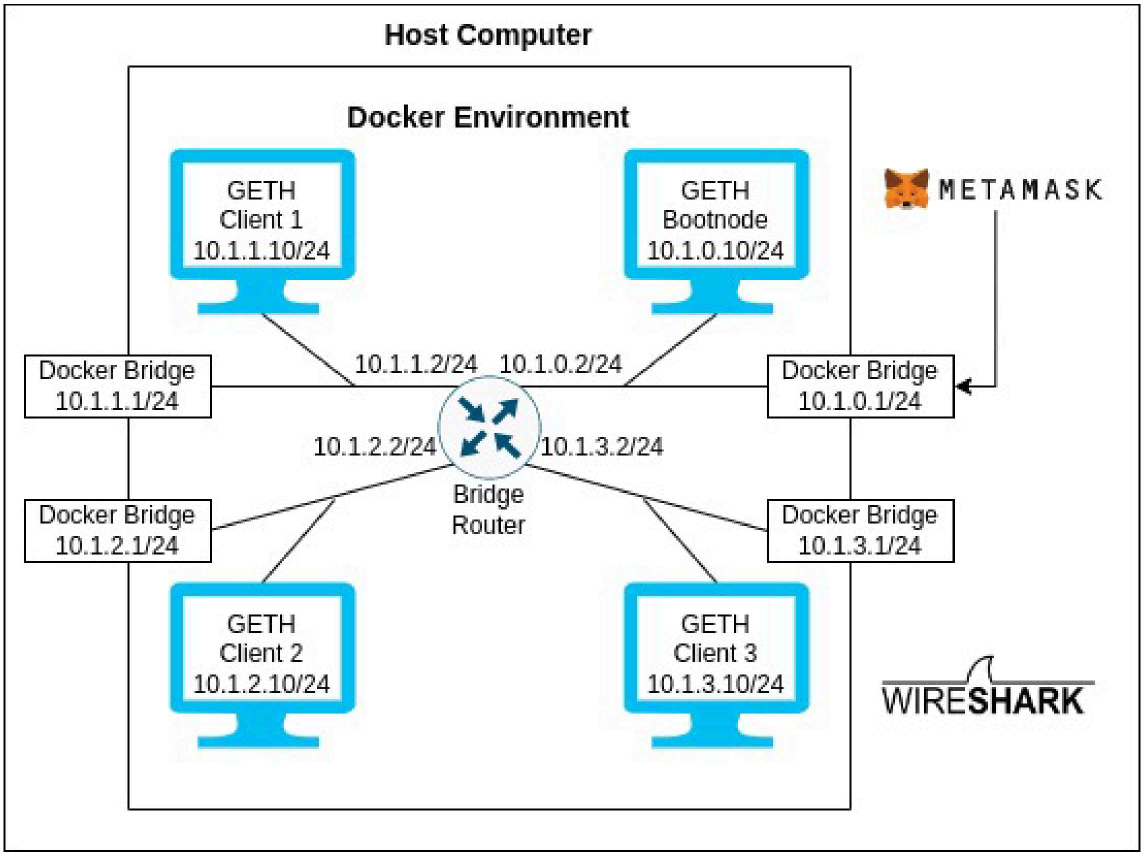


Figure 1: Docker Go Ethereum Private Network

Table 1: Docker Container Network Interfaces

Container Name	IP Address	UDP/TCP Port	Interfaces	HTTP RPC Port
geth-ubuntu-bootnode	10.1.0.10/24	30303	eth0	8545
geth-client-1	10.1.1.10/24	30304	eth0	-
geth-client-2	10.1.2.20/24	30305	eth0	-
geth-client-3	10.1.3.30/24	30306	eth0	-
bridge-router	10.1.0.2/24, 10.1.1.2/24, 10.1.2.2/24, 10.1.3.2/24	-	eth0, eth1, eth2, eth3	-

2.2 Ethereum Client Accounts

On an Ethereum node, and more specifically on clients such as Go Ethereum (GETH), accounts can be created and used to manage the ownership and transfer of the cryptocurrency associated with the Chain ID. Each account has its own private/public elliptic curve key pair, and is identifiable by its own unique address, derived from the account's public key. This keypair is used solely for account authentication, and in the event the keypair is lost, the user is no longer able to access their assets. It is important to note that this elliptic curve keypair is separate and distinct from that of the actual underlying Node's keypair, used for uniquely identifying the node in terms of discovery and communication. This account keypair is stored in what is known as a keystore file, located in the filesystem containing an encrypted version of the account secret key, along with the necessary parameters to decrypt it, requiring the use of the account password. The accounts created on their respective container/node account addresses are shown in Table 2. Clients like GETH can have several accounts associated with it, each of which is managed independently of the others. Each account has its own balance of ETH, or the chain's specified cryptocurrency, and can send and receive transactions on the Ethereum network.

Table 2: GETH Client Node Account Addresses

Container Name	Account Address
geth-ubuntu-bootnode	0x6DED7354774DA5056AE8E3C52484E2CDA3F6F788
geth-client-1	0x41159606B6240F725E969E3F1F342FF65904A4EC
geth-client-2	0x1F0CEBF80F05DE1213401C6D0A58E215C8CE635F
geth-client-3	0x11BEE17E6D6835AA46197990ADB681BA3A1B4435

2.3 Starting the Private Network

By starting the docker environment, effectively with "docker-compose up", the entire network will come alive, including the bridge router, the bootnode and the three nodes. Shortly after, the three nodes will reach out to their configured bootnode to join the network. Once joined they will then perform their normal peer-to-peer discovery along with authenticated and confidential communication. Below, depicts the command line output of each client, in this instance, periodically searching for new peers on the network, seen in Figure 2. Note the "peercount" on each, showing that they have 3 connected peers, not including themselves. As each of the clients are configured to mine for their respective accounts on startup, the command line output will also show their progress in mining potential new blocks and committing their work to their peers throughout the network, seen in Figure 3.

```
geth-bootnode          | INFO [02-27|19:28:18.419] Looking for peers
             peercount=3 tried=0 static=0
geth-client-1          | INFO [02-27|19:28:18.583] Looking for peers
             peercount=3 tried=0 static=0
geth-client-2          | INFO [02-27|19:28:19.346] Looking for peers
             peercount=3 tried=0 static=0
geth-client-3          | INFO [02-27|19:28:19.761] Looking for peers
             peercount=3 tried=0 static=0
```

Figure 2: GETH Clients Looking for Peers with Peer Count of 3

```
geth-client-2          | INFO [02-27|19:32:30.410] Successfully sealed new block
bf8d hash=f31cec..9831f1 elapsed=975.665ms
geth-client-2          | INFO [02-27|19:32:30.410] 🔨 mined potential block
1
geth-client-2          | INFO [02-27|19:32:30.410] Commit new sealing work
bd7c uncles=0 txs=0 gas=0     fees=0       elapsed="100.2µs"
geth-client-2          | INFO [02-27|19:32:30.410] Commit new sealing work
bd7c uncles=0 txs=0 gas=0     fees=0       elapsed="178.1µs"
geth-bootnode          | INFO [02-27|19:32:30.417] Imported new chain segment
 blocks=1 txs=0 mgas=0.000 elapsed=3.914ms   mgasps=0.000 dirty=62.65KiB
```

Figure 3: GETH Clients Mining Potential Blocks and Importing Chain Segments

2.4 Connecting MetaMask

Now that the network is up and running, it is time to show how it can be used, just like in a real-world scenario. Instead of using the GETH built in command line interface or CLI to view account balances and transact, MetaMask will be used. MetaMask is the leading self-custodial wallet, used to interact with the Ethereum blockchain or even private development Ethereum networks. It allows users to access their Ethereum wallet through a browser extension or mobile app, which can then be used to interact with decentralized applications. This interface provides a real-world experience when pulling up the individual accounts from any of the nodes on the custom private docker network [9].

GETH provides a way for third party applications to interact with the client by sending requests to the JSON-RPC API endpoint. This can be enabled via a flag when starting up GETH from the command line, specifically, in this scenario, the HTTPS transport will be used and utilizing the default RPC port 8545. MetaMask is extremely lightweight and can be installed as a browser extension on any popular browser, or even as an iPhone or Android application. Once installing MetaMask, from the Settings > Network page, this local private network can be connected to, using the RPC HTTPS endpoint port and the chain ID that each GETH node is configured with, seen Figure 4 [10].

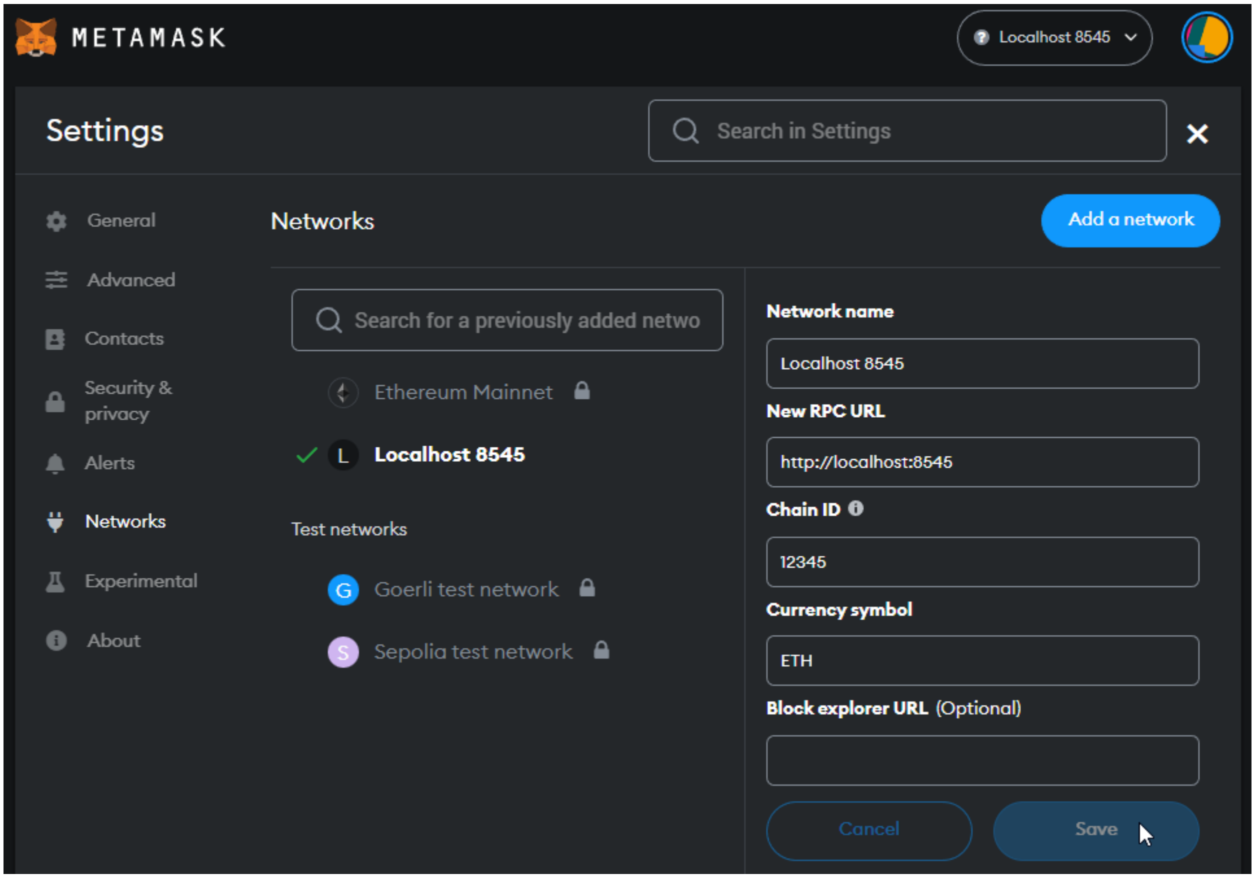


Figure 4: Connecting to Private Network from MetaMask

2.5 Connecting Accounts and Transacting ETH

Once the local private Ethereum network is connected just like how another network or even the Mainnet is connected, it is time to link the accounts that exist. This is as simple as importing the keystore file found on each of the GETH nodes, found in ~/.ethereum/keystore/, shown in Figure 5. Again, this keystore file contains the encrypted account private key along with the necessary AES parameters to decrypt it, which requires the use of the account password as well. After importing, these accounts can be seen from the profile dropdown menu located at the upper right-hand corner, while making sure to have the local private network selected as well. From here, we can now see the actual balance of the accounts as well, labeled manually to correspond to which node the account exists, shown in Figure 6. These accounts are named after the node they reside on in the private Ethereum network, this does not mean that "Node 1" has X amount of Ether. Nodes facilitate account connections to the network and facilitate transactions.

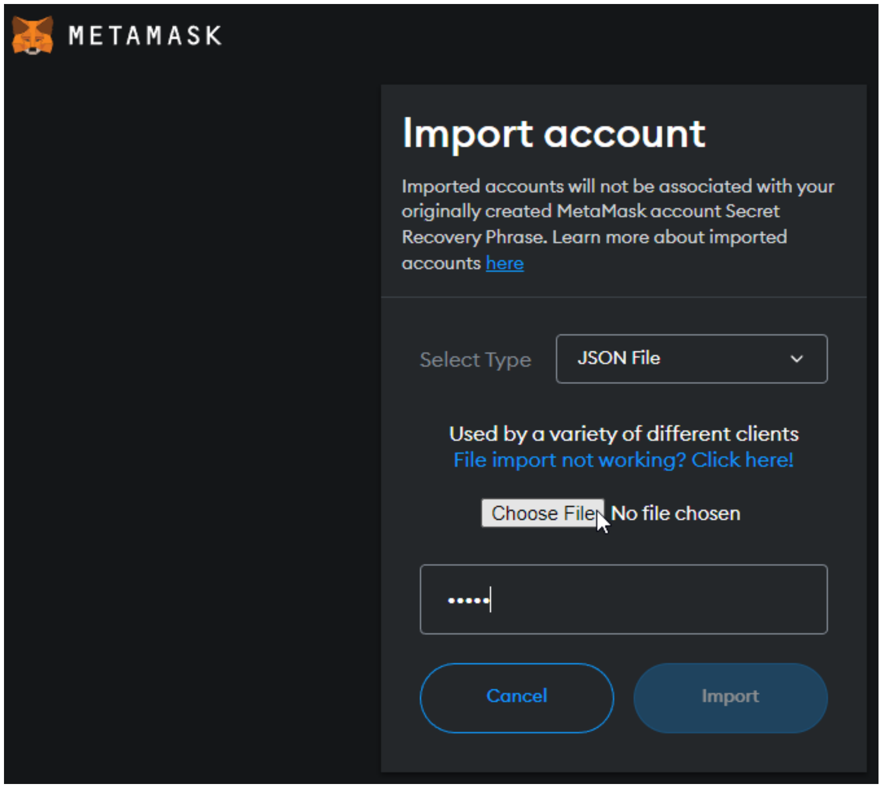


Figure 5: Importing Accounts Found on GETH Clients into MetaMask

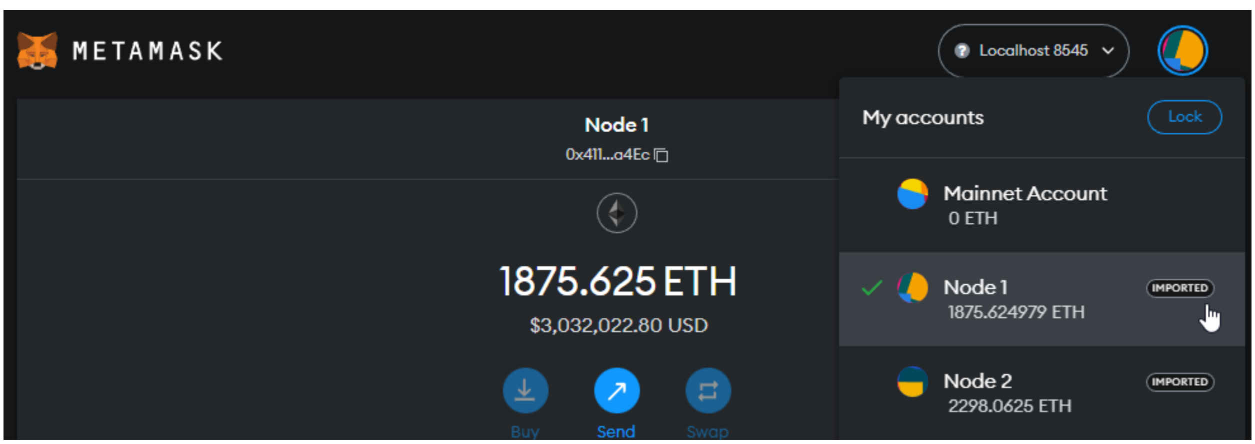


Figure 6: Showing Connected Accounts and their ETH Balance

It is important to note that the amount in USD is depicted as the current conversion rate from ETH to USD, however, in this test network as the chain started from scratch, the difficulty was rather low with a lack of competition as well. Lastly, to complete the scenario, Account/Node 1 will transact with Account/Node 2, sending 200 ETH to Node 2, steps shown in sequential order in Figures 7 through 10.

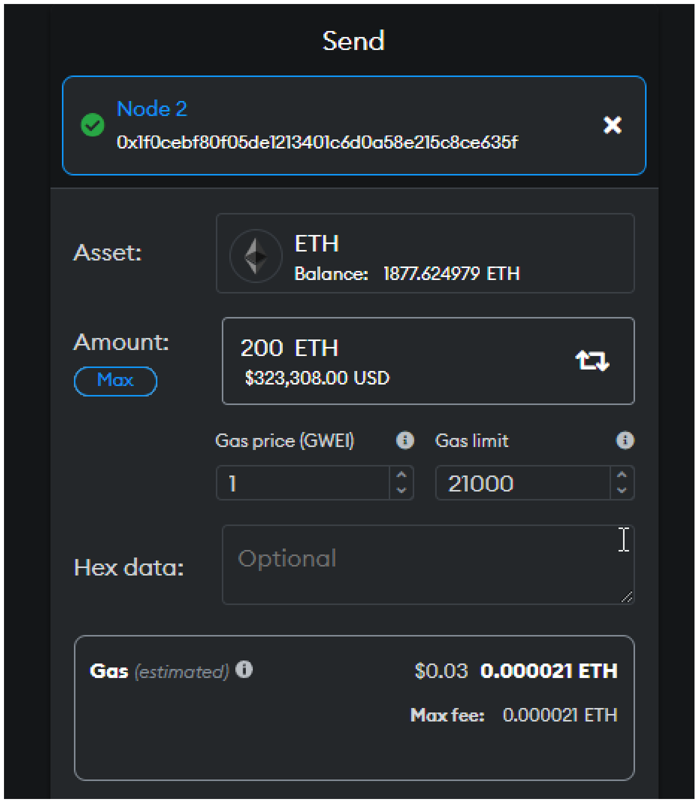


Figure 7: Sending 200 ETH from Node 1 to Node 2

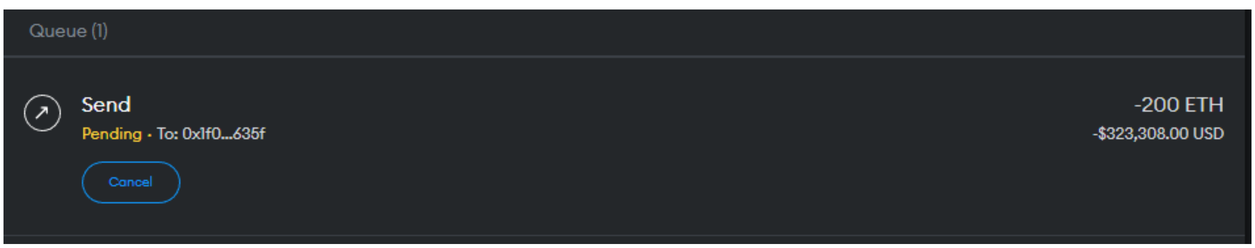


Figure 8: Queued/Pooled Transaction Awaiting Validation

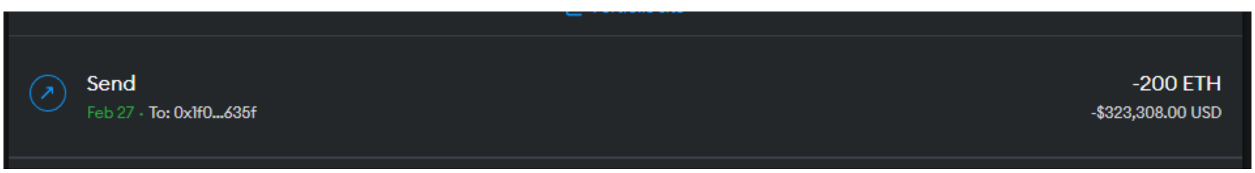


Figure 9: Validation & Verification of 200 ETH Sent

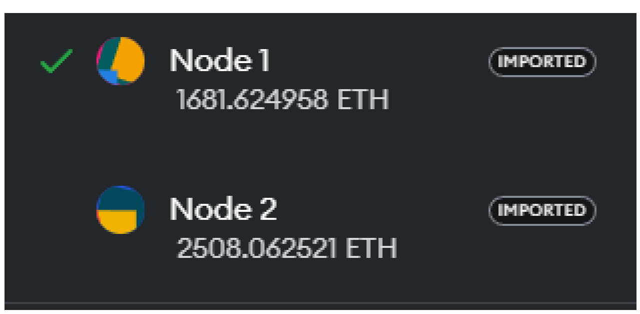


Figure 10: Updated Balances of Node 1 and Node 2 Accounts

2.6 Scenario Discussion

Most of what has been shown above is the typical use case of a account and nodes on an Ethereum network. Typically, a user would create an account and a corresponding wallet on an exchange, not having to actually deal with the setup of their own Ethereum node, unless of course the individual wanted to mine using their own hardware. However, the steps above mimic that of a real world scenario, where a user can manage their own account, account balance and transact amongst node addresses all from the user interface connected to a client residing on the Ethereum network.

What if however, you were looking to learn more about what happened exactly when a node joined the network, or see exactly what information was propagated throughout the network when Account/Node 1 sent Account/Node 2 200 ETH. How would we know which node actually mined the valid block that pulled in the pooled transaction?

Much of this can be read about, with Ethereum's own documentation, found on their webpage, or even through source code found in the various implementations like GETH. The next chapter will begin to discuss the inner workings of the Ethereum network, giving a background of much needed information while also introducing two existing but very limited in functionality Ethereum network packet dissectors which can be used to aid in the visualization of the discovery mechanism used through the GETH nodes on the network. Then, in subsequent chapters, the creation of a new dissector will be discussed, using it to visualize this real-world scenario, and clarify much of the documentation regarding the DEVP2P protocol suite.

3. Related Work and Literature Review

In this chapter, we will explore the related work in this field, starting with a brief overview of the history and critical pieces of the Ethereum network. This will include a deeper understanding of Ethereum nodes, networks, and the protocols they use to communicate to understand the requirements for a new dissector. We will then survey the current state of network documentation and examine the role of GETH in the network. Finally, we will discuss existing dissectors for the Ethereum network and highlight their strengths and weaknesses. This will provide a foundation for the subsequent chapters, where the Ethereum Network packet dissector for DEVP2P will be introduced and utilized.

3.1 Discussion of Ethereum Networks

There are two main types of Ethereum networks: public and private. Public networks are comprised of nodes worldwide, residing on different machines throughout the internet. Each public network has a uniquely identifiable chain ID, sometimes referred to as the network ID, which Ethereum nodes use to denote which chain/network clients use to communicate. The Mainnet is the live Ethereum network that hosts actual transactions and smart contracts denoted by a chain ID of 1, chains other than the Mainnet that are public Ethereum networks are considered "testnets." Testnets are alternative networks used for testing and experimentation and serve as sandboxes for developers to test their smart contracts and applications in a safe environment without the risk of losing real Ether. However, some testnets are also used for actual alternative currencies, other than Ether (ETH), but still using the underlying Ethereum network technology stack.

The two public testnets client developers maintain for the Ethereum chain are Sepolia and Goerli. Sepolia is a network for contract and application developers to test their applications. The Goerli network lets protocol developers test network upgrades and lets stakers test running validators. There are other networks that are not specifically maintained by the Ethereum community and can even use alternative currencies. These alternative chains can use most of the core Ethereum network protocols and include some popular networks like Polygon ($MATIC), Binance Smart Chain ($BNB), Avalanche C-Chain ($AVAX) and many others [11].

Regarding the proposed Ethereum Network packet dissector for DEVP2P, it can be used on any test net, such as Ropsten, Rinkeby, or Kovan. Furthermore, the dissector can be used to capture and analyze packets sent and received by nodes on the mainnet, test nets, or any network running the execution layer network protocols found in DEVP2P and even in the LIBP2P discovery mechanism, which uses DiscoveryV5. This all attests to how versatile this dissector can be, which can help developers identify issues and optimize the performance of their applications on a range of environments or networks.

In addition to public networks, Ethereum can also be deployed on private networks which in this context, private only means reserved or isolated, rather than protected or secure. Private networks are usually created for development and testing purposes specifically. Private networks are not open to the public and may have different rules and parameters than the main net or test nets. For the purposes of the type of network used in support of the creation and testing of the proposed dissector, a private network that matches the Ethereum pre-merge proof-of-work consensus algorithm will be used [12].

3.2 Discussion of Ethereum Nodes/Clients

A “node” is any instance of an Ethereum client, a computer running any Ethereum software, forming a peer-to-peer network. A client is more specific and is known as an actual implementation of Ethereum software implementing the necessary Ethereum network protocols or data validation algorithms. Pre-Merge Ethereum consisted of a single type of client, an execution client, such as Go Ethereum (GETH), running the execution layer network protocols known as DEVP2P. Now, post-Merge, consensus clients are added to fully integrate with execution clients, both required if Ethereum Mainnet connectivity is necessary. Effectively splitting the work into an execution client and a consensus client, where data/block validation is handled on the execution client, and the consensus mechanism and chain is handled on the consensus client. This addition of a consensus client handles the new implementation of the proof-of-stake consensus algorithm, its peer-to-peer network based on LIBP2P. A consensus algorithm is a process used to achieve agreement amongst peers about the validity of some distributed data, which would be a block of data as it relates to blockchain. Both execution clients and consensus clients can be run on their own, run together communicating by a local RPC connection, or tightly coupled in the same software as a single execution/consensus client, as shown in Figures 11 and 12 respectively [13]. The deployment of these client’s matter, depending on the consensus algorithm and network, but for the purpose of this document, we will be using Go Ethereum, an execution client, running by itself without a consensus client.

The Ethereum technology stack is meant to be diverse, providing a base set of requirements for any node/client implementation to utilize and join a network. With this, many implementations of both execution clients and consensus clients are found and fully supported in a range of programming languages. Execution clients include GETH (Go Ethereum), written in GO, Nethermind written in C#, Besu, written in Java; and Ekula, written in Rust. Consensus client implementations include Lighthouse (Rust), Lodestar (TypeScript), Prysm (GO). All these clients differ in architecture, functionality, and performance but all utilize the same core Ethereum guidelines and Ethereum Network protocols.

The execution client, Go Ethereum (GETH) will be the main focus throughout this document, as GETH has been a core part of Ethereum since the beginning. GETH was the original Ethereum implementation, supporting all the development and testing, making Ethereum what it is today, and is still known as the most used execution client. Being an execution client, GETH supports handling transaction validation, deployment, and execution of smart contracts and contains an embedded computer known as the Ethereum Virtual Machine. To connect to the Mainnet, GETH must run alongside a consensus client, effectively creating a full Ethereum node [14].

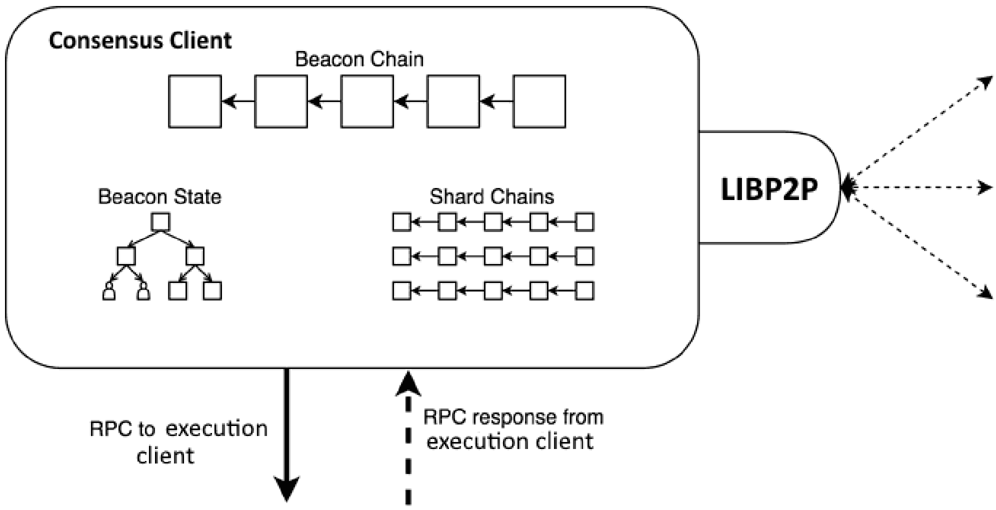


Figure 11: Ethereum Consensus Client

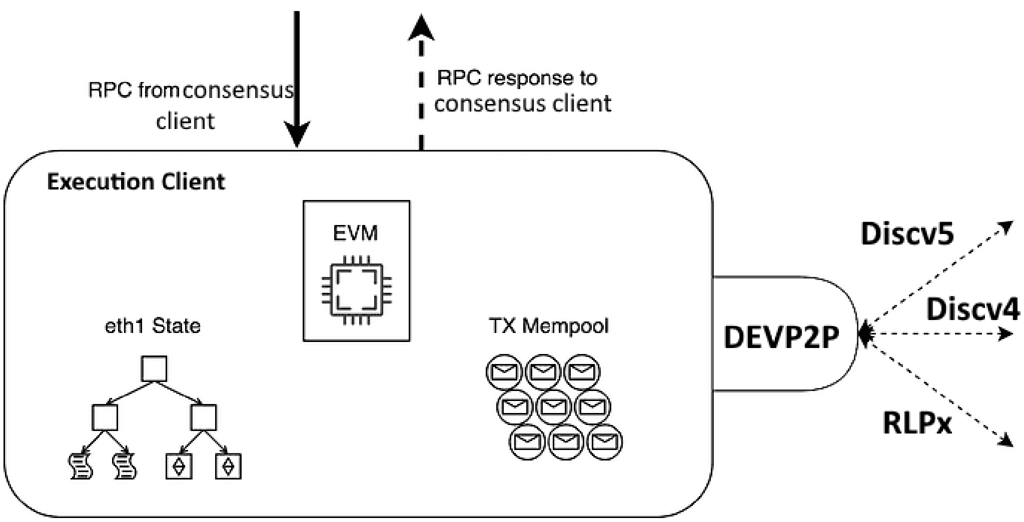


Figure 12: Ethereum Execution Client

3.3 Discussion of Ethereum Network Protocols

Each type of client, execution or consensus has its own network, DEVP2P, and LIBP2P respectively, both with their own individual protocols for handling discovery, and authenticated/encrypted communication. Pre-Merge, the Ethereum Mainnet only consisted of execution clients, where DEVP2P was used to facilitate all the communications between nodes, using the proof-of-work consensus algorithm. DEVP2P, as will be discussed in great detail in

Chapter 4, provides protocols for UDP discovery, like DiscoveryV4 and DiscoveryV5, along with a TCP-based authenticated and encrypted messaging amongst nodes using RLPx. This means that an execution client, such as GETH, on a proof-of-work private/development network/chain is the best way to see the full use of DEVP2P and its multitude of protocols and message types. However, this does not mean DEVP2P is not utilized in a post-merge proof-of-stake consensus algorithm deployment. DEVP2P DiscoveryV4 is still used as the primary discovery mechanism amongst execution clients, and RLPx is being used for synchronization and propagation along with the LIBP2P consensus clients [15].

As the main contributor to the development of Ethereum, GETH also was the front-runner in implementing and testing the successor to DiscoveryV4, DiscoveryV5. DiscoveryV5, compared to DiscoveryV4, provides confidentiality by masking the contents of packets, making it more dynamic for use amongst arbitrary nodes, increasing performance regarding node identity, and no longer relying on the system clock of nodes. GETH is the only implementation of an execution client that supports DiscoveryV5, as support for DiscoveryV5 was paused for execution clients as talks of the merge arose. DiscoveryV5 was chosen as the discovery mechanism for consensus clients instead of LIBP2P's own discovery implementation/schema. However, we will not be looking at a consensus client throughout this document, nor be looking at LIBP2P's specific DiscoveryV5 implementation. This means that the GETH execution client can also discover consensus clients and is a great way to understand the inner workings of DiscoveryV5 without spinning up a complete LIBP2P network with a consensus client [16].

3.4 Discussion of the Proof-of-Work Consensus Algorithm

Proof-of-work (PoW) is a consensus algorithm used in blockchain technology to verify and add new transactions or any data (blocks) to the blockchain. In a PoW system, miners compete to solve a complex mathematical puzzle, a hash function, to add a new valid block to the blockchain. There is a multitude of different implementations of PoW mechanisms, like that found in Bitcoin, whereas Ethereum's specific algorithm is called Ethash. Mining a new block involves selecting a set of pending transactions from a pool for validation then creating a block containing those transactions. The miner then attempts to find a solution to the hash function that meets a certain difficulty level which is adjusted periodically to maintain a target block time and prevent the network from becoming too congested [17].

The miners must find a specific hash value at or below the target hash value determined by the difficulty level, visualized in Figures 13 and 14. This target hash is calculated by what is known as the "difficulty level" which is calculated and incremented using previous block difficulty levels. This means the difficulty increases as more blocks are mined. Miners achieve this target hash by repeatedly changing a value called a "nonce" in the block header that produces a hash value using SHA 256 that is at or below the target hash when combined with the block's header and other inputs. Once a miner finds a valid solution, they broadcast the new block to the network. The other nodes in the network then verify that the solution is correct by checking that the hash value meets the target hash/difficulty level and that the transactions in the block are valid by checking their hash values. Once most nodes in the network verify the block, it is added to the blockchain, and the miner who found the solution is rewarded with new cryptocurrency units as an incentive for their work [18].

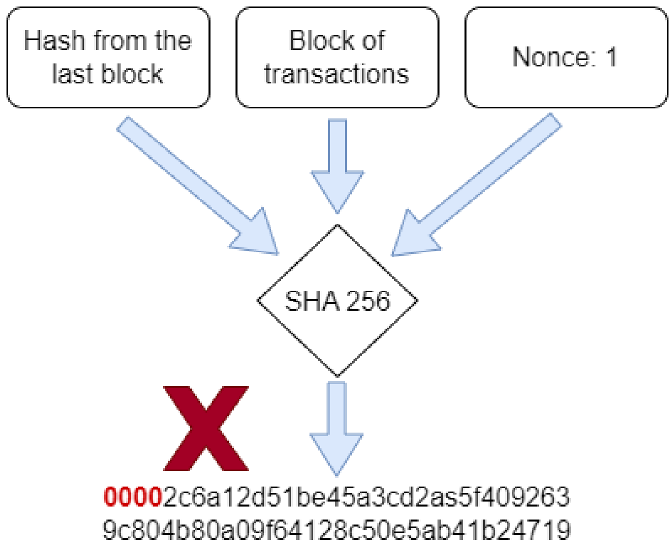


Figure 13: Invalid Proof-of-Work Hash Value

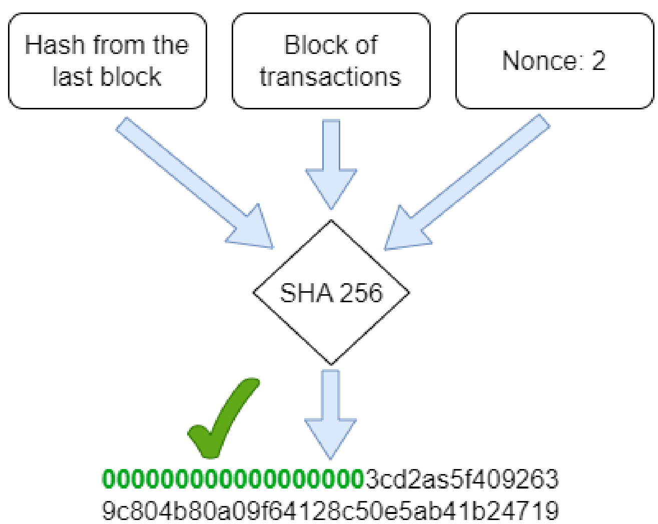


Figure 14: Valid Proof-of-Work Hash Value

3.5 Literature & Documentation Review

Documentation quality is essential for developers and users to understand the workings of a system fully. Ethereum and the Ethereum network has a significant amount of documentation available through various sources, including the official Ethereum website and multitude of README files found from the Ethereum GitHub. The documentation provided by Ethereum maintainers and community members includes in-depth explanations of the Ethereum toolkit, including protocol usage, smart contracts, the EVM, and the Solidity programming language. Ethereum also provides whitepapers, tutorials, and specifications for developers to use and community members to use while also providing transparency into research and development roadmaps with Ethereum Improvement Proposals or EIPs [6].

Additionally, specific implementations of Ethereum software such as Go Ethereum, GETH, the most widely used Ethereum client, provides documentation, start guides and tutorials, to aid new users and developers for starting their own client. The GETH source code is extensive, used as the primary development platform for Ethereum execution clients, and its documentation helps developers understand how the client works and how they can interact with it [14]. Many times, throughout the creation of the dissector discussed in later chapters, the GETH source code is used to provide DEVP2P and Ethereum implementation and design specific insights into the creation of the dissector and PYDEVP2P.

Despite the vast amount of documentation available for the Ethereum network, some areas of the protocol need to be better documented, which can create difficulties for developers. In addition, some of the documentation can be challenging for those with a deep technical understanding of the Ethereum network. Especially with the Ethereum Merge, many documentations instantly became outdated; source code deprecated, and exact deployment and usage of Ethereum network-specific protocols left undocumented. Furthermore, the specification and documentation do not provide real-world data to help guide the reader to further understanding. However, using a network packet dissector, many of these specifications, claims, and gray areas can be proven, verified, and uncovered in a digestible manner to aid in documentation and education.

3.6 Existing Dissector Implementations

Network packet dissectors like Wireshark capture and analyze network traffic to decode the data transmitted over the network. This can be done either in real-time or with pre-recorded traffic, providing visibility into the communication amongst different devices. Specifically, tools like Wireshark are used in conjunction for debugging purposes to help diagnose network problems by revealing malformed or misconfigured network data. Such tools are also proven valuable educational tools, providing detailed and real-world views into the underlying network protocols and helping visually understand low-level topics. In addition, dissected packets are displayed in an easily understandable structure or schema, conveying the contents of the network packets in a digestible manner.

Two Ethereum packet dissector implementations exist, created in 2018 and abandoned shortly after that, including implementations in C and LUA. Both were created to provide transparency of the inner workings of the Ethereum network protocol and their subsequent messages exchanged between communicating nodes.

Setting up these two types of dissectors is quite different, first looking at the C Wireshark dissector from PegaSys, now known as Consensus, a market-leading blockchain technology company building developer tools to enterprise solutions [19]. Written in C, this dissector must be compiled alongside the source-code of Wireshark, specifically Wireshark version 2.6.2 [8]. This means that a proper development environment will need to be set up, with all the C and third-party dependencies locally installed to correctly compile the dissector and the Wireshark source code into an executable.

A great ReadMe can be found from the source code of the C Ethereum Dissector from PegaSys (ConsenSys) which walks through pulling both the Wireshark source code, the dissector source code, and utilizing Ninja, a small build system for building executables from source [20]. After building, the custom version of Wireshark with the built-in C dissector can then be run. Wireshark directly states that the preferred language for creating dissectors is C, due to its performance, and its larger range of functionality as it is built directly into the source.

The LUA dissector on the other hand was created as a Wireshark plugin [7]. LUA is a powerful light-weight programming language designed for extending applications and used in conjunction with Wireshark as a language for prototyping and scripting dissectors. Wireshark has a built-in LUA runtime and an API where LUA plugins can be loaded and utilize the API function calls to access important packet information [21]. The LUA dissector was created by BCSEC, a blockchain security group which aimed to elevate the security of the entire blockchain ecosystem. Even Though this group no longer exists, the source code for their Ethereum DEVP2P Wireshark LUA dissector still does and provides a great starting implementation for creating a Wireshark dissector plugin.

Wireshark has a built-in LUA runtime environment, so using the LUA dissector is as easy as placing the .LUA code itself, without any compilation steps necessary, is right in the correct plugins folder used by Wireshark. Then, a typical installation of Wireshark, no matter the version, will automatically pull in this LUA dissector plugin and run it automatically. This ability to develop and test using a LUA dissector without having to rebuild and compile the entirety of the Wireshark source makes its usefulness clear for development purposes. However, as mentioned earlier, some functionality for LUA dissectors could be improved, such as creating heuristics reports like the C dissector provides.

The C dissector and the LUA dissector are both able to dissect Discovery V4. DiscoveryV4 as mentioned earlier is a UDP-based discovery mechanism for Ethereum clients and is part of the DEVP2P protocol specifications. DiscoveryV4 packets are sent in the clear, without encryption, and provide a very simple method for nodes to join the peer-to-peer network with 4 main packet types, PING, PONG, FindNode and Neighbors. The PING and PONG packets deal with peer liveliness and endpoint proof, while the FindNode and Neighbors deal with the actual discovery of other nodes based on secp256k1 public key identities. Every node has a cryptographic identity, a key on the secp256k1 elliptic curve, and this public key of the nodes serves as its unique identifier or “node ID”. This allows packets in DiscoveryV4 to be signed, validated and authenticated with Elliptic Curve Digital Signature. Lastly, there are two other packet types, ENRRequest and ENRResponse, added to the protocol specification in October 2019 via an Ethereum Improvement Proposal EIP-868 to enable authoritative resolution of Ethereum Node Records or ENR’s in DiscoveryV4. An ENR contains specific network endpoint information about a node, it also holds information regarding protocol version information as well as a compressed secp256k1 public key.

Despite their differences with installing the two dissectors, using them is quite the same, however they do have their minor differences and nuances. One main thing with the C dissector is the ability to click on the individual fields in the DiscoveryV4 packets and see the exact byte that field value correlates with, while this functionality is lost with the LUA dissector. The LUA dissector also seemed more unfinished, as all the expiration and date fields were left as hex values instead of being parsed into human readable dates. Shown below, is a DiscoveryV4 FindNode packet, first from the C dissector followed by the dissection from the LUA dissector, shown in Figures 15 and 16. Each shows the typical header information found on each DiscoveryV4 packet, the hash of the message, signature information and the type of the packet. Then the differences, where the C dissector clips the full target field, which is actually the 64-byte secp256k1 public key of the node that is being searched, while the LUA dissector displays it in full, inaccurately listing the field as “hash”.

Figure 15: C Dissector DiscoveryV4 FindNode Packet

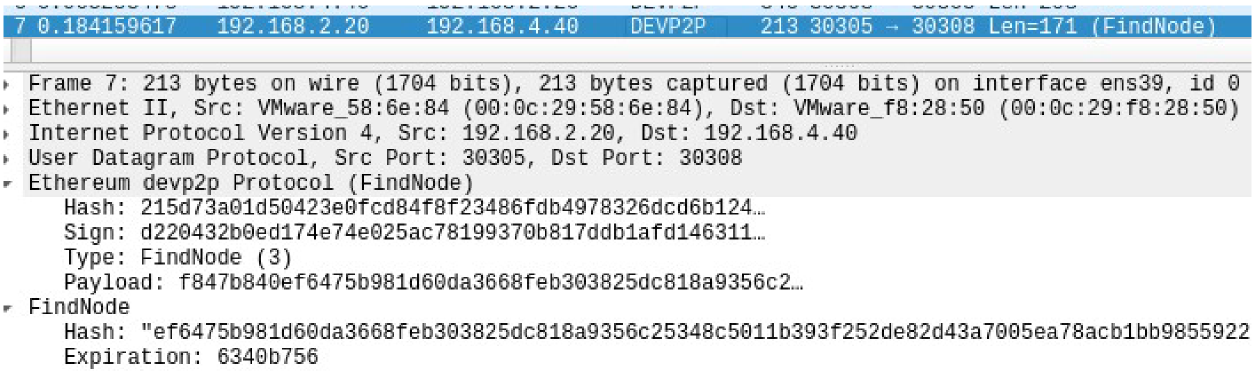


Figure 16: LUA Dissector DiscoveryV4 FindNode Packet

Both the C and LUA dissector implementations have significant limitations, aside from compilation times, and LUA runtime performance. To start, since these implementations are several years old they are not compatible with the newest specification release for DiscoveryV4. This new specification mentioned earlier, EIP-868, adds an ENR-sequence field to both the PING and PONG packets, as well as adding the ENRResponse and ENRRequest packet types. Because of these modified and added packets, the dissectors both fail to fully dissect the PING and PONG packets found in the latest version of any execution client such as Go Ethereum (GETH), shown in Figure Figures 17 and 18, as well as recognizing and dissecting the ENRRequest and ENRResponse packets. Shown below, examples from both dissectors, where the Ping packet is either not able to be dissected at all, or not showing the ENR-sequence field, and of course both ENRRequest and ENRResponse, message type number 5 and 6 respectively not dissected by either dissector, seen in Figures 19 and 20.

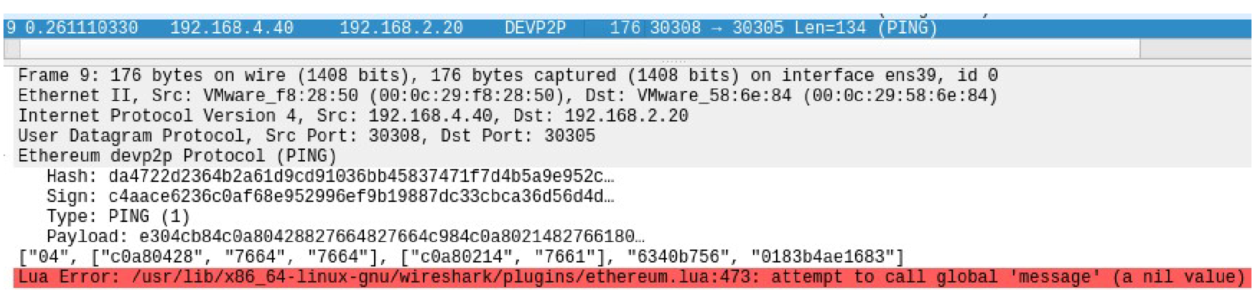


Figure 17: LUA Dissector Ping Packet Unable to Dissect

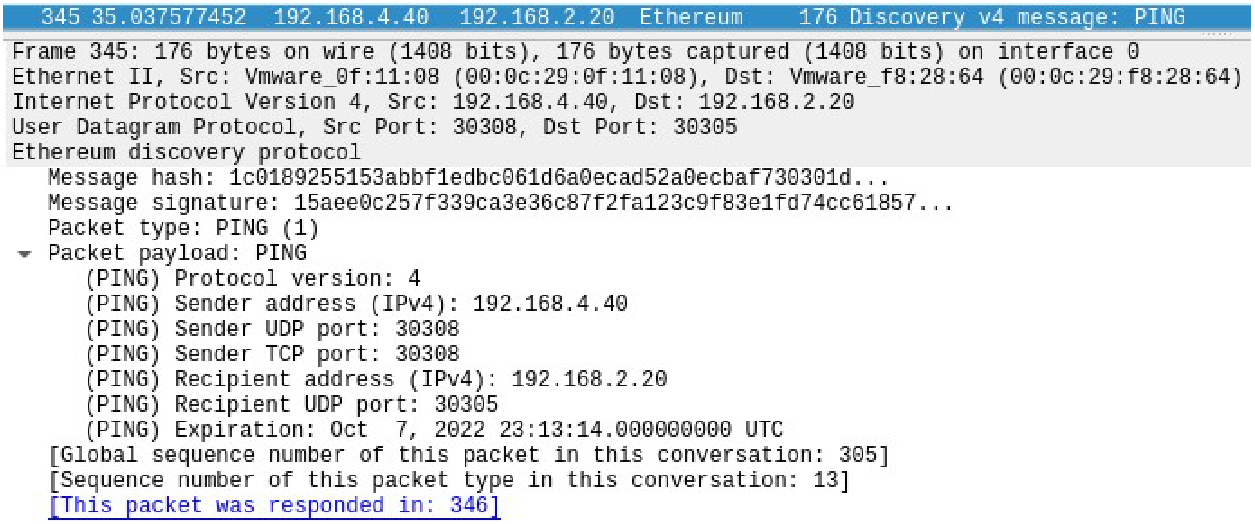


Figure 18: C Dissector Not Fully Dissecting Ping Packet

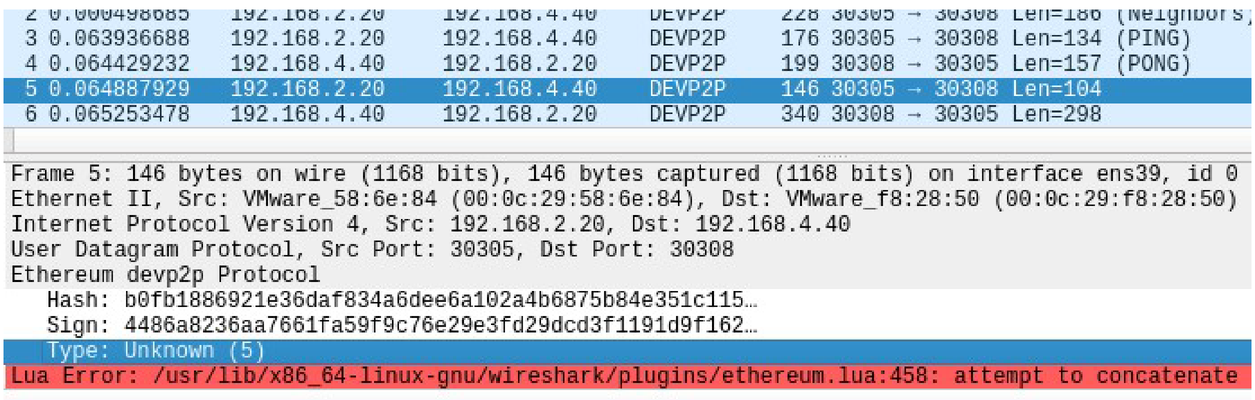


Figure 19: LUA Dissector ENRRequest Un-dissected Packet

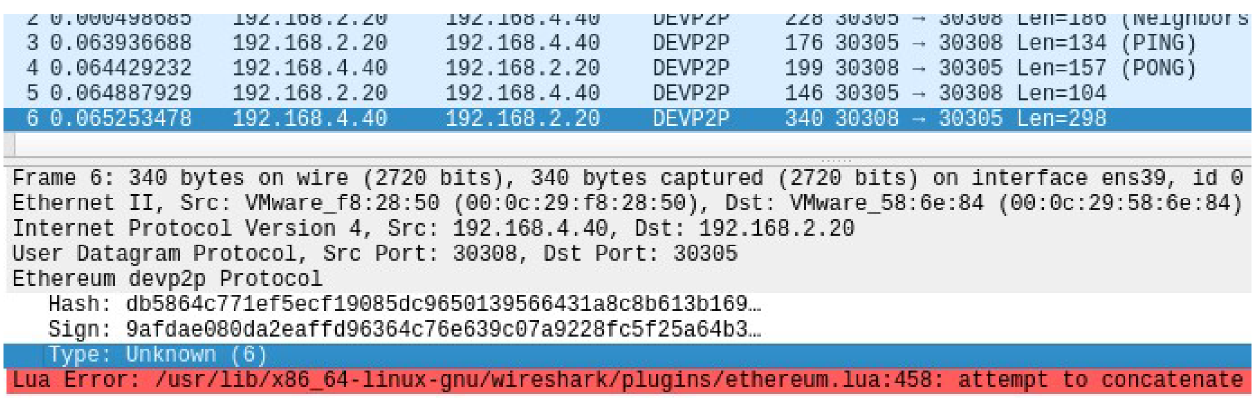


Figure 20: LUA Dissector ENRResponse Un-dissected Packet

These dissectors, despite their lack of support of the newest DiscoveryV4 specification, also do not provide support and dissection of DiscoveryV5 and RLPx. DiscoveryV5 being a newer encrypted discovery mechanism while RLPx being the main TCP-based encrypted communication channel for Ethereum execution clients. The lack of support for these protocols is not due to the fact of being outdated, however ConsenSys directly states that dissecting RLPx "is somewhat complicated, as TCP connections are encrypted with an AES symmetric key derived per-session via ECIES (Elliptic Curve Integrated Encryption Scheme), which means that even if the dissector had access to the private key of the local node, it would not be enough to decrypt communications" [5]. This is the main hindrance for both dissectors to finish the dissector, and the main reason why support for these dissectors were dropped.

3.7 Conclusion

Overall, while very useful, the limitations of existing documentation and existing dissectors highlights the need for a new Ethereum packet dissector implementation that can handle the latest Ethereum protocol features and support a wider range of protocols, such as DiscoveryV5, and RLPx including the ETH and SNAP sub-protocols. Such a dissector could improve the ability of developers and researchers to analyze the Ethereum network and gain a deeper understanding of its operation. A tool like this can also aid in reinforcing that which is read from the documentation and ReadMe's, or finding the gaps, by visually proving the actual protocols and messages used in a live Ethereum network environment. Even complexity was cited as the main reason to halt development on the dissectors, with the emergence of newer protocols and the added complexity with the consensus layer and clients, it becomes even more imperative to create a dissector that solves the gaps.

The next chapter will discuss the creation of a new dissector, one that is a LUA dissector plugin for Wireshark, originally based on the LUA dissector but with little to no similarities in its final form.

4. Creating the Dissector

4.1 Packet Dissector Design

To create a new dissector that supports the latest messages found in DEVP2P, it is important to go over what is required to meet this goal. Several components are needed, such as a private/development network, which allows for the creation of an entire Ethereum network to be spun up quickly and modified rapidly. This dissector will also be in the form of a LUA plugin for Wireshark, which allows for quick development and the ability to support any version of Wireshark, without the need to compile everything. This keeps the dissector portable, and easily installed and used throughout the community.

First, reviewing the specific functional requirements for this new LUA dissector is essential. As stated before, this dissector must be able to handle not only DiscoveryV4 but also decrypt and decipher DiscoveryV5 along with RLPx messages and its related sub-protocols ETH and SNAP. This means that the dissector will have to handle the decoding of RLP, which is not related to RLPx, while also handling ECIES, which entails sessions with symmetric and public key encryption using elliptic curve cryptography. In addition, the dissector must maintain the typical requirements for a Wireshark dissector, such as live dissections from network interfaces or through captured PCAP files with the ability to display the contents of packets/messages clearly in the Wireshark user interface. To handle ECIES decryption, the dissector must be able to hold information about the Ethereum nodes, specifically their private keys and IP addresses. RLP is the encoding used by all DEVP2P messages, which stands for "Recursive Length Prefix" and is used for arbitrary data structures in a compact format. Lastly, this dissector should be easily installed and utilized in a development environment that can quickly be spun up for educational or demonstration purposes. On top of all of this, it is the utmost goal of the dissector and subsequent PYDEVP2P Python library, which will be discussed in great detail, to provide a readily accessible, and clear implementation of elliptic curve calculations done without the use of C or 3rd party dependencies to also add educational value.

As stated earlier, the primary focus for this dissector is DEVP2P, the protocol suite used amongst execution layer clients, such as GO Ethereum. DEVP2P was created during the time when the Ethereum Mainnet was using the proof-of-work consensus algorithm, therefore, to see some of the DEVP2P messages, a proof-of-work development network is required. Specifically, DiscoveryV4, DiscoveryV5 and RLPx including sub-protocols ETH and SNAP will be dissected, each holding their individual messages.

After understanding the general goals and requirements of the dissector, we can now look at the individual pieces that fit together to make it all possible. Seen below in Figure 21, the software architecture diagram depicts each component that is necessary for a full environment, broken down into two main categories, software/tools that are directly related to the function of the dissector, and then software/tools related to the custom development Docker network and GO Ethereum nodes that will be used to capture DEVP2P network traffic via Wireshark on all nodes' interfaces in the middle.

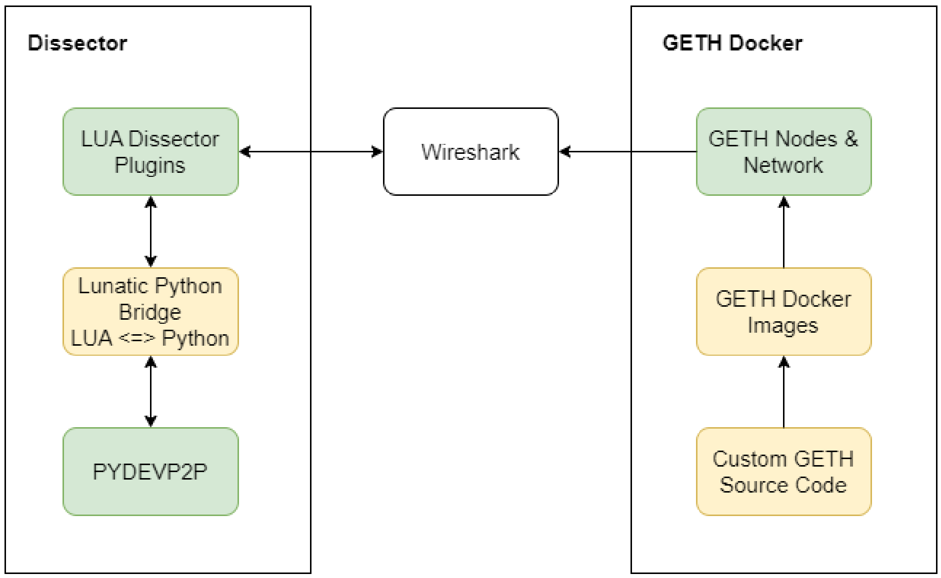


Figure 21: Dissector Software Architecture Diagram

Docker is a platform and tool for creating, deploying, and managing applications using containerization technology which means they are self-contained and isolated from the underlying host system. It allows developers to package an application and its dependencies into a docker image and deploy lightweight, portable containers that can run consistently across different environments, from development to production [22]. As they are so lightweight compared to traditional virtual-machine technology, multiple docker containers can be spun up quickly and even virtualize network connectivity amongst the host machine and containers.

As it is used for the dissector, custom docker images are built with the required networking tools and custom GO Ethereum source code which will be discussed in later sections. This custom image is then used to create multiple docker containers, each interconnected with different subnets and fully simulating a small private Ethereum network. Since the docker containers are able to connect to the host through a bridged network connection, a normal install of Wireshark on the host allows it to capture the packet traffic transferred amongst these GETH docker containers.

Once the DEVP2P packets are captured by Wireshark, the LUA dissector plugin comes into play. The LUA Wireshark plugin acts as an interface, first registering with Wireshark the exact packets it wishes to dissect, usually based off packet schema and/or port numbers. Then, Wireshark sends these packets to the LUA plugin to be dissected and decoded, which makes the LUA plugin responsible for displaying the contents of the packets appropriately in the user interface of Wireshark. This process is done for each incoming packet, either in a live packet capture or from a standalone PCAP file.

Lastly, there is PYDEVP2P, a Python library and toolkit to aid in decryption, decoding, and dissecting DEVP2P messages along with Lunatic Python. Lunatic Python is a two-way bridge between Python and LUA, allowing these languages to inter-communicate. This gives the ability for each language to invoke built-in functions from the other language. This project specifically uses this implementation to allow LUA to call specific functions from PYDEVP2P which handles all the heavy lifting for RLP decoding, ECIES deciphering, and node state management. The main reason for this is the lack of cryptographic library support in LUA, as well as snappy compression/decompression support found in Python. As it is used throughout this project, PYDEVP2P provides the bulk of the implementation for providing a dissector for DEVP2P, as well as including a limited dependency ECIES implementation specific to Ethereum ECC. In order to match certain implementation specifics to DEVP2P and Go Ethereum due to lack of documentation, it should be noted that a lot of the design characteristics and data techniques for decryption were recreated in a pythonic approach utilizing the Go Ethereum source code directly [14].

Wireshark provides an interface for LUA dissectors known as the dissector API, which provides functions for dissecting network protocols, accessing protocol fields, and creating new protocol fields. In Wireshark, the Lua dissectors are stored in the "plugins" directory, which is automatically loaded by Wireshark. Specifically, in this case, the Wireshark LUA dissectors will be located in the ~/.local/lib/wireshark/plugins directory found on Linux operating systems. LUA plugins are registered with Wireshark by initializing a short protocol name, full name, description and a list of ports to bind with [23]. When a Lua dissector is registered with Wireshark, it is called whenever a packet that matches the protocol, port, or schema is captured. The dissector then analyzes the packet and creates a tree of protocol fields, which can be viewed in the Wireshark GUI. This tree allows for nested fields and values, to allow the user to easily view hierarchical information easier.

In terms of this DEVP2P dissector, once the LUA plugin receives the captured packet information, it then calls the associated Python function for the payload of the received message. This is handled by Lunatic Python, using the shared object binary, “python.so” that is loaded into LUA to allow LUA to interface with Python functions [24]. These Python functions are found in PYDEVP2P, located in the “bridge.py” file, holding all of the functions needed by the LUA plugins, supporting DiscoveryV4, DiscoveryV5, and RLPx. From there, depending on the function, the payload of the captured network traffic is sent off to different sub-modules found in PYDEVP2P, supporting state management, RLP decoding, ECIES, and more, seen in Figure 22.

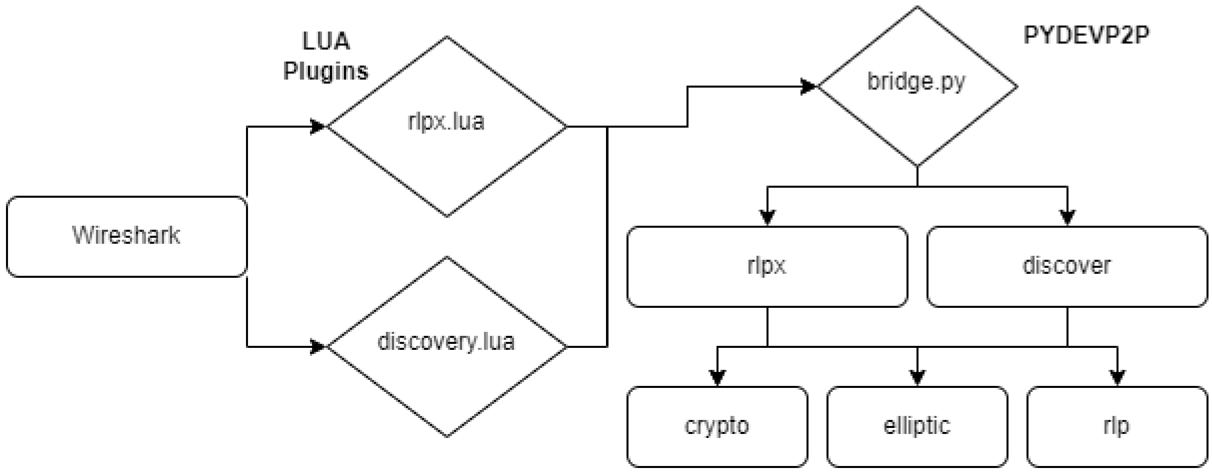


Figure 22: Wireshark, LUA, PYDEVP2P Flow Diagram

In the coming dissector sections, each of these modules found in PYDEVP2P will be discussed in greater details. PYDEVP2P was designed to be a standalone library for DEVP2P, not just to be used as a dissector, therefore, each sub-module can be used independently, providing Ethereum-specific cryptographic, elliptic curve, and RLP tools and functions. However, as stated previously, bridge.py is provided as both an interface with the LUA Wireshark plugins, and also as a statement to how PYDEVP2P can be used for various functionality.

4.2 Creating the Network

As stated earlier, a custom docker network will be used throughout the creation of this dissector, as seen in the Scenario chapter. This network contains 4 GETH containers and 1 Ubuntu Router container, facilitating networking amongst the containers so that they can reside on different subnets. Docker is a necessary component, as it provides a method to spin up an entire private Ethereum network quickly and dynamically in seconds, running lightweight custom source code. There are several components to creating a custom docker environment, such as a dockerfile, a docker image, a docker container, and lastly, what is known as a docker-compose file.

A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. This image contains possible source code, packages, and other dependencies to run a docker container. Throughout the development of the dissector, the dockerfile and the images saw many variations, first starting with a dockerfile that pulled in a preconfigured GO Ethereum docker image first; however, as the dissector progressed, custom GO Ethereum source code had to be installed in order to expose the private session keys during the RLPx handshake.

In the final implementation of the dockerfile, the docker images are built with a slim Ubuntu 22.04 docker image, then containing the necessary commands to install the necessary APT dependencies such as GIT and GOLANG. The next main piece in the dockerfile pulls in the forked custom GO Ethereum source code, which again allows for the exposure of the session keys during the RLPx handshake to allow for proper decryption and dissection. The dockerfiles are also set up to take in an "ACCOUNT_PASSWORD" as an argument, which is the password used to set up the default account for the GETH client. This is the same password that is used when adding the account in MetaMask. Next, the "genesis.json" is loaded into the image, which is used as a configuration file for the GETH client, defining the "chainId" 12345 and the type of consensus mechanism to use. Please see the document materials list in the appendix to view the dockerfile for these custom containers.

When initializing the GETH client utilizing the "geth init ~/genesis.json" command, this creates what is known as the nodekey, found in ~/.ethereum/geth/nodekey in the docker containers. This is the unique elliptic curve secret/private key that is used for node authentication and identification. It is also possible to override this key to make it static (outside of initialization) for development purposes, which will be done in this case. After the creation of the dockerfile, the "docker build" command can then be used to create the actual docker images, each with their own corresponding tag according to their uniquely chosen account password, "boot" for the boot node of the network, followed by "node1", "node2", and "node3" and lastly the Ubuntu docker image for the router. In the GETH docker repository, <see contribution>, the "build-dockers.sh" shell script is used to automatically build these 5 total images, with their corresponding tags.

The last piece to the docker test environment is what is known as the docker-compose file. A docker-compose file is used to define and manage multi-container Docker applications, allowing the definition of services, also known as a container, along with networks and volumes which are files and directories that can be piped into the containers from the host machine. With docker-compose, it is possible to start, stop, and restart all the docker containers with a single command, effectively spinning up the entire GO Ethereum test network in a single click. [26].

It is in the docker-compose where the individual subnets are defined for the individual containers. Usually, when manually assigning networks like this, the containers will lose connection amongst themselves, however, that is precisely where the "Ubuntu-with-tools" docker image comes into play, creating the "bridge-router" service. This docker container will act as a gateway for each of the other containers, holding a docker-compose network interface for each of the different subnets. As container communication via different subnets is not a standard use case, the end result in the docker-compose file is somewhat work around. Usually, containers with special communication needs are handled with what is known as an "overlay" driver; however, the containers in this scenario require communication with the host computer, as Wireshark needs to be able to capture and dissect the network traffic [45]. The workaround in this case is to manually issue an "iptables" firewall rule change on the "bridge-router" container startup, and "ip route" modifications on startup to each of the individual node containers.

The different parameters are set for the individual containers within each service in the docker-compose file. This includes setting up the ip address of the container, the ports to expose to the host machine, the image to use, and of course the startup command. This startup command is what issues the ip route changes, to set the default route up to send network traffic through the Ubuntu bridge-router container instead of the default docker bridge interface. Also in this command is what starts up Go Ethereum, issuing the command “geth” with several different parameters all specifically chosen for this private development network [25]. Each node container has a static “nodekeyhex” assigned during the container's startup using this “geth” command. This again allows for the private elliptic curve key of the node to be set, which the dissector can, in turn, use for RLPx and other dissections. Shown below in Table 3, the list of GETH nodes, including their static private/public keys.

Table 3: GETH Nodes and Corresponding Static Private/Public Keys

Node (IP Address)	Static Elliptic Curve Private/Public Keys

Bootnode (10.1.0.10)	Private Key:
	`3028271501873c4ecf501a2d3945dcb64ea3f27d6f163af45eb23ced9e92d85b`
	Public Key:
	`2c4b6808e788537ca13ab4c35e6311bc2553b65323fb0c9e9a831303a1059b87`
	`54aab13dbb78c03a7a31beee5c2f2fb570393f056d54fa83ebd7e277039cc7b6`

Node 1 (10.1.1.10)	Private Key:
	`4622d11b274848c32caf35dded1ed8e04316b1cde6579542f0510d86eb921298`
	Public Key:
	`c35c2b7f9ae974d1eee94a003394d1cc18135e7fe6665e6b4f221970f1d9d59f`
	`6a58e76763803bcc9097eba4c91fd08b30405e65c53272b8635348e37f93cedc`

Node 2 (10.1.2.20)	Private Key:
	`816efc6b019e8863c382fe94cefe8e408d53697815590f03ce0a5cbfdd5f23f2`
	Public Key:
	`1ae68ad9b2b095b5366d9a725a184bf1a6a5e101a4e6a3de62b38b07eac2c8fe`
	`365e8a184004191c96d2f365f3c116c5dfbb92247635cf49a730f02908d6e397`

Node 3 (10.1.3.30)	Private Key:
	`3fadc6b2fbd8c7cf1b2292b06ebfea903813b18b287dc29970a8a3aa253d757f`
	Public Key:
	`e98d53b2a12bdb4441d825d4b0a1c4255b880c2f657c0adece61cbe11c5869ae`
	`35fd6bc956b3f8a2364b314eda761ebb570764c127efd5c114910a71ddfc7c4a`

The GETH command line arguments for the Bootnode differ slightly compared to the other geth-client services in the docker-compose file, shown below in Figure 23. Specifically, the Bootnode does not get assigned a “bootnode enode” as it is one. The Bootnode is also assigned the “http” flag along with the “http.addr” and “http.port” flags to set up the http JSON-RPC interface. This is setup with address 0.0.0.0 to listen from any incoming connections, along with the port 8545. This is what allows MetaMask to connect to the Bootnode to connect to accounts in the network.

```yaml
geth-ubuntu-bootnode:
  hostname: geth-ubuntu-bootnode
  cap_add:
    - NET_ADMIN
  networks:
    bridge-net-0:
      ipv4_address: 10.1.0.10
  ports:
    - 30303:30303/udp
    - 8545:8545/tcp
  env_file:
    - .env
  image: geth-custom:boot
  container_name: geth-bootnode
  depends_on:
    - bridge-router
  command: >-
    sh -c "ip route del default && ip route add default via 10.1.0.2 &&
    geth --nodekeyhex 3028271501873c4ecf501a2d3945dcb64ea3f27d6f163af45eb23ced9e92d85b
    --networkid 12345
    --v5disc
    --http
    --http.addr 0.0.0.0
    --http.port 8545
    --netrestrict 10.1.0.0/22
    --nat extip:10.1.0.10
    --port 30303"
```

Figure 23: Docker Compose GETH Bootnode Service

Each geth-client service, including the Bootnode is also set up with its own specific UDP port to carry out the DEVP2P discovery and RLPx communications. This is done utilizing the "—port <port #>" flag for the geth command line argument. The Bootnode is given 30303, followed by Node 1 30304, 30305 for Node 2 and 30306 for Node 3. These ports are very important and each of these ports related to the ports the dissector is registered to, and each will be seen in subsequent dissections found in the dissection chapter. Lastly, each client and Bootnode is passed the "—nat extip:<ip address>" flag. This tells the GETH client that the outward interface to communicate and connect to the network is a NAT and tells GETH the specific interface to communicate through.

Lastly, found in the GETH clients, Node 1, Node 2 and Node 3 specifically, shown in Figure 24, are flags that support the auto unlocking of accounts that are located on the client. This just allows for easily logging in via the command line and accessing account funds and sending transactions. This is done using the "—allow-insecure-unlock", "—unlock 0" which defines which account to unlock, followed by "—pass /root/password", which is the file path of the password file for the account. For a GETH client to automatically become a miner on the network after startup and connection to a network, the "—mine" flag will turn the client into a miner, followed by "—miner.threads 1".

```
command: sh -c "ip route del default && ip route add default via 10.1.1.2 &&
  geth --bootnodes enode://2c4b6808e788537ca13ab4c35e6311bc2553b65323fb0c9e9a831303a10 \
  59b8754aab13dbb78c03a7a31beee5c2f2fb570393f056d54fa83ebd7e277039cc7b6@10.1.0.10:30303
  --nodekeyhex 4622d11b274848c32caf35dded1ed8e04316b1cde6579542f0510d86eb921298
  --v5disc
  --networkid 12345
  --allow-insecure-unlock
  --unlock 0
  --password /root/password
  --netrestrict 10.1.0.0/22
  --nat extip:10.1.1.10
  --port 30304"
```

Figure 24: Docker Compose GETH Client Service Command for Node 1

Finally, with all of this, the entire docker network can be spun up with a single command: "docker-compose up". This will automatically create the network interfaces connecting each of the GETH nodes to the Ubuntu bridge router, along with the Docker internal bridged network necessary for Wireshark to capture the network traffic along with MetaMask to connect to the Bootnode using the exposed 8545 TCP port.

4.3 Node Discovery Mechanisms

Looking back at the scenario in chapter 2, the first thing we saw when starting the network with "docker-compose up" was all the peers connecting. From the command line interface, starting with the bootnode, the "peer count" is displayed, showing the amount of connected and authenticated peers the node has discovered. This, of course, starts with Discovery, either DiscoveryV4 or DiscoveryV5, depending on the type of client. DiscoveryV5 was implemented in GETH, but production-level use was halted as DiscoveryV5 moved more to the consensus layer clients and was not implemented in the other execution clients. However, we will still provide the dissector for DiscoveryV5 as it is essential to understand the differences with DiscoveryV4.

Ethereum's DEVP2P Discovery protocols are used for discovering and connecting to other nodes on the Ethereum network. The protocol is a part of the more extensive DEVP2P networking protocol used by Ethereum nodes to communicate with each other. The main goal of the discovery protocol is to enable nodes to find and connect to other nodes on the network without the need for a central server or authority. In addition, the protocol allows nodes to discover and share information about other nodes, such as their IP addresses, port numbers, and public keys.

4.3.1 DiscoveryV4 Dissection

Connection to an Ethereum network relies on a pre-authenticated or validated node that facilitates new nodes connecting to the network. This node is known as the Bootnode. In the scenario, the bootnode is the first node to come online, as in the docker-compose, each service/container relies on the bootnode to start first. After starting the bootnode, the other nodes can connect to the bootnode; this is facilitated by the “enode” flag and value in the “geth” initialization command. This enode is the unique node identifier for the bootnode, meaning that when the other nodes startup, they will try to connect to the bootnode immediately. Seen below in Figure 25 a sequence diagram of the DiscoveryV4 messages sent between the bootnode and other nodes. It is important to note that these messages can be between two nodes, not specifically a node and a bootnode.

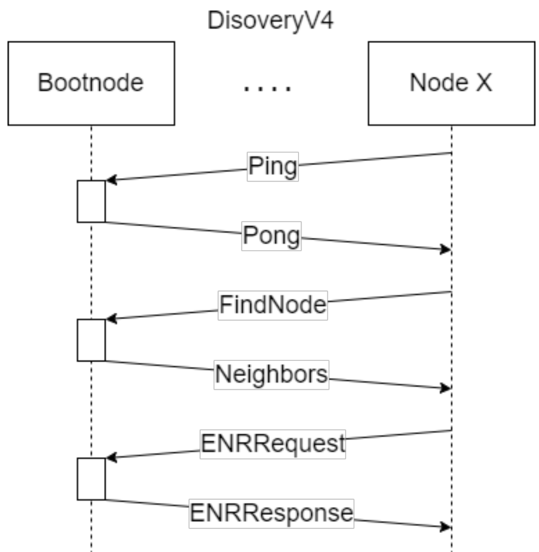


Figure 25: DiscoveryV4 Message Sequence Diagram

The first step in the DEVP2P DiscoveryV4 protocol is the Ping/Pong exchange. In this step, a node sends a Ping message to another node, which is then expected to respond with a Pong message. If the sender of the Ping message does not receive a Pong message or any sort of communication within a 12 hour period, the sender drops that node from their own dictionary of known nodes. From the scenario for example, Node 1 when it comes online would immediately send a Ping to the bootnode, where the bootnode would then respond with a Pong, letting Node 1 it is online [27].

After the Ping/Pong exchange, the next step is the Findnode/Neighbors exchange. In this step, the node that sent the Ping message sends a Findnode message to the receiving node. The receiver responds with a Neighbors message, which contains a list of up to 16 node IDs that are closest to the requested node ID. If the requested node ID is the receiver's own node ID, the receiver will return its own node ID as the only neighbor. This exchange is how the dictionary of known nodes, known as a Kademlia Table is populated, growing the list of nodes that the node is directly connected with. From the scenario, Node 1, after a valid Ping/Pong exchange, would send a FindNode message to the bootnode, where the bootnode would send a Neighbors message listing out the up to 16 nodes it knows about.

If the sender of the Ping message is interested in obtaining additional information about the receiver, the final step is the ENRRequest/ENRResponse exchange. In this step, the sender sends an ENRRequest message to the receiver, which requests the receiver's Ethereum Node Record (ENR). The ENR is a record that contains information about the node, such as its IP address, the secp256k1 compressed public key, tcp/udp ports, and other metadata listed out in key-value pairs. The receiver responds with an ENRResponse message, which contains the requested ENR.

DiscoveryV4 messages are sent as UDP datagrams, with each packet starting with a header, containing the hash of the message, signature and the packet type. Every node has a cryptographic identity, a key on the secp256k1 elliptic curve. The public key of the node serves as the identifier or the “node ID”, where this public key corresponds to the private key that we passed in with the “nodekeyhex” flag in the “geth” command line startup for the node containers. The signature is encoded as a byte array of length 65 as the concatenation of the signature values r, s, and the recovery id, v. The packet type is a single byte defining the type of the message, from 0x01 defining a Ping message, 0x02 Pong, 0x03, FindNode, 0x04 Neighbors, 0x05 ENRRequest, 0x06 ENRResponse.

For both DiscoveryV4 and DiscoveryV5, the “discovery.lua” script is used as the plugin and the interface for Wireshark, while also using PYDEVP2P as the backend for the bulk of the dissection. The plugin interfaces with the LUA Wireshark API registering a dissector for the UDP ports from 30303 to 30308 and naming the dissector protocol as “devp2p” for both DiscoveryV4 and DiscoveryV5. Upon a new packet, that matches the plugin registration for the ports, the payload of the packet is sent to the PYDEVP2P bridge using the “handleDiscv4Msg(srcaddr, dstaddr, payload, pinfo.visited, pinfo.number)” function. This is possible again using Lunatic Python, where LUA can call a function that is written in python using the python shared object binary.

This function is found in “bridge.py” which is the main interface for all the LUA plugins, including all the functions for DiscoveryV5 and RLPx messages. From there, the “discover” sub-module is used, calling “decodeDiscv4()”found in “discover/v4wire/decode.py”. This function pulls out the fields from the header, which are of static lengths. First the hash, which is 32 bytes long, followed by the signature, 65 bytes long, then the packet type, the first byte after the signature. The hash can then be verified, checking the equality of the hash field (32 bytes) and the keccak256Hash of all the data after the first 32 bytes of the payload. This verification is crucial as both DiscoveryV4 and DiscoveryV5 are registered with the same dissector, and the packet is first checked to see if it is a valid DiscoveryV4 packet type, and if not, it will then error out and try again with DiscoveryV5. These three fields make up the header on all DiscoveryV4 messages, shown below in Figure 26, the output of just the header information for a dissected PING packet.

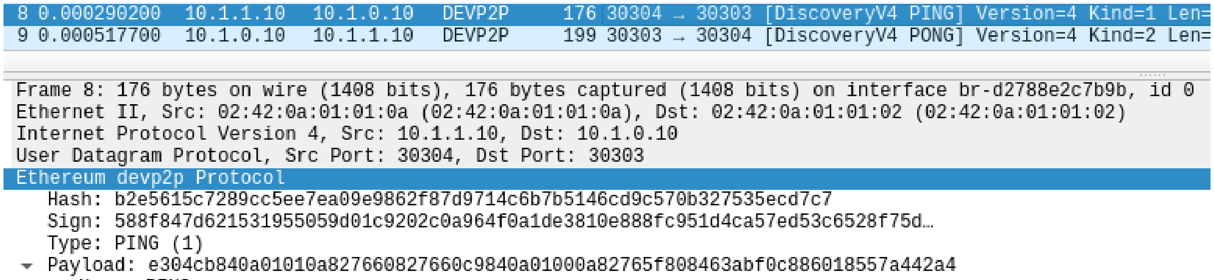


Figure 26: DiscoveryV4 Dissected Header Fields

After extracting the three fields, including verifying the message hash, the signature can be used to recover the sender's public key. This signature is created using the Elliptic Curve Digital Signature Algorithm (ECDSA) and by signing the message hash using the private elliptic curve key. The recipient of this message can then recover the public key using the elliptic curve cryptography found in the "elliptic" sub-module in PYDEVP2P which will be discussed in greater depth in Chapter 5.1. This implementation of ECDSA, including ECIES, is specific to Ethereum and is done solely with Python, without the use of any third-party dependencies. Using this ECDSA allows for non-repudiation, providing identification and proof of origin, authentication, and data integrity, however causing a significant performance impact.

Next, the message type byte can be used to determine the payload schema. The payload is encoded using RLP, a binary encoding method that allows for sending dynamic data structures and schema of data. RLP encoding can handle lists, strings, and bytes, where each field is preceded by a single byte determining the structure of the data. This byte can also determine the length of the data it represents if it is a list. In addition, RLP can be deeply nested, where each value or field in the data is preceded by an identifying byte. From there, the known type of the message can be RLP decoded into key/value pairs specific to the message type [28].

This decoding is taken care of by the "Packet" found in the "msg.py" in the v4wire package under discover. Here, the constructor of this class utilizes the message type, creating a sub-class depending on the message type, and automatically decodes the information. Each message schema is represented using tuples, where each tuple starts with the field name, followed by the field value, where the value could be another class or RLP schema, representing depth. Seen below in Figure 27, the schema definition for the Ping and Pong messages, where sub-schema definitions are being used as well, in the form of "FromInfo" and "ToInfo", seen in Figure 28. Each of these values found in the schema denote a RLP object instantiation that gives the rules for what constitutes for serialization and deserialization. This is necessary as there are values that must be deserialized into human readable time, or ip addresses, or even plain hex values. These RLP utilities are found in the "RLP" sub-module in PYDEVP2P, providing many custom types used throughout all the dissections.

```
class Packet(object):
    """
    Packet is implemented by all message types.
    """
    class Ping(RLPMessage):
        fields = (("Version", big_endian_int), ("Sender_Info", FromInfo), ("Recipient_Info",
                ToInfo), ("Exipration", date_value), ("ENR_Sequence_Num", big_endian_int))

    class Pong(RLPMessage):
        fields = (("Recipient_Info", ToInfo), ("Ping_Hash", hex_value),
                ("Expiration", date_value), ("ENR_Sequence_Num", big_endian_int))
```

Figure 27: Ping and Pong Class RLP Schema Definitions

```
class FromInfo(RLPMessage):
    fields = (("IP_Address", ip_address), ("UDP_Port",
            big_endian_int), ("TCP_Port", big_endian_int))


class ToInfo(RLPMessage):
    fields = (('IP_Address', ip_address),
            ('UDP_Port', big_endian_int), ("None", binary))
```

Figure 28: FromInfo and ToInfo RLP Schema Definitions

After successful RLP decoding, the individual fields can be converted into a python dictionary, which can then be sent back to the LUA plugin to be iterated over, creating the Wireshark tree for the DiscoveryV4 protocol and then displayed in Wireshark. Therefore, relating back to the scenario, the first step when a node comes online, or when a node wants to test connectivity with another node, the node will send a Ping message, seen in Figure 29. This message includes the protocol version, always 4, along with the sender and recipient information, specifically the IP address of each node and their respective ports for both UDP and TCP. Next, there is an expiration field, which is the validity window of the Ping message. Lastly, the ENR sequence number, which is a 64-bit unsigned integer and is mostly used to denote the version of the ENR of the sending node. This ENR is incremented if anything in the node's ENR changes, therefore making the other node send an ENRRequest to request the updated ENR information of the node.

The recipient then replies with a Pong message (0x02), with the same header found on every DiscoveryV4 message, followed by the recipient info, and the hash of the Ping that requested this Pong message, seen in Figure 30. Notice that the "Ping Hash" field in the Pong message matches the "Hash" field in the header of the Ping message that requested the Pong.

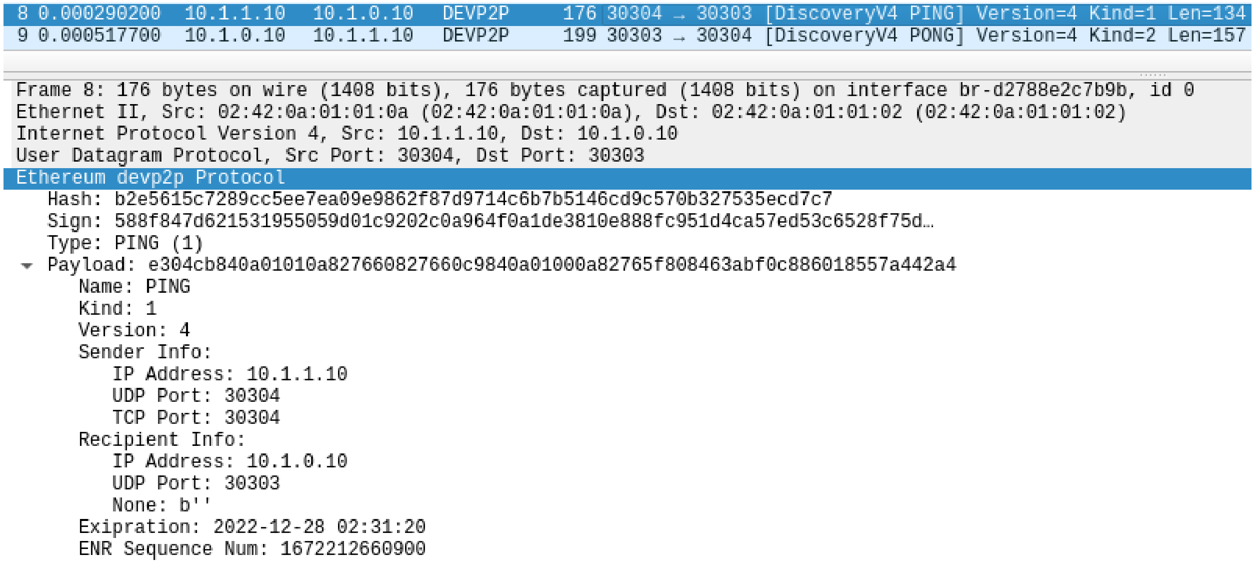


Figure 29: DiscoveryV4 Ping Packet Node1 to Bootnode

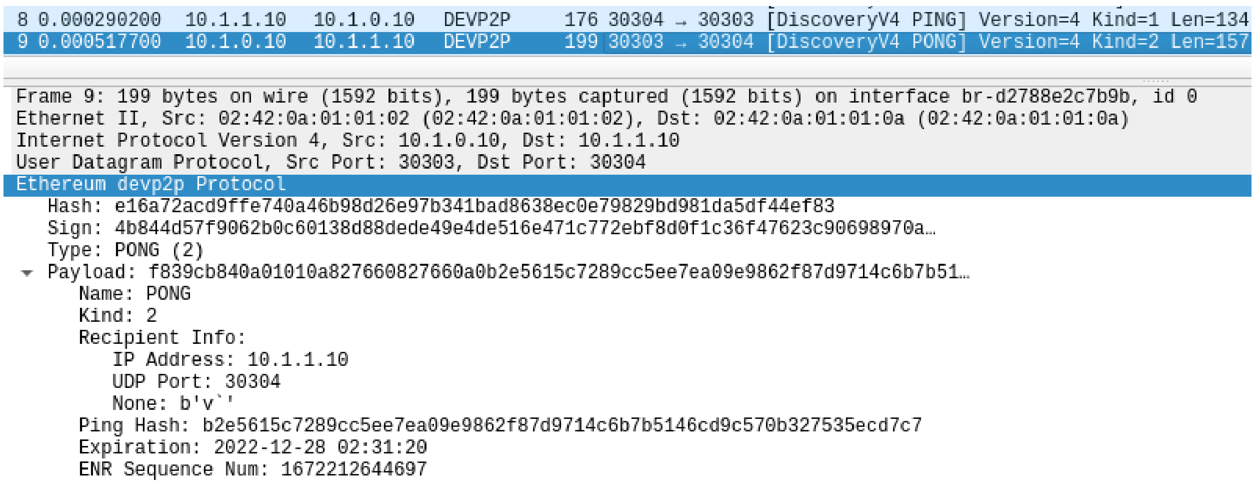
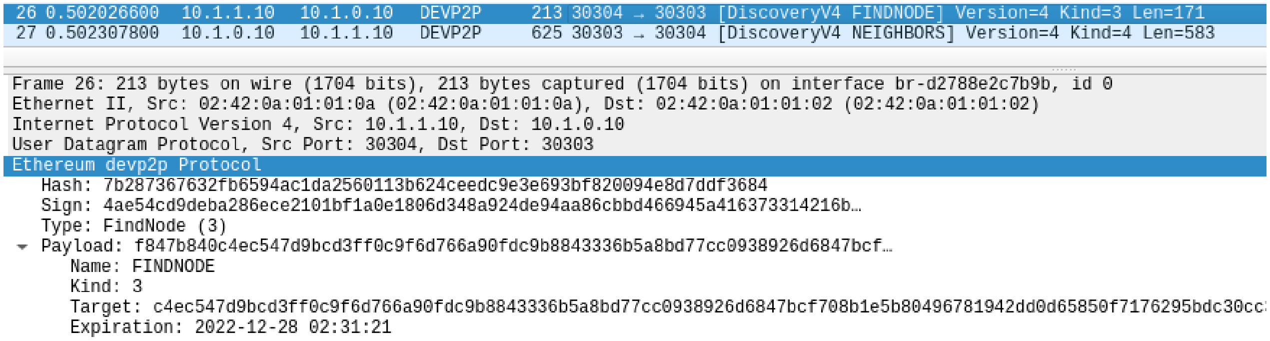


Figure 30: DiscoveryV4 Pong Packet Bootnode => Node1

The FindNode packet is used by a node to discover other nodes in the network. When a node sends a FindNode packet, including the target node ID seen in Figure 31. The target node ID is the 64-byte secp256k1 public key representing the node that the sender is trying to find. The receiving node will then respond with a Neighbors packet, seen in Figure 32, that contains a list of nodes in its routing table that are closest to the target node ID. Both the FindNode and Neighbors packets are important for maintaining the connectivity and robustness of the Ethereum network by allowing nodes to discover and connect to other nodes in the network.



```
26 0.502026600  10.1.1.10  10.1.0.10  DEVP2P  213 30304 → 30303 [DiscoveryV4 FINDNODE] Version=4 Kind=3 Len=171
27 0.502307800  10.1.0.10  10.1.1.10  DEVP2P  625 30303 → 30304 [DiscoveryV4 NEIGHBORS] Version=4 Kind=4 Len=583

Frame 26: 213 bytes on wire (1704 bits), 213 bytes captured (1704 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:0a (02:42:0a:01:01:0a), Dst: 02:42:0a:01:01:02 (02:42:0a:01:01:02)
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.0.10
User Datagram Protocol, Src Port: 30304, Dst Port: 30303
Ethereum devp2p Protocol
    Hash: 7b287367632fb6594ac1da2560113b624ceedc9e3e693bf820094e8d7ddf3684
    Sign: 4ae54cd9deba286ece2101bf1a0e1806d348a924de94aa86cbbd466945a416373314216b…
    Type: FindNode (3)
    Payload: f847b840c4ec547d9bcd3ff0c9f6d766a90fdc9b8843336b5a8bd77cc0938926d6847bcf…
        Name: FINDNODE
        Kind: 3
        Target: c4ec547d9bcd3ff0c9f6d766a90fdc9b8843336b5a8bd77cc0938926d6847bcf708b1e5b80496781942dd0d65850f7176295bdc30cc
        Expiration: 2022-12-28 02:31:21
```

Figure 31: DiscoveryV4 FindNode Packet Node1 => Bootnode

```
27 0.502307800  10.1.0.10  10.1.1.10  DEVP2P  625 30303 → 30304 [DiscoveryV4 NEIGHBORS] Version=4 Kind=4 Len=583

Frame 27: 625 bytes on wire (5000 bits), 625 bytes captured (5000 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
Internet Protocol Version 4, Src: 10.1.0.10, Dst: 10.1.1.10
User Datagram Protocol, Src Port: 30303, Dst Port: 30304
Ethereum devp2p Protocol
    Hash: c17405153364e36c8313c0b53b62203ab1df2c7a36b34adddc742f91128f0c2c
    Sign: f4a45dc22a683853607679348974 48cc47969ab60b68b88085c44c1d8e8a66ea60b6deff…
    Type: Neighbors (4)
    Payload: f901e2f901daf84d8434e7a56c82765f82765fb840715171f50508aba88aecd1250af392…
        Name: NEIGHBORS
        Kind: 4
        Nodes:
            Nodes #1:
                IP Address: 52.231.165.108
                UDP Port: 30303
                TCP Port: 30303
                Node ID: 715171f50508aba88aecd1250af392a45a330af91d7b90701c436b618c86aaa1589c9184561907bebbb56439b8f8787bc
            Nodes #2:
                IP Address: 3.209.45.79
                UDP Port: 30303
                TCP Port: 30303
                Node ID: 22a8232c3abc76a16ae9d6c3b164f98775fe226f0917b0ca871128a74a8e9630b458460865bab457221f1d448dd9791d2
            Nodes #3:
                IP Address: 65.108.70.101
                UDP Port: 30303
                TCP Port: 30303
                Node ID: 2b252ab6a1d0f971d9722cb839a42cb81db019ba44c08754628ab4a823487071b5695317c8ccd085219c3a03af063495b
```

Figure 32: DiscoveryV4 Neighbors Packet Bootnode => Node1

Lastly, if a node would like more information about a node, in the form of an Ethereum Node Record (ENR) or if the ENR Sequence number has changed from a received Ping/Pong, then the node will send an ENRRequest, shown dissection in Figure 33. This request is sent directly to the recipient, with an expiration timestamp again to provide a message validity window. The recipient then makes sure the sender is a valid node, as in the recipient has contacted the sender with a valid Ping/Pong exchange in the past 12 hours. The recipient then sends a ENRResponse, see dissection in Figure 34, which holds the ENR for the node sending the ENRResponse. The ENR holds important information including the hash of the request, the signature of the record contents, followed by the sequence number and a list of arbitrary key/value fields pertaining to the node. These key value fields contain node identifier information like the IP address, tcp/udp ports, and secp256k1 compressed public key.

```
73 6.040676598  10.1.1.10  10.1.0.10  DEVP2P   146 30304 → 30303 [DiscoveryV4 ENRREQUEST] Version=4 Kind=5 Len=104
74 6.040938798  10.1.0.10  10.1.1.10  DEVP2P   340 30303 → 30304 [DiscoveryV4 ENRRESPONSE] Version=4 Kind=6 Len=298

Frame 73: 146 bytes on wire (1168 bits), 146 bytes captured (1168 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:0a (02:42:0a:01:01:0a), Dst: 02:42:0a:01:01:02 (02:42:0a:01:01:02)
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.0.10
User Datagram Protocol, Src Port: 30304, Dst Port: 30303
Ethereum devp2p Protocol
   Hash: 6060288299a1dbb487f56969680d14c13150f9e03c793d8a61598be30bacceba
   Sign: ca4b0343065ead7b86a005a71a7670438122b4ec4cdc05e68bf099ec0c3150f875b9457d…
   Type: ENRRequest (5)
 ▾ Payload: c58463abf0ce
      Name: ENRREQUEST
      Kind: 5
      Expiration: 2022-12-28 02:31:26
```

Figure 33: DiscoveryV4 ENRRequest Packet Node1 => Bootnode

```
74 6.040938798  10.1.0.10  10.1.1.10  DEVP2P   340 30303 → 30304 [DiscoveryV4 ENRRESPONSE] Version=4 Kind=6 Len=298

Frame 74: 340 bytes on wire (2720 bits), 340 bytes captured (2720 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
Internet Protocol Version 4, Src: 10.1.0.10, Dst: 10.1.1.10
User Datagram Protocol, Src Port: 30303, Dst Port: 30304
Ethereum devp2p Protocol
   Hash: 21f3a9ac033313e3df49a7b75d109fd53a730a65efb4c4010f4e194173029bd4
   Sign: c70cf7f8bb3818025d4c0c74dfd434b95964b322a8655cf5b801756fc1545f546eaec4af…
   Type: ENRResponse (6)
 ▾ Payload: f8c6a06060288299a1dbb487f56969680d14c13150f9e03c793d8a61598be30baccebaf8…
      Name: ENRRESPONSE
      Kind: 6
      Request Hash: 6060288299a1dbb487f56969680d14c13150f9e03c793d8a61598be30bacceba
      Signature: 30d7ef935580a7abf905a17b176b3f7ed09e092670332274fd0b870f896d45455d029af9196cd202f23d7d5eee2445ed5e9959e09e1ba821f126573c844e2d59
      Sequence #: 1672212644697
      eth: Fork Hash: c18145ad
      id: v4
      ip: 10.1.0.10
      secp256k1: 022c4b6808e788537ca13ab4c35e6311bc2553b65323fb0c9e9a831303a1059b87
      snap: N/A
      tcp: 30303
      udp: 30303
```

Figure 34: DiscoveryV4 ENRResponse Packet Bootnode => Node1

4.3.2 DiscoveryV5 Dissection

DiscoveryV5 was created as a logical successor to DiscoveryV4, fixing many of DiscoveryV4's shortcomings. In the rationale documentation for DiscoveryV5, many goals are laid out, for example, fixing endpoint proof. This issue comes from DiscoveryV4, where the existing mutual endpoint verification may be unreliable. One node may assume that the other node knows about a recent Ping/Pong exchange, sending a FindNode message. However, if this other node does not store information reliably, or drops this information, then a new Ping/Pong exchange would have to take place, followed by another FindNode [29].

Other goals of DiscoveryV5 include the requirement for knowledge of a destination node ID for communication. The goal is to make obtaining a logical node ID expensive before any discovery communications because in DiscoveryV4, any message could provoke a response from a node using just the node's IP address alone. DiscoveryV5 also mitigates replay prevention and fixes the "expiration" field issue at the end of all the DiscoveryV4 messages. The issue came from a requirement that all system clocks must be synced to guarantee message validity; this obviously caused an issue in a protocol used globally with several implementations. Lastly, DiscoveryV5 provides message obfuscation by introducing an encryption scheme and handshake. However, this does not ensure complete confidentiality but aids with issues such as traffic amplification, replay, and packet authentication. This "masking" of data protects against passive eavesdroppers;

however, as discussed in the Analysis chapter, the encryption scheme and handshake are not forward-secure and active participants can access node information by simply asking for it.

As stated, DiscoveryV5 is primarily used in consensus layer clients; however, it is found in GETH solely for proof-of-concept and developmental purposes. However, the GETH implementation is complete and robust and allows us to see its use in an execution client and compare the results with DiscoveryV4. Furthermore, discoveryV5 communication is “opt-in” for GETH clients, simply using the “—v5disc” flag in conjunction with the “geth” command when starting up the GETH client. Interestingly, when setting this flag to use DiscoveryV5, DiscoveryV4 is active, in parallel, but completely separate from one another, not sharing information received. This is partly because GETH still must discover other execution clients who solely use DiscoveryV4.

Discovery communication is encrypted and authenticated using session keys, established in the handshake. A handshake can be initiated by either side of communication at any time. Relating back to the scenario, Node 1 wants to communicate with the Bootnode, therefore Node 1 must have a copy of the Bootnode’s ENR in order to communicate with it. If Node 1 has session keys from prior communication with the Bootnode, it encrypts its request with those keys. If no keys are known, it initiates the handshake by sending an ordinary message packet with random message content for example a Ping or a FindNode, shown as the first message sent at the top of the sequence diagram in Figure 35 [30].

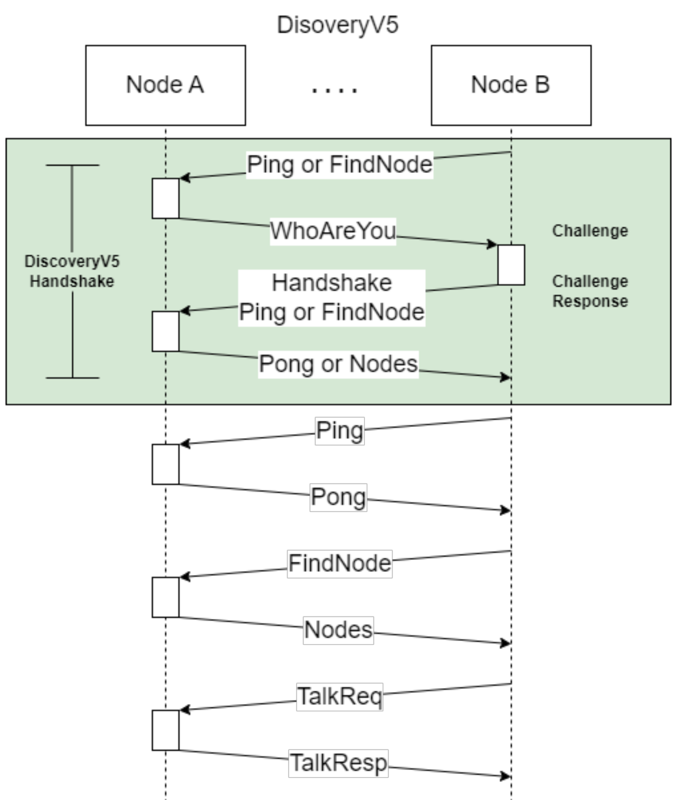


Figure 35: DiscoveryV5 Message Sequence Diagram

The Bootnode will receive this message packet and extract the source node ID from the packet header, if the Bootnode has session keys from a prior communication with Node 1, then it will attempt to decrypt the message data. If the decryption and authentication of the message succeeds, then there is no need for a handshake and the Bootnode can simply respond to the request from Node 1. However, if the decryption fails or like in this case, where there are no session keys set up because Node 1 is communicating with the Bootnode for the first time, the Bootnode then initiates a handshake by responding with a “WhoAreYou” packet.

Node 1 then receives the challenge sent by the Bootnode, which is a uniquely generated “id-nonce”. Node 1 then resends the original request packet, either a “Ping” or a “FindNode” message, but this time in the form of a handshake packet. This packet contains three parts in addition to the message: id-signature, ephemeral-pubkey, and the record. Node 1 derives the new session keys utilizing Elliptic Curve Diffie Hellman, which will be discussed in greater detail in the Analysis chapter.

When the Bootnode receives the Handshake message packet, it first loads back the WhoAreYou challenge that it sent earlier. The Bootnode then performs key derivation using its own static private key and the ephemeral-pub key from the handshake message. Using the resulting session keys, the message payload in the handshake message can be attempted to be decrypted and authenticated. Upon valid decryption and authentication, the Bootnode can then respond to the message, with either a Pong or a Nodes message, thus resulting in the end of the DiscoveryV5 handshake stage.

Following the handshake, similar messages that are seen in DiscoveryV4 are sent, such as Ping/Pong, and FindNode/Nodes, where Nodes is like DiscoveryV4’s Neighbors message. However, in the Nodes packet for DiscoveryV5, every Node ID is now seen as a full ENR entry. Lastly, there are several other packet type specifications, but only two with formal implementation, such as the TalkReq (0x05) and TalkResp (0x06) messages. TalkReq sends an application-level request for pre-negotiating connections made through another application-specific protocol. The recipient of the TalkReq must respond with a TalkResp message containing the response. It is important to note that both of these messages were not able to be captured in the private Ethereum network utilizing GETH nodes. Therefore, their true use and implementation could vary.

Now, let’s take a closer look at dissecting DiscoveryV5. The same “discovery.lua” plugin is used to register DiscoveryV5 with Wireshark, and on incoming packets, first DiscoveryV4 is tested, and without a valid hash and message layout, DiscoveryV5 is then tested, by then calling the “handleDiscv5Msg()” in “bridge.py”. DiscoveryV5 requires knowledge of the other nodes’ public key prior to communication, therefore, dissection is handled differently than DiscoveryV4. Each node is initialized at the top of “bridge.py” with the corresponding private key and IP address, creating a "Node" which is found in "node.py". A Node in terms of PYDEVP2P holds all the state full information needed for dissection, such as challenges, session keys, ephemeral keys and more. This Node class is responsible for handling all of the peer connections for the Node, handling all DiscoveryV5 and RLPx dissection which will be discussed in greater detail later. Seen in Figure 36, the top-level Node class, which utilizes the Discv5Codec class that handles all the encoding and decoding for DiscoveryV5, also handling sessions, including session keys and previous handshakes.

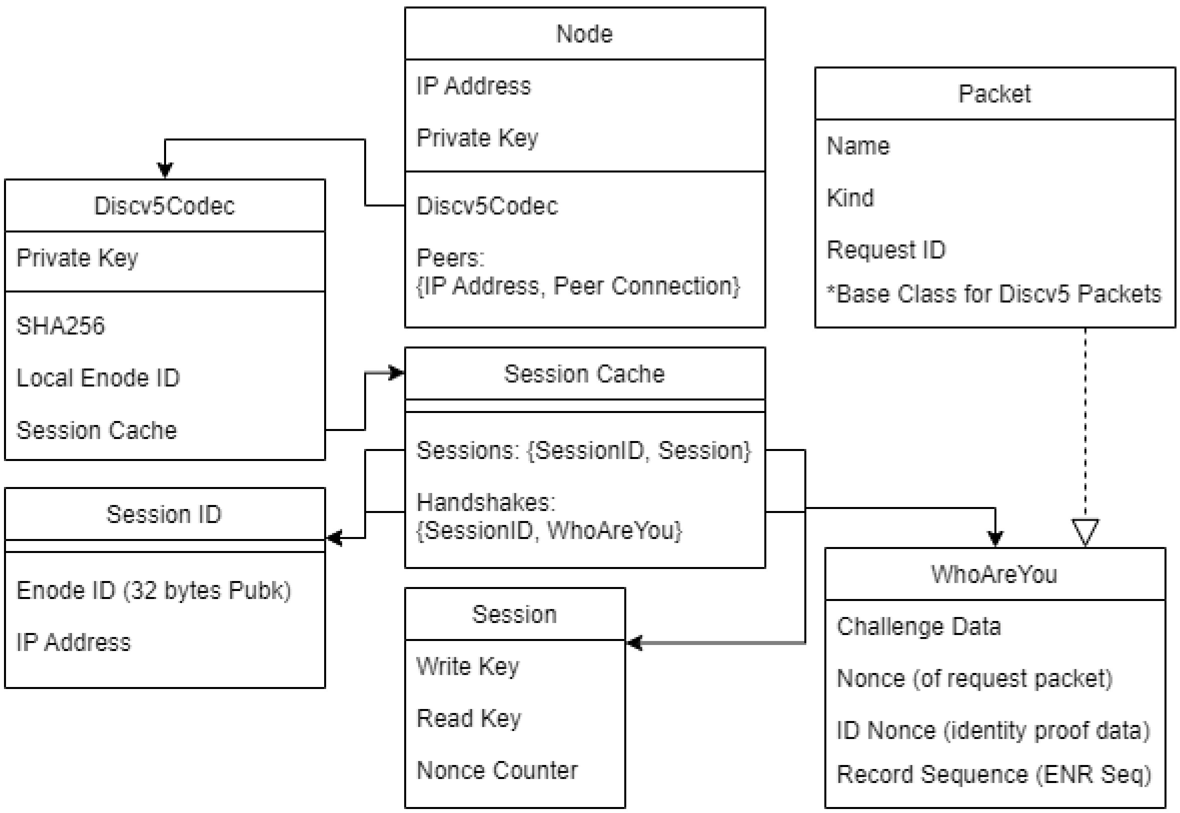


Figure 36: DiscoveryV5 Class Diagram

This means that in order to dissect DiscoveryV5 and RLPx prior knowledge of the node's IP addresses and their private elliptic curve key will need to be known. So, the source IP and destination IP address are used to pull in the correct Nodes, therefore pulling in all the state information, including previous peer connections, challenges, or keys setup amongst peers. Shown in Figure 37, the instantiation of the known Nodes including their IP address and their elliptic curve private key. This dictionary of Node classes is used for both DiscoveryV5 and RLPx dissection.

```python
boot_priv_static_k = "3028271501873c4ecf501a2d3945dcb64ea3f27d6f163af45eb23ced9e92d85b"
node1_priv_static_k = "4622d11b274848c32caf35dded1ed8e04316b1cde6579542f0510d86eb921298"
node2_priv_static_k = "816efc6b019e8863c382fe94cefe8e408d53697815590f03ce0a5cbfdd5f23f2"
node3_priv_static_k = "3fadc6b2fbd8c7cf1b2292b06ebfea903813b18b287dc29970a8a3aa253d757f"

all_nodes: dict[str, Node] = {
    "10.1.0.10": Node("10.1.0.10", hex_to_bytes(boot_priv_static_k)),
    "10.1.1.10": Node("10.1.1.10", hex_to_bytes(node1_priv_static_k)),
    "10.1.2.20": Node("10.1.2.20", hex_to_bytes(node2_priv_static_k)),
    "10.1.3.30": Node("10.1.3.30", hex_to_bytes(node3_priv_static_k))
}
```

Figure 37: PYDEVP2P Bridge Node Creation with Private Keys

DiscoveryV5 header information is "masked" using symmetric encryption in order to avoid static identification of protocol firewalls. The header starts with a Masking IV which is 16 bytes, then using the local nodes Enode ID, or the first 16 bytes of the public key, a new AES CTR cipher can be set up with the IV as the MaskingIV and the key Enode ID. From there, after decrypting the header, all the information can be pulled out, like the Protocol ID, Version, Flag, Nonce, Auth Size and Type of the message. The types/flags of the message payload can be either "Message", "WhoAreYou", or "Handshake". As stated before, if the message is sent prior to a handshake with proper session keys setup, then the first message payload will be UNKNOWN, as seen in Figure 38. Here, the header is able to be unmasked, however, the payload data of the message is not able to be decrypted, therefore triggering the start of the handshake amongst nodes. In this case, between Node 1 and the Bootnode.

```
29 0.700993500  10.1.1.10  10.1.0.10  DEVP2P   133 30304 → 30303 [DiscoveryV5 MESSAGE UNKNOWN/v5]
30 0.701203600  10.1.0.10  10.1.1.10  DEVP2P   105 30303 → 30304 [DiscoveryV5 WHOAREYOU WHOAREYOU/
31 0.701498200  10.1.1.10  10.1.0.10  DEVP2P   412 30304 → 30303 [DiscoveryV5 HANDSHAKE FINDNODE/v
32 0.701997700  10.1.0.10  10.1.1.10  DEVP2P  1125 30303 → 30304 [DiscoveryV5 MESSAGE NODES/v5] Ve

Frame 29: 133 bytes on wire (1064 bits), 133 bytes captured (1064 bits) on interface br-d2788e2c7b9b,
Ethernet II, Src: 02:42:0a:01:01:0a (02:42:0a:01:01:0a), Dst: 02:42:0a:01:01:02 (02:42:0a:01:01:02)
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.0.10
User Datagram Protocol, Src Port: 30304, Dst Port: 30303
Ethereum devp2p Protocol
  Header: 98f01e6a04292ed2ee9d8340b067d4f710e41d5967a53bb8316e90d80af81d0b835f81de…
      Iv: 98f01e6a04292ed2ee9d8340b067d4f7
      Protocolid: 110404070241845
      Version: 1
      Flag: 0
      Nonce: dae0a471e5e24884cffaf6b6
      Authsize: 32
      Authdata: b91ffa042b49f544a12ecb8502aca6521dd791ce915de0bb8740cd464e2e58de
      Src: b91ffa042b49f544a12ecb8502aca6521dd791ce915de0bb8740cd464e2e58de
      Type: MESSAGE
  Payload: 4fac76f377379fcb4f838988d9ea61c7f366d9ac
      Nonce: dae0a471e5e24884cffaf6b6
```

Figure 38: DiscoveryV5 Unknown Packet Node1 => Bootnode

Next, a WhoAreYou packet is sent, where the "authdata" section contains information for the identity verification procedure. The "message" part of the WhoAreYou packet is always empty, and the "nonce" part of the message is set to the "nonce" field of the message that caused the WhoAreYou packet. We can see with the dissected packets, that the Nonce field in both Figures

38 and 39 match. One major thing to note here is that the dissector is actively listening to the packets, however it is always receiving the messages in the context of the receiver. This means that the dissector has to retroactively set up handshakes after they have taken place. For example, the dissector will receive a WhoAreYou packet, same as Node 1, the Bootnode will already know that it sent the WhoAreYou packet and stored that information. So the dissector also needs to store this information for the Bootnode, which in this case is the “source node”. That way, when a handshake message is received, this information is already there to properly setup the session keys on the dissector side.

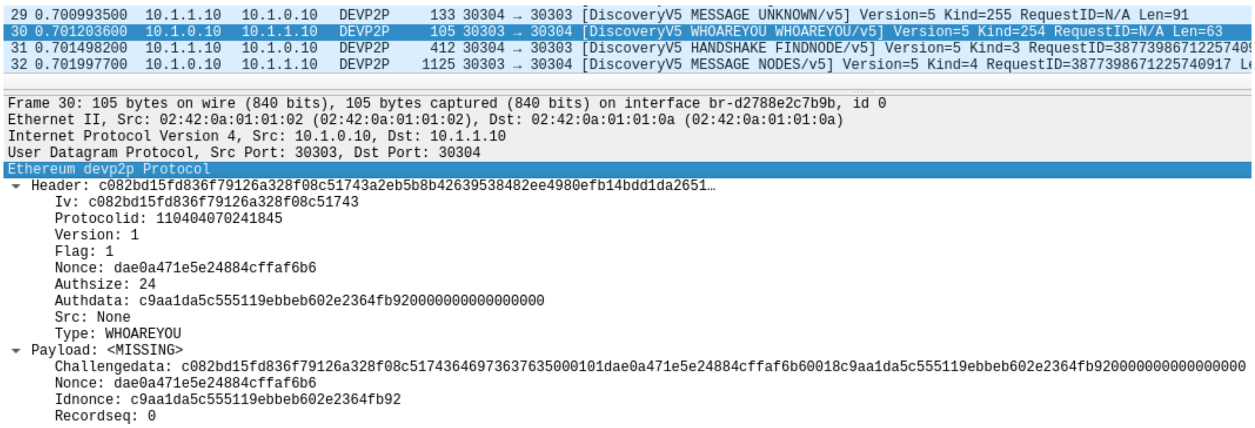


Figure 39: DiscoveryV5 WhoAreYou Packet Bootnode => Node1

This is where most of the complexity of the dissector comes from, and we will see much more of this when dealing with RLPx dissection. As a listener, the dissector must store information for both the receiver and the sender once the dissector receives a specific packet. So, now, Node 1 sends the Handshake FindNode packet back to the Bootnode, where Node 1 has effectively already set up their session keys, but the dissector must do this retroactively once the Handshake message is captured. Seen in Figure 40, the dissected output of the Handshake FindNode packet, where the payload of the message is now seen. Here both sides have successfully set up their session, by verifying the ID Signature decoding the public key, and deriving the session keys.

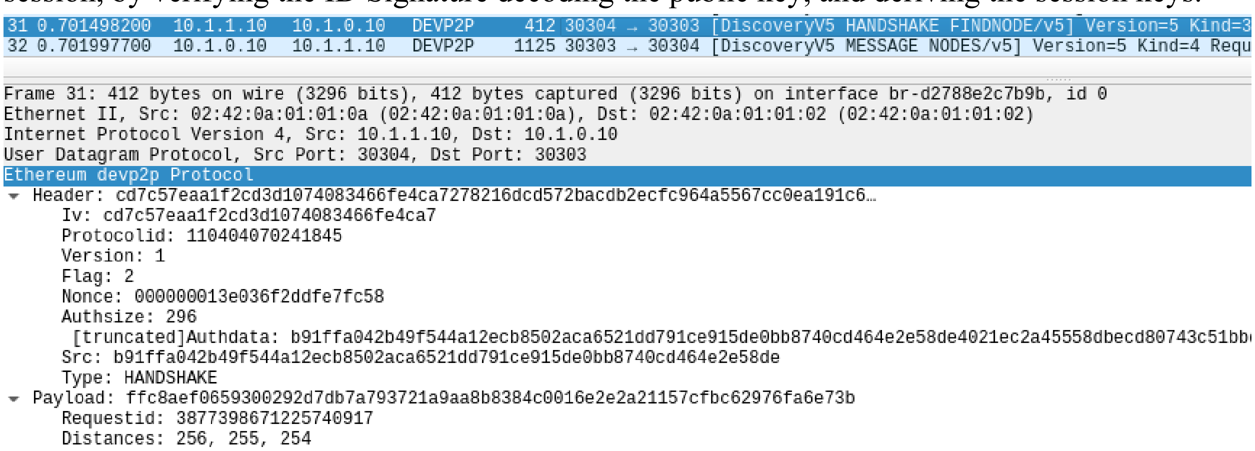


Figure 40: DiscoveryV5 FindNode Packet Node1 => Bootnode

After the handshake and session key setup, dissection proceeds as normal just like DiscoveryV5. Seen in Figure 41, the response to the FindNode Handshake packet, where each entry in the Nodes packet is a full Ethereum Node Record (ENR).

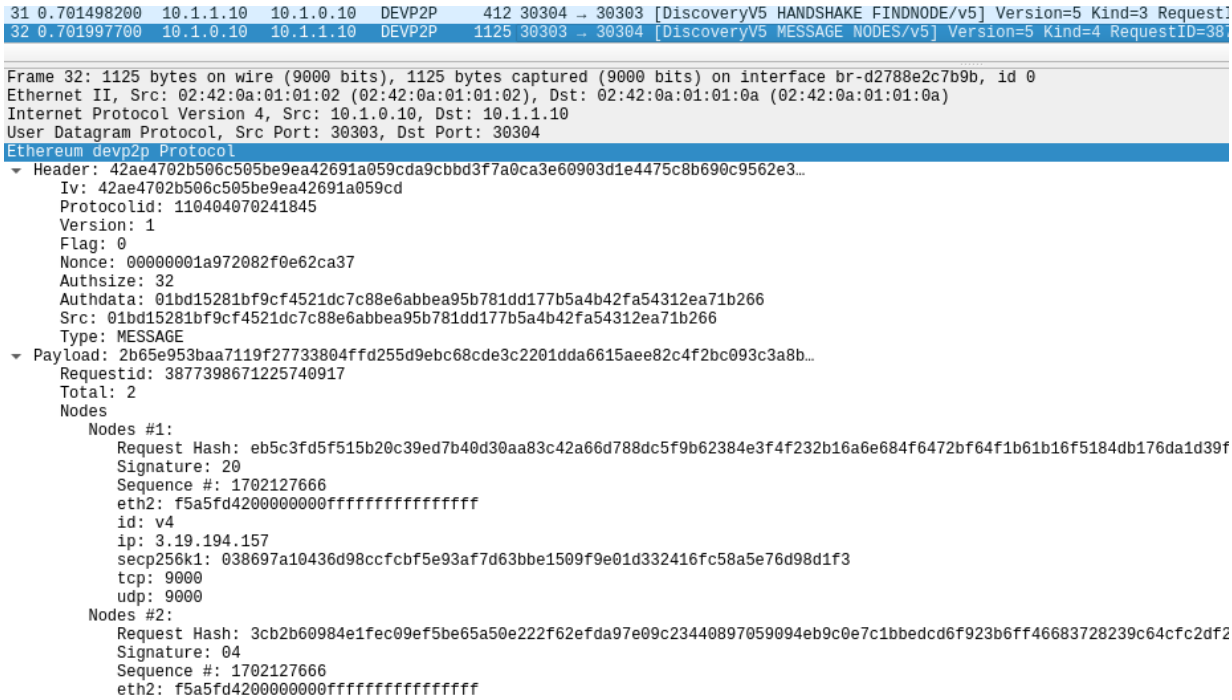


Figure 41: DiscoveryV5 Nodes Bootnode => Node1

Each message after the handshake is classified with a type/flag of "MESSAGE" and is labeled with the "Kind" of Ping, Pong, FindNode, Nodes, TalkReq, TalkResp, etc. Seen below, Figure 42, Node 1 pinging for liveliness the Bootnode and responding with a Pong in Figure 43. Lastly, another FindNode/Nodes exchange between Node 1 and the Bootnode, seen in Figures 44 and 45, this time outside of the Handshake. An important note is that the TalkReq/TalkResp packets were unable to be populated throughout the network, therefore unable to be captured and dissected.

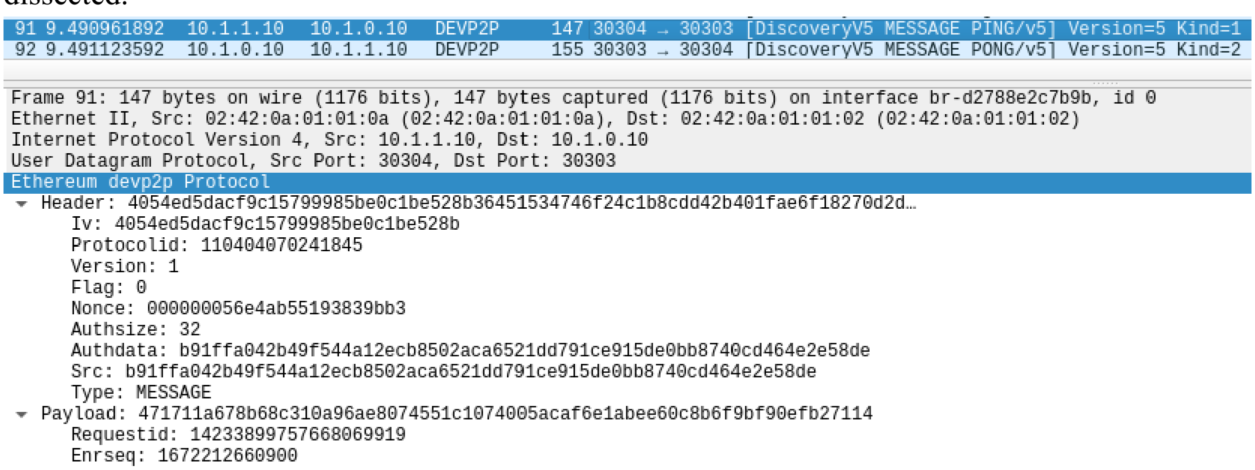


Figure 42: DiscoveryV5 Ping Packet Node1 => Bootnode

```
91 9.490961892  10.1.1.10  10.1.0.10  DEVP2P  147 30304 → 30303 [DiscoveryV5 MESSAGE PING/v5] Version=5
92 9.491123592  10.1.0.10  10.1.1.10  DEVP2P  155 30303 → 30304 [DiscoveryV5 MESSAGE PONG/v5] Version=5

Frame 92: 155 bytes on wire (1240 bits), 155 bytes captured (1240 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
Internet Protocol Version 4, Src: 10.1.0.10, Dst: 10.1.1.10
User Datagram Protocol, Src Port: 30303, Dst Port: 30304
Ethereum devp2p Protocol
  Header: 1a5d55616405b2c41827db3a6e006e979b6d98767ab1bffcbc109bf46a4c2c2d6bcdffcc…
      Iv: 1a5d55616405b2c41827db3a6e006e97
      Protocolid: 110404070241845
      Version: 1
      Flag: 0
      Nonce: 00000007f835234fef9d16d2
      Authsize: 32
      Authdata: 01bd15281bf9cf4521dc7c88e6abbea95b781dd177b5a4b42fa54312ea71b266
      Src: 01bd15281bf9cf4521dc7c88e6abbea95b781dd177b5a4b42fa54312ea71b266
      Type: MESSAGE
  Payload: f938aae15ead44d1ec31516cf9ecb8db131226da1ceb8caf50a8736a451c021eee511bfc…
      Requestid: 14233899757668069919
      Enrseq: 1672212644697
      Toport: 30304
```

Figure 43: DiscoveryV5 Pong Packet Bootnode => Node1

```
93 9.500325892  10.1.1.10  10.1.0.10  DEVP2P  142 30304 → 30303 [DiscoveryV5 MESSAGE FINDNODE/v5] Version=5 Kind=3
94 9.500463492  10.1.0.10  10.1.1.10  DEVP2P  309 30303 → 30304 [DiscoveryV5 MESSAGE NODES/v5] Version=5 Kind=4 Rec

Frame 93: 142 bytes on wire (1136 bits), 142 bytes captured (1136 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:0a (02:42:0a:01:01:0a), Dst: 02:42:0a:01:01:02 (02:42:0a:01:01:02)
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.0.10
User Datagram Protocol, Src Port: 30304, Dst Port: 30303
Ethereum devp2p Protocol
  Header: b4134710477e7e25e295fcb0917a0343ed2b85641e3d7f08bff6c2ce9e1a940cefba572f…
      Iv: b4134710477e7e25e295fcb0917a0343
      Protocolid: 110404070241845
      Version: 1
      Flag: 0
      Nonce: 00000006f8d44ff4c763107a
      Authsize: 32
      Authdata: b91ffa042b49f544a12ecb8502aca6521dd791ce915de0bb8740cd464e2e58de
      Src: b91ffa042b49f544a12ecb8502aca6521dd791ce915de0bb8740cd464e2e58de
      Type: MESSAGE
  Payload: c2c7f2a01272a9eab88b55ede265a6d26d285e7091e0b3f500e55c19de
      Requestid: 3576569588916442984
      Distances: N/A
```

Figure 44: DiscoveryV5 FindNode Packet Node1 => Bootnode

```
93 9.500325892  10.1.1.10  10.1.0.10  DEVP2P  142 30304 → 30303 [DiscoveryV5 MESSAGE FINDNODE/v5] Version=5 Kind=3 RequestID=3576569588916442984 Len=16
94 9.500463492  10.1.0.10  10.1.1.10  DEVP2P  309 30303 → 30304 [DiscoveryV5 MESSAGE NODES/v5] Version=5 Kind=4 RequestID=3576569588916442984 Len=267

Frame 94: 309 bytes on wire (2472 bits), 309 bytes captured (2472 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
Internet Protocol Version 4, Src: 10.1.0.10, Dst: 10.1.1.10
User Datagram Protocol, Src Port: 30303, Dst Port: 30304
Ethereum devp2p Protocol
  Header: a1c216d8b2c83fdfe7c199d4bfca41367f57af879c7ca07521104cc538be20ed3ad06f63…
      Iv: a1c216d8b2c83fdfe7c199d4bfca4136
      Protocolid: 110404070241845
      Version: 1
      Flag: 0
      Nonce: 000000083b4525416938ddf6
      Authsize: 32
      Authdata: 01bd15281bf9cf4521dc7c88e6abbea95b781dd177b5a4b42fa54312ea71b266
      Src: 01bd15281bf9cf4521dc7c88e6abbea95b781dd177b5a4b42fa54312ea71b266
      Type: MESSAGE
  Payload: 9525cf0dece813a48394ec15f5eb56a2ef82cb4bbe85f052916f9b04c1277cef5943f49d…
      Requestid: 3576569588916442984
      Total: 1
      Nodes
        Nodes #1:
          Request Hash: 30d7ef935580a7abf905a17b176b3f7ed09e092670332274fd0b870f896d45455d029af9196cd202f23d7d5eee2445ed5e9959e09e1ba821f126573c844e2d59
          Signature: 018557a40359
          Sequence #: 6648936
          eth: Fork Hash: c18145ad
          id: v4
          ip: 10.1.0.10
          secp256k1: 022c4b6808e788537ca13ab4c35e6311bc2553b65323fb0c9e9a831303a1059b87
          snap: N/A
          tcp: 30303
          udp: 30303
```

Figure 45: DiscoveryV5 Nodes Packet Bootnode => Node1

4.4 Authenticated Node Communication

RLPx is a cryptographic peer-to-peer protocol suite which provides a general-purpose transport utilizing TCP and interface for applications to communicate via a P2P network. The protocol carries encrypted messages belonging to one or more 'capabilities' which are negotiated during connection establishment. RLPx is the only authenticated communication channel for execution clients, carrying data for all the application-level needs [31]. RLPx doesn't stand for anything specifically, however it is named after the RLP serialization formation as most of the underlying message payloads are encoded with RLP. The capabilities are sub protocols that are used to exchange messages between nodes, depending on the type of client. For example, the ETH subprotocol is used to exchange Ethereum blockchain data, while there exists the SHH subprotocol to exchange Whisper messages, or LES for light clients.

An RLPx connection is first established by a TCP connection and agreeing on ephemeral key material for further encrypted and authenticated communication. This process that creates the session keys is known as the "RLPx Handshake" and is carried out between the "initiator" or the node who opened the TCP connection, and the "recipient", the node who accepted the connection. An RLPx connection occurs after the node discovery phase, where nodes first join the network, then create secure connections between nodes to facilitate their application level data transfers.

As seen in Figure 46, the initiator in this case is Node B, where the recipient is Node A. Generally, the initiator first connects with the recipient by sending an "Auth Init" message, the recipient then accepts this message, decrypts, and verifies the authenticity of the message. The recipient, Node A, then sends an "Auth Ack" message using the "remote-ephemeral-key" and the "nonce" which was sent in the initialization message from Node B. Node A also derives secrets and sends the first encrypted frame containing a "P2P Hello" message. This P2P Hello message is the first packet sent over the connection and sent only once by both sides upon session initialization with the handshake. No messages are sent until both sides of the handshake send and receive a P2P message. Lastly, Node B receives the P2P Hello message and derives the same shared secrets and encrypts and sends its own P2P Hello message to Node A. Thus, completing the RLPx Handshake, where both sides have shared ephemeral keys, derived shared secrets and have generated the "AES secret" and "MAC secret" which are used for the session's encryption/decryption and message authentication.

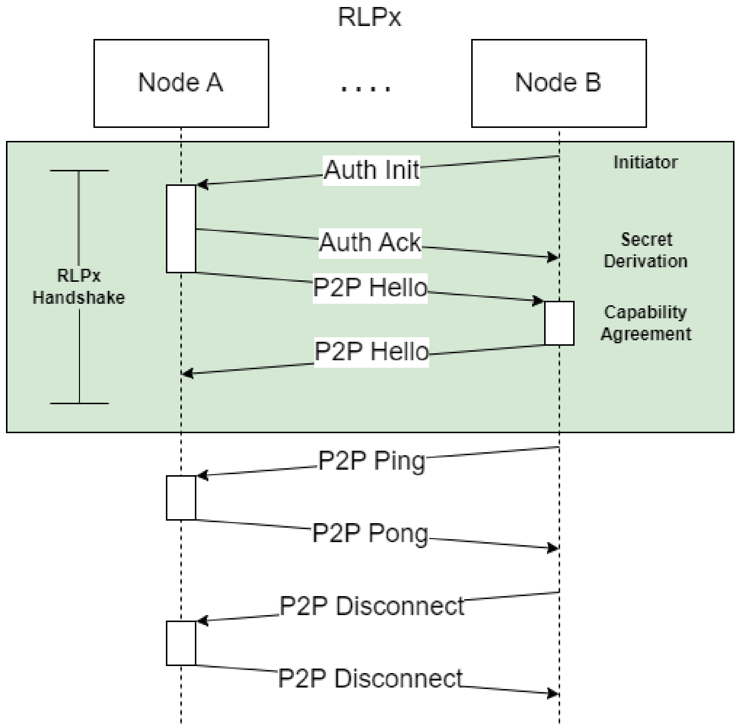


Figure 46: RLPx Handshake & P2P Capability Message Sequence Diagram

All messages following the initial handshake are associated with a "capability". Any number of capabilities can be used concurrently on a single RLPx connection. A capability is identified by a short ASCII name, with a max of eight characters, and a version number. The capabilities supported on either side of the connection are exchanged within the Hello message (0x00) found at the end of the RLPx Handshake seen above in Figure 46. The standard capability that is always supported between both sides of the connection is known as the "p2p" capability.

In this scenario and dissector, only the ETH and SNAP sub protocols are seen, as the clients are running on a proof-of-work consensus algorithm network, and running GETH clients that support SNAP. ETH is used to exchange blocks, transactions and other data regarding block information between nodes. SNAP is used to facilitate the exchange of Ethereum state snapshots between peers. The other "p2p" capability messages include a Disconnect (0x01) which is used to inform the peer that a disconnection is imminent, including a reason for why the peer wants to disconnect. Lastly, there also exists a Ping (0x02) and a Pong (0x03) message for RLPx session liveliness.

Now, let's look deeper into the actual dissection of the RLPx Handshake messages followed by the "p2p" capability messages. First a new dissector plugin must be registered with Wireshark, specifically named "rlpx.lua". This dissector registers the protocol name "rlpx" with a description of "Ethereum RLPx Protocol", with the same standard ports that were used with discovery but for TCP, ports 30303 to 30308. As far as the LUA dissector is concerned, there are two main types of packets, a handshake packet and a normal RLPx packet which would carry the payload data of a capability for instance. Luckily, in the handshake packets, AuthInit and AuthAck, the first two bytes are non-encrypted, meaning this can be used to tell if the packet is a handshake or standard RLPx message. These first two bytes represent the size of the payload that is encrypted, so using this, the LUA dissector is able to calculate the payload size and check if it equals the first two bytes. Seen below in Figure 47, the LUA implementation to get the first two bytes of the payload, then checking if the length of the entire payload minus the "auth-size" is equal to two, which is the left over size representing the size of the size field.

```
    local auth_size = tvb(offset, 2)
    if (tvb:len() - auth_size:int() == 2) then
```

Figure 47: rlpx.lua Parsing RLPx Auth Size Field

This check is crucial in verifying if the packet is a handshake message or a normal standard message, as different functions in the "bridge.py" are called accordingly. Lastly, still in the "rlpx.lua" we can check if the known port of the packet, which again is 30303 to 30308, is associated with the sender or the receiver of the packet. If the source port is a known port, then this message is an AuthAck packet, otherwise, this message is an AuthInit packet. Lastly, the "handleRLPxHandshakeMsg()" function is called from the "bridge.py".

RLPx dissections require knowledge of the node's static private keys, that is the private key that is associated with the "nodekeyhex". This again utilizes the "Node" class to handle the state information for the peer, including the peers that the node is connecting to, utilizing the "Peer Connection" class. Seen in Figure 48, the class architecture for RLPx messages. Each Node gets an associated Peer Connection upon initialization of an RLPx Handshake, this effectively creates a graph. Nodes can have multiple Peer Connections, and a Peer Connection is technically just a wrapper for a node to node connection to hold all the state information between the two. So, a Peer Connection is two Nodes, one considered as a parent, which is the "own" Node, and the other. The first part is the "Handshake State" class, which deals with incoming Auth Init and Auth Acks, and also deals with the creation of the "Secrets". The secrets hold all of the derived information between the two, including the ephemeral keys, random public and private keys, and session keys including the AES and MAC keys. The details of the handshake cryptography will be gone into greater detail in the Analysis chapter. Once the secrets are generated for both parties, this then creates a "Session State" which handles the decoding, decryption and dissection of all messages after the handshake.

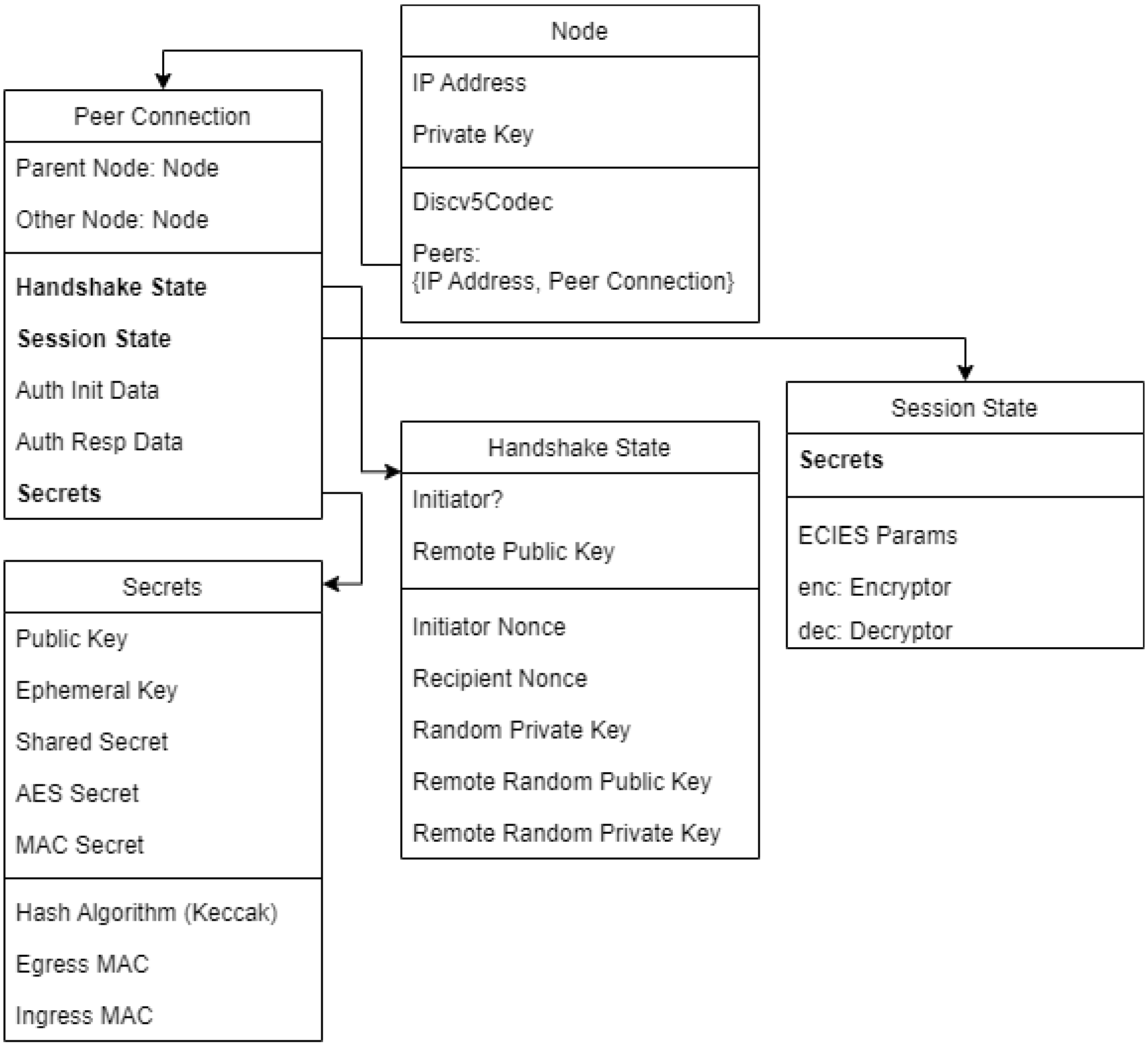


Figure 48: PYDEVP2P RLPx Class Flow Diagram

4.4.1 Handshake ECIES Decryption

Continuing on with the dissection of the RLPx Handshake messages, the packet payload is sent through the Node class, then calling the "read_handshake_msg()" function found in "handshake.py", which decrypts the data utilizing the node's static private key. This decryption implements ECIES (Elliptic Curve Integrated Encryption Scheme) Decryption where there cryptosystem used by RLPx is as follows:

- The elliptic curve secp256k1 with a generator G
- KDF(k, len): the NIST SP 800-56 Concatenation Key Derivation Function
- MAC(k, m): HMAC using the SHA-256 hash function.
- AES(k, iv, m): the AES-128 encryption function in CTR mode

So, let's say the Bootnode receives and Auth Init message from Node 1, seen in Figure 50. Node 1, will then need to decrypt this message, which was encrypted by the Bootnode. Node 1 will receive the following for the ciphertext: $R \,||\, iv \,||\, c \,||\, d$, where first Node 1 will pull out the ephemeral public key, also known as the ECDH (Elliptic Curve Diffie Helman) public key from the ciphertext which is R. Using this R, Node 1 is able to generate the shared secret (S) such that $S = Px$ where $(Px, Py) = k_{node\,1} * R$. That is the private static private key of Node 1 multiplied by the ephemeral public key R. This creates a point on the elliptic curve secp256k1, Px and Py, where Px is the actual shared secret [31].

Then, the encryption and authentication keys can be derived utilizing the NIST SP 800-56 Concatenation Key Derivation Function. Next, Node 1 verifies the authenticity of the message by checking whether the trailing message authentication tag d equals $MAC(sha256(K_m), iv \,||\, c)$. Lastly, obtaining the plaintext by using symmetric decryption utilizing the IV and the ciphertext and the AES derived key. All of which is found in the "crypto/ecies.py" module in PYDEVP2P. Below, in Figure 49, is a great depiction of the steps required for ECIES encryption [32].

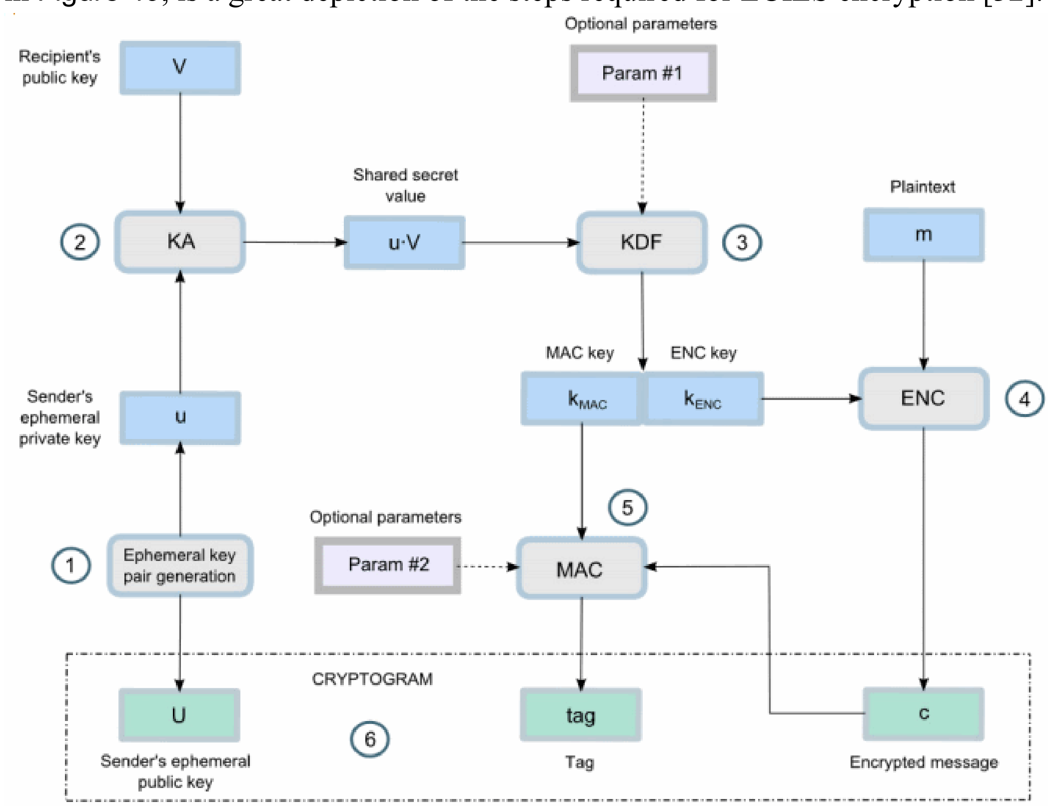


Figure 49: ECIES Hybrid Encryption Scheme

Seen below, in Figure 50, the contents of the dissected Auth Init packet sent from Node 1 to the Bootnode. Containing the following:

- Signature:
- InitiatorPubkey: The static public key of the Node 1
- Nonce: randomly generated nonce for the Init message
- Version: 04

```
11 0.000666800  10.1.1.10  10.1.0.10  RLPX     523 39436 → 30303 [HANDSHAKE] AUTH INIT
12 0.000709100  10.1.0.10  10.1.1.10  TCP       66 30303 → 39436 [ACK] Seq=1 Ack=458 Win=64768 Len=0 TSval=1157360977 TSecr=3258813120
13 0.000836500  10.1.0.10  10.1.1.10  DEVP2P   199 30303 → 30304 [DiscoveryV4 PONG] Version=4 Kind=2 Len=157
14 0.000913500  10.1.1.10  10.1.0.10  DEVP2P   199 30304 → 30303 [DiscoveryV4 PONG] Version=4 Kind=2 Len=157
15 0.000954400  10.1.0.10  10.1.1.10  DEVP2P   176 30303 → 30304 [DiscoveryV4 PING] Version=4 Kind=1 Len=134
16 0.001249900  10.1.1.10  10.1.0.10  DEVP2P   199 30304 → 30303 [DiscoveryV4 PONG] Version=4 Kind=2 Len=157
17 0.001279600  10.1.0.10  10.1.1.10  RLPX     456 30303 → 39436 [HANDSHAKE] AUTH ACK
18 0.001286100  10.1.1.10  10.1.0.10  TCP       66 39436 → 30303 [ACK] Seq=458 Ack=391 Win=64128 Len=0 TSval=3258813120 TSecr=1157360977
19 0.001490800  10.1.0.10  10.1.1.10  RLPX     274 30303 → 39436 [P2P Hello] Type=Hello Code=0 Len=160
20 0.001495900  10.1.1.10  10.1.0.10  TCP       66 39436 → 30303 [ACK] Seq=458 Ack=599 Win=64128 Len=0 TSval=3258813120 TSecr=1157360977
21 0.001523300  10.1.1.10  10.1.0.10  RLPX     274 39436 → 30303 [P2P Hello] Type=Hello Code=0 Len=160
22 0.001846500  10.1.0.10  10.1.1.10  RLPX     178 30303 → 39436 [ETH Status] Type=Status Code=0 Len=64
23 0.001986200  10.1.1.10  10.1.0.10  RLPX     178 39436 → 30303 [ETH Status] Type=Status Code=0 Len=64
24 0.058675300  10.1.0.10  10.1.1.10  TCP       66 30303 → 39436 [ACK] Seq=711 Ack=778 Win=64640 Len=0 TSval=1157361035 TSecr=3258813121

Frame 11: 523 bytes on wire (4184 bits), 523 bytes captured (4184 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:0a (02:42:0a:01:01:0a), Dst: 02:42:0a:01:01:02 (02:42:0a:01:01:02)
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.0.10
Transmission Control Protocol, Src Port: 39436, Dst Port: 30303, Seq: 1, Ack: 1, Len: 457
Ethereum RLPx Protocol
  Auth Size: 455
▾ Data: 04b403956bb978aa4c18cfe89ea09e7429d8792dbf6d521afd06986af1bbbd42415ff1c1…
    Signature: beb5ee854a4d733c3abf672d2c05eaf1750cdeef55c4063fe3e092c86bf200981b7bd981c03920d094bcff6b09772f03f761d8145920391854fc3ae3f9f94d3601
    InitatorPubkey: c35c2b7f9ae974d1eee94a003394d1cc18135e7fe6665e6b4f221970f1d9d59f6a58e76763803bcc9097eba4c91fd08b30405e65c53272b8635348e37f93cedc
    Nonce: 36b25e9c26a8b4c07d3ac33ef7a765a4e286d1ee66aaff7d37bad45ed421d77e
    Version: 04
    RandomPrivKey: 1919b264437eeb7ea1e14f658ed7b78c2ff8ff69cdc74b75dab0dadb130ed16a
```

Figure 50: RLPx Auth Init Packet Node1 => Bootnode

Following the Auth Init, the Bootnode then sends an Auth Ack message to Node 1, seen in Figure 51. This message contains the following:

- RandomPubkey: Ephemeral random public key of the Bootnode
- Nonce: randomly generated nonce for the Ack message
- Version: 04

```
11 0.000666800  10.1.1.10  10.1.0.10  RLPX     523 39436 → 30303 [HANDSHAKE] AUTH INIT
12 0.000709100  10.1.0.10  10.1.1.10  TCP       66 30303 → 39436 [ACK] Seq=1 Ack=458 Win=64768 Len=0 TSval=1157360977 TSecr=3258813120
13 0.000836500  10.1.0.10  10.1.1.10  DEVP2P   199 30303 → 30304 [DiscoveryV4 PONG] Version=4 Kind=2 Len=157
14 0.000913500  10.1.1.10  10.1.0.10  DEVP2P   199 30304 → 30303 [DiscoveryV4 PONG] Version=4 Kind=2 Len=157
15 0.000954400  10.1.0.10  10.1.1.10  DEVP2P   176 30303 → 30304 [DiscoveryV4 PING] Version=4 Kind=1 Len=134
16 0.001249900  10.1.1.10  10.1.0.10  DEVP2P   199 30304 → 30303 [DiscoveryV4 PONG] Version=4 Kind=2 Len=157
17 0.001279600  10.1.0.10  10.1.1.10  RLPX     456 30303 → 39436 [HANDSHAKE] AUTH ACK
18 0.001286100  10.1.1.10  10.1.0.10  TCP       66 39436 → 30303 [ACK] Seq=458 Ack=391 Win=64128 Len=0 TSval=3258813120 TSecr=1157360977
19 0.001490800  10.1.0.10  10.1.1.10  RLPX     274 30303 → 39436 [P2P Hello] Type=Hello Code=0 Len=160
20 0.001495900  10.1.1.10  10.1.0.10  TCP       66 39436 → 30303 [ACK] Seq=458 Ack=599 Win=64128 Len=0 TSval=3258813120 TSecr=1157360977
21 0.001523300  10.1.1.10  10.1.0.10  RLPX     274 39436 → 30303 [P2P Hello] Type=Hello Code=0 Len=160
22 0.001846500  10.1.0.10  10.1.1.10  RLPX     178 30303 → 39436 [ETH Status] Type=Status Code=0 Len=64
23 0.001986200  10.1.1.10  10.1.0.10  RLPX     178 39436 → 30303 [ETH Status] Type=Status Code=0 Len=64
24 0.058675300  10.1.0.10  10.1.1.10  TCP       66 30303 → 39436 [ACK] Seq=711 Ack=778 Win=64640 Len=0 TSval=1157361035 TSecr=3258813121

Frame 17: 456 bytes on wire (3648 bits), 456 bytes captured (3648 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
Internet Protocol Version 4, Src: 10.1.0.10, Dst: 10.1.1.10
Transmission Control Protocol, Src Port: 30303, Dst Port: 39436, Seq: 1, Ack: 458, Len: 390
Ethereum RLPx Protocol
  Ack Size: 388
▾ Handshake AUTH ACK
    RandomPubkey: 87ee6a1cf75a8a43815a5802d391143b6f11bd6f8e83d2bed52aca0bb7653dd6f6cc5bd499fc149dc09924c8587aaacbca9a8fce6a47c0435ae8f40a53e18a0f
    Nonce: be4642ef93143627c0257abb856f7fc59fa94298407cc4c9b79510bc51b0c1b9
    Version: 04
    RandomPrivKey: d51e031d25f11b0d962d6ce103b647af3d76a99a8b3440a4944807e21add82b4
```

Figure 51: RLPx Auth Ack Packet Bootnode => Node1

4.4.2 Exposing the Random Private Key

Just like with the dissection of DiscoveryV5 the state information has to be propagated for both sides of the exchange, which adds complexity as the dissector is capturing the packets retroactively, like it is the recipient. So, when an AuthMsg is received, the sender Node, in this case Node 1 must be populated with the information, such as its own nonce, random public keys, etc. Each side must know about their own randomly generated private key (RandomPrivKey), and in turn know their own random public key (RandomPubKey). This step is done in the background, and is completely obscured to the dissector, meaning without the random private key, and the random public key of the other node, the dissector would not be able to generate the shared secrets to dissect anything after the AuthInit and AuthAck messages.

Now, to mitigate this issue, we exposed the RandomPrivKey by adding this field to both the AuthInit and AuthResp packets found in the GETH source code. This allows for the dissector to be run completely by itself, and whenever a new handshake occurs, these random keys are immediately shared "in the clear" via the handshake messages in RLPx. These keys are found in the struct of both the definitions for the AuthMsgV4 and AuthRespV4 found in the GETH source code, still requiring the ECIES encryption as shown above. Shown below in Figure 52 lines 396 and 407 were added to the GETH source code "/p2p/rlpx/rlpx.go" to expose the RandomPrivKey. This is the explanation for why the custom GETH docker images had to be created in the first place for proper RLPx dissection to take place. Also, lines 575 and 596 were added in the same "rlpx.go" file to add the RandomPrivKey to the AuthInit and AuthResp structure prior to sending it out, seen in Figures 53 and 54.

```
390  // RLPx v4 handshake auth (defined in EIP-8).
391  type authMsgV4 struct {
392      Signature       [sigLen]byte
393      InitiatorPubkey [pubLen]byte
394      Nonce           [shaLen]byte
395      Version         uint
396      RandomPrivKey   [32]byte
397
398      // Ignore additional fields (forward-compatibility)
399      Rest []rlp.RawValue `rlp:"tail"`
400  }
401
402  // RLPx v4 handshake response (defined in EIP-8).
403  type authRespV4 struct {
404      RandomPubkey  [pubLen]byte
405      Nonce         [shaLen]byte
406      Version       uint
407      RandomPrivKey [32]byte
408
409      // Ignore additional fields (forward-compatibility)
410      Rest []rlp.RawValue `rlp:"tail"`
411  }
```

Figure 52: Exposing the RandomPrivKey to the AuthInit and AuthResp Messages in GETH

```
    msg := new(authMsgV4)
    copy(msg.Signature[:], signature)
    copy(msg.InitiatorPubkey[:], crypto.FromECDSAPub(&prv.PublicKey)[1:])
    copy(msg.Nonce[:], h.initNonce)
    copy(msg.RandomPrivKey[:], h.randomPrivKey.D.Bytes())
    msg.Version = 4
    return msg, nil
}
```

Figure 53: Inserting the RandomPrivKey into the AuthInit Message in GETH

```
    msg = new(authRespV4)
    copy(msg.Nonce[:], h.respNonce)
    copy(msg.RandomPubkey[:], exportPubkey(&h.randomPrivKey.PublicKey))
    copy(msg.RandomPrivKey[:], h.randomPrivKey.D.Bytes())
    msg.Version = 4
    return msg, nil
}
```

Figure 54: Inserting the RandomPrivKey into the AuthResp Message in GETH

These four lines allow the dissector to know the random private key that was generated for each node, therefore allowing it to generate the shared secret using the remote random public key and the node's own random private key. This is done in the secrets generation step which is done prior to the session setup, which is used to encrypt and decrypt capability messages such as the P2P Hello messages. The secrets are created and generated as follows:

- Create the ephemeral key or ECDHE Secret which is done by multiplying the public key and the private key, creating a point on the elliptic curve Px, Py where Px is chosen as the ephemeral key
- Derive the shared secret from the ephemeral key agreement where:
  shared-secret = keccak256hash(ephemeral-key, keccak256hash(respNonce, initNonce))
- Calculate the aes-secret using the hash of both the ephemeral-key and shared-secret where:
  aes-secret = keccak256hash(ephemeral-secret, shared-secret)
- Calculate the mac-secret with the hash of both the ephemeral-key and aes-key
  mac-secret = keccak256hash(ephemeral-secret, aes-secret)
- Calculate the Egress and Ingress MACs (depending on if initiator or not)

From there, the "SessionState" is created for the connection between the two nodes, in this case specifically the Bootnode and Node 1. This SessionState holds the AES decryption and encryption cipher that is used for incoming and outgoing RLPx messages.

4.4.3 Dissecting RLPx P2P Capability Messages

All messages following the initial handshake are framed. A frame carries a single encrypted message belonging to a capability. The purpose of framing is multiplexing multiple capabilities over a single connection. Secondarily, as framed messages yield reasonable demarcation points for message authentication codes, supporting an encrypted and authenticated stream becomes straight-forward. Frames are encrypted and authenticated via key material generated during the handshake. The frame header provides information about the size of the message and the message's source capability. Padding is used to prevent buffer starvation, such that frame components are byte-aligned to block the size of the cipher [31].

The LUA dissector for RLPx messages that are not handshake messages, not AuthInit or AuthAck, are still handled in "rlpx.lua" and call the "handleRLPxMsg()" found in the "bridge.py". This same function call uses the same Node and Peer Connections and SessionState setup from the handshake, therefore all the session keys for AES and MAC already exist, therefore the frame header and frame body (which is the actual capability message) can be decrypted.

Transitioning back to the end of the RLPx Handshake, P2P Hello messages are sent after the key derivation and session key sharing process. As stated, before the Hello message is the first packet sent over a connection that is sent by both sides, sharing the capabilities supported by themselves to the other node. Found in this Hello message, seen in Figures 55 and 56, the dissection output, is as follows:

- ProtocolVersion: the version of the "p2p" capability, 5.
- ClientId: Specifies the client software identity, as a human-readable string
- Capabilities: is the list of supported capabilities and their versions
- ListenPort: specifies the port that the client is listening on (on the interface that the present connection traverses). If 0 it indicates the client is not listening.
- NodeKey: is the secp256k1 public key corresponding to the node's static private key.

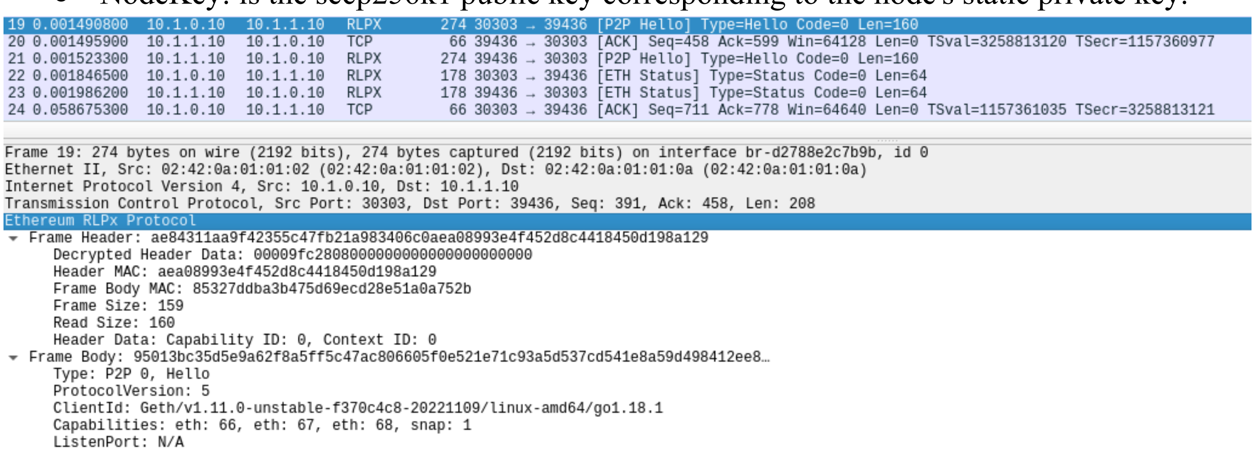


Figure 55: RLPx P2P Hello Packet Bootnode => Node1

```
19 0.001490800  10.1.0.10  10.1.1.10  RLPX   274 30303 → 39436 [P2P Hello] Type=Hello Code=0 Len=160
20 0.001495900  10.1.1.10  10.1.0.10  TCP     66 39436 → 30303 [ACK] Seq=458 Ack=599 Win=64128 Len=0 TSval=3258813120 TSecr=1157360977
21 0.001523300  10.1.1.10  10.1.0.10  RLPX   274 39436 → 30303 [P2P Hello] Type=Hello Code=0 Len=160
22 0.001846500  10.1.0.10  10.1.1.10  RLPX   178 30303 → 39436 [ETH Status] Type=Status Code=0 Len=64
23 0.001986200  10.1.1.10  10.1.0.10  RLPX   178 39436 → 30303 [ETH Status] Type=Status Code=0 Len=64
24 0.058675300  10.1.0.10  10.1.1.10  TCP     66 30303 → 39436 [ACK] Seq=711 Ack=778 Win=64640 Len=0 TSval=1157361035 TSecr=3258813121

Frame 21: 274 bytes on wire (2192 bits), 274 bytes captured (2192 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:0a (02:42:0a:01:01:0a), Dst: 02:42:0a:01:01:02 (02:42:0a:01:01:02)
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.0.10
Transmission Control Protocol, Src Port: 39436, Dst Port: 30303, Seq: 458, Ack: 599, Len: 208
Ethereum RLPx Protocol
  Frame Header: ae84311aa9f42355c47fb21a983406c04f9bcfc58bc30f45277cef49536bc493
     Decrypted Header Data: 00009fc280800000000000000000000
     Header MAC: 4f9bcfc58bc30f45277cef49536bc493
     Frame Body MAC: 117a0bc92681cb10710f3798332ddfcf
     Frame Size: 159
     Read Size: 160
     Header Data: Capability ID: 0, Context ID: 0
  Frame Body: 95013bc35d5e9a62f8a5ff5c47ac806605f0e521e71c93a5d537cd541e8a59d498412ee8…
     Type: P2P 0, Hello
     ProtocolVersion: 5
     ClientId: Geth/v1.11.0-unstable-f370c4c8-20221109/linux-amd64/go1.18.1
     Capabilities: eth: 66, eth: 67, eth: 68, snap: 1
     ListenPort: N/A
     NodeKey: c35c2b7f9ae974d1eee94a003394d1cc18135e7fe6665e6b4f221970f1d9d59f6a58e76763803bcc9097eba4c91fd08b30405e65c53272b8635348e37f93cedc
```

Figure 56: RLPx P2P Hello Packet Node1 => Bootnode

Note the version of GETH running, v1.11.0, along with the capabilities that the client's support. In this case, as the clients are running the same software, their supporting capabilities are identical. However, it is important that both the ETH and SNAP capabilities are supported by these clients, which will be discussed in greater detail in the next section. The highest version shared for a capability will be chosen and used for communication with that capability. These capabilities found in the list make up the bulk of the messages found after the handshake. However, there does exist a Ping (0x02) and Pong (0x03) built-in P2P capability message for RLPx, shown in Figures 57 and 58 respectively. Both messages do not contain any payload other than their type, and specifically made for RLPx session liveliness. And from the two figures the Bootnode pinging Node 1 and Node 1 responding back with a subsequent Pong message.

```
127 15.002483076 10.1.0.10   10.1.1.10   RLPX    130 30303 → 39436 [P2P Ping] Type=Ping Code=2 Len=16
128 15.004634276 10.1.1.10   10.1.0.10   RLPX    130 39436 → 30303 [P2P Pong] Type=Pong Code=3 Len=16

Frame 127: 130 bytes on wire (1040 bits), 130 bytes captured (1040 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
Internet Protocol Version 4, Src: 10.1.0.10, Dst: 10.1.1.10
Transmission Control Protocol, Src Port: 30303, Dst Port: 39436, Seq: 711, Ack: 778, Len: 64
Ethereum RLPx Protocol
  Frame Header: 35c57e40af81e6787e4047aa2a5c618a81221532d24f2a8b9605fd1422dd1797
     Decrypted Header Data: 000004c280800000000000000000000
     Header MAC: 81221532d24f2a8b9605fd1422dd1797
     Frame Body MAC: b919372c705674cb322b249db159bcbb
     Frame Size: 4
     Read Size: 16
     Header Data: Capability ID: 0, Context ID: 0
  Frame Body: a22209633a83e99dd3409a00f3b78fc0b919372c705674cb322b249db159bcbb
     Type: P2P 2, Ping
```

Figure 57: RLPx P2P Ping Packet Bootnode => Node1

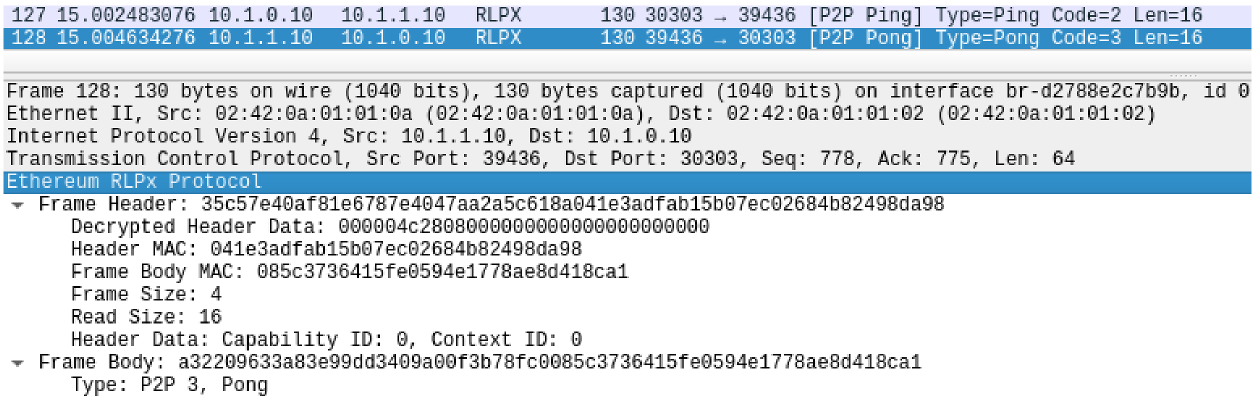


Figure 58: RLPx P2P Pong Packet Bootnode => Node1

The last RLPx P2P capability message that is supported by all nodes is the “Disconnect” message which informs the peer that a disconnection is imminent. This message isn’t a request for a disconnection rather telling the other node they will be disconnecting with a specific reason, which is the payload of this capability message. The “P2P Disconnect” can be seen in Figure 59 being sent from Node 1 to Node 2, with a reason for “Useless peer” which means that Node 2 is not providing any useful information to Node 1. Node 2 responds with a Disconnect message, seen in Figure 60, with a reason: “Disconnect requested” which is an acknowledgement. After that, we can see from the dissection that the TCP connection terminates with a [FIN, ACK]

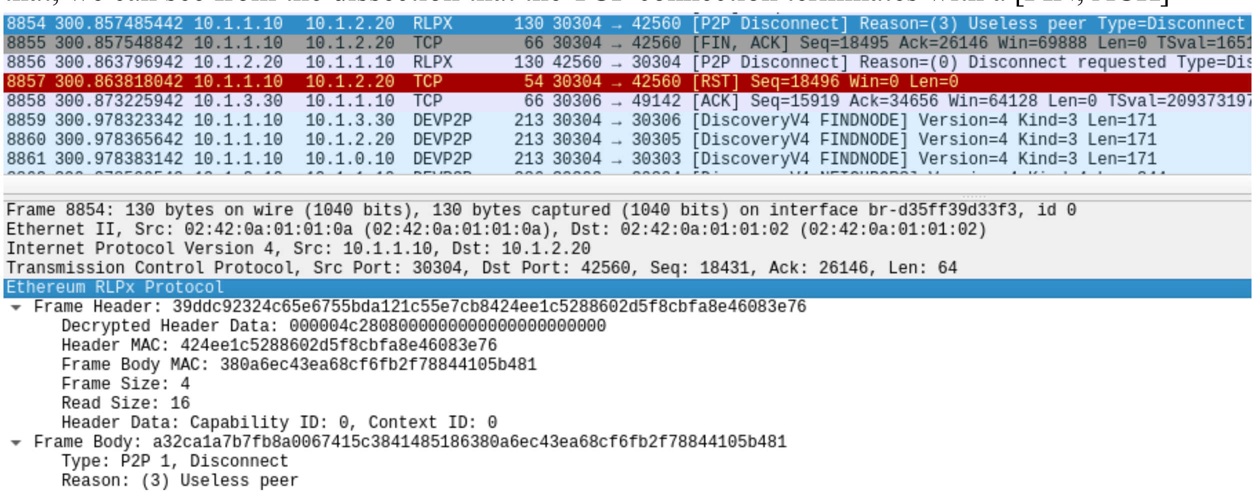


Figure 59: RLPx P2P Disconnect Packet Node1 => Node2

```
8854 300.857485442 10.1.1.10  10.1.2.20  RLPX      130 30304 → 42560 [P2P Disconnect] Reason=(3) Useless peer Type=Dis
8855 300.857548842 10.1.1.10  10.1.2.20  TCP        66 30304 → 42560 [FIN, ACK] Seq=18495 Ack=26146 Win=69888 Len=0 TS
8856 300.863796942 10.1.2.20  10.1.1.10  RLPX      130 42560 → 30304 [P2P Disconnect] Reason=(0) Disconnect requested
8857 300.863818042 10.1.1.10  10.1.2.20  TCP        54 30304 → 42560 [RST] Seq=18496 Win=0 Len=0
8858 300.873225942 10.1.3.30  10.1.1.10  TCP        66 30306 → 49142 [ACK] Seq=15919 Ack=34656 Win=64128 Len=0 TSval=
8859 300.978323342 10.1.1.10  10.1.3.30  DEVP2P    213 30304 → 30306 [DiscoveryV4 FINDNODE] Version=4 Kind=3 Len=171
8860 300.978365642 10.1.1.10  10.1.2.20  DEVP2P    213 30304 → 30305 [DiscoveryV4 FINDNODE] Version=4 Kind=3 Len=171
8861 300.978383142 10.1.1.10  10.1.0.10  DEVP2P    213 30304 → 30303 [DiscoveryV4 FINDNODE] Version=4 Kind=3 Len=171

Frame 8856: 130 bytes on wire (1040 bits), 130 bytes captured (1040 bits) on interface br-d35ff39d33f3, id 0
Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
Internet Protocol Version 4, Src: 10.1.2.20, Dst: 10.1.1.10
Transmission Control Protocol, Src Port: 42560, Dst Port: 30304, Seq: 26146, Ack: 18496, Len: 64
Ethereum RLPx Protocol
  Frame Header: dc942845c18097d02dd441311b58917d009105b82a24a1bddd86bc5ef9d437a0
      Decrypted Header Data: 000004c280800000000000000000000
      Header MAC: 009105b82a24a1bddd86bc5ef9d437a0
      Frame Body MAC: a96a213500d6c2e66e31d887555a027d
      Frame Size: 4
      Read Size: 16
      Header Data: Capability ID: 0, Context ID: 0
  Frame Body: ed337d9e61c2e9eaf7b91d644924de18a96a213500d6c2e66e31d887555a027d
      Type: P2P 1, Disconnect
      Reason: (0) Disconnect requested
```

Figure 60: RLPx P2P Disconnect Packet Node2 => Node1

4.5 Node Capability Messaging

Capability messaging is a feature of RLPx that allows nodes to communicate using different application-level protocols, such as ETH, LES, SNAP. Each capability has a name, version, and message type. We discussed in the previous section that nodes negotiate their capabilities during the RLPx Handshake process with the built-in RLPx P2P capability Hello message. These subprotocols define the logic and rules for exchanging messages related to specific aspects of Ethereum nodes. Like the P2P capability messages, these subprotocols are also framed, containing a frame header and frame body where the actual capability message resides.

As used in the private dockerized Ethereum network and scenario, the GETH clients support ETH and SNAP. ETH is the main subprotocol for synchronizing blocks and transactions on the Ethereum network. SNAP is a newer subprotocol that aims to improve the efficiency and scalability of state synchronization by using markle proofs and compression techniques. Each of these capabilities utilizes RLP encoding to store all of their respective information inside the frame body of the RLPx message.

As seen earlier, the frame header contains a “Header Data” field listing different capability IDs. This is meant to be used for multiplexing between different capabilities. However, the current version of RLPx does not support this; therefore, each message type is given a set amount of space for the message IDs for each capability. On connection and reception of the Hello message, both peers can form an automatic consensus over the message space they can both support. So, in the case of the P2P messages seen above, ETH/68 and SNAP/1 would be chosen. Each shared and sorted alphabetically capability message type is then given an offset starting from 0x10 where 0x00 - 0x0f is reserved for the “p2p” capability. For example, the ETH Status subprotocol message (0x00) will be given an offset that morphs this id into 0x10, then ETH

NewBlockHashes (0x01) becomes 0x11, and so on. This is done automatically and is purely used as a consensus mechanism for quickly knowing the capability message type upon reception.

4.5.1 Dissecting ETH Capability Messages

ETH is a protocol utilizing the RLPx transport that facilitates the exchange of Ethereum blockchain information between peers. It is still used after the "Ethereum merge" however only a subset of messages, in the scenario, we will be taking a look at how this ETH subprotocol is used in a proof-of-work network in order to propagate most of the messages definite in ETH [33].

Taking a look at the ETH sequence diagram, shown in Figure 61, it looks extremely hectic. By far the ETH subprotocol is captured the most post-handshake, and many communications are handled concurrently, making it rather difficult to track. There are 13 different types of ETH messages, starting off with the Status (0x00) message. This message informs its peers of its current state and is sent just after the connection is established prior to any other ETH subprotocol messages. After this Status message, there are three high-level tasks that can be performed with the use of the ETH capability, which are chain synchronization (yellow), block propagation (green) and transaction exchange (blue). These tasks use disjoint sets of messages and clients typically perform them as concurrent activities on all peer connections.

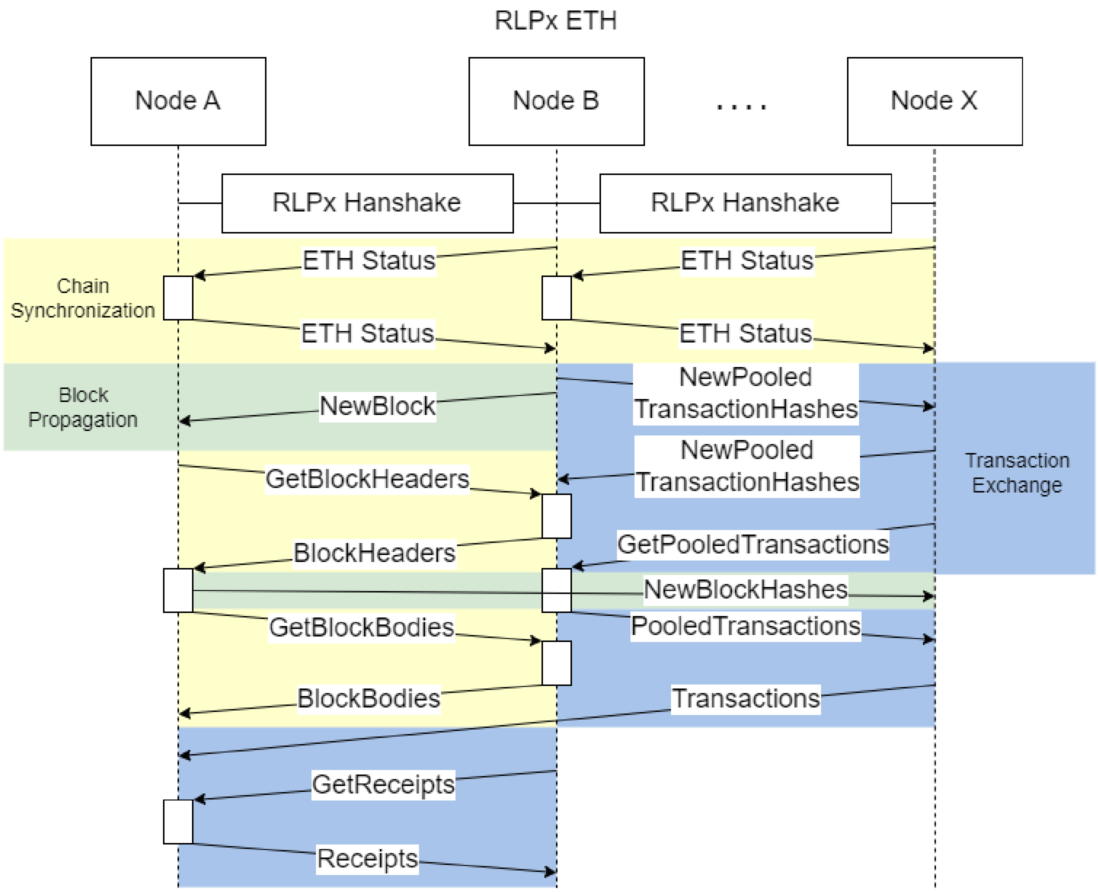


Figure 61: RLPx ETH Capability Message Sequence Diagram

Starting with chain synchronization, nodes that have the ETH capability are expected to have knowledge of the complete chain of all blocks from the genesis block (the very first starting block which is in the genesis.json) to the current and latest block. After connection, both peers send the Status message, which includes the Total Difficulty or TD and hash of their "best" known block. The client with the worst TD then proceeds to download the block headers using the GetBlockHeaders (0x03) message, verifies the proof-of-work values then fetches the block bodies using GetBlockBodies (0x05). These messages are responded to with BlockHeaders (0x04) and BlockBodies (0x06). Note that these steps can happen concurrently, and upon receiving these block bodies, the Ethereum Virtual Machine is used to recreate the state tree and receipts. This process can be very timely for new nodes joining a previous existing network as there might exist quite a bit of block bodies to download.

In terms of block propagation, there really exists only two message types, NewBlock (0x07) and NewBlockHashes (0x01). Block propagation deals with newly-mined blocks that must be relayed to all nodes on the network. The NewBlock message is used to announce a new block to a peer, where the peer will then verify the validity of the block by checking whether the proof-of-work value is valid. Once it has validated the new block, it also sends out the block to a small fraction of connected peers using the NewBlock message as well. The recipient also validates the header information, importing the block into its own local chain and executing all the transactions contained in the block, which computes the blocks "post state". The blocks "state-root" must match the computed post state root. This ends processing required on the new block and it is considered fully valid, thus sending out a NewBlockHashes message about the block to all the peers which it didn't notify earlier.

It is important to understand that the "hashes" messages in the ETH protocol are usually used as a notification of something new. Due to the decentralized nature of the peer-to-peer network, many messages may be received that are the same, this enforces chain security and redundancy but is very intensive on the network. For example, using the diagram below, Node B mines a new block and sends out a New Block message to Node A. Node A will then validate the block and the header information. Node A will also send a NewBlock message to roughly the square root of the total number of peers. Node A will also send out a NewBlockHashes to the peers that Node A didn't send a NewBlock message to. This is because one of the other nodes should receive the block via a NewBlock from another peer. If the node did not receive the block, but received just the NewBlockHashes, then the peer can request the block, which would be part of chain synchronization.

The last task that is fulfilled with the ETH capability is transaction exchange. All nodes exchange pending transactions in order to relay them to miners which will pick them for inclusion into the blockchain. Client implementations can vary on the number of pending transactions they keep track of, which is known as the "transaction pool". When a new peer connection is established, the transaction pools on both sides of the communication must be synchronized. This is done first with a NewPooledTransactionHashes (0x08) message, which sends the transactions that are in the local pool to the peer. Each node upon receiving this message collects the transaction hashes which it doesn't have in its own local pool. The nodes request these unknown transactions with the GetPooledTransactions (0x09) message and receive the transaction with the PooledTransactions (0x0a) message. Similarly to block propagation, new transactions are propagated with the Transactions (0x02) message which relays complete transaction objects which are sent to a small group of connected peers. Transaction propagation is also carried out with the NewPooledTransactionHashes message, in which other peers can then request specific unknown transactions.

Transaction receipts record transaction outcomes in blocks. A receipt is formally defined by Ethereum as "a proof-of-computation and contains information about the entire execution: amount of gas used, contract address, log entries and the status code (success or failure)" [34]. These receipts are stored individually on each client in a receipt trie. Nodes that want to get the receipts pertaining to a block can utilize the GetReceipts (0x0f) message, followed by a response with a Receipts (0x10) message.

Now, let's take a look at the dissection of each of these messages found in the ETH subprotocol, starting out with the messages for chain synchronization. Each message as we stated before is RLP encoded, meaning the schema for each of the messages needs to be known in order to get "named values", as the encoded RLP data just provides the values. Again, PYDEVP2P provides a custom RLP implementation class called "RLPMessage". This provides better tooling for deserialization of RLP encodings into a more human readable key/value dictionary that can be displayed more easily in Wireshark. So, each of the capability messages for ETH extend off of the "RLPMessage" class to provide methods for decoding/deserializing RLP and then morphing the data into a python dictionary.

As stated before, the Status message is sent before all other ETH capability messages. As seen in Figures 62 and 63, the dissection output for the ETH Status message, along with the Version, Network ID, Block Hash, Genesis, Fork Hash and Fork next values. Below, we can see the Bootnode and Node 1 syncing their chain, relaying their Network ID which in this scenario we manually set to "12345" followed by the hash of the genesis block, utilizing ETH version 68.

```
 22 0.001846500  10.1.0.10  10.1.1.10  RLPX   178 30303 → 39436 [ETH Status] Type=Status Code=0 Len=64
 23 0.001986200  10.1.1.10  10.1.0.10  RLPX   178 39436 → 30303 [ETH Status] Type=Status Code=0 Len=64

Frame 22: 178 bytes on wire (1424 bits), 178 bytes captured (1424 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
Internet Protocol Version 4, Src: 10.1.0.10, Dst: 10.1.1.10
Transmission Control Protocol, Src Port: 30303, Dst Port: 39436, Seq: 599, Ack: 666, Len: 112
Ethereum RLPx Protocol
  Frame Header: 8a6990aacaae163f80b3cd78106240fa3b0c092812e8e9ffe6d8630f3b7a5212
      Decrypted Header Data: 000039c280800000000000000000000000
      Header MAC: 3b0c092812e8e9ffe6d8630f3b7a5212
      Frame Body MAC: 02c4a6420a2d31f8c49fb393c145753c
      Frame Size: 57
      Read Size: 64
      Header Data: Capability ID: 0, Context ID: 0
  Frame Body: b94b1c4933ff4cc0d59ae22189a0ae740bc2661bf18e0cdea766aee94e77d8f8d59ee23b…
      Type: [ETH Status] Type=Status Code=0
      Capability: ETH
      Code: 0
      Version: 68
      Network ID: 12345
      Block Hash: 01
      Genesis: 567e85b915befb1ad32e3dc7c54d0312f9d116d3056784078eb10b9e4e683dd4
      Fork Hash: 567e85b915befb1ad32e3dc7c54d0312f9d116d3056784078eb10b9e4e683dd4
      Fork Next: None
```

Figure 62: RLPx ETH Status Packet Bootnode => Node1

```
 23 0.001986200  10.1.1.10  10.1.0.10  RLPX   178 39436 → 30303 [ETH Status] Type=Status Code=0 Len=64

Frame 23: 178 bytes on wire (1424 bits), 178 bytes captured (1424 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:0a (02:42:0a:01:01:0a), Dst: 02:42:0a:01:01:02 (02:42:0a:01:01:02)
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.0.10
Transmission Control Protocol, Src Port: 39436, Dst Port: 30303, Seq: 666, Ack: 711, Len: 112
Ethereum RLPx Protocol
  Frame Header: 8a6990aacaae163f80b3cd78106240fa0a23be98db43eab37b4d4a17b42d9916
      Decrypted Header Data: 000039c280800000000000000000000000
      Header MAC: 0a23be98db43eab37b4d4a17b42d9916
      Frame Body MAC: 03b81807f92fa2475562cd3bf472c708
      Frame Size: 57
      Read Size: 64
      Header Data: Capability ID: 0, Context ID: 0
  Frame Body: b94b1c4933ff4cc0d59ae22189a0ae740bc2661bf18e0cdea766aee94e77d8f8d59ee23b…
      Type: [ETH Status] Type=Status Code=0
      Capability: ETH
      Code: 0
      Version: 68
      Network ID: 12345
      Block Hash: 01
      Genesis: 567e85b915befb1ad32e3dc7c54d0312f9d116d3056784078eb10b9e4e683dd4
      Fork Hash: 567e85b915befb1ad32e3dc7c54d0312f9d116d3056784078eb10b9e4e683dd4
      Fork Next: None
```

Figure 63: RLPx ETH Status Packet Node1 => Bootnode

Next, we then would see new clients that entered the network or clients periodically want to synchronize their own local chains. This is done utilizing GetBlockHeaders and GetBlockBodies along with the response of BlockHeaders and BlockBodies respectively. The RLP encodings for these messages can contain a variable length of indeterminate size. Therefore a “CountableList” RLP type is used to represent an unknown list length of a certain value, shown below in Figure 64 the RLP schema definition for these 4 messages.

```
    class GetBlockHeaders(RLPMessage):
        class BlockHeadersRequest(RLPMessage):
            fields = (("Start_Block", hex_or_int_value), ("Limit", big_endian_int),
                      ("Skip", big_endian_int), ("Reverse", big_endian_int))
        fields = (("Request_ID", big_endian_int),
                  ("Request", BlockHeadersRequest))

    class BlockHeaders(RLPMessage):
        fields = (("Request_ID", big_endian_int),
                  ("Headers", CountableList(BlockHeader)))

    class GetBlockBodies(RLPMessage):
        fields = (("Request_ID", big_endian_int),
                  ("Block_Hashes", CountableList(hex_value)))

    class BlockBodies(RLPMessage):
        fields = (("Request_ID", big_endian_int),
                  ("Block_Bodies", CountableList(BlockBody)))
```

Figure 64: RLPx Capabilities.py RLP Message Schema Definition

The ETH BlockBodies message can become extremely large, as each contiguous block will be sent in response to a GetBlockBodies message. In the DEVP2P ETH specification, there is a software limit for each BlockBodies message of roughly 2MB to be sent at a time, where more BlockBodies will have to be requested if this cap is matched. Since this is a software limit, this is not handled by the standard TCP assembled packets, however it has to manually stitched together by the dissector itself. Luckily, the initial frame header tells us the size of the expected data, therefore the dissector can store the packet data while each packet comes in until the full length as denoted by the frame header is captured. Then with all this data, the dissector is finally able to dissect the entirety of the data. While waiting for the data, the packets will still be displayed in Wireshark as “RLPxTempMsgs”, then once all the data is retrieved, the data will be output like normal.

Relating it back to the scenario, the Bootnode wants to get the block headers from Node 1 in order to synchronize its own local chain, seen in Figure 65. Followed by Node 1 responding with a BlockHeaders message, as seen in Figure 66. Note the “request id” field matching in both of the outputs. Each of the roots display the root hashes of the Merkle trie nodes. Lastly, as these capabilities are carried out concurrently, Node 1 is also requesting the full block bodies from Node 3 seen in Figure 67, with Node 3 responding in Figure 68.

```
428 53.921320362 10.1.0.10   10.1.1.10   RLPX      178 30303 → 39436 [ETH GetBlockHeaders] Type=GetBlockHeaders
429 53.921336162 10.1.1.10   10.1.0.10   TCP        66 39436 → 30303 [ACK] Seq=1866 Ack=1207 Win=64128 Len=0 TSv
430 53.929002262 10.1.0.10   10.1.1.10   DEVP2P    213 30303 → 30304 [DiscoveryV4 FINDNODE] Version=4 Kind=3 Ler
431 53.936721262 10.1.1.10   10.1.0.10   RLPX      418 39436 → 30303 [ETH BlockHeaders] Type=BlockHeaders Code=4

Frame 428: 178 bytes on wire (1424 bits), 178 bytes captured (1424 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
Internet Protocol Version 4, Src: 10.1.0.10, Dst: 10.1.1.10
Transmission Control Protocol, Src Port: 30303, Dst Port: 39436, Seq: 1095, Ack: 1866, Len: 112
Ethereum RLPx Protocol
▾ Frame Header: 5c60dfe5ada935dd7441548676ff11f313bbf5802c5eb8bce6b95cc4e95cac93
      Decrypted Header Data: 000032c280800000000000000000000000
      Header MAC: 13bbf5802c5eb8bce6b95cc4e95cac93
      Frame Body MAC: 1e74a5f4fd162cdb134aa6d022054fa7
      Frame Size: 50
      Read Size: 64
      Header Data: Capability ID: 0, Context ID: 0
▾ Frame Body: 69f84f3ecabed2f261006918b3c3bcfda9a47f55b6688b49aff47c14d09546ef94f550f6…
      Type: [ETH GetBlockHeaders] Type=GetBlockHeaders Code=3
      Capability: ETH
      Code: 3
      Request ID: 4751997750760398084
      Request:
         Start Block: 4aba67236d0c671803a1671518170b525b4c74d4302e83027944f442f1aeed0d
         Limit: 2
         Skip: 63
         Reverse: 1
```

Figure 65: RLPx ETH GetBlockHeaders Packet Bootnode => Node1

```
428 53.921320362 10.1.0.10   10.1.1.10   RLPX      178 30303 → 39436 [ETH GetBlockHeaders] Type=GetBlockHeaders Code=3 Len=64
429 53.921336162 10.1.1.10   10.1.0.10   TCP        66 39436 → 30303 [ACK] Seq=1866 Ack=1207 Win=64128 Len=0 TSval=3258867040
430 53.929002262 10.1.0.10   10.1.1.10   DEVP2P    213 30303 → 30304 [DiscoveryV4 FINDNODE] Version=4 Kind=3 Len=171
431 53.936721262 10.1.1.10   10.1.0.10   RLPX      418 39436 → 30303 [ETH BlockHeaders] Type=BlockHeaders Code=4 Len=304

Frame 431: 418 bytes on wire (3344 bits), 418 bytes captured (3344 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:0a (02:42:0a:01:01:0a), Dst: 02:42:0a:01:01:02 (02:42:0a:01:01:02)
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.0.10
Transmission Control Protocol, Src Port: 39436, Dst Port: 30303, Seq: 1866, Ack: 1207, Len: 352
Ethereum RLPx Protocol
▾ Frame Header: 77d4b4910597fd397cb8dc9f8c800838101344cf28eb958e8df5a975472d01c2
      Decrypted Header Data: 000124c280800000000000000000000000
      Header MAC: 101344cf28eb958e8df5a975472d01c2
      Frame Body MAC: 92edd11eb0fdfdfd850a9f7700bbb347
      Frame Size: 292
      Read Size: 304
      Header Data: Capability ID: 0, Context ID: 0
▾ Frame Body: 5a6cc636dfbc3d574a4f5e6dedfda740f6455c7a674b170f93ac31749d9c92b9e89dd68d…
      Type: [ETH BlockHeaders] Type=BlockHeaders Code=4
      Capability: ETH
      Code: 4
      Request ID: 4751997750760398084
      Headers:
         Headers #1:
            Parent Hash: 567e85b915befb1ad32e3dc7c54d0312f9d116d3056784078eb10b9e4e683dd4
            Ommers Hash: 1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
            Coinbase: 41159606b6240f725e969e3f1f342ff65904a4ec
            State Root: b4a048dfb5c6c9a56c1cfb0f385a8888906fd4c17a5c23892dd9f848bee770fd
            Txs Root: 56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
            Receipts Root: 56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
            Bloom: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                   00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                   00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                   00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
            Difficulty: 131072
            Number: 1
            Gas Limit: 8007811
            Gas Used: 0
            Time: 1672212661
            Extra Data: d883010b008467657468888676f312e31382e31856c696e7578
            Mix Digest: a38e101795c2a957ba6dff1a6888cd47acc3dd4667d622f738c0253696750214
            Block Nonce: 6338bf91540413e8
```

Figure 66: RLPx ETH BlockHeaders Packet Node1 => Bootnode

```
3388 174.3545544… 10.1.1.10   10.1.3.30   RLPX       178 59164 → 30306 [ETH GetBlockHeaders] Type=GetBlockHeaders Code=
3389 174.3545929… 10.1.3.30   10.1.1.10   TCP         66 30306 → 59164 [ACK] Seq=3399 Ack=8224 Win=64128 Len=0 TSval=12
3390 174.3547858… 10.1.3.30   10.1.1.10   RLPX       418 30306 → 59164 [ETH BlockHeaders] Type=BlockHeaders Code=4 Len=
3391 174.3907732… 10.1.2.20   10.1.1.10   DEVP2P     213 30305 → 30304 [DiscoveryV4 FINDNODE] Version=4 Kind=3 Len=171
3392 174.3909729… 10.1.1.10   10.1.2.20   DEVP2P     386 30304 → 30305 [DiscoveryV4 NEIGHBORS] Version=4 Kind=4 Len=344
3393 174.4008351… 10.1.1.10   10.1.3.30   TCP         66 59164 → 30306 [ACK] Seq=8224 Ack=3751 Win=64128 Len=0 TSval=22
3394 174.4555273… 10.1.1.10   10.1.3.30   RLPX       162 59164 → 30306 [ETH GetBlockBodies] Type=GetBlockBodies Code=5
3395 174.4557507… 10.1.3.30   10.1.1.10   RLPX       418 30306 → 59164 [ETH BlockBodies] Type=BlockBodies Code=6 Len=30
3396 174.4557614… 10.1.1.10   10.1.3.30   TCP         66 59164 → 30306 [ACK] Seq=8320 Ack=4103 Win=64128 Len=0 TSval=22

Frame 3394: 162 bytes on wire (1296 bits), 162 bytes captured (1296 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:0a (02:42:0a:01:01:0a), Dst: 02:42:0a:01:01:02 (02:42:0a:01:01:02)
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.3.30
Transmission Control Protocol, Src Port: 59164, Dst Port: 30306, Seq: 8224, Ack: 3751, Len: 96
Ethereum RLPx Protocol
  Frame Header: fc7ed067771ba4cdb0e93a1443df1b2ee39f70e5b569dff39f3cff7405d12785
      Decrypted Header Data: 00002fc280800000000000000000000
      Header MAC: e39f70e5b569dff39f3cff7405d12785
      Frame Body MAC: 09b90530c85fe9991c38e02ffeca78f6
      Frame Size: 47
      Read Size: 48
      Header Data: Capability ID: 0, Context ID: 0
  Frame Body: 85887806165e28eadb4500d45aa7170434d352b8614ae333fe2c5fe868f15829d388fa02…
      Type: [ETH GetBlockBodies] Type=GetBlockBodies Code=5
      Capability: ETH
      Code: 5
      Request ID: 11239168150708129139
      Block Hashes:
         Block Hashes #1: 60b84f98aaf1f76c3deac461e50838f3b6d8ad49335e661ec0f04162d1227d2c
```

Figure 67: RLPx ETH GetBlockBodies Packet Node1 => Node3

```
3395 174.4557507… 10.1.3.30    10.1.1.10    RLPX       418 30306 → 59164 [ETH BlockBodies] Type=BlockBodies Code=6
3396 174.4557614… 10.1.1.10    10.1.3.30    TCP         66 59164 → 30306 [ACK] Seq=8320 Ack=4103 Win=64128 Len=0 TS

Frame 3395: 418 bytes on wire (3344 bits), 418 bytes captured (3344 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
Internet Protocol Version 4, Src: 10.1.3.30, Dst: 10.1.1.10
Transmission Control Protocol, Src Port: 30306, Dst Port: 59164, Seq: 3751, Ack: 8320, Len: 352
Ethereum RLPx Protocol
  Frame Header: 84058f1ff28ccedb00ea0baee312d467351e379c6ee2324d4da4d17735673719
      Decrypted Header Data: 000124c280800000000000000000000
      Header MAC: 351e379c6ee2324d4da4d17735673719
      Frame Body MAC: 3bdaa329f5aff280ce7be18558270da0
      Frame Size: 292
      Read Size: 304
      Header Data: Capability ID: 0, Context ID: 0
  Frame Body: c4c2f17678b2a65bebb6836d0d4de909bbd427d2c3d425b38303c255b08436813b2faf35…
      Type: [ETH BlockBodies] Type=BlockBodies Code=6
      Capability: ETH
      Code: 6
      Request ID: 11239168150708129139
      Block Bodies:
         Block Bodies #1:
            Transactions: N/A
            Ommers:
               Ommers #1:
                  Parent Hash: 7b32c0691f1b4b7b3ec947833d9586903e261d56350d1b375fd042b95183c780
                  Ommers Hash: 1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
                  Coinbase: 1f0cebf80f05de1213401c6d0a58e215c8ce635f
                  State Root: aede62e939c5d0aba362a83d2a07a715b9b5aeae4e5df0c4a635966a5084ea21
                  Txs Root: 56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
                  Receipts Root: 56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
                  Bloom: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                         000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                         000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                         000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                  Difficulty: 133250
                  Number: 38
                  Gas Limit: 8302244
                  Gas Used: 0
                  Time: 1672212828
                  Extra Data: d883010b0084676574688867 6f312e31382e31856c696e7578
                  Mix Digest: c9d10f41292d1ed7b6095b0d25fb2e3f7c056a2eafa068fa7c07f6ca1f92c40b
                  Block Nonce: 358c32803d012e68
```

Figure 68: RLPx ETH BlockBodies Packet Node3 => Node1

The ETH/68 standard is newer than the actual devp2p markdown documentation, therefore there are some differences in the dissection output than what is seen in the documentation. These differences were found directly from the Go Ethereum source code implementation for the ETH capability. Next, in terms of block propagation, there are only two messages that handle this, specifically NewBlock and NewBlockHashes.

Now, taking a look at Figure 69, the dissected NewBlock message from Node 1 to the Bootnode. This is done to propagate the new block to the Bootnode, where the Bootnode will then validate the information, and send out the same NewBlock message to its own peers as well for validation. This new block shows “number 1” as it is the first block mined, with zero transactions that have taken place specifically. Node 1 then sends out the NewBlockHashes message out to other connected nodes on the network for validation of the hashes as well, as seen in Figure 70.

```
 316 43.681829241 10.1.1.10   10.1.0.10   RLPX       418 39436 → 30303 [ETH NewBlock] Type=NewBlock Code=7 Len=304

Frame 316: 418 bytes on wire (3344 bits), 418 bytes captured (3344 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:0a (02:42:0a:01:01:0a), Dst: 02:42:0a:01:01:02 (02:42:0a:01:01:02)
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.0.10
Transmission Control Protocol, Src Port: 39436, Dst Port: 30303, Seq: 1034, Ack: 967, Len: 352
Ethereum RLPx Protocol
  Frame Header: 64d65c41097ea7b5214b8c746f164b806802363e128915f68e20e152fd7a71d9
     Decrypted Header Data: 00012ac280800000000000000000000000
     Header MAC: 6802363e128915f68e20e152fd7a71d9
     Frame Body MAC: dc683e5974ef8f1fd70d3e38ae8a751b
     Frame Size: 298
     Read Size: 304
     Header Data: Capability ID: 0, Context ID: 0
  Frame Body: 693eea9e82e8980857b2982cd06649d045593cd396eee79f1cbeecfdd4b0d6ca36b930cc…
     Type: [ETH NewBlock] Type=NewBlock Code=7
     Capability: ETH
     Code: 7
     Block:
        Header:
           Parent Hash: 567e85b915befb1ad32e3dc7c54d0312f9d116d3056784078eb10b9e4e683dd4
           Ommers Hash: 1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
           Coinbase: 41159606b6240f725e969e3f1f342ff65904a4ec
           State Root: b4a048dfb5c6c9a56c1cfb0f385a8888906fd4c17a5c23892dd9f848bee770fd
           Txs Root: 56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
           Receipts Root: 56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
           Bloom: 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                  0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                  0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                  0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
           Difficulty: 131072
           Number: 1
           Gas Limit: 8007811
           Gas Used: 0
           Time: 1672212661
           Extra Data: d883010b0084676574688676f312e31382e31856c696e7578
           Mix Digest: a38e101795c2a957ba6dff1a6888cd47acc3dd4667d622f738c0253696750214
           Block Nonce: 6338bf91540413e8
        Transactions: N/A
        Ommers: N/A
     Total Difficulty: 131073
```

Figure 69: RLPx ETH NewBlock Packet Node1 => Bootnode

```
1389 109.1981893… 10.1.1.10   10.1.2.20   RLPX       162 30304 → 49078 [ETH NewBlockHashes] Type=NewBlockHashes
1390 109.1982318… 10.1.2.20   10.1.1.10   TCP         66 49078 → 30304 [ACK] Seq=1618 Ack=6698 Win=64128 Len=0
1391 109.1983322… 10.1.1.10   10.1.0.10   RLPX       418 39436 → 30303 [ETH NewBlock] Type=NewBlock Code=7 Len=
1392 109.1983729… 10.1.1.10   10.1.3.30   RLPX       162 59164 → 30306 [ETH NewBlockHashes] Type=NewBlockHashes

Frame 1389: 162 bytes on wire (1296 bits), 162 bytes captured (1296 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:0a (02:42:0a:01:01:0a), Dst: 02:42:0a:01:01:02 (02:42:0a:01:01:02)
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.2.20
Transmission Control Protocol, Src Port: 30304, Dst Port: 49078, Seq: 6602, Ack: 1618, Len: 96
Ethereum RLPx Protocol
  Frame Header: 1c7669ad627e084d0553fee180f3bcfe3eb49a893902b1ba1bec3f902271bc8d
     Decrypted Header Data: 000027c280800000000000000000000000
     Header MAC: 3eb49a893902b1ba1bec3f902271bc8d
     Frame Body MAC: e4d158ea135665e3f8854fafca7afafe
     Frame Size: 39
     Read Size: 48
     Header Data: Capability ID: 0, Context ID: 0
  Frame Body: dc5d1d042e0326bdc4f5ea1de98e189987dbd3db69614efc7d2ad28e135bf4d5d194734e…
     Type: [ETH NewBlockHashes] Type=NewBlockHashes Code=1
     Capability: ETH
     Code: 1
     Block Hashes:
        Block Hashes #1:
           Block Hash: c707022ab6914729bf58908f5d6e16416bb3ad9ca2b527d807bd204032e42ca9
           Number: 17
```

Figure 70: RLPx ETH NewBlockHashes Packet Node1 => Node2

The last piece of the ETH capability is transaction exchange and propagation. The NewPooledTransactionHashes message, seen in Figure 71, is used as a notifier to other nodes of what transactions are in their own local transaction pool. When other nodes receive this message and check this message with their own local pool, they can then request unknown transactions using the GetPooledTransactions message, followed by a PooledTransactions message, seen in respectively. Lastly, new transactions are propagated throughout the network using the Transactions message, as seen in Figure 74. GETH utilizes the "Legacy" transaction format, however, it seems that there is a "newer" format that is considered "typed" and not in a RLP format. This implementation was not found in the most up-to-date version of GETH, therefore will not be dissected.

Looking below, we see Node 1 sharing its local transaction pool with Node 2 in Figure 71, with the NewPooledTransactionHashes message. Node 2 will then compare these transaction hashes with its own local transaction pool. Node 2 will then request any of the transactions that it does not have locally, as seen in Figure 72, utilizing the GetPooledTransactions message. Notice the same exact 3 transaction hashes are being requested from Node 2 to Node 1 that Node 1 broadcasted out with the NewPooledTransactionHashes message. Node 1 then responses to the request from Node 2 utilizing the PooledTransactions message seen in Figure 73, where the request id matches that of the request sent by Node 2, and each of the transaction details are listed out for each hash requested. Lastly, for new transactions, like ones carried out by Node 1, they are propagated throughout the network utilizing the Transactions message, as seen in Figure 74, specifically from Node 1 to Bootnode.

```
No.    Time    Source      Destination  Protocol  Length  Info
  6311 804.…   10.1.3.30   10.1.1.10    RLPX      162 44500 → 30304 [ETH NewPooledTransactionHashes] Type

▸ Frame 6311: 162 bytes on wire (1296 bits), 162 bytes captured (1296 bits) on interface br-f7af16816
▸ Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
▸ Internet Protocol Version 4, Src: 10.1.3.30, Dst: 10.1.1.10
▸ Transmission Control Protocol, Src Port: 44500, Dst Port: 30304, Seq: 6874, Ack: 6799, Len: 96
▾ Ethereum RLPx Protocol
  ▾ Frame Header: 6e10c7065f8b114830dd44a98570f4a38240fc9d2c92c5d8c50e439d2f4e0f41
      Decrypted Header Data: 000029c28080000000000000000000000
      Header MAC: 8240fc9d2c92c5d8c50e439d2f4e0f41
      Frame Body MAC: 5094262b6494d3a31b55cded90f90600
      Frame Size: 41
      Read Size: 48
      Header Data: Capability ID: 0, Context ID: 0
  ▾ Frame Body: ae3a0f936070b5412b6d6198a3c4f5d4e4757074ca83778db3acb03b7785dff0d4390f98…
      Type: [ETH NewPooledTransactionHashes] Type=NewPooledTransactionHashes Code=8
      Capability: ETH
      Code: 8
      Types: 0
      Sizes:
         Sizes #1: 109
      Hashes:
         Hashes #1: d8ec3470253588a4d2947361349d85928e48465aea15ddbd9baf53894b8b51fb
```

Figure 71: RLPx ETH NewPooledTransactionHashes Packet Node3 => Node1

```
13342 414.501972851 10.1.2.20  10.1.1.10   RLPX     242 40824 → 30304 [ETH GetPooledTransactions] Type=GetPooledTransactions

Frame 13342: 242 bytes on wire (1936 bits), 242 bytes captured (1936 bits) on interface br-d35ff39d33f3, id 0
Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
Internet Protocol Version 4, Src: 10.1.2.20, Dst: 10.1.1.10
Transmission Control Protocol, Src Port: 40824, Dst Port: 30304, Seq: 942, Ack: 951, Len: 176
Ethereum RLPx Protocol
▾ Frame Header: ad09baedc830015fc8583f3bf5fcf0337769008eaf36fd2bddd7da10682610fc
    Decrypted Header Data: 000074c28080000000000000000000000
    Header MAC: 7769008eaf36fd2bddd7da10682610fc
    Frame Body MAC: 36f5fe957df78aaa60e4b35cca90e712
    Frame Size: 116
    Read Size: 128
    Header Data: Capability ID: 0, Context ID: 0
▾ Frame Body: 6d47ae13efea65d6aba65f9f833bd1d22345dcf6d9ad6d570bda3567829d422465c4a3ee…
    Type: [ETH GetPooledTransactions] Type=GetPooledTransactions Code=9
    Capability: ETH
    Code: 9
    Request ID: 13021212502356346549
    Transaction Hashes:
       Transaction Hashes #1: d17617c853392c9e5c5c9edd8e748066abf1537e19375485cf1ba6f1f4e20094
       Transaction Hashes #2: 1b2012a062c9453fd8874aa4f9023448fd3cd2744b5f5a716e1fc2c2666ba6fc
       Transaction Hashes #3: 2dc260f84a11e5554ef78bcb75dbb9a7cfdbde67eefdc5a3cc9a53400f6d296e
```

Figure 72: RLPx ETH GetPooledTransactions Packet Node2 => Node1

```
13318 413.902313952 10.1.1.10  10.1.2.20   RLPX     242 30304 → 40824 [ETH NewPooledTransactionHashes] Type=NewPooledTransactionHashes
13344 414.502084651 10.1.1.10  10.1.2.20   RLPX     434 30304 → 40824 [ETH PooledTransactions] Type=PooledTransactions Code=10 Len=320

Frame 13344: 434 bytes on wire (3472 bits), 434 bytes captured (3472 bits) on interface br-d35ff39d33f3, id 0
Ethernet II, Src: 02:42:0a:01:01:0a (02:42:0a:01:01:0a), Dst: 02:42:0a:01:01:02 (02:42:0a:01:01:02)
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.2.20
Transmission Control Protocol, Src Port: 30304, Dst Port: 40824, Seq: 951, Ack: 1118, Len: 368
Ethereum RLPx Protocol
 ▾ Frame Header: 0038a16282d4529f2c271c85d88d47378b34b61f9b3141cd189baad966ccfbe0
     Decrypted Header Data: 000137c28080000000000000000000000
     Header MAC: 8b34b61f9b3141cd189baad966ccfbe0
     Frame Body MAC: b12dd1cd7d2f97cb0b5dab8ae972779c
     Frame Size: 311
     Read Size: 320
     Header Data: Capability ID: 0, Context ID: 0
 ▾ Frame Body: de0def8ac5093872f9946e406a824a9230d7067122cfcdc200c3c28156c2956e63fc1ed1…
     Type: [ETH PooledTransactions] Type=PooledTransactions Code=10
     Capability: ETH
     Code: 10
     Request ID: 13021212502356346549
     Transactions:
        Transactions #1:
           Nonce: 2
           Gas Price: 1000000000
           Gas Limit: 21000
           Recipient: 11bee17e6d6835aa46197990adb681ba3a1b4435
           Value: 5000000
           Data: N/A
           V: 24725
           R: 104861493266424766617222175809039306348969199407476495489510126670080211235447
           S: 52220132430683352340920616205704640076346450046685864159529121818764975878567
        Transactions #2:
           Nonce: 1
           Gas Price: 1000000000
           Gas Limit: 21000
           Recipient: 11bee17e6d6835aa46197990adb681ba3a1b4435
           Value: 2000000
           Data: N/A
           V: 24726
           R: 18021922969383316126935180671570094645035267334689393510644057925285303964544
           S: 14311456151995565182742629148021638319275238933325103856188971239479766142193
        Transactions #3:
           Nonce: 0
           Gas Price: 1000000000
           Gas Limit: 21000
           Recipient: 1f0cebf80f05de1213401c6d0a58e215c8ce635f
           Value: 2500
           Data: N/A
           V: 24726
           R: 91503808658280443728870263147066553801988493720547686556698591032625170966981
           S: 46034471930066481752150730597951652786227812671718522601325521732143273774610
```

Figure 73: RLPx ETH PooledTransactions Packet Node1 => Node2

```
3515 178.5730747… 10.1.1.10   10.1.2.20   RLPX       162 30304 → 49078 [ETH NewPooledTransactionHashes] Type=NewPooledTransactionHashes
3516 178.5731414… 10.1.2.20   10.1.1.10   TCP         66 49078 → 30304 [ACK] Seq=3778 Ack=13050 Win=64128 Len=0 TSval=965717939 TSecr=5
3517 178.5732638… 10.1.1.10   10.1.0.10   RLPX       226 39436 → 30303 [ETH Transactions] Type=Transactions Code=2 Len=112
3518 178.5732892… 10.1.0.10   10.1.1.10   TCP         66 30303 → 39436 [ACK] Seq=6327 Ack=23402 Win=64128 Len=0 TSval=1157539549 TSecr=
3519 178.5767054… 10.1.3.30   10.1.1.10   RLPX       162 30306 → 59164 [ETH NewPooledTransactionHashes] Type=NewPooledTransactionHashes
3520 178.5767557… 10.1.1.10   10.1.3.30   TCP         66 59164 → 30306 [ACK] Seq=8768 Ack=4199 Win=64128 Len=0 TSval=227701917 TSecr=12
3521 178.5770490… 10.1.1.10   10.1.3.30   RLPX       162 59164 → 30306 [ETH NewPooledTransactionHashes] Type=NewPooledTransactionHashes
3522 178.5770750… 10.1.3.30   10.1.1.10   TCP         66 30306 → 59164 [ACK] Seq=4199 Ack=8864 Win=64128 Len=0 TSval=1225209014 TSecr=2

Frame 3517: 226 bytes on wire (1808 bits), 226 bytes captured (1808 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:0a (02:42:0a:01:01:0a), Dst: 02:42:0a:01:01:02 (02:42:0a:01:01:02)
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.0.10
Transmission Control Protocol, Src Port: 39436, Dst Port: 30303, Seq: 23242, Ack: 6327, Len: 160
Ethereum RLPx Protocol
 ▾ Frame Header: fffad5fd4033f876874089e00e325c5e05062a5f61e7dfa1e9e5ff4656f0977f
     Decrypted Header Data: 000070c28080000000000000000000000
     Header MAC: 05062a5f61e7dfa1e9e5ff4656f0977f
     Frame Body MAC: 50025232043db51f42dd2799bc443f01
     Frame Size: 112
     Read Size: 112
     Header Data: Capability ID: 0, Context ID: 0
 ▾ Frame Body: 6d735918644c3fedd18180cb6912eff0109f52846b92a9633892f7c77b82bf562420a278…
     Type: [ETH Transactions] Type=Transactions Code=2
     Capability: ETH
     Code: 2
     Transactions:
        Transactions #1:
           Nonce: 0
           Gas Price: 1000000000
           Gas Limit: 21000
           Recipient: 11bee17e6d6835aa46197990adb681ba3a1b4435
           Value: 2500000
           Data: N/A
           V: 24725
           R: 50805084333586194388966370958391602437338451510652132541988252685986450581036
           S: 36428224884624016312651150295410362968840842274684184919434425858579114415856
```

Figure 74: RLPx ETH Transactions Packet Node1 => Bootnode

Lastly, receipts are found within the blocks themselves, however if new clients come online or other clients want to verify transactions they are able to request receipts view the GetReceipts message, followed by a Receipts message response as seen in Figures 75 and 76. In the dissection environment, this was normally seen when newer clients would come online and had to synchronize transactions that took place prior to joining the network.

```
4018 192.5709562… 10.1.1.10   10.1.0.10   RLPX      162 39436 → 30303 [ETH GetReceipts] Type=GetReceipts
4019 192.5712223… 10.1.0.10   10.1.1.10   RLPX      162 30303 → 39436 [ETH Receipts] Type=Receipts Code=

Frame 4018: 162 bytes on wire (1296 bits), 162 bytes captured (1296 bits) on interface br-d2788e2c7b9b,
Ethernet II, Src: 02:42:0a:01:01:0a (02:42:0a:01:01:0a), Dst: 02:42:0a:01:01:02 (02:42:0a:01:01:02)
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.0.10
Transmission Control Protocol, Src Port: 39436, Dst Port: 30303, Seq: 26954, Ack: 17735, Len: 96
Ethereum RLPx Protocol
 ▾ Frame Header: 6a2cf3fc5cfa28690780947981dbae737d6e2e3d3cb2c75caa195ea2a0465569
      Decrypted Header Data: 00002fc28080000000000000000000000
      Header MAC: 7d6e2e3d3cb2c75caa195ea2a0465569
      Frame Body MAC: 136521816eb556b0587d2565aca2b1f1
      Frame Size: 47
      Read Size: 48
      Header Data: Capability ID: 0, Context ID: 0
 ▾ Frame Body: 9ce1fb989158ccad08dea44849ad3ecae6247ea98a9d0f49f541f9f9e2c3d89e0ceb3df7…
      Type: [ETH GetReceipts] Type=GetReceipts Code=15
      Capability: ETH
      Code: 15
      Request ID: 13126262220165910460
      Block Hashes:
         Block Hashes #1: ab6e9c0061abc6f5555bb9e5365193ef38ca56c65214022082d14c52c53ceb55
```

Figure 75: RLPx ETH GetReceipts Packet Node1 => Bootnode

```
4019 192.5712223… 10.1.0.10   10.1.1.10   RLPX      162 30303 → 39436 [ETH Receipts] Type=Receipts Code=16 Len=48

Frame 4019: 162 bytes on wire (1296 bits), 162 bytes captured (1296 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
Internet Protocol Version 4, Src: 10.1.0.10, Dst: 10.1.1.10
Transmission Control Protocol, Src Port: 30303, Dst Port: 39436, Seq: 17735, Ack: 27050, Len: 96
Ethereum RLPx Protocol
 ▾ Frame Header: a79b6093600fd575a73e3cdfce55b70219339405eed761a180cc860c462ecc9b
      Decrypted Header Data: 00002ec28080000000000000000000000
      Header MAC: 19339405eed761a180cc860c462ecc9b
      Frame Body MAC: 4c15cdd5a6f9f32efe876f206ac3e10b
      Frame Size: 46
      Read Size: 48
      Header Data: Capability ID: 0, Context ID: 0
 ▾ Frame Body: 4f2d7e09f3f7a3a309d8a9c94e8ff5723bff8f544886117aa49f027f8007f059746dd044…
      Type: [ETH Receipts] Type=Receipts Code=16
      Capability: ETH
      Code: 16
      Request ID: 13126262220165910460
      Receipts:
         Receipts #1:
            Receipts #1 #1:
               Post State Or Status: 1
               Cumulative Gas: 21000
               Bloom: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                      00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                      00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                      00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
               Logs: N/A
```

Figure 76: RLPx ETH Receipts Packet Bootnode => Node1

4.5.2 Dissecting SNAP Capability Messages

The SNAP protocol runs on the RLPx transport facilitating the exchange of Ethereum state snapshots between peers. The protocol was originally an optional extension for peers that supported the capability; however, with the release of ETH/67, SNAP has become mandatory for state management amongst peers. The SNAP protocol aims to make dynamic snapshots of current states available for peers, allowing for semi-real-time data retrieval. The SNAP protocol is meant to run side-by-side with the ETH protocol, meaning it cannot be run without the ETH protocol. The SNAP synchronization mechanism enables peers to retrieve and verify all the account and storage data without downloading intermediate Merkle trie nodes. This allows the final state trie to be reassembled locally, drastically reducing the networking load [35].

In Ethereum, the state trie is a Merkle tree comprised of leaves that contain valuable data, and each node above is the hash of 16 children. Syncing from the tree's root (the hash embedded in a block header), the only way to download everything is to request each node individually [36]. A trie node is a node in the trie data structure. In Ethereum, the trie nodes are used to store the state of the blockchain. The state of the blockchain is the current state of all accounts and contracts on the blockchain. The state is stored in a Merkle Patricia Trie, which is a modified version of a Patricia Trie [37]. For example, every block header stores the roots of three trie structures: stateRoot, transactionRoot, and receiptsRoot. The state trie represents a mapping between account addresses and the account states. The account state includes the balance, nonce, codeHash and storageRoot.

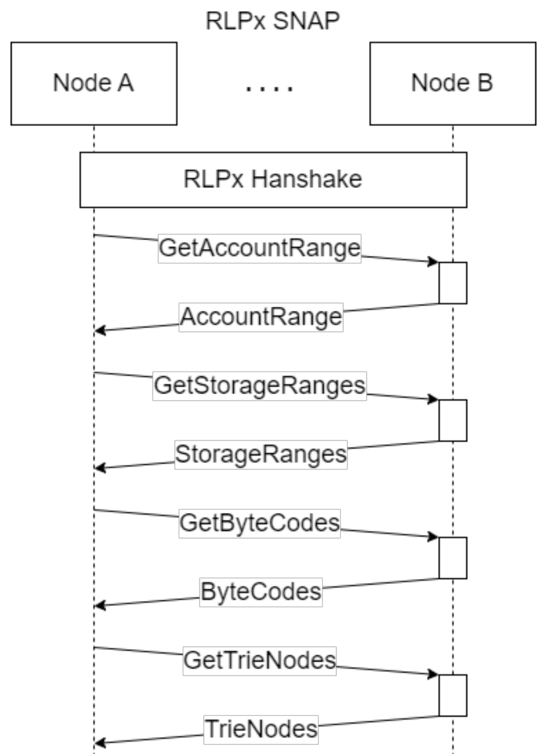


Figure 77: RLPx SNAP Capability Message Sequence Diagram

SNAP is used for getting quick snapshots to quickly build the Ethereum state locally, and the typical sequence flow of the SNAP message can be seen in the above sequence diagram in Figure 77. This starts off with a GetAccountRange (0x00) message, as seen in Figure 78. This message requests an unknown number of accounts from a given account trie, intended to fetch a large number of subsequent accounts from a remote node and reconstruct a state subtrie locally. The response message, AccountRange (0x01), seen in Figure 79, returns a number of consecutive accounts and the Merkle proofs for the entire range. Each SNAP message has a mandatory “request id” field, which is used to track which response message correlates with which request/get message.

```
435 53.963732462 10.1.0.10  10.1.1.10  RLPX   194 30303 → 39436 [SNAP GetAccountRange] Type=GetAccountRange
436 53.963778562 10.1.0.10  10.1.1.10  RLPX   146 30303 → 39436 [ETH GetBlockHeaders] Type=GetBlockHeaders (
437 53.995379062 10.1.1.10  10.1.0.10  TCP     66 39436 → 30303 [ACK] Seq=2618 Ack=1495 Win=64128 Len=0 TSva
438 53.995499062 10.1.1.10  10.1.0.10  RLPX   130 39436 → 30303 [ETH BlockHeaders] Type=BlockHeaders Code=4
439 53.995637962 10.1.0.10  10.1.1.10  RLPX   146 30303 → 39436 [ETH GetBlockHeaders] Type=GetBlockHeaders (
440 53.995792662 10.1.1.10  10.1.0.10  RLPX   130 39436 → 30303 [SNAP AccountRange] Type=AccountRange Code=1

Frame 435: 194 bytes on wire (1552 bits), 194 bytes captured (1552 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
Internet Protocol Version 4, Src: 10.1.0.10, Dst: 10.1.1.10
Transmission Control Protocol, Src Port: 30303, Dst Port: 39436, Seq: 1287, Ack: 2618, Len: 128
Ethereum RLPx Protocol
  Frame Header: 9c41e84674c67b76033581ef8ebd44a52f61f725d38b85a8471fa72e49da1c98
      Decrypted Header Data: 000041c2808000000000000000000000
      Header MAC: 2f61f725d38b85a8471fa72e49da1c98
      Frame Body MAC: bb081f7b0b38048cd5d887a325007044
      Frame Size: 65
      Read Size: 80
      Header Data: Capability ID: 0, Context ID: 0
  Frame Body: 899abcef6bf268e6baa32ba75b86f6ec3d1cd867515a7caccaed4666001c33165364f5e6…
      Type: [SNAP GetAccountRange] Type=GetAccountRange Code=0
      Capability: SNAP
      Code: 0
      Request ID: 1976235410884491574
      Root Hash: 56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
      Starting Hash: 0000000000000000000000000000000000000000000000000000000000000000
      Limit Hash: 0fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
      Stop Bytes: 65536
```

Figure 78: RLPx SNAP GetAccountRange Packet Bootnode => Node1

```
435 53.963732462 10.1.0.10  10.1.1.10  RLPX   194 30303 → 39436 [SNAP GetAccountRange] Type=GetAccountRange
436 53.963778562 10.1.0.10  10.1.1.10  RLPX   146 30303 → 39436 [ETH GetBlockHeaders] Type=GetBlockHeaders (
437 53.995379062 10.1.1.10  10.1.0.10  TCP     66 39436 → 30303 [ACK] Seq=2618 Ack=1495 Win=64128 Len=0 TSva
438 53.995499062 10.1.1.10  10.1.0.10  RLPX   130 39436 → 30303 [ETH BlockHeaders] Type=BlockHeaders Code=4
439 53.995637962 10.1.0.10  10.1.1.10  RLPX   146 30303 → 39436 [ETH GetBlockHeaders] Type=GetBlockHeaders (
440 53.995792662 10.1.1.10  10.1.0.10  RLPX   130 39436 → 30303 [SNAP AccountRange] Type=AccountRange Code=1

Frame 440: 130 bytes on wire (1040 bits), 130 bytes captured (1040 bits) on interface br-d2788e2c7b9b, id 0
Ethernet II, Src: 02:42:0a:01:01:0a (02:42:0a:01:01:0a), Dst: 02:42:0a:01:01:02 (02:42:0a:01:01:02)
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.0.10
Transmission Control Protocol, Src Port: 39436, Dst Port: 30303, Seq: 2682, Ack: 1575, Len: 64
Ethereum RLPx Protocol
  Frame Header: 47c74975e96c932e0bbea1e5a6f2cd40a572a84e36bd70e894bc396712e12fb2
      Decrypted Header Data: 00000fc2808000000000000000000000
      Header MAC: a572a84e36bd70e894bc396712e12fb2
      Frame Body MAC: a185d3d8b03b0530b1048bb25f60443c
      Frame Size: 15
      Read Size: 16
      Header Data: Capability ID: 0, Context ID: 0
  Frame Body: f7524f1e7498b55324d2ebee8b66e083a185d3d8b03b0530b1048bb25f60443c
      Type: [SNAP AccountRange] Type=AccountRange Code=1
      Capability: SNAP
      Code: 1
      Request ID: 1976235410884491574
      Accounts: N/A
      Proof: N/A
```

Figure 79: RLPx SNAP AccountRange Packet Node1 => Bootnode

Next, the SNAP protocol allows the request of the storage slots of multiple accounts’ storage tries, which could even be a single account. This message is GetStorageRanges (0x02). As we know in this private Ethereum network environment and scenario, each client only has a single account associated with it. This message is responded to with the StorageRanges (0x03) message. However, we was not able to propagate the GetStorageRanges and StorageRanges messages utilizing the private Ethereum network with the latest GETH clients. However, the dissector does support them, but this is not verified. See below for the information that would be dissected in these messages:

GetStorageRanges:

- Request ID: Integer
- Root Hash: Hex Value
- Account Hashes: List of Hex Values
- Starting Hash: Hex Value
- Limit Hash: Hex Value
- Response Bytes: Integer

StorageRanges:

- Request ID: Integer
- Slots: List of Slot:
  - Slot Hash: Hex Value
  - Slot Data: Hex Value
- Proof: List of Hex Values

Lastly, with four messages remaining, there exists the GetByteCodes (0x04) message which requests a number of contracts byte-codes by hash. This allows retrieving the code associated with accounts retrieved via the GetAccountRange message but GetByteCodes is needed during healing too. Healing is a cleansing of the local state of the node. ByteCodes (0x05) is sent in response to GetByteCodes which returns a number of requested contract codes in the same order as the requests but there might be some gaps if not all codes were available. Next, the GetTrieNodes (0x06) message, seen in Figure 80, is used to request a number of state (either account or storage) Merkle trie nodes by path. This message is responded to by the TrieNodes (0x07) message, seen in Figure 81, which returns the requested number of state tire nodes.I was not able to propagate the GetByteCodes and ByteCodes messages utilizing the private Ethereum network with the latest GETH clients. However, the dissector does support them, but this is not verified. See below for the information that would be dissected in these messages:

GetByteCodes:

- Request ID: Integer
- Hashes: List of Hex Values
- Bytes: Integer

ByteCodes:

- Request ID: Integer
- Codes: List of Hex Values

```
 3294 129.539243560 10.1.2.20  10.1.1.10   RLPX     178 42560 → 30304 [SNAP GetTrieNodes] Type=GetTrieNodes
 4860 179.881323353 10.1.2.20  10.1.1.10   RLPX     258 42560 → 30304 [SNAP TrieNodes] Type=TrieNodes Code=7

Frame 3294: 178 bytes on wire (1424 bits), 178 bytes captured (1424 bits) on interface br-d35ff39d33f3, ic
Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
Internet Protocol Version 4, Src: 10.1.2.20, Dst: 10.1.1.10
Transmission Control Protocol, Src Port: 42560, Dst Port: 30304, Seq: 4578, Ack: 6255, Len: 112
Ethereum RLPx Protocol
 ▾ Frame Header: 01be3be4ff7f2d6ce9d2ce4429ebab35cb818737750c256ecdd079926df9d64b
      Decrypted Header Data: 000035c280800000000000000000000000
      Header MAC: cb818737750c256ecdd079926df9d64b
      Frame Body MAC: 9ad3f898cc39808aca907b992cb3f772
      Frame Size: 53
      Read Size: 64
      Header Data: Capability ID: 0, Context ID: 0
 ▾ Frame Body: 6f2f880d34495a203a2246a7c06880327a3b930dfe57d1dbe9e35c8a7de9c41f39b3ba6e…
      Type: [SNAP GetTrieNodes] Type=GetTrieNodes Code=6
      Capability: SNAP
      Code: 6
      Request ID: 4893789450120281907
      Root Hash: b4a048dfb5c6c9a56c1cfb0f385a8888906fd4c17a5c23892dd9f848bee770fd
      Paths:
         Paths #1:
            Paths #1 #1: 00
      Bytes: 524288
```

Figure 80: RLPx SNAP GetTrieNodes Packet Node1 => Bootnode

```
 3294 129.539243560 10.1.2.20  10.1.1.10   RLPX     178 42560 → 30304 [SNAP GetTrieNodes] Type=GetTrieNodes Code=6 Len=64
 4860 179.881323353 10.1.2.20  10.1.1.10   RLPX     258 42560 → 30304 [SNAP TrieNodes] Type=TrieNodes Code=7 Len=144

Frame 4860: 258 bytes on wire (2064 bits), 258 bytes captured (2064 bits) on interface br-d35ff39d33f3, id 0
Ethernet II, Src: 02:42:0a:01:01:02 (02:42:0a:01:01:02), Dst: 02:42:0a:01:01:0a (02:42:0a:01:01:0a)
Internet Protocol Version 4, Src: 10.1.2.20, Dst: 10.1.1.10
Transmission Control Protocol, Src Port: 42560, Dst Port: 30304, Seq: 9074, Ack: 10143, Len: 192
Ethereum RLPx Protocol
 ▾ Frame Header: edf2626dc92d46ce9eb9c46ae1c9f90031d7f28fb66bab81bc996f4d74e23e76
      Decrypted Header Data: 000087c280800000000000000000000000
      Header MAC: 31d7f28fb66bab81bc996f4d74e23e76
      Frame Body MAC: d72700415fd7cfd6d13ab741dc17519d
      Frame Size: 135
      Read Size: 144
      Header Data: Capability ID: 0, Context ID: 0
 ▾ Frame Body: 9345ba16d5d47ec6d9bff6dc217f52a2f5c8762daef12a3cb37392ea5c4bd3c2547dff1a…
      Type: [SNAP TrieNodes] Type=TrieNodes Code=7
      Capability: SNAP
      Code: 7
      Request ID: 3337066551442961397
      Nodes:
         Nodes #1: f871a03b78532ba1b5b69666061243d4694e4e4162b1c20e0bf25963ebded08510e2bfb84ef84c0188f9ccd8a1c5005ee0a056e81f171bcc55a6ff8345e692c0
                   f86e5b48e01b996cadc001622fb5e363b421a0c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470
```

Figure 81: RLPx SNAP TrieNodes Packet Node1 => Bootnode

4.6 Recap and Discussion

After looking at all that is to offer with the dissector, let us take a step back and talk about what can be learned from the development of the dissector. The dissector provides a minimal third-party dependency method for dissecting DiscoveryV4, DiscoveryV5, and RLPx sub-protocols, including ETH and SNAP. Creating the dissector was no easy task, solely going off the Go Ethereum source code and the minimal documentation found in the markdown documents in the Ethereum DEVP2P repository. Implementation specifics were often only touched on if digging deep into the source code to figure out how specifically they are deriving the keys, or very often, what public key they are using, whether it is compressed or not.

The dissector proves a viable tool for DEVP2P dissection while touching on a range of topics from encoding/decoding with RLP, elliptic curve cryptography and its use in ECIES, along with elliptic curve digital signature and Diffie-Hellman. The dissector even deals with reassembling

TCP packets and uses SNAPPY for decompression. Besides this range of topics the dissector touches, it overcomes the main hurdle attributed specifically by ConsenSys, specifically RLPx decryption and automatically grabbing the exposed random private keys and using them for session key derivation.

Lastly, PYDEVP2P provides tooling for a range of capabilities and tooling for not only DEVP2P but a python-only zero-dependency elliptic curve cryptography implementation. This could be easier because many implementations use built-in C libraries for performance-intensive elliptic curve calculations. All six message types for DiscoveryV4 were dissected and displayed in Wireshark, while all eight messages for DiscoveryV5 were dissected; however, only six were proven and displayed in Wireshark. Lastly, with RLPx, 2 Handshake messages, four built-in P2P capability messages, 13 ETH messages, and 8 SNAP messages were dissected. Therefore, this dissector provides dissection, decryption, and decoding capabilities for a total of 41 message types spanning three different protocol types in the suite of DEVP2P.

5. Security Analysis with the Dissector

Network packet dissectors, like the one created and explained in great detail in the previous chapter, are great for analyzing specific pieces of network traffic and subsequent packet data in a human-readable format. As discussed, dissectors can help identify malicious traffic, such as malware, denial-of-service attacks, or authorized access attempts. They can even help monitor network activity and detect anomalies or suspicious patterns, either actively or after the fact, with captured network traffic.

Dissectors have also been pointed to for helping students learn about popular network protocols, data structures, and communication patterns in our daily network traffic. Teachers have also used them to demonstrate network concepts, such as the TCP 3-way handshake, or even cryptographic concepts regarding TLS/SSL and, more specifically, in this case, elliptic curve cryptography.

In the industry and the open-source community, dissectors help developers and analysts test and debug their network applications or protocols. They are proven to help developers understand how other network applications or protocols work and intersect together while also providing tooling to help diagnose and resolve network, connectivity, or performance issues. Network packet dissectors provide a way to dig deeper and see the actual underlying packet information transferred between network hosts, removing any abstraction.

As we have seen with the multitude of packet dissections in the previous chapter, a great deal of information can be unraveled, uncovering the mysteries behind DEVP2P and understanding and proving the Ethereum documentation. However, what we saw is considered the best-case scenario, where the network is set up to provide all the results to develop and create the dissector. Dissectors shine when there is a “rainy day” scenario or when something goes wrong, and the network traffic needs to be debugged to diagnose possible configuration issues or connectivity problems.

Throughout this chapter, we will provide several ways this dissector can be used, proving its usefulness to the community, educators, developers, and researchers. With the help of our main contribution PYDEVP2P, we will first walk through how the Elliptic Curve Digital Signature Algorithm (ECDSA) is used regarding DiscoveryV4. We will first look closer at the captured packets and then dig deeper into how the digital signature is used to recover the sender's identity and prove the message's authenticity. Secondly, we will discuss the methods used for obfuscating the network traffic in DiscoveryV5 and its use of Elliptic Curve Diffie-Hellman (ECDH) while using the dissected packets to verify the goals laid out for DiscoveryV5 and compare the security improvements and implementations with DiscoveryV4. These elliptic curve cryptography (ECC) algorithms are all found in PYDEVP2P without the use of 3rd party dependencies for elliptic curve calculations, all helping to provide transparency and accessibility to these topics for educators and the community. Lastly, we will utilize the dissector to track a transaction from

Node 1 to Node 2, as we saw in the scenario, from a transaction propagation throughout the network to seeing the transaction make it to the blockchain and ultimately to the target account.

5.1 DiscoveryV4 ECDSA Performance & Security Analysis

This section will cover what the Elliptic Curve Digital Signature Algorithm (ECDSA) is and how it is used in DiscoveryV4, covered in detail in Chapter 4.3.1, to recover the public key of the sender node while also breaking down some of the technical details behind the elliptic curve. This section will also cover the ECDSA implementation in PYDEVP2P, which can be utilized as a great educational tool to understand elliptic curve operations in a pure-python implementation. Now, revisiting the scenario in Chapter 2.3, "Starting the Private Network", we saw that when the network started, the nodes on the network found each other and connected to one another as denoted by the "peercount." As we have discussed, this is all done through Ethereum node discovery, specifically DiscoveryV4 in Ethereum execution clients. However, what we will cover in this section is how exactly nodes are able to validate and verify the identity of the sender and authenticity of DiscoveryV4 packets.

Let's recount the contents found in a DiscoveryV4 ping packet, covered in great detail in Chapter 4 Section 3.1, from Node 1 (10.1.1.10) to the Bootnode (10.1.0.10) utilizing the DEVP2P Wireshark dissector's output shown in Figure 82. What we are specifically keying in on here is the "Sign" field, which represents the elliptic curve digital signature on the contents of the Ping packet from Node 1 to the Bootnode.

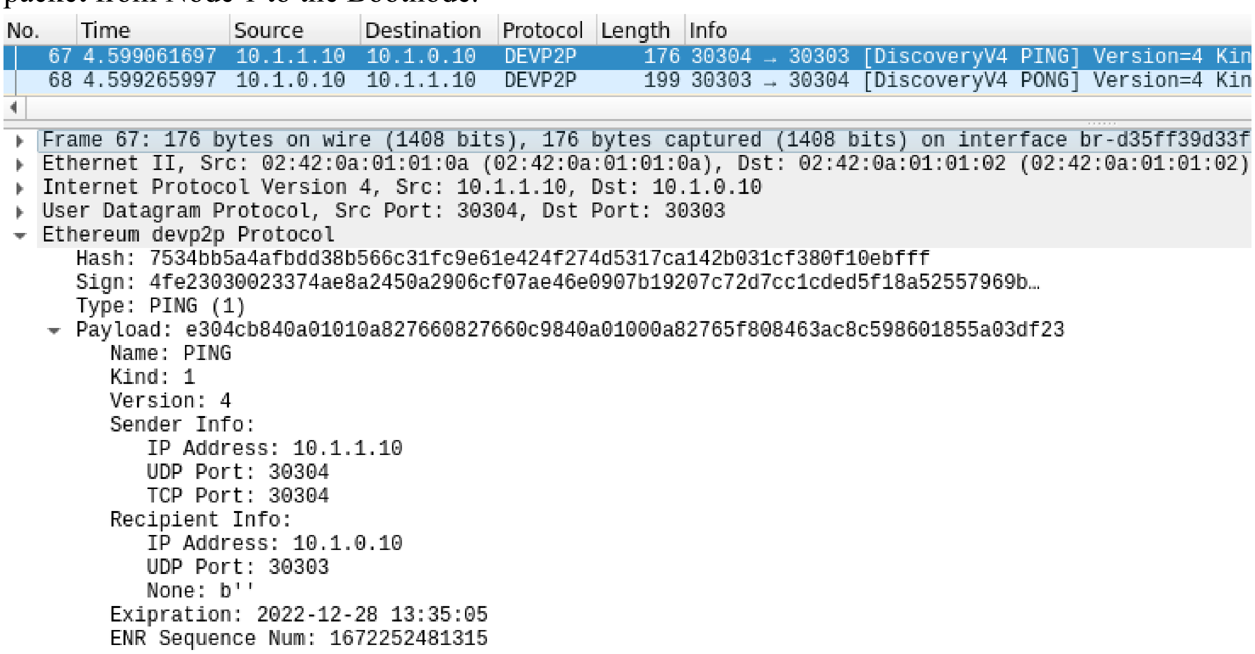


Figure 82: DiscoveryV4 Ping Packet Node1 to Bootnode

Now, let's think about this packet from the perspective of the Bootnode who is receiving this Ping packet. Based on the contents alone, the Bootnode could look at the "Sender Info" fields without using the digital signature to figure out who sent this packet. This includes the sender's IP address, 10.1.1.10, and the ports used, by the sender for TCP/UDP which is 30304. These fields can be checked with the sender's IP address found in the IP layer of the packet, but other than this, there is really no way for the Bootnode to know the identity of the sender, such as their elliptic curve public key or the actual authenticity of the packet contents. Meaning, without the use of the signature, anyone could have sent this information to the Bootnode, and the Bootnode has no exact way to authenticate and verify the identity of the sender. This is, of course, where the digital signature field comes into play.

First, the Bootnode can verify message integrity using the "Hash" field, which is found in the first 32 bytes in all the DiscoveryV4 packet headers. This hash is calculated using the keccak256 hash, part of the SHA-3 family of algorithms to compute the hash of an input to a fixed length output. The input can be of a variable length, but the result will always be fixed to 32 bytes. This is a one-way cryptographic hash function, which cannot be decoded in reverse [38]. Therefore, this hash found in the DiscoveryV4 message header can be checked for equality to the hash of the message contents following the first 32 bytes of the message, which includes the signature field.

Before the Bootnode can verify the message and recover the public key of the sender node, let's dig deeper into the cryptography behind elliptic curves. ECDSA relies on the math of the cyclic groups of elliptic curves over finite fields and on the difficulty of the elliptic curve discrete logarithm problem (ECDLP) [39]. The basis of this problem is what makes elliptic curve cryptography secure, where plainly put, the private key, a random integer in the range from 0 to (N -1), where N is the order of the curve, is not able to be uncovered from a known public key which is a point on the elliptic curve calculated from $privKey * G$. The point value G is known as the generator point, a defined point on the elliptic curve. This point is found on the actual ellptic curve, which is comprised by a specific equation template, known as a weistress equation: $y^2 = x^3 + Ax + B$, where A and B are constants picked by different elliptic curve definitions [40]. For parity, users of elliptic curve cryptography must share the same elliptic curve parameters and agree on the same generator point G, including N, which defines the length of private keys. In the secp256k1 elliptic curve, which is used by Bitcoin and Ethereum, the constants are $A = 0$ and $B = 7$ and specifically for Ethereum the generator point Gx and Gy including the N:

N: `115792089237316195423570985008687907852837564279074904382605163141518161494337`
Gx: `55066263022277343669578718895168534326250603453777594175500187360389116729240`
Gy: `32670510020758816978083085130507043184471273380659243275938904335757337482424`

So, as was just stated, using the nodes private static key, the public key for that node can be created using the private key multiplied with G, the generator point on the elliptic curve. Thus, yielding the private and public keys for the Bootnode and Node 1 shown below. Of course, only the nodes know their own private keys, and the public keys are what can be shared with other nodes, also defining the node's unique identity. The public key of Node 1, shown below in Table 4 is what we will cross reference with the public key recovered from the signature field found in the Ping packet.

Table 4: Bootnode and Node 1 Private and Public Static Keys

Node (IP Address)	Private / Public Keys

Bootnode (10.1.0.10)	Private Key:
	`3028271501873c4ecf501a2d3945dcb64ea3f27d6f163af45eb23ced9e92d85b`
	Public Key:
	`2c4b6808e788537ca13ab4c35e6311bc2553b65323fb0c9e9a831303a1059b875`
	`4aab13dbb78c03a7a31beee5c2f2fb570393f056d54fa83ebd7e277039cc7b6`

Node 1 (10.1.1.10)	Private Key:
	`4622d11b274848c32caf35dded1ed8e04316b1cde6579542f0510d86eb921298`
	Public Key:
	`c35c2b7f9ae974d1eee94a003394d1cc18135e7fe6665e6b4f221970f1d9d59f6`
	`a58e76763803bcc9097eba4c91fd08b30405e65c53272b8635348e37f93cedc`

Next, the Bootnode can then recover the sender's static public key, also referred to as the node key using just the signature, the 65 bytes following the hash, along with the message data, which is all the data following the hash and the signature. The signature consists of 2 values, R and S, and in some cases, like with Ethereum ECDSA signatures, V. Without going into the complete specifics, the signer encodes a random point R (representing only by the x coordinate) followed by the S portion which is the computed value of the message hash H using the signer's private key *privKey*, which is the proof that the message signer knows their own private key. Signature verifications decodes the proof value, S, from the signature back to its original point R, using the public key and the message hash H and compares the x-coordinate of the recovered R with the r value from the signature [41].

Nevertheless, how would the Bootnode recover the public key of the sender, specifically Node 1? This can be done utilizing the ECDSA public key recovery algorithm laid out in section 4.1.6 of "Standards for Efficient Cryptography: Elliptic Curve Cryptography." This algorithm is handy for self-signed signatures and valuable in bandwidth-constrained environments where a full certificate may not be viable. Using the ECDSA signature (r, s, v), and all of the elliptic curve parameters, it is possible to determine the public key Q of the signer. Due to the nature of the elliptic curve, there is the possibility for several public keys to be recovered from the signature that resides on the elliptic curve. This is mitigated with the extra byte in the signature, held in the "V" value of the signature. This value holds which public key is correct out of the three possibilities, which could be the values 0, 1, or 2.

Lastly, let's relate this back to how PYDEVP2P implements the ECDSA public key recovery algorithm. All the ECDSA specifics must be incorporated into the dissector in order to achieve the proper results, which has not been done before by any other dissectors. This calculation is done utilizing Jacobian points, which is a way of representing a point on an elliptic curve using three coordinates (x, y, z) instead of using the standard cartesian coordinates (x, y) [42]. This allows for faster arithmetic operations on the curve, such as addition and multiplication of points, mitigating costly modular arithmetic. The function, shown in Figure 83 as "ecdsa_raw_recover", found in pydevp2p/elliptic/curve.py, uses the secp256k1 curve parameters, which are defined as global variables, then utilizing jacobian point arithmetic, produces the public key, Q, that is recovered from the signature.

```python
def ecdsa_raw_recover(msghash: bytes, rsv: tuple[int, int, int]):
    """ecdsa_raw_recover recovers the public key from the signature.

    Args:
        msghash (bytes): The message hash to recover the public key from
        rsv (tuple[int, int, int]): The signature to recover the public key from

    Returns:
        bytes: The public key recovered from the signature
    """
    # https://www.secg.org/sec1-v2.pdf page 47-48
    # https://gist.github.com/nlitsme/dda36eeef541de37d996
    N = secp256k1.N
    A = secp256k1.A
    B = secp256k1.B
    H = secp256k1.H
    P = secp256k1.P
    G = secp256k1.G
    r, s, v = rsv

    x = r

    ysqaure = (x**3 + A * x + B)
    beta = pow(ysqaure, (P+1)//4, P)
    y = beta if beta % 2 == v else -beta
    R = (x, y)

    m = bytes_to_int(msghash)
    # R * s
    Rs = jacobian_multiply(to_jacobian(R), s)
    # G * m // r
    Gz = jacobian_multiply(to_jacobian(G), (N - m) % N)

    # (R * s - G * m)
    Qr = jacobian_add(Rs, Gz)
    Q = jacobian_multiply(Qr, inv(r, N))
    Q = from_jacobian(Q)

    return encode_pubkey(Q, "bin_electrum")
```

Figure 83: PYDEVP2P ECDSA Raw Public Key Recovery

The output of this above "ecdsa_raw_recover" function, returns the recovered public key from the msghash and the elliptic curve digital signature, which in this case, will be the public key of the sender/signer of the message. In this case, related back to the scenario, would be Node 1's public key, recovered from the message.

While ECDSA public key recovery from the signature allows for the recovery of the public key and verifying the identity of the sender without knowing their public key in advance, there are shortcomings to this method. This operation is computationally expensive and can consume many CPU resources. The performance impact of ECDSA public key recovery from the signature can be significant, especially for nodes that receive much traffic from unknown peers. This can lead to increased latency, reduced throughput, and higher energy consumption [43]. This leads to the main cause of DiscoveryV4's downfall, traffic amplification attacks, a form of denial-of-service. This DoS takes place by simply creating a significant amount of fake nodes on the network and spamming DiscoveryV4 messages. This forces the nodes to try and recover and verify signatures, but also requires the recipient of these messages to respond with DiscoveryV4 messages. This leads into specific mitigations for this issue by dropping known nodes that have not responded in the last 12 hours. This also creates what is known as an "endpoint proof" system, where verified connections are stored, and the signature will not necessarily be rechecked from that endpoint within a specific time limit. This causes problems in terms of security and scalability throughout the network, where this endpoint proof is unreliable, causing costly retries and also causing fake authentication of endpoints.

5.2 DiscoveryV5 Masking and Confidentiality

This section will analyze the DEVP2P DiscoveryV5 protocol, covered in detail in Chapter 4.3.2, focusing on its masking and confidentiality features, and what the dissector has to overcome in order to provide these dissected results. We will start by understanding the security goals laid out by the DiscoveryV5 documentation, which are mainly to mitigate endpoint proof, require destination node ID for communication, and provide message obfuscation and confidentiality. Then, we will examine how these goals are achieved by the protocol design and implementation, utilizing the dissector and subsequent packets found. Finally, we will discuss some of the cryptographic aspects of the protocol, such as the use of elliptic curve cryptography, specifically ECDH, and how its use effectively hinders denial-of-service and replay attacks.

In response to specific shortcomings of DiscoveryV4, the Ethereum team laid out specific goals for DiscoveryV5. One main goal was to replace the DiscoveryV4 endpoint proof, as it is unreliable and slow due to retries. DiscoveryV5 also requires knowledge of the destination Node ID for communication, which makes it harder for other nodes to obtain the Node ID. Fake nodes are also less likely to provoke responses knowing just the node's IP address alone. Next, DiscoveryV5 also must obfuscate the traffic to prevent accidental packet mangling or trivial packet sniffing while also providing a way to prevent packet replay attacks or peer amplification attacks [29]. Throughout this section, we will look at how DiscoveryV5 is implemented to prevent or achieve the goals set out above, which are found directly in the specification of DiscoveryV5.

Now, let us look at how the header information of DiscoveryV5 packets is "masked" using symmetric encryption. Primarily, as stated before, this masking and obfuscation of the header data are to prevent static and passive identification of the protocol information. When a packet is received, the message is laid out in three portions, the masking-if, the masked header, and the message itself. The masked header contains the actual packet header, which starts with a fixed-size "static-header" followed by a variable-length "authdata" section.

Decrypting the masked header starts by the recipient constructing an AES/CTR stream cipher using its own node ID as the key and using the IV from the packet. This means that the sender of the DiscoveryV5 packet has to know the Node ID of the destination prior to sending the packet. Then using this AES/CTR stream cipher, the recipient can then decrypt the static-header, and verify the contents and successful decryption by checking the "protocol-id" field, which is always "discv5". If this protocol ID is correct, then the "authdata" can also be unmasked using the same cipher. Shown in Figure 84, the unmasked header information found from the DEVP2P Wireshark dissector's output.

```
Ethereum devp2p Protocol
▾ Header: 1a5d55616405b2c41827db3a6e006e979b6d98767ab1bffcbc109bf46a4c2c2d6bcdffcc…
      Iv: 1a5d55616405b2c41827db3a6e006e97
      Protocolid: 110404070241845
      Version: 1
      Flag: 0
      Nonce: 00000007f835234fef9d16d2
      Authsize: 32
      Authdata: 01bd15281bf9cf4521dc7c88e6abbea95b781dd177b5a4b42fa54312ea71b266
      Src: 01bd15281bf9cf4521dc7c88e6abbea95b781dd177b5a4b42fa54312ea71b266
      Type: MESSAGE
```

Figure 84: DiscoveryV5 Unmasked Packet Header

Shifting over to PYDEVP2P, we can see how this DiscoveryV5 header unmasking can be accomplished, which effectively proves that the documentation for DiscoveryV5 is accurate. Below, as seen in Figure 85, we can see this three step process for unmasking the header, which is found in "pydevp2p/discover/v5wire/encoding.py" from lines 182 to 200.

```
# Unmask the static header.
head = Header(input[:sizeofMaskingIV])
mask = head.mask(self.localEnodeID)
staticHeader_masked = input[sizeofMaskingIV:Header.STATIC_SIZE]
staticHeader_unmasked = mask.decrypt(staticHeader_masked)

# Decode and verify the static header.
head.setStaticHeader(staticHeader_unmasked)
remainingInput = len(input) - Header.STATIC_SIZE
if not head.checkValid(remainingInput):
    print(
        f"{framectx()} Discv5Codec decode(input, fromAddr) Err Static Header Invalid")
    return None, None, None

# Unmask auth data
authDataEnd = Header.STATIC_SIZE + bytes_to_int(head.authSize)
authData_masked = input[Header.STATIC_SIZE:authDataEnd]
authData_unmasked = mask.decrypt(authData_masked)
head.authData = authData_unmasked
```

Figure 85: PYDEVP2P Unmasking the DiscoveryV5 Header

They are first unmasking the static header, which is accomplished by setting up an AES CTR stream cipher utilizing the “mask” function where the masking IV is used from the message along with the Node’s local ID. This sets up all the fields in the static header, like the protocol ID, version, flag, nonce, authorize and type. Next, we see in lines 190 and 191, that the validity of the header is checked, following this the auth data is then unmasked, using the same AES CTR stream cipher. This masking provides the necessary header obfuscation for the UDP payload data to prevent passive eavesdroppers and message identification tools. With the use of the dissector, along with PYDEVP2P, it allows for the DiscoveryV5 documentation to be verified utilizing a live Ethereum network with actual data sent between nodes. This dissector also provides educators and researchers the tools to dig deeper into the implementation specifics and understand and learn critical elliptic curve cryptography topics.

What is interesting about DiscoveryV5 is that the header is always masked in this manner, even after a successful handshake between nodes. Now, let us focus on the actual handshake that takes place between two nodes with regard to DiscoveryV5. Let us recall the handshake process between two nodes, Node A and Node B, where neither has communicated before, meaning no prior session keys have been formed. Node A sends an ordinary message to Node B, such as a Ping or FindNode message. Node B then receives this message and extracts the source Node Id from the packet header. Node B then initiates the handshake by responding with a WhoAreYou packet which includes a uniquely generated “id-nonce” field. Node A receives this WhoAreYou packet and proceeds with a handshake message, by resending the original packet they sent, including three new pieces to the message: “id-signature,” “ephemeral-pubkey” and “record”.

With all this information, Node A is then able to derive the new session keys, utilizing Elliptic Curve Diffie-Hellman (ECDH).

Elliptic Curve Diffie-Hellman (ECDH) is an anonymous key agreement scheme that allows two parties, each having an elliptic-curve public/private key pair to establish a shared secret key over an insecure channel [44]. ECDH is very similar to the classical Diffie-Hellman Key Exchange algorithm, however it uses ECC point multiplication instead of modular exponentiations. The protocol is based on the mathematical problem of finding the discrete logarithm of a point on an elliptic curve. The basic ECDH algorithm is quite trivial:

- Both parties (Node A and Node B) agree on a public elliptic curve and a base generator point G on the curve
- The sender generates a random private key (ephemeral-privK) and computes their public key (ephemeral-pubK)
- The recipient generates a random private key and public key
- Both parties exchange their public keys through the insecure channel
- Node A calculates the sharedKey = nodeBPubK * nodeAPrivK
- Node B calculates the sharedKey = nodeAPubK * nodeBPrivK
- Now both Node A and Node B have the same sharedKey

The security of ECDH relies on the assumption that it is hard to compute the private keys given the public keys and the generator point G. This is known as the elliptic curve discrete logarithm problem (ECDLP). Implementations of ECDH vary significantly, and in the case of Ethereum DiscoveryV5, it becomes much more complicated with the use of challenge and authentication data, along with key derivation functions that provide two keys, a “writeKey” and a “readKey.”

So, let’s go back to how Node A derives the session keys. After Node A receives the challenge data from the WhoAreYou message, it will generate a random private key known as the “ephemeral-key” along with a corresponding public key “ephemeral-pubkey”. This ephemeral key is used in conjunction with Node B’s static public key to perform the Diffie-Hellman key agreement. With the static public key of Node B (destination static public key), and the private ephemeral key, Node A is now able to compute the “sharedKey” also known as the “shared secret”. Shown in Figure 86, below, the actual implementation of the creation of the shared secret which is from the static-public-key and ephemeral-key or the ephemeral-pubkey and the static-private-key, for the initiator and the recipient respectively.

```python
def ecdh(pubk: bytes, privk: bytes) -> bytes:
    # ecdh creates a shared secret.
    eph_key = multiply(pubk, privk)
    shr_key = decode_pubkey(eph_key)
    first = bytes_to_int(b'\x02') | ((shr_key[1] >> 0) & 1)

    return int_to_bytes(first) + int_to_bytes(shr_key[0])
```

Figure 86: PYDEVP2P DiscoveryV5 ECDH Function Returning Shared Secret

It doesn't stop there however, Node A uses the shared secret (master key), the unmasked challenge data from the WhoAreYou message (salt), along with the text "discovery v5 key agreement" concatenated with the source Enode ID and the destination Enode ID (context) with a key derivation function. This key derivation function derives one or more keys from a master key using the HMAC-based KDF defined in RFC5869. The actual implementation of this from PYDEVP2P can be found from lines 50 to 65 in "pydevp2p/discover/v5wire/crypto.py", shown below in Figure 87. This specifically outputs two keys, the write key and the read key which is in the perspective of the node, meaning the other node will have the same two keys, just flipped. So, let's say Node A wants to send something to Node B, Node A will use the "write key" where then Node B will use its own "read key" which is actually the same key.

```python
50  def deriveKeys(hash, privk: bytes, pubk: bytes, n1: bytes, n2: bytes, challenge: bytes) -> Session:
51      # deriveKeys creates the session keys.
52      text = "discovery v5 key agreement".encode('utf-8')
53      info = text + n1 + n2
54      if len(info) != len(text) + len(n1) + len(n2):
55          print("deriveKeys(hash, privk, pubk, n1, n2, challenge) Err Invalid Info Len")
56          return None
57
58      eph = ecdh(pubk, privk)
59      if eph is None:
60          print("deriveKeys(hash, privk, pubk, n1, n2, challenge) Err Unable to Generate Shared Secret")
61          return None
62
63      writeKey, readKey = HKDF(
64          master=eph, key_len=16, salt=challenge, hashmod=hash, num_keys=2, context=info)
65      return Session(writeKey, readKey, 0)
```

Figure 87: PYDEVP2P DiscoveryV5 Key Derivation and Session Initiation

Finally, when Node B receives this handshake message, same key derivation can occur, this time using its own static private key and the ephemeral public key sent from Node A in the handshake packet. Thus finalizing the session, and as seen in the PYDEVP2P implementation, creates a "Session" which are the respective write/read keys for the nodes to be used for encrypting the message payload for all subsequent DiscoveryV5 messages. The ephemeral key is able to be extracted from the unmasked "authdata" from the DiscoveryV5 header which is not shown in the dissected packet, but the authdata is structured as follows, shown in Figure 88. In the authdata, there exists three static fields, the 32 byte Source Enode ID, followed by the size of the signature and the size of the public key. Then following this, the signature, the ephemeral public key and the ENR record.

```python
class HandshakeAuthData:
    SIZE = 32 + 1 + 1  # Only h or { srcID, sigSize, pubkSize }

    def __init__(self, authData: bytes) -> None:
        self.srcEnodeId = authData[:32]  # 32 bytes pubk
        self.sigSize = authData[32:33]  # uint8 byte 8 bits
        self.pubkSize = authData[33:34]  # uint8 byte 8 bits
        # Trailing variable-size data.
        self.sig: bytes = None
        self.pubk: bytes = None
        self.record: bytes = None
```

Figure 88: PYDEVP2P Handshake Auth Data Schema

5.3 Tracking a Transaction using the Dissector

This section will be laid out to show the steps to use the DEVP2P dissector to track a transaction propagated throughout the network. As discussed in detail in Chapter 4.4, this type of communication is facilitated by RLPx, encrypted TCP messages that the dissector must decrypt, decode, and dissect for the contents to be viewed properly in Wireshark. This section proves the dissector's educational value and gives a specific use case for the dissector. First, let us revisit the scenario, looking specifically at the transaction between Account/Node 1 and Account/Node 2, where the account created on Node 1 sent 200 ETH to the account created on Node 2. Before the dissector, the inner workings, and exchanges between the nodes on the network would have been obscured, encrypted, and impossible to track. However, now with the use of the dissector, it is now possible to fully trace a transaction through the network packets sent amongst the nodes on the network. It is even possible to see which node might choose the block to mine it into the chain, then follow each of the nodes in the network, validating this new block in this private Ethereum network.

Ethereum transactions are actions initiated by an externally owned account (EOA), which is an account managed by a human, not a contract. For example, in our scenario, if Account/Node 1 sends Account/Node2 200 ETH, Node 1’s account must be debited, and Node 2’s account must be credited. Remember, the actual nodes are not the accounts; they are simply a facilitator for these accounts to connect to and transact on the Ethereum network. This state-changing action, credit and debit, takes place within a transaction. When a transaction is submitted on the Ethereum network, it is broadcasted to all the nodes (clients) that run the network, like, in our case, Bootnode, Node 1, Node 2, and Node 3. The nodes validate the transaction and add it to their pool of pending transactions. The pending transactions are then selected by miners who try to include them in new blocks. Miners are incentivized to choose transactions that pay higher fees (called gas) per unit of computation (called gas limit).

So, let’s take a look at a slightly different example, where we are using an account (that was originally created on Node 1) to transact 100 Ether (ETH) to the account that was created on Node 2. This is carried out by the account’s public address, where the Node 1 account address:

"0x41159606b6240f725e969e3f1f342ff65904a4ec," is going to send 100 ETH to the Node 2 account address: "0x1f0cebf80f05de1213401c6d0a58e215c8ce635f". Looking at Figure 89, we can see the transaction confirmation that it was executed from MetaMask. Notice the amount of 100 ETH followed by the Gas Limit, which is the maximum amount of gas units the account is willing to spend in order for the transaction to be processed. A great way to think about gas is it provides fuel to the network, incentivizing others to perform network operations to keep everything going.

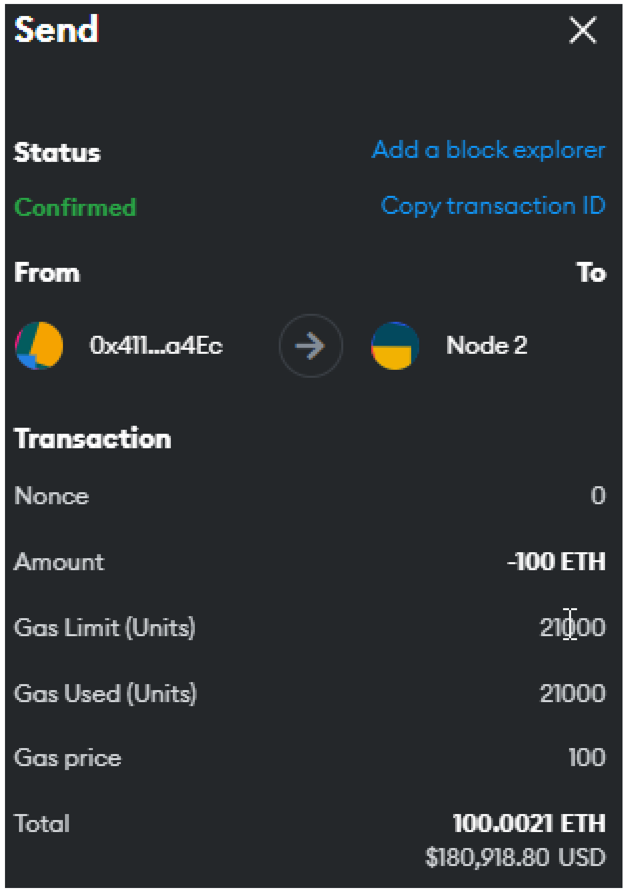


Figure 89: Node 1 Account Sending Node 2 Account 100 ETH

Upon sending this transaction, while having the dissector running, capturing on the interface of the Bootnode and Node 1, we see the following packets captured, seen in Figure 90. We first see the ETH capability Transactions message, which again is used to propagate a new transaction throughout the network. This is first sent from the Bootnode to Node 2, making sense as we connect MetaMask to our private network through the Bootnode. As stated earlier, only one node or the square root of the number of connected nodes will get the Transactions message, while the others will get the NewPooledTransactionHashes message. This notifies the other nodes in the network that a new transaction hash is in the Bootnode's local transaction pool. From there, we can see this chain of Transaction messages, propagating the full new transaction throughout the network, originally from the Bootnode => Node 2, then in packet #1921 Node 2 => Node 1, then finally Node 1 => Node 3. This fully propagates the transaction throughout the network, therefore a message like GetPooledTransactionHashes is unnecessary in this case.

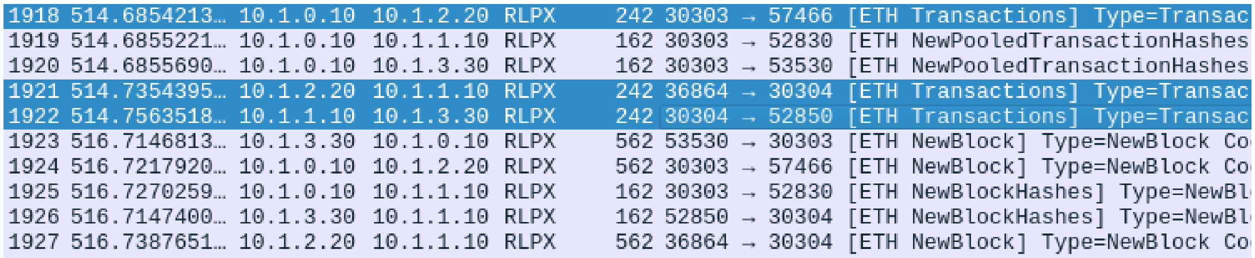


Figure 90: Dissector Packet Captures After Sending Transaction

This transaction propagation automatically puts the transaction in the local pool if the node is a valid miner on the network, which, in the case of this private network, each of the nodes are miners, except the Bootnode. Now, before we move on to the next step, let's dig deeper on what information about the transaction the dissector is able to provide us with. So, clicking on one of the Transactions messages, we see the following dissection of this ETH capability message, seen in Figure 91.

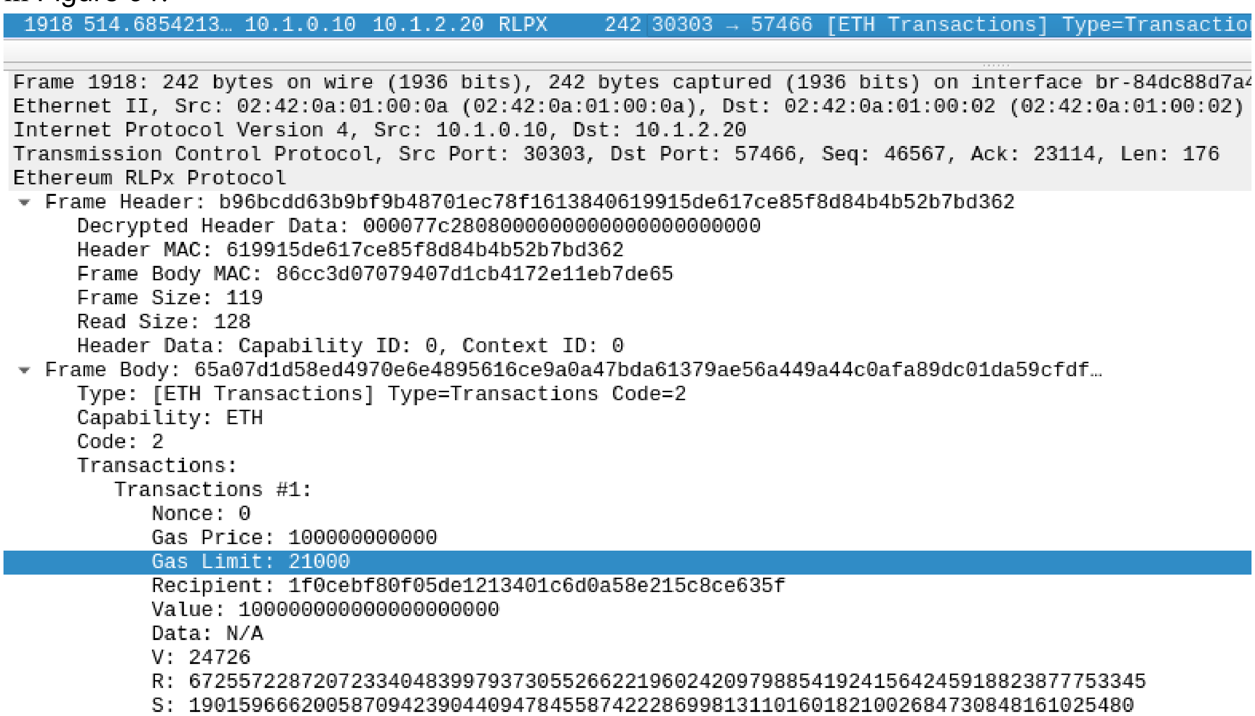


Figure 91: RLPx ETH Transactions Message 100 ETH from Bootnode to Node 2

Looking at the above packet capture of the ETH capability Transaction message from Bootnode to Node 2, we can break down the following fields under the "Transactions #1" tree as follows:

- Nonce: (0) A number that represents how many transactions the sender node has made.
- Gas Price: (100000000000) The amount of ether the sender is willing to pay per unit of gas.

- Gas Limit: (21000) The maximum amount of gas the sender is willing to spend on the transaction.
- Recipient: (1f0cebf80f05de1213401c6d0a58e215c8ce635f) The address of the account or contract that will receive the ether or execute the function call.
- Value: (100000000000000000000) The amount of ether to be transferred to the recipient (if any).
- Data: (N/A) The input data for the contract function call (if any).
- V, R, S: The signature values that prove that the sender has authorized the transaction.

The nonce field in an Ethereum transaction is a number that indicates how many transactions have been sent from the sender's address starting at 0, which we can see in the above figure. It is used to prevent double-spending and replay attacks on the network. The nonce must be equal to the current number of transactions sent by the sender, otherwise the transaction will be rejected by the nodes. The nonce is incremented by one for each subsequent transaction sent by the same address. This is able to be validated by block validators which iterate over the existing validated blocks and make sure the nonce increments per account transaction.

Next, we can see the gas price of 100000000000, which is the gas price expressed in WEI, which is the smallest unit of Ether, where one Ether is equal to 10^18 WEI. However, gas is usually expressed in terms GWEI, or giga-wei, which means one billion WEI, therefore one GWEI is 10^9 WEI, or specifically 100000000000 WEI is 100 GWEI, which is what was set up in the transaction in MetaMask. Similarly, the transaction value field shows a value of 100000000000000000000 WEI, when converted to ETH, is 100 ETH. This smallest unit method prevents Ethereum from using decimals or fractions in its transactions and ensures that all values are integers.

The recipient of the 100 ETH is shown as 0x1f0cebf80f05de1213401c6d0a58e215c8ce635f, which is the expected same account address found on Node 2. Lastly, taking a closer look at the V, R, and S values which are Ethereum's extended elliptic curve digital signatures used to sign transactions to save storage and bandwidth. These 3 values, coupled with the hash of the transaction, allow the ability to recover the public key of the signer, which is the sender of the transaction. This is done just like in the public key recovery from the signature found in DiscoveryV4. This transaction hash is the same hash that we saw getting broadcasted out by the Bootnode with the NewPooledTransactionHashes message that was sent to the peers that the Bootnode did not send the Transactions message to, specifically Node 1 and Node 3. This dissection of the packet can be seen in Figure 92 below.

```
1919 514.6855221… 10.1.0.10  10.1.1.10 RLPX     162 30303 → 52830 [ETH NewPooledTransactionHashes] T
1920 514.6855690… 10.1.0.10  10.1.3.30 RLPX     162 30303 → 53530 [ETH NewPooledTransactionHashes] T

Frame 1919: 162 bytes on wire (1296 bits), 162 bytes captured (1296 bits) on interface br-84dc88d7a
Ethernet II, Src: 02:42:0a:01:00:0a (02:42:0a:01:00:0a), Dst: 02:42:0a:01:00:02 (02:42:0a:01:00:02)
Internet Protocol Version 4, Src: 10.1.0.10, Dst: 10.1.1.10
Transmission Control Protocol, Src Port: 30303, Dst Port: 52830, Seq: 30479, Ack: 40410, Len: 96
Ethereum RLPx Protocol
▾ Frame Header: 50f7cf47c5cf2f4153e910a46d73a8e239ed5c0ad735a590693b5d863ffc5f97
     Decrypted Header Data: 000029c28080000000000000000000000
     Header MAC: 39ed5c0ad735a590693b5d863ffc5f97
     Frame Body MAC: 05093e50305af10605b76bff9c3de6ae
     Frame Size: 41
     Read Size: 48
     Header Data: Capability ID: 0, Context ID: 0
▾ Frame Body: 456274165b63fa5459c8d73761b3d26481d6d1d36b53094fea8611426c0b3d942797e513…
     Type: [ETH NewPooledTransactionHashes] Type=NewPooledTransactionHashes Code=8
     Capability: ETH
     Code: 8
     Types: 0
     Sizes:
        Sizes #1: 113
     Hashes:
        Hashes #1: 75cc828f236e030954b1b589cfede496baf5ca58c33c8c2553626907bbd38c9a
```

Figure 92: RLPx ETH NewPooledTransactionHashes Message from Bootnode to Node 1

The next message we see is the ETH capability NewBlock message, which is propagated when a new block is created and sent out to be propagated by the other nodes for validation. This dissection is seen below in Figure 93. In this dissected packet, we will only go over a few things related to the transaction that is attached to this new block. First, noticing that Node 3 is the first node that has issued this block to the network, and as it turns out, is the one that created this block. This can be seen by the "Coinbase" field, where it is equal to the address of the account that mined the block, specifically "0x11bee17e6d6835aa46197990adb681ba3a1b4435" which is equal to that of Node 3's account. This would mean that Node 3 is the actual "winner" of this block, which will be added to the blockchain, which will be shown below. The gas used is the exact same as the "gas limit" selected in the transaction as well, followed by tacked onto the bottom of the block in the "transactions" field, the transaction we saw before that was propagated throughout the network.

```
1923 516.7146813… 10.1.3.30  10.1.0.10  RLPX     562 53530 → 30303 [ETH NewBlock] Type=NewBlock Code=7

Transmission Control Protocol, Src Port: 53530, Dst Port: 30303, Seq: 37434, Ack: 25879, Len: 496
Ethereum RLPx Protocol
▾ Frame Header: 24d9d50ff1a653fc3f026f3de703046d8d182d61cd74ad8629ad29483ceaac0c
     Decrypted Header Data: 0001b3c28080000000000000000000000
     Header MAC: 8d182d61cd74ad8629ad29483ceaac0c
     Frame Body MAC: 0ddacbb086d843cb2b3a76bd4ab4f296
     Frame Size: 435
     Read Size: 448
     Header Data: Capability ID: 0, Context ID: 0
▾ Frame Body: f328ac3c8341a444a2d53f0b1624a2c6d465268ea7226a3e71eb8aa81e5bf8b53096fac5…
     Type: [ETH NewBlock] Type=NewBlock Code=7
     Capability: ETH
     Code: 7
     Block:
        Header:
           Parent Hash: de028e23c804e7a77de953db217e8c4b74a384fe8cf14ce4a99bec7f4dada2c6
           Ommers Hash: 1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
           Coinbase: 11bee17e6d6835aa46197990adb681ba3a1b4435
           State Root: 367f6f52e342abe23af179a0d120778c503a39ae3722cfbd42b5bd43ff57bec9
           Txs Root: cd7091331b23981bd4a47480f03e9214952e0c07a4941f27654ea1adb5ff66f8
           Receipts Root: 056b23fbba480696b65fe5a59b8f2148a1299103c4f57df839233af2cf4ca2d2
           Bloom: 0000000000000000000000000000000000000000000000000000000000000000000000000000000000
                  0000000000000000000000000000000000000000000000000000000000000000000000000000000000
                  0000000000000000000000000000000000000000000000000000000000000000000000000000000000
                  0000000000000000000000000000000000000000000000000000000000000000000000000000000000
           Difficulty: 141915
           Number: 159
           Gas Limit: 9342856
           Gas Used: 21000
           Time: 1679443005
           Extra Data: d883010b0084676574688867 6f312e31382e31856c696e7578
           Mix Digest: f3a2d442922cb69c83a4ec36fddeab8a7705657dae0c91ff5011d317ff1224fa
           Block Nonce: 625cacca10f57352
        Transactions:
           Transactions #1:
              Nonce: 0
              Gas Price: 100000000000
              Gas Limit: 21000
              Recipient: 1f0cebf80f05de1213401c6d0a58e215c8ce635f
              Value: 100000000000000000000
              Data: N/A
              V: 24726
              R: 67255722872072334048399793730552662219602420979885419241564245918823877753345
              S: 19015966620058709423904409478455874222869981311016018210026847308481610254 80
        Ommers: N/A
     Total Difficulty: 21695423
```

Figure 93: RLPx ETH New Block Propagation from Node 3 to Bootnode

The same propagation takes place with this "NewBlock" message like the "Transactions" message, where first Node 3 sends it to the Bootnode, followed by Bootnode => Node 2, then Node 2 => Node 1, seen below in Figure 94. Same goes for the "NewBlockHashes" message that is sent out and propagated throughout the network. This also notifies the other nodes that did not receive the "NewBlock" in totality to request more information with the GetBlockHeaders message and the entire block with the "GetBlockBodies".

```
1923 516.7146813… 10.1.3.30  10.1.0.10  RLPX     562 53530 → 30303 [ETH NewBlock] Type=Ne
1924 516.7217920… 10.1.0.10  10.1.2.20  RLPX     562 30303 → 57466 [ETH NewBlock] Type=Ne
1925 516.7270259… 10.1.0.10  10.1.1.10  RLPX     162 30303 → 52830 [ETH NewBlockHashes] T
1926 516.7147400… 10.1.3.30  10.1.1.10  RLPX     162 52850 → 30304 [ETH NewBlockHashes] T
1927 516.7387651… 10.1.2.20  10.1.1.10  RLPX     562 36864 → 30304 [ETH NewBlock] Type=Ne
```

Figure 94: RLPx ETH New Transaction Block Propagation Throughout the Network

The last thing to make sure is to track if this block actually made it into the chain, meaning, it has been mined and validated. This can be done by taking a look at the block hash found in the "NewBlockHashes" message, seen in Figure 95, and comparing it with the subsequent "NewBlock" message, as in the one that comes after this transaction block. Take a close look at the "Block Hash" field in the bottom, right before the block number "159". When looking at the next "NewBlock" message dissected, seen in Figure 96, this same block hash is now listed as the "Parent Hash", as in the preceding block hash, meaning this block has now made it to the full chain.

Thus, showing how a transaction can be tracked and traced throughout the network traffic utilizing the provided dissector as discussed in the previous chapter. All the steps in terms of transaction propagation, validation, then block propagation, and block validation and viewing the block in the actual chain can be seen through the use of the DEVP2P dissector.

```
 1925 516.7270259… 10.1.0.10  10.1.1.10  RLPX     162 30303 → 52830 [ETH NewBlockHashes] Typ

Frame 1925: 162 bytes on wire (1296 bits), 162 bytes captured (1296 bits) on interface br
Ethernet II, Src: 02:42:0a:01:00:0a (02:42:0a:01:00:0a), Dst: 02:42:0a:01:00:02 (02:42:0a
Internet Protocol Version 4, Src: 10.1.0.10, Dst: 10.1.1.10
Transmission Control Protocol, Src Port: 30303, Dst Port: 52830, Seq: 30575, Ack: 40410,
Ethereum RLPx Protocol
 ▾ Frame Header: b7037c52e99a7e415f86ba4f112082c9bf4adbae3156003eed48825ad0382764
     Decrypted Header Data: 000028c2808000000000000000000000
     Header MAC: bf4adbae3156003eed48825ad0382764
     Frame Body MAC: 5ab6b111c0323c9ca976a9a64955e81e
     Frame Size: 40
     Read Size: 48
     Header Data: Capability ID: 0, Context ID: 0
 ▾ Frame Body: d44f5b14046ba4d67b871adf6baf2e2f0716ec6e32c3edcaefc57a57bdbcfca3ae77a9a0…
     Type: [ETH NewBlockHashes] Type=NewBlockHashes Code=1
     Capability: ETH
     Code: 1
     Block Hashes:
        Block Hashes #1:
           Block Hash: ca1b8b41975086179bff804244101142d4532e2d16f883ef2cfc3ce813385627
           Number: 159
```

Figure 95: RLPx ETH NewBlockHashes Message from Bootnode to Node 1

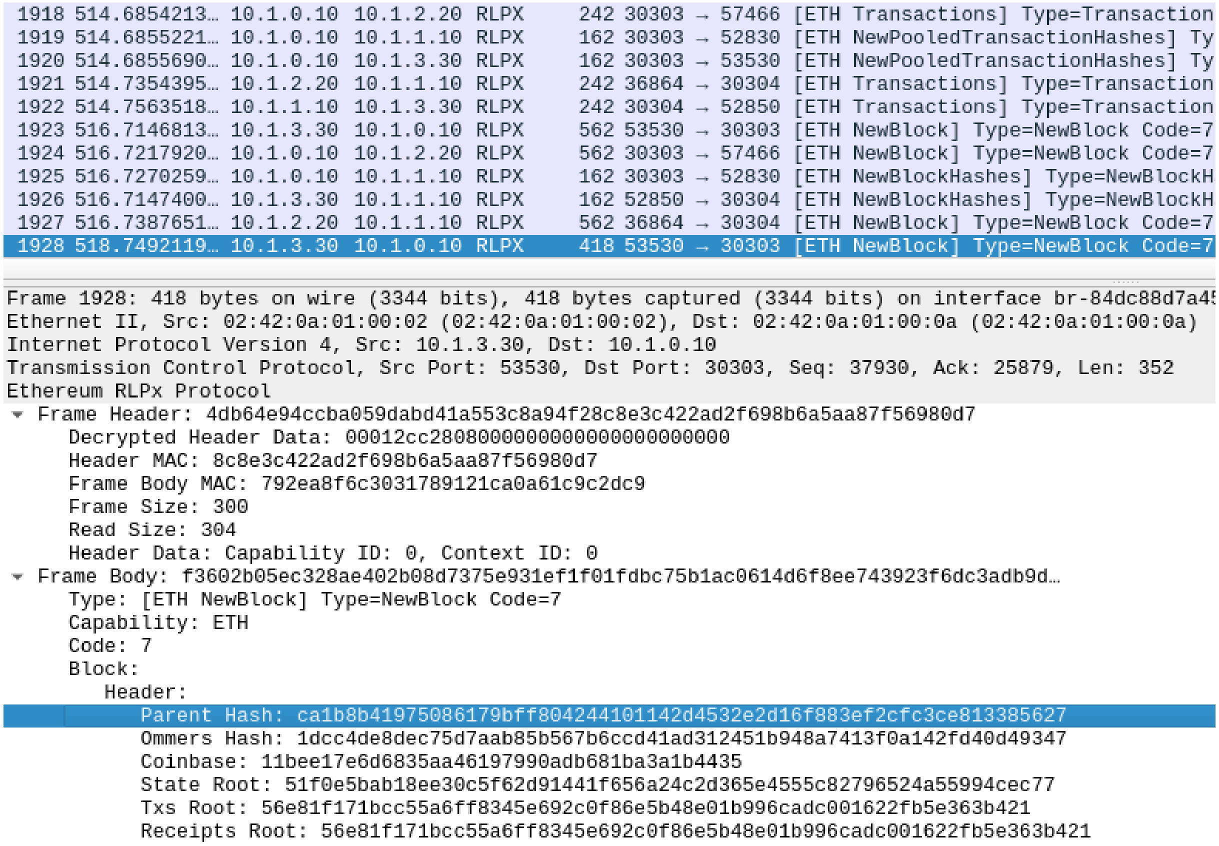


Figure 96: RLPx ETH NewBlock with Previous Block as Parent Hash

6. Conclusion

6.1 Introduction and Recap

This thesis presents a novel approach to creating a Wireshark dissector for Ethereum's DEVP2P suite of peer-to-peer protocols, including DiscoveryV4, DiscoveryV5, and RLPx with the ETH and SNAP subprotocols. As we have discussed, Ethereum networks facilitate intercommunication amongst Ethereum networked nodes, providing for decentralized applications and accounts. Therefore, many contributions were covered to satisfy the requirements for creating a Wireshark dissector plugin that supports RLP decoding and ECIES decryption. First, creating a private Ethereum docker network was discussed, utilizing a custom Go Ethereum source. Next, the actual implementation of the LUA Wireshark plugins was covered; first, the "discovery.lua" plugin supporting the DEVP2P discovery protocols, followed by "rlpx.lua" which allows for the dissection of RLPx, including the ETH and SNAP subprotocols. Then we dug deeper into PYDEVP2P, the python-based, minimal third-party dependency library providing tools for RLP decoding, ECIES decryption, and dissection helper function. Lastly, the technical details behind Elliptic Curve Digital Signature Algorithm (ECDSA), Elliptic Curve Diffie-Hellman, are analyzed utilizing the dissector and PYDEVP2P.

Wireshark is a widely used network analysis tool that allows users to inspect and decode network packets. However, Wireshark does not support Ethereum's DEVP2P protocols natively, which limits the ability of researchers and developers to monitor and understand the behavior of Ethereum nodes. On top of this, Wireshark supports the use of dissector plugins, which are add-ons to Wireshark's dissection capability. As we discussed, the current two dissector plugins provided by BCSEC Org and ConsenSys do not fully support the latest message structure of DiscoveryV4 and provide zero support for DiscoveryV5 and RLPx. To address this gap, this thesis develops a custom Wireshark dissector plugin in LUA that can parse and display DEVP2P packets in a user-friendly format. The plugin leverages PYDEVP2P, a python-based library developed to assist with decoding RLP (Recursive Length Prefix) and decrypting ECIES (Elliptic Curve Integrated Encryption Scheme) used by DEVP2P protocols.

Furthermore, this thesis creates a private docker network with custom Go Ethereum images that generate real-to-life DEVP2P traffic for development, testing, and analysis purposes. Using this dissector plugin and environment, this thesis demonstrates how the Wireshark dissector can analyze various aspects of DEVP2P packet flow, such as tracking the propagation of a transaction throughout the network, analyzing the DiscoveryV4 Elliptic Curve Digital Signature Algorithm (ECDSA) and DiscoveryV5's use of Elliptic Curve Diffie-Hellman (ECDH). This thesis contributes to the field of blockchain research, Ethereum community members, and educators by providing a practical tool for studying Ethereum's peer-to-peer communication layer and enhancing the transparency and security of decentralized applications.

6.2 Dissection & Analysis Results

This thesis has presented the design and implementation of a Wireshark dissector plugin to dissect DEVP2P's DiscoveryV4, DiscoveryV5, and RLPx protocols. The dissector plugin can decode and display various messages exchanged between Ethereum nodes, either live in a real-time network or after the fact with packet captures. Furthermore, the dissector plugin also supports decoding DiscoveryV5 and RLPx messages using the session keys derived from the handshake process.

The process of creating the dissector and the multitude of the dissector's capabilities is shown in Chapter 4. Specifically, the DEVP2P dissector supports all of the messages found in DiscoveryV4, discussed in Chapter 4.3.1, including Ping, Pong, FindNode, Neighbors, ENRRequest, and ENRResponse. The previous dissectors, including the LUA dissector plugin by BSECORG and the C dissector by ConsenSys, cannot fully dissect the newer message schema for the Ping and Pong messages due to the new "enr-seq" field. These previous dissectors also do not support the newest ENRRequest and ENRResponse packets described in EIP-868, which were added to the protocol in October 2019.

Next, the new dissector supports the latest implementation of DiscoveryV5, discussed in Chapter 4.3.2, including Ping, Pong, FindNode, Nodes, TalkReq, and TalkResp. The previous dissectors did not support this due to the nature of the protocol obfuscating the packet header information and the ECDH handshake to exchange session keys for encrypted communication. However, this new DEVP2P dissector provides all the capabilities to maintain the sessions created amongst known nodes on the network, seamlessly decrypting and deciphering captured network data.

The DEVP2P dissector plugin, can analyze and decipher the authenticated and encrypted communication between Ethereum nodes facilitated by RLPx. This includes the handshake process of creating session keys between nodes using the AuthInit and AuthAck RLPx messages, followed by the built-in RLPx capability "P2P" Hello messages, as shown in Chapter 4.4. The dissector supports the other RLPx P2P messages, Ping, Pong, and Disconnect. As RLPx is used as a TCP transport for multiple capabilities, the dissector can decode, decrypt, and dissect ETH and SNAP, the two main sub-protocols or capabilities under RLPx. These protocols support block propagation, chain synchronization, and transactions, followed by state management and synchronization with SNAP. The dissector supports the 2 RLPx handshake messages, 4 RLPx P2P messages, 13 ETH capability messages, and 6 SNAP capability messages.

The range of messages this dissector supports makes it possible to research, analyze and study the behavior and performance of the DEVP2P protocols. As was shown, this new Wireshark dissector plugin can be used to understand how nodes discover each other using the discovery protocols, establish encrypted connections using RLPx, and exchange information amongst connected peers regarding blockchain status and transactions. The dissector can also reveal the details of the message formats, such as the RLP encoding and decoding, the packet headers and trailers, and the message types and contents. This tool allows for an easily accessible view of the inner workings of DEVP2P and assists researchers, the general blockchain community, and educators in similar fields.

6.3 Limitations & Future Work

The dissector is a novel contribution to Ethereum network analysis, as it is the first tool to dissect all three DEVP2P protocols in a unified and user-friendly interface. The dissector can help researchers and developers understand the behavior and performance of the Ethereum network and identify and mitigate potential security threats. The dissector can also aid educators by making elliptic curve cryptography more accessible in real-world applications while helping develop and test new protocols or features for the Ethereum network while providing a solid foundation for future improvements and extended support for existing protocols.

However, the dissector also has some limitations and drawbacks that must be addressed in future work:

- The dissector requires custom Go Ethereum source code that includes the random private keys generated by each node during the RLPx handshake used for session key sharing and encryption of subsequent packets. The dissector will not support RLPx dissection with official GETH or Ethereum clients.
- The dissector requires Python to be used with PYDEVP2P, which handles the main logic of dissection, decoding and decryption behind the scenes. This adds complexity and overhead to the setup and execution of the dissector, requiring the Python PIP package of PYDEVP2P to be installed, along with LUA and the Lunatic-Python bridge.
- The dissector has an incomplete message bug that causes some messages to be truncated or skipped when they are larger than a certain size. This bug affects the accuracy and completeness of the dissection results when a handshake packet or a malformed packet is captured.
- The dissector does not show the unmasked “authdata” of DiscoveryV5 messages, which contains essential information such as node ID, signature, and the ephemeral public key in a clear human-readable format on the Wireshark display.

As Ethereum and its underlying network continues to grow in complexity and evolve over time, there are some possible improvements or extensions for future work are:

- Utilize the dissector in a proof-of-stake environment to see what DEVP2P protocols and messages are used in an execution client in a proof-of-stake consensus algorithm network. It would be interesting to see which DEVP2P RLPx capability messages are adapted or unused by new protocols for proof-of-stake.
- Dissect LIBP2P and compare it with DEVP2P. LIBP2P is another peer-to-peer networking stack used by Ethereum consensus clients. Comparing and contrasting LIBP2P with DEVP2P in terms of features, performance, and security would be useful.
- Add the dissection of LES, PIP, WIP, and other RLPx sub-protocols. The dissector currently only supports the ETH and SNAP sub-protocols. These other sub-protocols are used for different purposes, such as Light Ethereum Subprotocol (LES) support, Parity Light Protocol (PIP) support, and Ethereum Witness Protocol (WIT). The dissector should be extended to support these sub-protocols as well.
- Investigate network discovery leaking, identified as an issue seen in DiscoveryV4. Network discovery leaking is a problem where Ethereum nodes setup for specific chain/network IDs incorrectly communicate and discover nodes on other Ethereum networks.
- Implement a DiscoveryV4 and DiscoveryV5 denial-of-service (DoS) attack. The DiscoveryV4 protocol is vulnerable to denial-of-service (DoS) attacks that can flood nodes with fake ping or pong messages. These messages consume the bandwidth and processing resources of nodes and may prevent them from responding to legitimate messages. The dissector can help to detect and monitor such attacks by capturing such malicious packets on the network.
- Implement an RLPx Known Plaintext Attack. RLPx protocol uses AES-CTR encryption with a fixed IV (initialization vector) for each message. This makes it susceptible to a known plaintext attack that can recover the encryption key if an attacker knows some plaintext-ciphertext pairs. The dissector can help to avoid this attack by randomizing the IV for each message or using a different encryption scheme.
- Perform Wireshark statistical analysis. Wireshark provides various statistical network traffic analysis tools, such as graphs, charts, tables, filters, etc. The dissector can leverage these tools to perform a more advanced and comprehensive analysis of DEVP2P protocols, such as throughput, latency, packet loss, message distribution, node behavior, etc.

6.4 Final Thoughts

The design, implementation, and analysis of the results of the dissector prove its usefulness to the community, educators, developers, and researchers. The LUA Wireshark dissector plugin and the PYDEVP2P library provide all the tools necessary for educators, researchers, and developers to understand on a deeper level the inner workings of the Ethereum network. This dissector also allows the visualization of popular cryptography concepts, utilizing Elliptic Curve Cryptography

(ECC) while also understanding how Recursive Length Prefix (RLP) encoding. Many hurdles were overcome throughout the creation of this dissector, whether it needed to be updated documentation or the implementation of Ethereum-specific ECIES technicalities. Most of the technical details sprinkled throughout this document were acquired directly from the Ethereum DEVP2P GitHub specification page and the most-used execution client source code, Go Ethereum. As a result, these methods are not easily accessible to the broader Ethereum community, analysts, and, most important, educators. The contributions discussed throughout this document allow for greater accessibility into Ethereum node network communication while also overcoming hurdles noted by one of the largest blockchain technology solution companies, ConsenSys. This includes the Go Ethereum docker network, which can spin up a full-fledged private Ethereum network using a single command, followed by the LUA Wireshark plugin and PYDEVP2P library, which can be installed in a few easy steps. As discussed, the PYDEVP2P library also provides easy-to-understand elliptic curve cryptography implementations, allowing educators and students to get hands-on access to understand such low-level concepts in a real-world environment. This document's primary goal has been to provide all the tools necessary to support future enhancements, provide an easily accessible tool for educators to display ECIES cryptography techniques, and for security analysts to increase the robustness of peer-to-peer blockchain networks further.

7. Appendix

7.1 Supplemental Materials

Please refer to the following list of contributions, submitted along with this document and found online:

- Lua-devp2p-wireshark-dissector - https://github.com/jmkemp20/lua-devp2p-wireshark-dissector
- PYDEVP2P - https://github.com/jmkemp20/pydevp2p
- Lunatic-Python - https://github.com/jmkemp20/lunatic-python
- Go-Ethereum - https://github.com/jmkemp20/go-ethereum
- GETH-Docker - https://github.com/jmkemp20/geth-docker

7.2 Environment Setup

Step 1) Install Wireshark and specific LUA version

```
sudo apt-get update && sudo apt-get upgrade
sudo apt-get install lua5.2 liblua5.2-dev wireshark python3.10
```

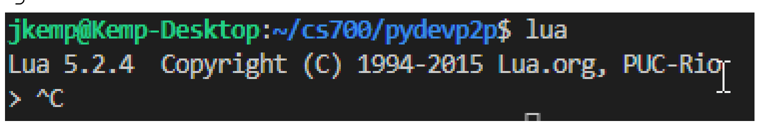


Figure 97: Checking the Version of LUA

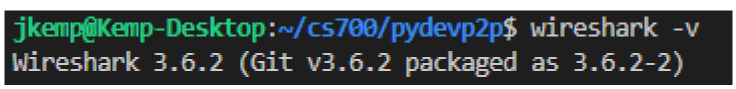


Figure 98: Checking the Version of Wireshark

Step 2) Change permissions and Copy Lunatic Python LUA ⇔ Python Bridge binary

```
sudo chmod +x python.so
cp python.so /usr/local/lib/lua/5.2/.
```

* This mitigates the need to manually build lunatic-python

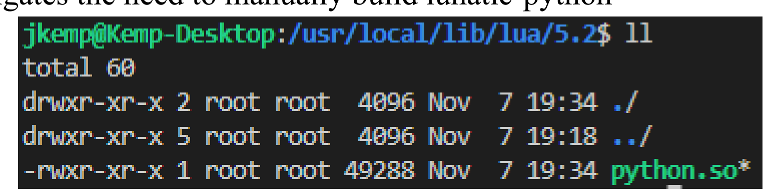


Figure 99: Python LUA Library In LUA Directory

Step 3) Clone both the LUA Dissector and PYDEVP2P

```
    git clone https://github.com/jmkemp20/lua-devp2p-wireshark-
dissector.git
```

```
git clone https://github.com/jmkemp20/pydevp2p.git
```

Step 4) Install the PYDEVP2P PIP Package from source (should also use sudo)

```
cd pydevp2p
pip install -e .
sudo pip install -e .
```

```
jkemp@Kemp-Desktop:~/cs700/pydevp2p$ pip install -e .
Defaulting to user installation because normal site-packages is not writeable
Obtaining file:///home/jkemp/cs700/pydevp2p
  Preparing metadata (setup.py) ... done
```

Figure 100: Installing PYDEVP2P Using PIP

```
Installing collected packages: pydevp2p
  Attempting uninstall: pydevp2p
    Found existing installation: pydevp2p 0.1.0
    Uninstalling pydevp2p-0.1.0:
      Successfully uninstalled pydevp2p-0.1.0
  Running setup.py develop for pydevp2p
Successfully installed pydevp2p-0.1.0
```

Figure 101: Successfully Installed PYDEVP2P PIP Package

Step 5) Next, create the directory for Wireshark plugins (local user and root)

```
cd ~/.local/lib
mkdir wireshark   (if doesn't exist)
cd wireshark && mkdir plugins
cd plugins
```

Step 6) Now, Symbolic link (or copy over) the .lua dissectors

```
sudo ln -s <location of cloned dissector>/rlpx.lua rlpx.lua
sudo ln -s <location of cloned dissector>/discovery.lua discovery.lua
```

```
jkemp@Kemp-Desktop:~/.local/lib/wireshark/plugins$ ll
total 12
drwxr-xr-x 2 jkemp jkemp 4096 Nov 28 00:43 ./
drwxr-xr-x 3 jkemp jkemp 4096 Nov  7 18:16 ../
lrwxrwxrwx 1 root  root    62 Nov 28 00:43 discovery.lua -> /home/jkemp/cs700/lua-devp2p-wireshark-dissector/discovery.lua*
lrwxrwxrwx 1 root  root    57 Nov  7 18:31 rlpx.lua -> /home/jkemp/cs700/lua-devp2p-wireshark-dissector/rlpx.lua*
```

Figure 102: Linking Dissector Plugins in Local Wireshark Plugin Directory

Step 7) Do the same for the Root user (if using Wireshark with sudo privileges)

```
root@Kemp-Desktop:~/.local/lib/wireshark/plugins# ll
total 12
drwxr-xr-x 2 root root 4096 Dec  4 03:56 ./
drwxr-xr-x 3 root root 4096 Nov  8 15:04 ../
lrwxrwxrwx 1 root root   62 Dec  4 03:56 discovery.lua -> /home/jkemp/cs700/lua-devp2p-wireshark-dissector/discovery.lua*
lrwxrwxrwx 1 root root   57 Nov  8 15:01 rlpx.lua -> /home/jkemp/cs700/lua-devp2p-wireshark-dissector/rlpx.lua*
root@Kemp-Desktop:~/.local/lib/wireshark/plugins#
```

Figure 103: Linking Dissector Plugins in Root Wireshark Plugin Directory

* These .lua files can also just be copied directly without needing to symbolically link to them

7.3 Testing with Local .pcapng Packet CaptureFile

* Note, the LUA dissector files register the UDP/TCP ports for discovery and RLPx, these are ports 30303 – 30308, this can be changed directly in discovery.lua and rlpx.lua.

```
wireshark -r final.pcapng
```

```
jkemp@Kemp-Desktop:~/cs700/lua-devp2p-wireshark-dissector$ wireshark -r final.pcapng
nl80211 not found.
Discv5Codec decodeMessage(fromAddr, head, headerData, msgData) No Known Key for 10.1.1.10 Initiate Handshake
Discv5Codec decodeMessage(fromAddr, head, headerData, msgData) No Known Key for 10.1.1.10 Initiate Handshake
Discv5Codec decodeMessage(fromAddr, head, headerData, msgData) No Known Key for 10.1.2.20 Initiate Handshake
Discv5Codec decodeMessage(fromAddr, head, headerData, msgData) No Known Key for 10.1.3.30 Initiate Handshake
[pydevp2p/discover/v4wire/decode.py 42] decodeDiscv4(input): Err Unable To Verify Hash
[pydevp2p/discover/v4wire/decode.py 42] decodeDiscv4(input): Err Unable To Verify Hash
```

Figure 104: Running Wireshark with Captured Packet File

* The above "errors" are normal, these are shown for the first packet in an RLPx handshake and for discv5 packets

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	02:42:0a…	Broadcast	ARP	42	Who has 10.1.1.2? Tell 10.1.1.10
2	0.000018800	02:42:0a…	02:42:0a:0…	ARP	42	10.1.1.2 is at 02:42:0a:01:01:02
3	0.000021800	10.1.1.10	10.1.0.10	DEVP2P	176	30304 → 30303 [DiscoveryV4 PING] Version=4 Kind=1 Len=134
4	0.000048200	10.1.1.10	10.1.0.10	TCP	74	39506 → 30303 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=168456292 TSecr=
5	0.000086100	10.1.0.10	10.1.1.10	TCP	74	30303 → 39506 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=18963
6	0.000095500	10.1.1.10	10.1.0.10	TCP	66	39506 → 30303 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=168456292 TSecr=1896397177
7	0.000127400	10.1.1.10	10.1.0.10	DEVP2P	176	30304 → 30303 [DiscoveryV4 PING] Version=4 Kind=1 Len=134
8	0.000167600	10.1.1.10	10.1.0.10	DEVP2P	133	30304 → 30303 [DiscoveryV5 MESSAGE UNKNOWN/v5] Version=5 Kind=255 RequestID=N/A Len=9
9	0.000281500	10.1.0.10	10.1.1.10	DEVP2P	199	30303 → 30304 [DiscoveryV4 PONG] Version=4 Kind=2 Len=157
10	0.000434800	10.1.0.10	10.1.1.10	DEVP2P	176	30303 → 30304 [DiscoveryV4 PING] Version=4 Kind=1 Len=134
11	0.000487800	10.1.1.10	10.1.0.10	RLPX	523	39506 → 30303 [HANDSHAKE] AUTH INIT
12	0.000517400	10.1.0.10	10.1.1.10	TCP	66	30303 → 39506 [ACK] Seq=1 Ack=458 Win=64768 Len=0 TSval=1896397178 TSecr=168456293
13	0.000572600	10.1.0.10	10.1.1.10	DEVP2P	199	30303 → 30304 [DiscoveryV4 PONG] Version=4 Kind=2 Len=157
14	0.000585000	10.1.1.10	10.1.0.10	DEVP2P	199	30304 → 30303 [DiscoveryV4 PONG] Version=4 Kind=2 Len=157
15	0.000686000	10.1.0.10	10.1.1.10	DEVP2P	176	30303 → 30304 [DiscoveryV4 PING] Version=4 Kind=1 Len=134
16	0.000843000	10.1.1.10	10.1.0.10	DEVP2P	199	30304 → 30303 [DiscoveryV4 PONG] Version=4 Kind=2 Len=157
17	0.000905000	10.1.0.10	10.1.1.10	RLPX	456	30303 → 39506 [HANDSHAKE] AUTH ACK
18	0.000910600	10.1.1.10	10.1.0.10	TCP	66	39506 → 30303 [ACK] Seq=458 Ack=391 Win=64128 Len=0 TSval=168456293 TSecr=1896397178
19	0.001105800	10.1.0.10	10.1.1.10	RLPX	274	30303 → 39506 [P2P Hello] Type=Hello Code=0 Len=160
20	0.001110300	10.1.1.10	10.1.0.10	TCP	66	39506 → 30303 [ACK] Seq=458 Ack=599 Win=64128 Len=0 TSval=168456293 TSecr=1896397178
21	0.001162100	10.1.1.10	10.1.0.10	RLPX	274	39506 → 30303 [P2P Hello] Type=Hello Code=0 Len=160

Figure 105: Viewing Dissected DEVP2P Packets in Wireshark

7.4 Live GETH Docker Startup

Step 1) Make sure Docker and Docker Compose are installed and running

```
jkemp@Kemp-Desktop:~/cs700/geth-private-docker$ docker -v
Docker version 20.10.23, build 7155243
jkemp@Kemp-Desktop:~/cs700/geth-private-docker$ docker-compose -v
docker-compose version 1.29.2, build 5becea4c
jkemp@Kemp-Desktop:~/cs700/geth-private-docker$
```

Figure 106: Checking the Version of Docker

```
jkemp@Kemp-Desktop:~/cs700/geth-private-docker$ sudo service docker status
 * Docker is running
```

Figure 107: Starting the Docker Service on the Host

Step 2) Clone the GETH-Docker Repository

```
git clone https://github.com/jmkemp20/geth-docker.git
```

```
cd geth-docker
```

Step 3) Build the custom docker images

```
./build-dockers.sh
```

* This will create 5 images, one for the “router” and 4 GETH nodes all using the dockerfile.manual file

Step 4) Next, startup JUST the router container

```
docker-compose up -d bridge-router
```

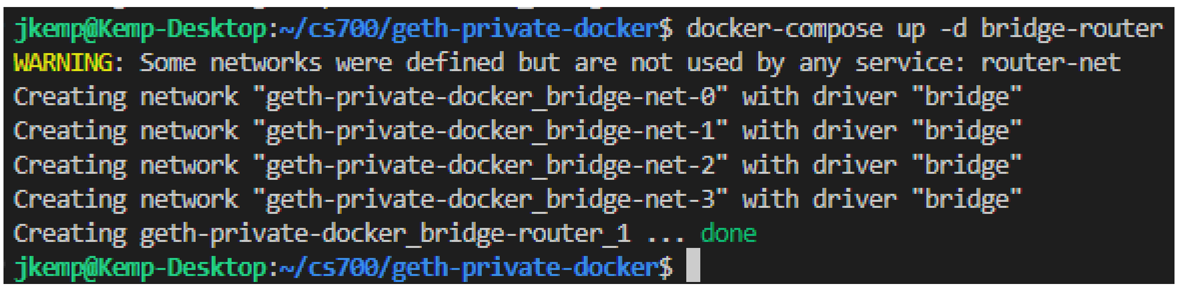


Figure 108: Running the Bridge Router Docker Container

Step 5) Then open up Wireshark and attach to the 10.1.0.1 or any 10.1.X.X network

```
sudo wireshark
```

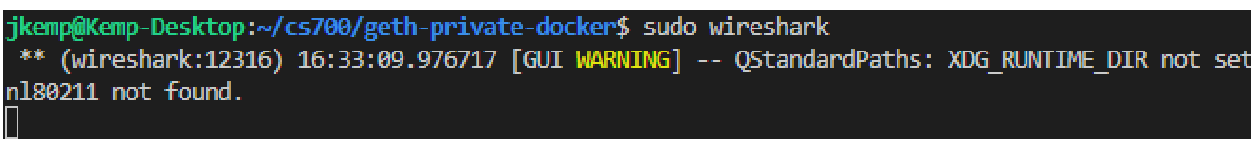


Figure 109: Starting Wireshark to Capture Live Network Traffic

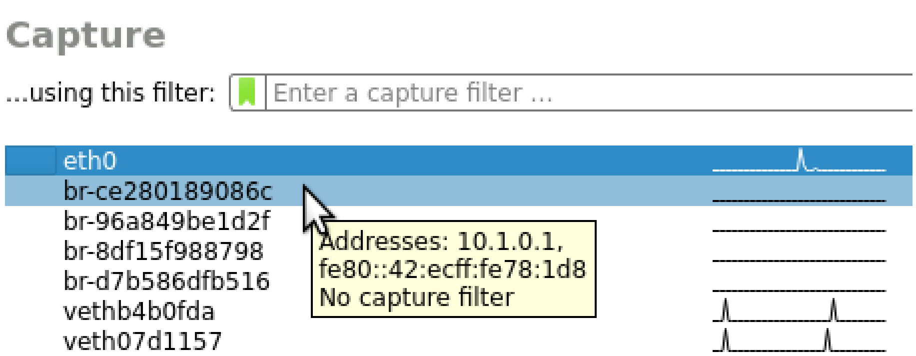


Figure 110: Selecting the Interface to Capture Packets On

Step 6) Finally, start up each GETH node/client container one-by-one

```
docker-compose up -d geth-ubuntu-bootnode
docker-compose up -d geth-client-1
docker-compose up -d geth-client-2
```

```
docker-compose up -d geth-client-3
```

7.5 Installing the Custom GO Ethereum Client from Scratch

Step 1) Clone the Custom GO Ethereum Source

```
git clone https://github.com/jmkemp20/go-ethereum.git
```

Step 2) Install GETH or all GO Ethereum Utilities

```
cd go-ethereum
make geth
make all    # for all utilities
```

Step 3) Run GETH

```
geth …
```

8. Glossary

- Node - A computer that runs software to verify blocks and transaction data on the Ethereum network
- Peer - Another node on the Ethereum network that a node communicates with
- Client - A software application that must be run on a computer to turn it into an Ethereum node
- Execution Client - A client that listens and executes transactions and maintains the latest state and database of all Ethereum data
- Consensus Client - Also known as Beacon Node or CL client, this client implements the proof-of-stake consensus algorithm which enables the network to achieve agreement based on validated data from the execution client
- Peer-to-peer Network - A decentralized network where nodes communicate with each other using standardized protocols
- Proof-of-Work - A consensus mechanism used to validate transactions and add new blocks to the chain. In PoW, nodes on the network compete to solve a complex cryptographic puzzle, and the first node to solve the puzzle is rewarded with ether and the right to add a new block to the blockchain.
- Proof-of-Stake - (PoS): A consensus mechanism used by the Post-Merge Ethereum blockchain that involves validators staking ether to participate in the network and validate transactions. In PoS, validators are chosen to validate blocks based on the amount of ether they have staked, and the probability of being chosen as a validator increases with the amount of ether staked.
- Accounts - (PoS): Digital identities on the Ethereum blockchain that can hold ether and other assets and execute smart contract functions. There are two types: externally owned accounts (EOAs) and contract accounts.
- DEVP2P - A set of network protocols that form the Ethereum peer-to-peer network for execution clients
- LIBP2P - A modular networking stack that enables peer-to-peer communication between consensus clients on the Ethereum network.
- Block - A package of data containing a set of transactions that have been verified and added to the Ethereum blockchain.
- Chain - The sequential arrangement of blocks in the Ethereum blockchain, which creates a decentralized ledger of all transactions on the network.
- Transaction - An operation that modifies the state of the Ethereum blockchain, such as transferring ether (ETH) or executing a smart contract.
- Receipt - A data structure that confirms the successful execution of a transaction on the Ethereum network, providing details such as gas used and contract addresses.
- Elliptic Curve Integrated Encryption Scheme (ECIES) - A public-key encryption algorithm used to securely transmit data between parties on the Ethereum network.

- Elliptic Curve Diffie Hellman Exchange (ECDHE) - A key agreement protocol that allows two parties to securely establish a shared secret key on the Ethereum network.
- Elliptic Curve Digital Signature Algorithm (ECDSA) - A digital signature algorithm used to verify the authenticity of transactions on the Ethereum blockchain.
- ENR - An Ethereum Node Record that contains metadata about a node on the Ethereum network, such as its IP address and public key.
- Node ID - A unique identifier assigned to each node on the Ethereum network, which is used to facilitate communication and routing.
- RLP - Recursive Length Prefix encoding, a compact data serialization format used to encode complex data structures such as Ethereum transactions and blocks.

9. References

[1] "What Is Ethereum? | Ethereum.org." *Ethereum.org*, 2015, ethereum.org/en/what-is-ethereum/.

[2] Stark, Josh, and Evan Van Ness. "The Year in Ethereum 2021." *Mirror.xyz*, 2021, stark.mirror.xyz/q3OnsK7mvfGtTQ72nfoxLyEV5lfYOqUfJIoKBx7BG1I.

[3] Nelson, Matt, and Clarissa Watson. "The State of the Merge: An Update on Ethereum's Merge to Proof of Stake in 2022 | ConsenSys." *ConsenSys*, 2022, consensys.net/blog/news/the-state-of-the-merge-an-update-on-ethereums-merge-to-proof-of-stake-in-2022/.

[4] ethereum. "Consensus-Specs/P2p-Interface.md at Dev · Ethereum/Consensus-Specs." *GitHub*, 26 Sept. 2022, github.com/ethereum/consensus-specs/blob/dev/specs/phase0/p2p-interface.md.

[5] Kripalani, Raúl. "Releasing Wireshark Dissectors for Ethereum ÐΞVp2p Protocols." *Medium*, ConsenSys Media, 14 Aug. 2018, media.consensys.net/releasing-wireshark-dissectors-for-ethereum-%C3%B0%CE%BEvp2p-protocols-215c9656dd9c.

[6] Ethereum Improvement Proposals. "Ethereum Improvement Proposals." *Ethereum Improvement Proposals*, 2023, eips.ethereum.org/.

[7] bcsecorg. "Bcsecorg/Ethereum_devp2p_wireshark_dissector: This Is Ethereum Devp2p Protocol Dissector Plugin for Wireshark." *GitHub*, 12 June 2018, github.com/bcsecorg/ethereum_devp2p_wireshark_dissector.

[8] ConsenSys. "ConsenSys/Ethereum-Dissectors: Wireshark Dissectors for Ethereum Devp2p Protocols." *GitHub*, 24 Aug. 2018, github.com/ConsenSys/ethereum-dissectors.

[9] “The Crypto Wallet for Defi, Web3 Dapps and NFTs | MetaMask.” *Metamask.io*, 2023, metamask.io/.

[10] “JSON-RPC Server | Go-Ethereum.” *Go-Ethereum*, go-ethereum, 2023, geth.ethereum.org/docs/interacting-with-geth/rpc.

[11] “Chainlist.” *Chainlist.org*, 2016, chainlist.org/.

[12] “Networks | Ethereum.org.” *Ethereum.org*, 2023, ethereum.org/en/developers/docs/networks/.

[13] “Nodes and Clients | Ethereum.org.” *Ethereum.org*, 2020, ethereum.org/en/developers/docs/nodes-and-clients/.

[14] “Home | Go-Ethereum.” *Go-Ethereum*, go-ethereum, 2013, geth.ethereum.org/.

[15] “Networking Layer | Ethereum.org.” *Ethereum.org*, 2023, ethereum.org/en/developers/docs/networking-layer/.

[16] Ethereum Foundation. “Eth1+Eth2 Client Relationship.” *Ethereum Research*, 7 Apr. 2020, ethresear.ch/t/eth1-eth2-client-relationship/7248.

[17] Alen Huskanović. “Proof of Work - What It Is and How Does It Work? - Async Labs - Software Development & Digital Agency.” *Async Labs - Software Development & Digital Agency*, 31 July 2018, www.asynclabs.co/blog/blockchain-development/proof-of-work-what-it-is-and-how-does-it-work/.

[18] “Proof-of-Work (PoW) | Ethereum.org.” *Ethereum.org*, 2022, ethereum.org/en/developers/docs/consensus-mechanisms/pow/.

[19] “About ConsenSys | ConsenSys.” *ConsenSys*, 2020, consensys.net/about/.

[20] “Ninja, a Small Build System with a Focus on Speed.” *Ninja-Build.org*, 2022, ninja-build.org/.

[21] "Lua." *Wireshark.org*, 2020, wiki.wireshark.org/Lua.

[22] Carey, Scott. "What Is Docker? The Spark for the Container Revolution." *InfoWorld*, 2 Aug. 2021, www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html.

[23] "Lua." *Wireshark.org*, 2020, wiki.wireshark.org/Lua.

[24] bastibe. "Bastibe/Lunatic-Python: A Two-Way Bridge between Python and Lua." *GitHub*, 2 Nov. 2021, github.com/bastibe/lunatic-python.

[25] "Command-Line Options | Go-Ethereum." *Go-Ethereum*, go-ethereum, 2023, geth.ethereum.org/docs/fundamentals/command-line-options.

[26] Ligios, Andrea. "Introduction to Docker Compose | Baeldung." *Baeldung*, 4 June 2019, www.baeldung.com/ops/docker-compose.

[27] "Devp2p/Discv4.Md at Master · Ethereum/Devp2p." *GitHub*, 6 Jan. 2023, github.com/ethereum/devp2p/blob/master/discv4.md.

[28] "Recursive-Length Prefix (RLP) Serialization | Ethereum.org." *Ethereum.org*, 2020, ethereum.org/en/developers/docs/data-structures-and-encoding/rlp/#:~:text=RLP%20standardizes%20the%20transfer%20of,objects%20in%20Ethereum's%20execution%20layer.

[29] "Devp2p/Discv5-Rationale.md at Master · Ethereum/Devp2p." *GitHub*, 7 Oct. 2020, github.com/ethereum/devp2p/blob/master/discv5/discv5-rationale.md.

[30] "Devp2p/Discv5-Theory.md at Master · Ethereum/Devp2p." *GitHub*, 31 Mar. 2022, github.com/ethereum/devp2p/blob/master/discv5/discv5-theory.md.

[31] "Devp2p/Rlpx.md at Master · Ethereum/Devp2p." *GitHub*, 2 Nov. 2022, github.com/ethereum/devp2p/blob/master/rlpx.md.

[32] “ECIES Hybrid Encryption Scheme - Practical Cryptography for Developers.” *Nakov.com*, 2023, cryptobook.nakov.com/asymmetric-key-ciphers/ecies-public-key-encryption.

[33] “Devp2p/Eth.md at Master · Ethereum/Devp2p.” *GitHub*, 30 June 2022, github.com/ethereum/devp2p/blob/master/caps/eth.md.

[34] Łukasz Żuchowski. “Ethereum: Everything You Want to Know about Gas - SoftwareMill Tech Blog.” *Medium*, SoftwareMill Tech Blog, 14 Nov. 2017, blog.softwaremill.com/ethereum-everything-you-want-to-know-about-the-gas-b7c8f5c17e7c.

[35] “Devp2p/Snap.md at Master · Ethereum/Devp2p.” *GitHub*, 2 Nov. 2021, github.com/ethereum/devp2p/blob/master/caps/snap.md. Accessed 29 Mar. 2023.

[36] “Geth V1.10.0 | Ethereum Foundation Blog.” *Ethereum Foundation Blog*, Ethereum Foundation Blog, 2021, blog.ethereum.org/2021/03/03/geth-v1-10-0.

[37] “Merkle Patricia Trie | Ethereum.org.” *Ethereum.org*, 2023, ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/.

[38] Tabora, Vincent. “Hashing Functions in Solidity Using Keccak256 - 0xCODE - Medium.” *Medium*, 0xCODE, 15 Feb. 2022, medium.com/0xcode/hashing-functions-in-solidity-using-keccak256-70779ea55bb0#:~:text=The%20keccak256%20(SHA%2D3%20family,cannot%20be%20decoded%20in%20reverse.

[39] ochekliye enigbe. “Elliptic Curves and the Discrete Log Problem - Ochekliye Enigbe - Medium.” *Medium*, Medium, 14 Feb. 2022, enigbe.medium.com/about-elliptic-curves-and-dlp-ed76c5e27497.

[40] "Weierstrass Equation of an Elliptic Curve." *Planetmath.org*, 2013, planetmath.org/weierstrassequationofanellipticcurve#:~:text=A%20Weierstrass%20equation%20for%20an,6%20are%20constants%20in%20K%20.

[41] "ECDSA: Elliptic Curve Signatures - Practical Cryptography for Developers." *Nakov.com*, 2023, cryptobook.nakov.com/digital-signatures/ecdsa-sign-verify-messages.

[42] Orion. "CS 463 Lecture." *Uaf.edu*, 2013, www.cs.uaf.edu/2015/spring/cs463/lecture/02_27_ECC_jacobi.html.

[43] Dominiek Ter Heide. "A Closer Look at Ethereum Signatures." *Hackernoon.com*, 16 Feb. 2018, hackernoon.com/a-closer-look-at-ethereum-signatures-5784c14abecc.

[44] "ECDH Key Exchange - Practical Cryptography for Developers." *Nakov.com*, 2023, cryptobook.nakov.com/asymmetric-key-ciphers/ecdh-key-exchange.

[45] "Networking Overview." *Docker Documentation*, 28 Mar. 2023, docs.docker.com/network/.