Custom Fast and Reliable Anonymity Network

Hao Zhang, Vidhi Goel, Venkata Srinivas Chowdary Reddy, Sivaramakrishnan Satymangalam Ramanthan, Gautam Bhatnagar, Mehak Soni
University of Southern California
{zhan849, vidhigoe, reddyv, satyaman, gautambh, mehakson} @usc.edu

***Abstract*- The existing communication systems that use symmetric key or public keys enable users to encrypt their data which mainly includes the payload. SSL, that is most commonly used in the internet, ensures security of the sensitive data communicated between hosts. But securing the host itself is not commonly seen. This project intends to provide security to the data as well as the hosts in an efficient and reliable manner. Being anonymous will enable a user to maintain his privacy, protect his identity and secure his communications.**

I. Introduction

The internet community is quickly changing and evolving as more users are connecting via personal devices like laptops, mobile phones, tablets, smart watch etc. With the advent of Internet of Things which enables physical objects embedded with network connectivity to exchange data, it has become necessary to protect the Confidentiality and Integrity of the data as well as of the physical objects themselves. Securing the identity of these physical objects is important as they contain granular information about the user, his family and households. As described in [4], IOT suffers from vulnerabilities that could be exploited to risk a user's safety by misuse of personal information. To safeguard the privacy of users, we propose the concept of Anonymity which hides the identity of physical objects like their MAC address and the information exchanged between the devices they own.

Fig. 1. Conventional network

Fig. 2. Anonymous network

Anonymity has always been important in real-world societal issues. These issues are becoming increasingly important as more people discover the digital world and find the need for anonymity in this new society. Anonymity is useful for people who want to ask technical questions that they don't want to admit they don't know the answer to, report illegal activities without fear of retribution and participate in

surveys. To provide anonymity, systems use cryptographic mechanisms, application proxies and onion routing [1] so that even an observer sniffing the network packets could not extract the data as well as routing information.

In our system, we intend to provide privacy to the users by hiding their devices and the messages exchanged from those devices. Our system also fulfills the security requirements of Confidentiality through cryptography. We perform Encryption/Decryption of the packet at the end points of each link to make sure that the packet does not travel in the clear using symmetric keys. To hide the identity of physical devices and the routing flow, we encrypt the packet header along with the payload. With this approach, an intruder sniffing the network packets can not learn neither the contents of message nor the routing information like source/destination address, application port numbers, etc.

II. Objectives

Although the name of this paper suggests "Anonymity" as the main objective, we support two other features i.e., Fast & Reliable. All these objectives make our Network one of the Best Custom Anonymous Network. Let's push down a little bit on these objectives as below.

- **Fast**
  In the current generation of Internet, everyone is expecting data transmission to be fast. To achieve this objective we are using custom packet instead of OSI packet that reduces the header size significantly. We are optimizing the routers to process packets and perform cryptographic operations in different threads. There are many more optimizations at routers, senders and receivers discussed later.
- **Reliable**
  Our Internet industry demands reliability along with fastness (throughput). To achieve this demand we designed a retransmission mechanism with the help of sequence numbers to keep track of lost packets. We perform retransmissions in batches to utilize the network resources efficiently. This way we could provide reliability even on highly lossy links.
- **Anonymity**
  This is the most important aspect of our project. Individuals, data centers, cloud systems and other networks that exchange sensitive information require anonymity. We present a completely anonymous solution where hosts send encrypted packets including the routing information. This ensures that an adversary sniffing the packets would not be able to decipher the host addresses, port numbers and other routing information along with the payload. To avoid replay of packets, every link uses a different key to encrypt the packets.

III. Project Overview

The project is designed to achieve anonymous transmission in the network. To demonstrate the same, we transfer files from various hosts to other hosts in the network. This file transfer should accomplish the three objectives discussed above. We also test random packet transfer to test the cryptographic functionalities and analyze the performance. To demonstrate our project we consider a Custom Network consisting of 3 routers and 6 nodes connected as shown below.

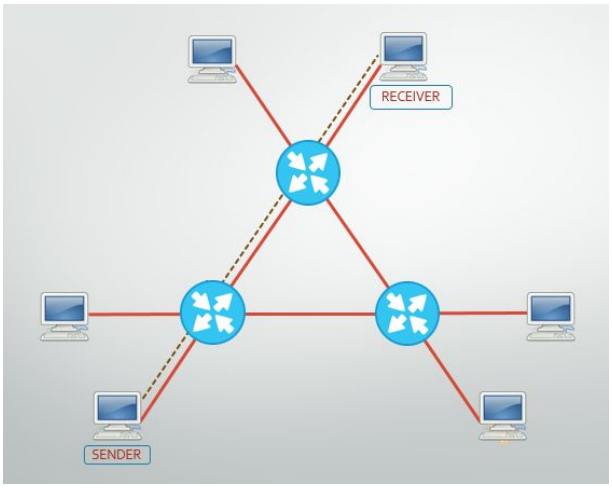

Fig. 3 Custom Network Topology

To understand single file transfer, consider that a file is sent from Sender node to Receiver node. The path followed by packet based on the shortest path algorithm is represented by a dashed line as shown in Fig. 3. Before performing the file transfer, we configure all our nodes with private keys using the Diffie Hellman Key Exchange. Thereafter, the file transfer takes place that involves packet generation, Encryption, Routing & Decryption. We are going to explain each and every step in the subsequent sections.

A. Key Exchange

In our network key exchange happens per link basis. We perform this step in the beginning to make sure all hosts and routers have a unique shared symmetric key based on the link and interface. We do the key exchange using Diffie–Hellman Key Exchange Protocol.

Diffie–Hellman key exchange (D–H) is a public key crypto system that is used to securely exchanging cryptographic keys over unsecured communication channel. It establishes a shared secret between two parties that can be used for secret communication for exchanging data over public network.

It has a fairly simple algorithm where two parties - A and B - agree on two very large numbers denoted by p and g.

- A selects a secret number a and sends X = (g^a *mod* p) to B.
- B selects a secret number b and sends Y = (g^b *mod* p) to A.
- A computes Y^a *mod* p as the symmetric key
- B computes X^b *mod* p as the symmetric key

A and B generate the symmetric key in their local machines and thereby ensuring its Confidentiality and Integrity. The important thing to note here is that even if an adversary acquires the shared X or Y, he can never get hold of the key as the secret numbers a and b never leave the local machines.

Due to its simplicity, Diffie Hellman key exchange has been extended to support Group Key Management [6]. The only drawback with the basic DH is that it does not authenticate the two parties performing the key exchange. A man-in-the-middle could send his public key to a legitimate user. To solve this problem, the practical implementations of DH use Authentication along with public key exchanges. Research work in [7] elaborates on integrating password authentication mechanisms with DH.

B. Packet Format

We are using a custom packet format. The reason why we choose custom packet over OSI packet format is that in OSI packet, we need to provide 3 headers - Ethernet, IP and TCP followed by the payload for reliable communication. Size of ethernet header is 14 Bytes, size of IP header is 20 Bytes, size of TCP header is 20 Bytes. Therefore all these header sizes sum up to 54 Bytes, and form a relatively large overhead, especially for small packets, and since OSI packets are designed to be universal, there are a lot of useless fields regarding to this project. Using OSI packets also means that we need to use system calls and trap into the kernel, which is time consuming.

To avoid the above mentioned shortcomings, we designed our custom packet header that contains all the relevant fields required to transmit data reliably from source to destination, and is also able to support additional functionalities at the same time. We need at least three critical information to perform communication between two host application instances:

- An identifier the destination machine - Destination Address
- An identifier of target application - Protocol
- An identifier of target application instance - Port

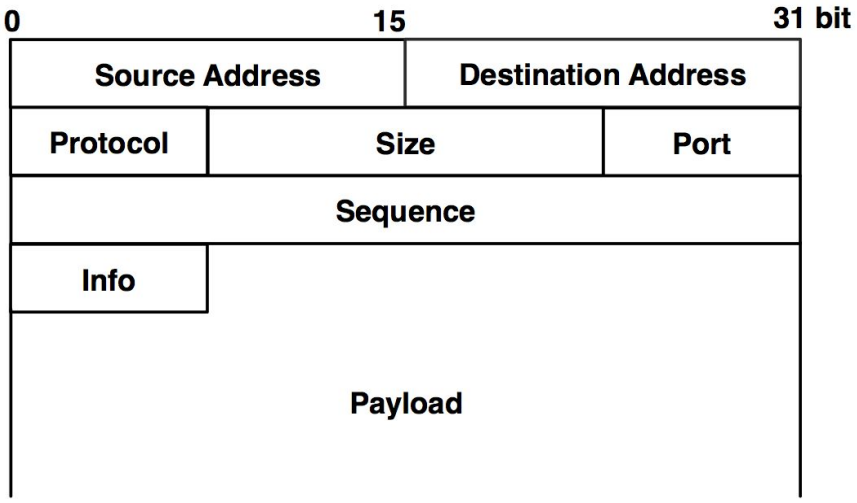

Fig. 4 Custom Packet Format

An illustration of packet header is shown in Fig. 4. We added additional fields to support functionalities such as dynamic routing and reliability. The first four fields serve routing purpose, and the next three fields provide reliability. The total size of our custom header is 13 bytes long which is 76% more efficient that OSI header. For file transfering, we send 1024 byte file segment per packet. Packet header data structures are shown in appendix A.

C. Encryption & Decryption

After Asymmetric Key exchange nodes/routers of each link will be having same Symmetric Key generated. With this Symmetric Key we are going to perform the Encryption of packets using the DES Algorithm. The Data Encryption Standard was once a predominant symmetric-key algorithm for the encryption of electronic data.

The DES has a 64-bit block size and uses a 56-bit key during execution (8 parity bits are stripped off from the full 64-bit key). DES is a symmetric cryptosystem, specifically a 16-round Feistel cipher. When used for communication, both sender and receiver must know the same secret key, which can be used to encrypt and decrypt the message, or to generate and verify a Message Authentication Code (MAC). The DES can also be used for single-user encryption, such as to store files on a hard disk in encrypted form [8].

While DES has 4 modes of operation, we chose to implement our encryption/decryption using CBC mode. To define CBC (Cipher Block Chaining) [8] is an advanced mode of operation which uses ECB encrypted blocks of ciphertext. Following are the important characteristics of CBC mode.

- In this mode of operation, each block of ECB encrypted ciphertext is XORed with the next plaintext block to be encrypted, thus making all the blocks dependent on all the previous blocks.
- This means that in order to find the plaintext of a particular block, you need to know the ciphertext, the key, and the ciphertext for the previous block.
- The first block to be encrypted has no previous ciphertext, so the plaintext is XORed with a 64-bit number called the Initialization Vector, or IV for short.
- So if data is transmitted over a network or phone line and there is a transmission error, the error will be carried forward to all subsequent blocks since each block is dependent upon the last.
- This mode of operation is more secure than ECB because the extra XOR step adds one more layer to the encryption process.

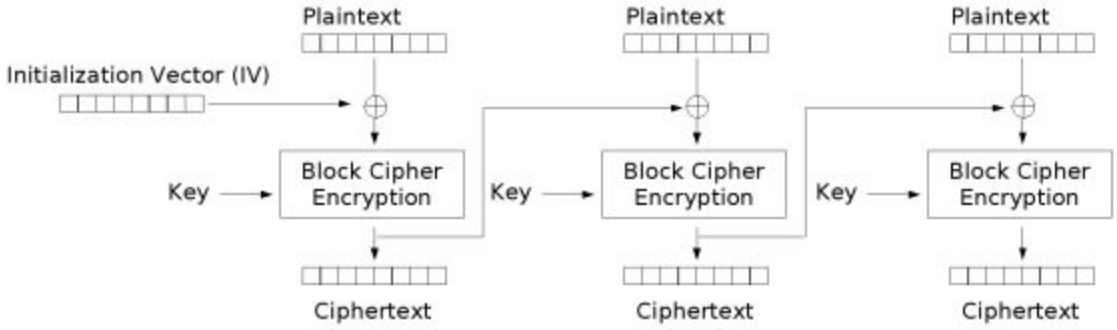

Fig. 5 DES CBC mode of encryption (Joshua Thijssen, 2010)

DES-CBC has lower performance than DES ECB as shown in Fig. 6 and Fig. 7. But it ensures higher security as it utilizes the chaining method where the cipher output of a block is used as input for encrypting the subsequent block. This makes the cryptanalysis of ciphertext very difficult, almost impossible.

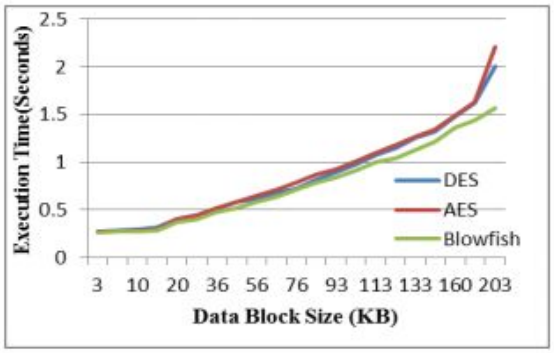

Fig. 6. Performance with ECB (source [9])

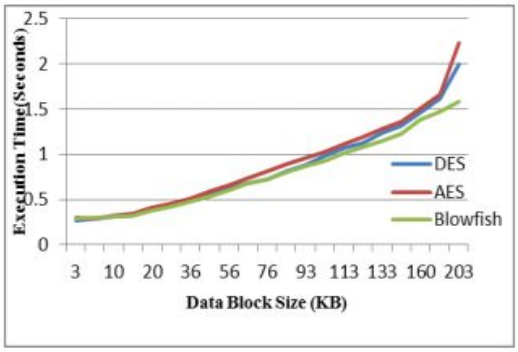

Fig. 7. Performance with CBC (source [9])

We utilize DES-CBC mode to encrypt the packets at sender and at every hop before retransmitting. We are using OpenSSL implementation of cryptographic algorithms in our project. It is an open source library that supports most of the security protocols and is cross platform with support on Linux and Windows based systems. Decryption is similar except that it takes ciphertext as the input and uses the key and IV to generate the plain text.

D. Router

Router runs in userland. It has one thread dealing with packet coming from every ethernet interface. Each thread has a sniff handler looking for any packet coming in. When a packet arrives from network interface, the sniffer brings it up to userland for processing. The first operation we perform at user space is decrypting the packet with the symmetric key of that link. Then the thread classifies the packet. Packet coming from control network should be filtered out; packets from end host applications need to be forwarded (i.e. file transfer); packets containing routing information query from other routers should be used to generate routing information reply. This step can be extended to support any additional functionality we might add to the network. If the thread need to forward the packet, it goes through the routing table and find out which interface this packet needs to be injected to. The routing table always contains the route with lowest cost. At this point the packet is ready to be forwarded. To forward a packet, the thread needs to fetch the injection handler of the interface, and use the injection handler to deliver the packet directly onto network interface from userland. Finally the network interface sends the packet out.

Figure 8 illustrates the router's workflow for packet forwarding:

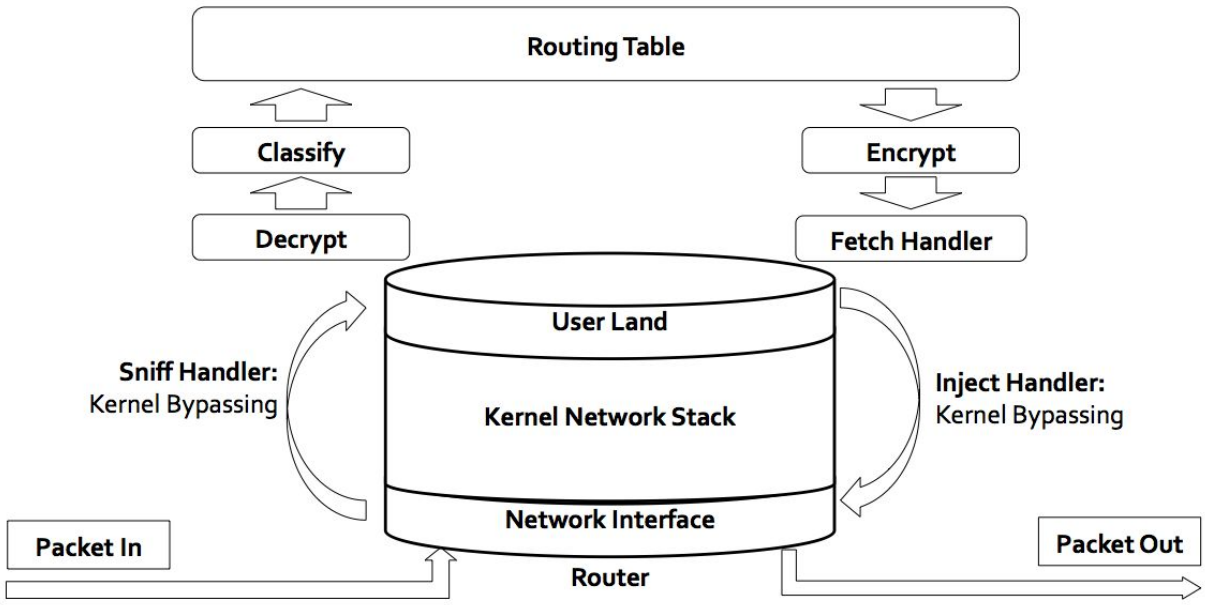

Fig. 8. Router Workflow for Forwarding Packet

E. Sender

The primary responsibility of the sender node is to break a file into packets and send it reliably. To do so, a sender node reads file chunks equivalent to one packet size at a time, constructs a packet with our custom header and payload and sends it out using raw sockets. On the sender side, we have a multithreaded-program, consisting of a sender thread and a receiver thread. The sender and receiver thread communicate with each other over a shared memory buffer, maintained in the form of an array.

Sender Thread- this thread is responsible for sending out file packets. Every time the sender thread sends out a packet, it marks the sequence number of this packet in the shared array to indicate that this packet had been sent out. The sender thread continuously looks at this shared array to know which packets it needs to send, or re-transmit in case they were lost.

Receiver Thread- this thread is responsible for receiving NACKs from the receiver and updating the shared array by marking packet sequence numbers that were not received by the receiver node.

F. Receiver

The primary responsibility of the receiver node is to receive the packets, re-order the packets of a flow according to sequence numbers and write them to the appropriate position in the file. On the receiver side, we have a multithreaded-program, consisting of a sender thread and a receiver thread. The sender and receiver thread communicate with each other over a shared memory buffer, maintained in the form of an array. This array is used to mark which packets have been received by the receiver and also to understand what packet sequence numbers to include while sending out NACKs to the sender node.

Sender Thread- this thread is responsible for going through the shared array and using the values in the shared array to construct NACK packets. We use a cumulative NACK mechanism, where each

NACK packet consists of a bit sequence and each bit represents whether a packet was received by the receiver node (1) or the packet was not received (0). Using such a bit sequence as the payload, we can ensure that we can represent approximately 8000 packets using just packet packet. The sender thread constructs NACK packets and sends them back to the sender node.

Receiver Thread- this thread is responsible for receiving file packets, validating that it is our custom packet and then using the packet sequence number to ensure that the content is written to the appropriate position in the file. The receiver thread then marks this packet sequence number in the shared array to indicate that it has been received.

G. Dynamic Routing

Since a fixed routing table would be harmful when some malfunctioning of the network such as broken link happens, the router needs to know the network condition and update its own routing table dynamically. Dynamic update of routing table is achieved on the router with the following state machine:

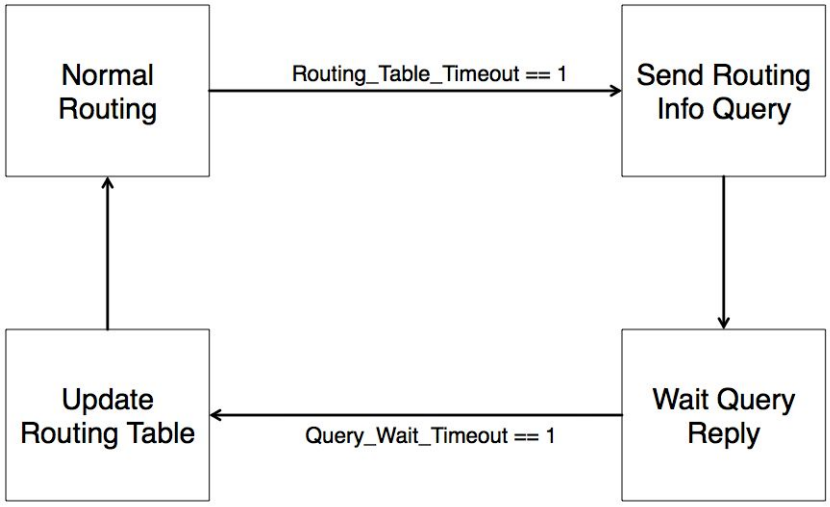

Fig. 9. Dynamic Routing State Machine

In addition to packet process threads, there is an additional scheduler thread running. The router's routing table is shared by all sniffer threads and the scheduler thread. Each routing table entry contains information of destination subnet address, interface to forward relative packets, and cost of this interface. Only the interface with lowest cost will be recorded, and therefore there is only one entry in the routing table for each destination subnet.

This router's routing table will expire after certain amount of time. After the routing table expires, the scheduler thread starts sending out a series of routing info queries on every ethernet interface that connects to a different router, and waits for query reply. The number of routing info query is configurable and the scheduler thread uses it to calculate the cost of the link. Routing information replies are received by the sniffer thread and stored in a piece of shared buffer so the scheduler thread can see it. Only the newest routing information reply is stored. On the other side, the if the other router receives a routing information query, its sniffer thread will send back its routing table as routing information reply.

Scheduler thread proceeds to update routing table after the waiting is timed out. For every query scheduler thread sends out, there is expected to be a query reply. If not every query is replied, the link is considered either to be lossy or to have a long delay. Though we do not distinguish between these two possibilities, both cases would indicate this link has a high cost. We calculate link cost using the following equation:

$$cost = default\ link\ cost + \frac{\#\ of\ routing\ info\ replies\ missing}{\#\ of\ routing\ info\ queries\ sent} \times add\ on\ cost$$

, in which both default cost and add-on cost are reconfigurable. The scheduler thread add the link cost to the cost to destination from the routing table received. If the cost to destination is less than than of the one in the router's own routing table, the scheduler thread updates its own routing table entry. Finally the scheduler thread goes back to normal routing.

There is no mutex to secure the routing table, which means it is possible that one sniff thread looks up the routing table while the scheduler is updating it. Based on the fact that link conditions does not change very frequently, if accessing routing table needs to lock the mutex, sniffer thread cannot read routing table while scheduler thread is checking it, even when there is nothing to be changed to the routing table. In addition, with mutex protection, sniffer thread also cannot access the routing table in parallel, although all sniffer threads only read the routing table. These two facts will severely influence the router's performance, and therefore we choose rather to route some packets using possibly outdated routing information.

IV. Experiment Setups

A. Symmetric Key generation using DH

We used Diffie Hellman key exchange as it uses a mechanism where the symmetric key is never exposed to the network. Below are the steps that we follow to perform the key exchange per link:

1. Read the large numbers 'p' and 'g' from a file. This file is same for both the exchanging parties and saved in their local machine
2. Generate the public key and the secret key at both parties locally.
3. Each party sends the public key to the other one
4. After receiving the public key, both parties generate the same symmetric key.

We used OpenSSL library to implement the above mentioned steps. Please find more details about DH implementation using OpenSSL library in Appendix B. Fig. 10 depicts the live execution of generating symmetric key on a host and a router.

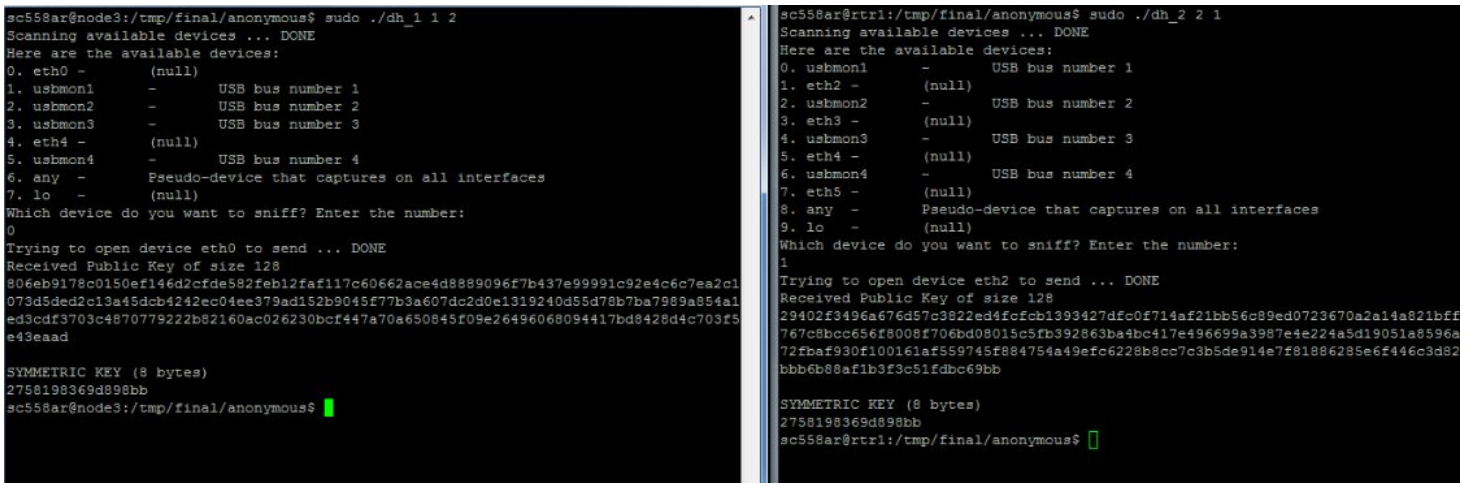

Fig 10. Symmetric key generation using DH

B. Encryption/Decryption

Here, we would present how a single packet flows from one host to another via router by going through decryption and encryption at every stage. The sender reads a small file, encrypts it with his link key and sends it to the router. The router which is already sniffing at all its interfaces connected to the hosts, will receive the encrypted packet, decrypt the entire packet using the shared link key and process it as explained in section III. It would again encrypt the packet with the link key it shares with the destination host and inject it to the appropriate interface. The encrypted packet will again travel through the network securing the header and the payload and would reach the destination, where the host decrypts it to access the payload. Please follow Appendix B to understand how we performed encryption/decryption using OpenSSL library.

Figure 11. shows the encrypted and the decrypted packet at Sender and Receiver. Important thing to note here is that the encrypted packet looks different at sender and receiver. This is because they communicate via router at two different links - one link connects sender to the router and the other link connects router to the receiver. The packet seen at the Left is encrypted using the sender - router key, while packet seen at the right is encrypted using router - receiver key.

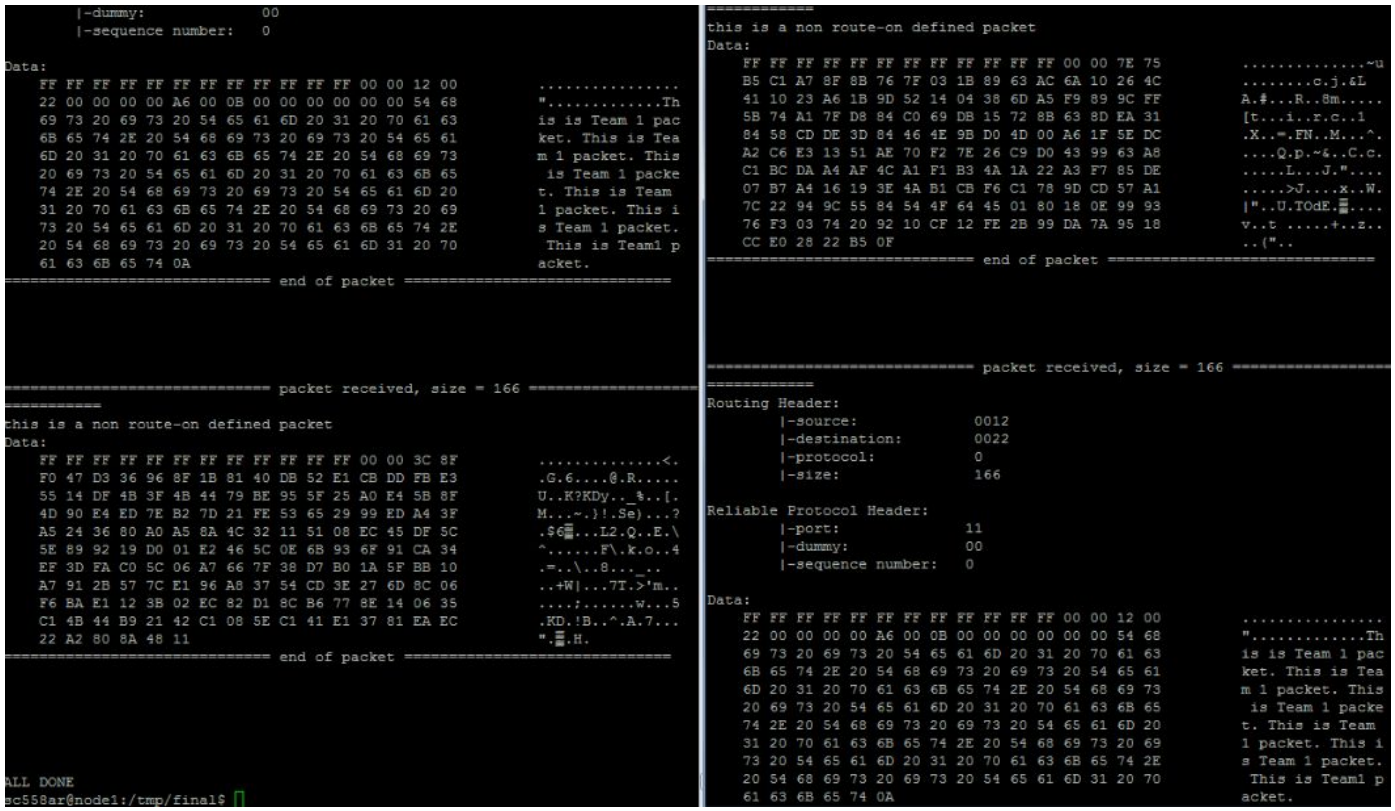

Fig 11. Packet in Encrypted/Decrypted form at Sender[Left] and Receiver[Right]

C. Full Duplex Support

We implemented Full duplex transmission using the round robin mechanism. Each host sends a file to two other hosts in different subnets via router. The sender would send packets to different hosts in a time sharing manner while sleeping for short intervals like 100 micro seconds between two successive transmissions. A brief delay ensures that the sender does not flood the router. To differentiate between two senders, we use a field called “port” in our custom header. Each connection from one host to another host is identified by the port field. On the receiver end, the machine receives packets from two different hosts. As already stated it would use the *port* field to differentiate between senders and save the payload in the respective file.

To demonstrate this functionality, we run three senders and three receivers simultaneously. All the links are already configured as shown in sub-section A. We start our router first to route packets from one host to another. Then, we start the receivers at all the 3 nodes so that they are ready to sniff packets at the respective interfaces. Finally, we start our senders simultaneously to start the full duplex transmission.

Figure 12. depicts the full duplex transmission with all senders on the top and all receivers at the bottom. Some of the most important observations are as follows. Firstly, Receivers don’t process any packet that does not comply to our header format. (The same happens at the router too). This improves the performance of individual nodes and overall performance of the system as well. Secondly, the receivers receives packets from alternating ports 11 and 12. This means that the router is processing the packets being sent from all the senders at the same time. Also, it depicts the usability of the round robin mechanism that we used at senders.

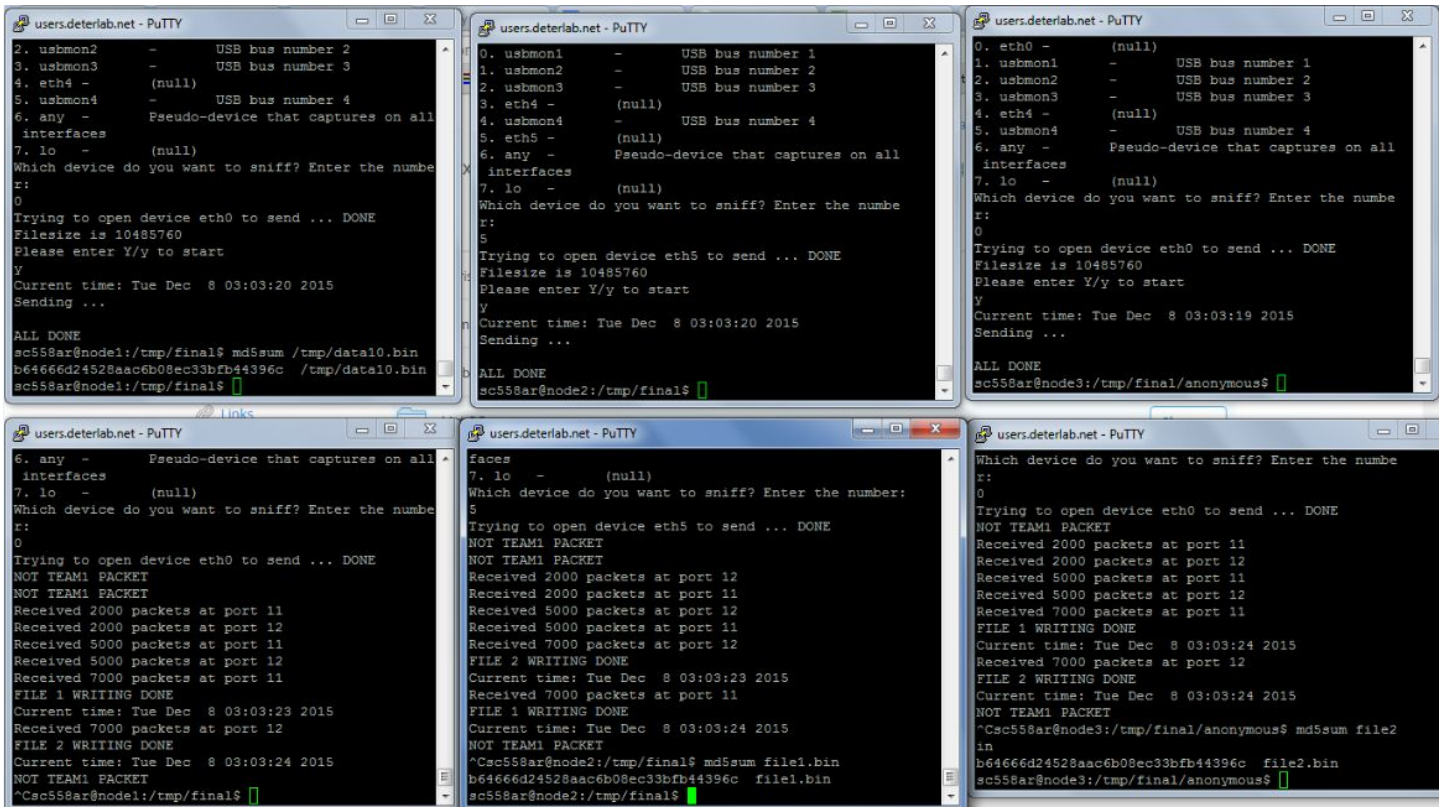

Fig 12. [Top] Senders for host1, host2 and host3 from left to right
[Bottom] Receivers for host1, host2 and host3 from left to right

Thus, we have successfully demonstrate the Full Duplex transmission capability of our system by exploiting time sharing and parallelism.

D. Max Performance

In this experiment we set up a 9-node experiment as shown in Figure 3. We first deploy sender and receiver on two host nodes that are not connected to the same router. Then we configure router / node addressing, and routing table as well (since we are using our own addressing mechanism). Router will be set to PERFORMANCE mode. Receiver is started prior to sender and wait for the file. A script is written to start the sender (see Appendix G): it takes a timestamp right before sender starts, and another timestamp right after sender ends. The difference between the two senders is the time taken (in nanoseconds) to transfer the file. The file size is 500MB. The link in the network has 1000 *Mbps* bandwidth, 0 *ms* delay, 0% packet loss.

E. Dynamic Routing

In this experiment we have same setup as Max Performance experiment. The only difference is that the router is set to DYNAMIC mode. Under this mode, the router is going to print out its routing table every time after it finishes sending routing info query and updating its own routing table. For demoing purpose, we set the router's routing table timeout to be 5 seconds, and therefore the router is going to exchange routing information with other router every 5 seconds. After the sender and receiver

start to transfer the file, we break up one link by setting link loss equal to 1, which means 100% packets on this link will be dropped. During the next routing information cycle, it is expected to observe that the router received 0 out of 5 routing information query reply on the broken link, and in the routing table , destination with injection interface onto the broken link will have their injection interface changed, and the corresponding link cost should also be changed.

V. Optimizations

After applying cryptography in our system, the performance decreases drastically due to the expensive cryptographic operations. The basic functionality needs to be upgraded to increase the performance of our system with OpenSSL. OpenSSL has a low processing speed of ~30511 kilobytes/second on a Intel(R) Atom(TM) CPU D525 @ 1.80GHz [11]. This open source library is a hard limitation for our system. To reach the maximum throughput provided this limitation, we performed best possible optimizations at router, sender and receiver and achieved our target.

Below is a description of our optimization mechanisms.

A. I/O Optimization

- **File Reading-** While sending the data we are reading a chunk of 1024 (payload size) bytes from the file. This operation can be performed via 5 methods.
  - fgetc : reads 1 bytes character into buffer from the specified file stream in a loop till 1024 bytes are read.
  - fread : reads a chunk of 1024 bytes into buffer from the specified file stream.
  - getc : low level system call to read 1 byte of data from specified file descriptor in a loop till 1024 bytes are read.
  - read : low level system call to read a chunk of 1024 bytes into buffer from the specified file descriptor.
  - mmap read : memory maps the source file into memory.

We devised an experiment to check the execution time of these operation on 500 MB file. The following graphs shows the comparison between them:

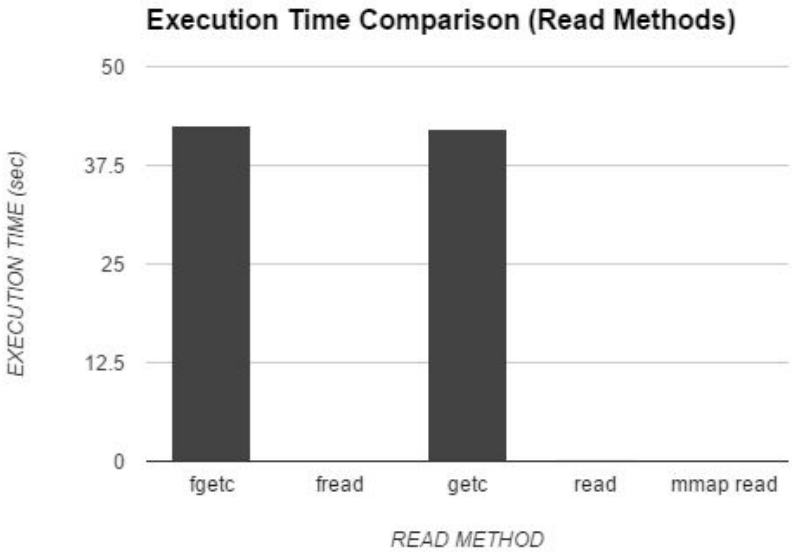

Fig. 13. Time Comparison for Read Methods

- **File Writing-** Writing the file involves reading the payload from the data packets and writing the payload to the target file. This can be done via 3 methods.
- fwrite : Writes the 1024 number of bytes to the specified file stream.
- write : Low level systems call to write specific amount of bytes (1024 in our case) to the specified file descriptor.
- mmap write : Writes the file memory mapped into the memory to the specified target file.

We devised an experiment similar to file reading section to check execution time to read+write 500MB of file. The following graph shows the comparison between them.

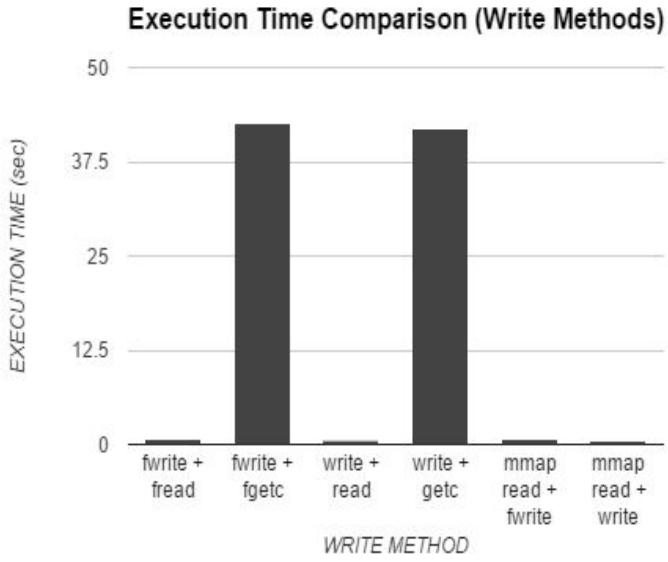

Fig. 14. Time Comparison for Write Methods

Although, write + read has lower execution time (in most cases) , the bare write() + read() are not stable. The execution time for read + write calls in worst case is around 4.5 sec. That is the reason we choose to implement with fwrite + fread combination.

- **Packet Sniffing**

We looked at different ways of sniffing packets to make it as fast as possible.

One way is to use the kernel network stack, but there are multiple factors that make it very slow. This is because it involves multiple system calls, packet sizes are large and we have to go through all the network layers. Due to all these reasons, we decided to go with other approaches that are faster.

Another approach involves implementing raw sockets in the traditional way, using a PF_PACKET flag. The problem with this approach is that it still involves system calls and has a limited buffer size, which means that we cannot process a large number of packets. Since we want to sniff a large number of packets and we want to ensure that it is fast, we decided to look at a more efficient implementation using raw sockets.

Raw sockets with a PF_RING flag [15] overcomes the limitations of using traditional raw sockets. It uses MMAP to map the memory directly to user space, due to which the sniffer is always sitting and waiting for incoming packets (and we avoid system calls). Also, this implementation has a configurable ring buffer, which means we have the memory to process more packets. Since PCAP is an efficient implementation of raw sockets with PF_RING, we use this approach to sniff packets faster. A throughput comparison between PF_PACKET and PF_RING is shown in Table 1.

TABLE I A THROUGHPUT COMPARISON BETWEEN PCAP AND RAW SOCKET WITH PF_PACKET[16]

Packet Size (Bytes)	**Raw Socket with PF_PACKET (Mbps)**	**PCAP (Mbps)**	**Improvement (%)**
64	6.81	7.97	**17.03**
128	95.95	99.30	**3.49**
256	173.68	274.10	**57.82**
512	327.57	393.53	**20.14**
1024	483.48	576.25	**19.19**
1514	885.65	984.43	**11.15**

We can see from the table that PCAP has about 20% improvement over a normal implementation of RAW socket with a packet size of 1024 bytes.

B. Encryption and Decryption Optimization

Although serial encryption and decryption using OpenSSL are slow, there are still space for optimization. First, encryption and decryption are computation intense, which means the performance is

computation resource bounded. Second, encryption and decryption processes have great memory locality which means such processes only operates on small memory segments. Finally, since OpenSSL encrypts and decrypts data block by block individually, there is no data dependency. With these characteristics, we can speed up encryption and decryption using parallelism. We use OpenMP (Open Multi-Processing) to parallelize encryption and decryption.

OpenMP is a high level thread based API with compiler directives which means it's easier to implement. It is thread based and is optimized particularly for shared memory parallel computation. OpenMP also supports data parallelism with certain level of hardware support based on the available architecture. Even though encryption and decryption can be implemented using pthread, as pthread is a general-purpose parallel tasking, OpenMP has it's advantages in doing parallel computation specifically. A sample implementation of OpenMP data parallel processing can be found in Appendix D.

C. Compiler Optimization

The code written does contain a lot of inefficiencies in the form of loops to read multiple segments of the file, check file receiving sequence, generate file retransmission packets; we also have branches to classify the different types of packets, decide whether to write the received packet into file. One of the solutions to overcome this deficiency is to use compiler flags. Compiler flags makes a lot of optimisations during compile time. For example, a common practise to increase the execution time of the program is to add large loops which does nothing but to add delay in the program. The compiler will ignore this unnecessary loop to make the execution faster if the compiler flags are active.

The compiler flags we turned on would optimize the code in ways such as loop unrolling, instruction reordering, short function inlining. Loop unrolling hardcode loop instructions same amount of time as it would be executed:

Before loop unrolling:

```
for (i = 0; i < 5; i++) {
      A[i]++;
}
```

After loop unrolling, the above code segment becomes:

```
A[0]++;
A[1]++;
A[2]++;
A[3]++;
A[4]++;
```

As loop counter `i` is hardcoded into instruction, it reduces CPU and memory cycles to load `i`, increment `i`, and compare `i` with `5`. It can save a lot of CPU cycles especially when the loop time is large. Instruction reordering is based on compile time branch prediction. Instead of waiting for condition to be generated, the code of one branch is executed no matter what the condition would be, and it starts over when the branch prediction is wrong. Though it would not save time when the prediction is wrong, it do save waiting time when the prediction is correct. Short function inlining automatically make short

functions inline by default, which means, to execute these short functions, CPU does not even need to use instructions such as `jmp` and `call` to jump to other instruction cache locations, or even link to other libraries when the function is called, because the next available machine instruction is already the target function, and is write in the CPU pipeline, ready to be executed. These compiler flags will also turn on optimizations based on the available hardware architecture, such as using advanced CPU instruction, stack optimization.

There are different levels of compiler optimisations, which can be specified by -Ox when compiling objects. For our implementation we have used -O2 and not -O3, as -O3 is not thread safe and includes flags such as `-ftree-loop-distribute-patterns`, which would break operations in a single loop and distribute them onto multiple processor cores. This degree of optimization is dangerous as it may cause unintended race conditions, if there are possible data dependencies that are not resolved during compilation time.

D. Logic Optimization

To improve performance on the sender and receiver side, it is important to ensure that we do not overload the network with too many packets at any time. It becomes important to send packets at the appropriate time and also to reduce the number of packets in the network.

- **Reduce packets in the network-** One of the optimizations we did was to use cumulative NACKs by constructing a bit sequence. Each bit in this sequence represents whether a packet was received (1) or not (0). By doing so, we represent approximately 8000 packets using a single NACK packet. For larger file sizes, this helping in an enormous reduction in the number of NACK packets that are sent over the network. As an example, assuming a 1GB file that is to be transferred, we would have about 1 million file packets that need to be sent to the receiver. Assuming that the routers in the network drop about 20% of the packets, we would lose 200,000 packets. Instead of sending 200,000 NACK packets, we can represent all 1 million packets with less than 150 NACK packets. This means that we are able to achieve a 99% reduction in the number of NACK packets sent over the network.
- **Tune timing for NACK packets-** It is very important for the receiver node to send back NACKs at the appropriate time. If the NACK packets are sent too often, it will congest the network and the sender node might be re-transmitting packets that have already been sent, but took a little longer to reach the receiver due to router processing time. If the receiver node waits too long to send the NACK packets, then the sender might have already sent out all packets and would just be waiting idle- this would definitely result in a suboptimal throughput. To ensure that the NACK packets are sent out at the right time, we have developed an algorithm that calculates this optimal time:

  Step 1: the receiver node uses the shared array to calculate the number of packets that are being sent by the sender node

  Step 2: the receiver node uses the number of packets being transmitted by the sender node to calculate the time it needs to wait before sending back NACK packets. This is easy to calculate because we already know how long it takes for the sender and each router to process 1 single packet.

  Step 3: after waiting for the appropriate amount of time, the receiver calculates the bit sequence and sends back the NACK packets.

VI. Performance

A. *Max Performance with PCAP*

We implemented the router with PCAP, added relative support and tested performance using iperf, with 1500 byte packet. Packets are neither encrypted nor decrypted during this test. The throughput of the router over 1000Gbit link is 954 *Mbps*. Here is a breakdown of data process latency:

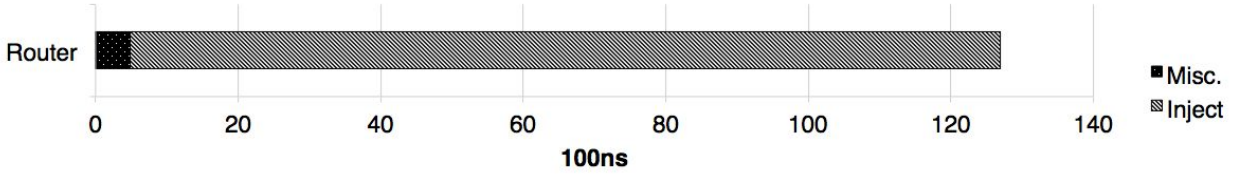

Fig. 15. Latency Breakup of Operations iperf packet support (1500 bytes per packet)

In this process, packet processing takes about 487 *ns*, as iperf is much more difficult to support than our self-designed functionality; and packet injection takes on average 12209 *ns*. We proved that the router architecture can reach line rate using PCAP implementation.

B. *Performance Breakup without Optimization*

Let's have a closer look at the performance aspect of our Custom Anonymity Network. With Anonymity our packets are getting encrypted and decrypted at every node. To determine the further areas of improvements we did a performance breakup. Here is a breakdown of all time-consuming operations to send one file segment from sender to receiver without any optimization. Reading file segment of size 1024 bytes takes 264 *ns*; encryption takes 15625 *ns*; decryption takes 15429 *ns*; injecting file segment into the network takes 8112 *ns*; writing file segment to receiving file takes takes 2145 *ns*. Additional sender miscellaneous logics, including file sequence recording, missing sequence checking, packet generating, takes 4609 *ns*; while that of receiver takes 5203 *ns*. The additional logic on router takes 82 *ns*, which is negligible.

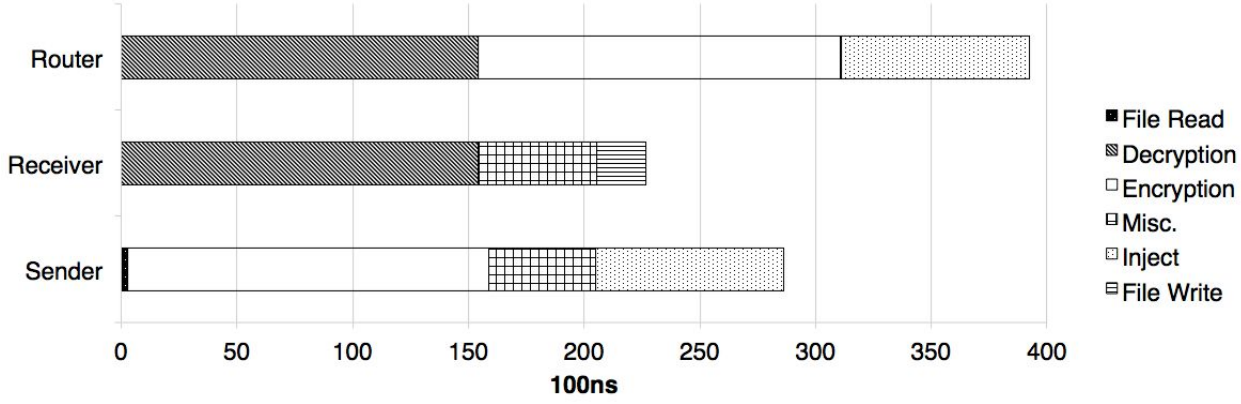

Fig. 16. Latency Breakup of Operations on One File Segment (without Optimization)

To sum up, a file segment takes on average 28616 *ns* on sender, 39254 *ns* on router, and 22676 *ns* on receiver. With this latency, even the data processings between two hops are pipelined perfectly, the maximum theoretical throughput is

$$(1024 \times 8)bit / (39254 \times 10^{-9})s = 208.69\ Mbps$$

C. Speedup Encryption and Decryption with OpenMP

We explored different level of parallelism by altering number of threads in OpenMP clause directives. For encryption, serial encryption takes 15625 ns, 2 thread encryption takes 8890 ns, 4 thread encryption takes 6374 ns, 8 thread encryption takes 5369 ns and 16 thread encryption takes 49695 ns. For decryption, serial decryption takes 15429 ns, 2 thread decryption takes 8861 ns, 4 thread decryption takes 6323 ns, 8 thread decryption takes 5209 ns and 16 thread decryption takes 48976 ns. Though latency seems to be optimal with 8 threads, it turned out that when the router is actually operating, 4 thread encryption/decryption has lower latency.

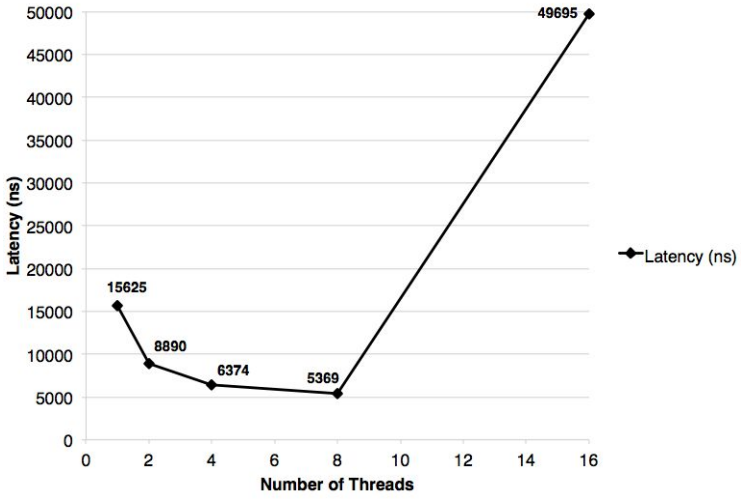

Fig. 17. Number of Threads vs Encryption Latency

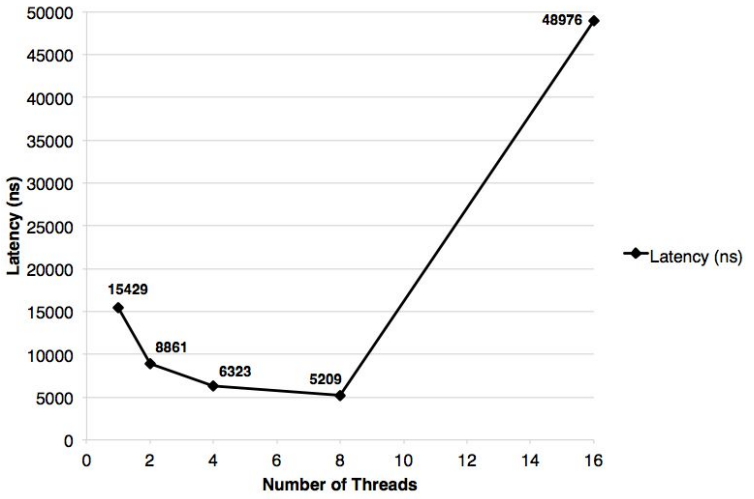

Fig. 18. Number of Threads vs Decryption Latency

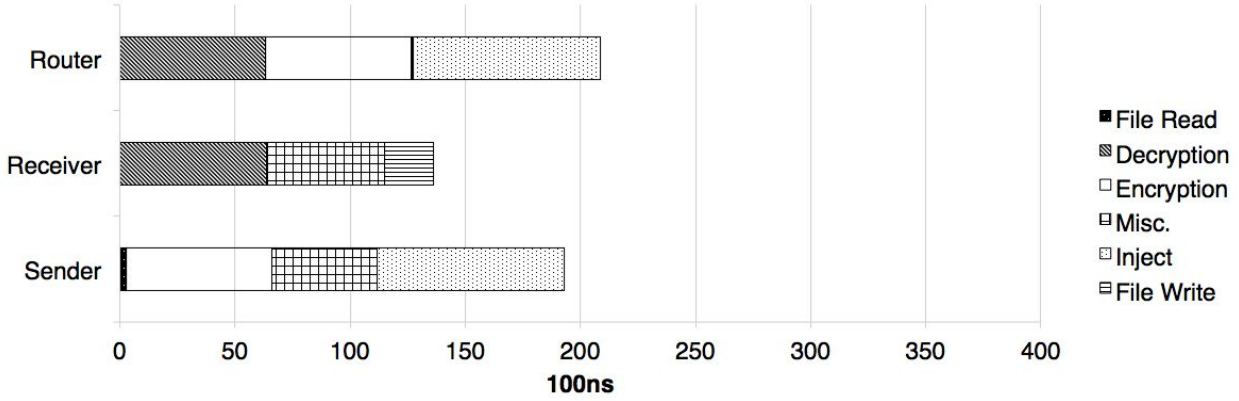

Fig. 19. Latency Breakup of Operations on One File Segment (with OpenMP)

With OpenMP optimization, the latency on sender reduced from 28616 *ns* to 19314 *ns*, the latency on the router reduced from 39254 *ns* to 20897 *ns* , and the latency on receiver reduced from 22676 *ns* to 13622 *ns*. Therefore we achieved the following speedups:

$$Speedup_{sender} = 28616 \div 19314 = 1.48$$
$$Speedup_{router} = 39254 \div 20897 = 1.88$$
$$Speedup_{receiver} = 22676 \div 13622 = 1.66$$

D. Speedup Miscellaneous Logics with Compiler Flags

As mentioned before, adding compiler flags will only speed up massive miscellaneous logics, and since the router's miscellaneous logics are minute compared with its other operations, compiler optimization only speeds up sender and receiver. This is useful for file packet generation, retransmission

packet generation and parsing, sequence checking, and packet classifying. With parallel encryption and decryption enabled, compiler flags further reduced sender latency from 18360 *ns* to 17951 *ns*; router latency from 20897 *ns* to 20871 *ns*; and receiver latency from 13622 *ns* to 12005 ns.

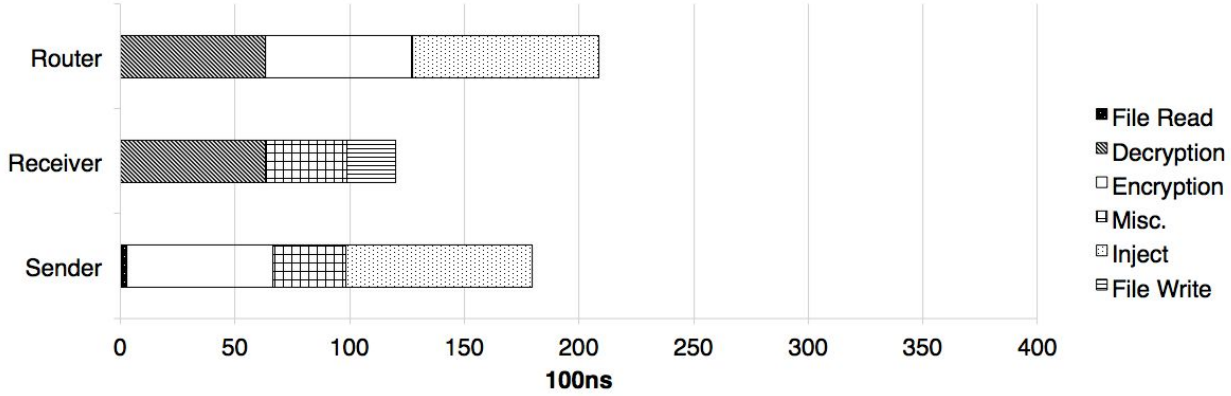

Fig. 20. Latency Breakup of Operations on One File Segment (with both OpenMP and Compiler Flags)

Therefore we achieved the following speedups:

$$Speedup_{sender} = 28616 \div 17951 = 1.69$$
$$Speedup_{router} = 39254 \div 20871 = 1.88$$
$$Speedup_{receiver} = 22676 \div 12005 = 1.88$$

With these optimizations, the router is able to achieve a theoretical maximum throughput of

$$(1024 \times 8)bit / (20871 \times 10^{-9})s = 392.51\ Mbps$$

We achieved 88% improvement with parallel encryption / decryption and compiler optimizations theoretically. In real practice, even we tuned the sender and receiver to their extremes, we still observes an approximate 5% packet loss at the router. The real maximum throughput we achieved is about 320 *Mbps*, which is 82% of the theoretic maximum throughput.

E. Router Pipelining

As the router's packet operation on each ethernet interface is always decryption, encryption, and injection. These three steps have similar time span, and therefore we can pipeline these operations and overlap latency. Since miscellaneous logic would take relatively ignorable length of time, we eliminated it for clearer illustration. A sample pipelining on router packet forwarding is shown as follows:

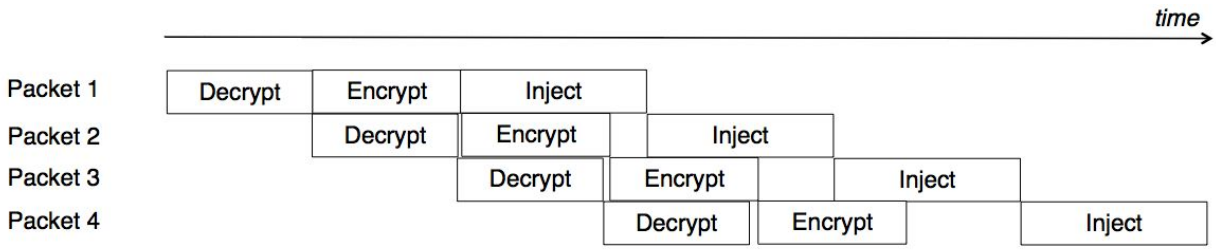

Fig. 21. Router Pipelining

From the illustration we can find that for large number of packet stream, the time router takes to process all of them is just the sum of time to injecting every packet once. Though due to time limit, we did not actually implemented it, we are providing the theoretical throughput and speedups.

Because with pipelining, the router processing rate can be close to line rate, we need to construct larger packets in order to speed up packet sniffing. As shown in Table 1, line-rate sniffing can be achieved when packet size is close to 1500 bytes. We tested packet injection latency using 1440 byte file segment, and it would take 11709 *ns* to inject the packet. Therefore, with router pipelining, the latency to process a single packet is further reduced down to 11709 *ns*. With this performance, The theoretical throughput of router is:

$$(1440 \times 8)bit / (11709 \times 10^{-9})s = 983.86\ Mbps$$

and the corresponding speedup is:

$$Speedup_{router} = 983.86 \div 208.69 = 4.71$$

VII. Future works

The OpenSSL library has a hard limitation on its processing speed which puts a barrier in achieving significant performance. We would like to improve the cryptographic performance of our system in future with the help of hardware acceleration mechanisms [12].

We would also like to include the “routing proxy” used in onion routing [14] to make the path followed by the packet hard to discover. This would obscure the sender by redirecting the encrypted packets to a different proxy every time. To avoid replay of encrypted packets, we would include storing of packet state information.

VIII. References

[1] Goldschlag, David, Michael Reed, and Paul Syverson. "Onion routing."*Communications of the ACM* 42.2 (1999): 39-41.

[2] Clarke, Ian, et al. "Freenet: A distributed anonymous information storage and retrieval system." *Designing Privacy Enhancing Technologies*. Springer Berlin Heidelberg, 2001.

[3] Berthold, Oliver, Hannes Federrath, and Stefan Köpsell. "Web MIXes: A system for anonymous and unobservable Internet access." Designing Privacy Enhancing Technologies. Springer Berlin Heidelberg, 2001.

[4] FTC Staff Report, “Internet of Things: Privacy & Security in a Connected World”. Jan 2015

[5] “Diffie–Hellman key exchange”, Wikipedia: The Free Encyclopedia, from https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange.

[6] Steiner, Michael, Gene Tsudik, and Michael Waidner. "Diffie-Hellman key distribution extended to group communication." Proceedings of the 3rd ACM conference on Computer and communications security. ACM, 1996.

[7] Boyko, Victor, Philip MacKenzie, and Sarvar Patel. "Provably secure password-authenticated key exchange using Diffie-Hellman." Advances in Cryptology—Eurocrypt 2000. Springer Berlin Heidelberg, 2000.

[8] VOCAL Technologies, “Data Encryption Standard (DES)”, Available from: http://www.vocal.com/cryptography/data-encryption-standard-des/

[9] Thakur, Jawahar, and Nagesh Kumar. "DES, AES and Blowfish: Symmetric key cryptography algorithms simulation based performance analysis."International journal of emerging technology and advanced engineering 1.2 (2011): 6-12.
[10] Joshua Thijssen, "Encryption operating modes: ECB vs CBC", December 2010, Available from: https://www.adayinthelifeof.nl/2010/12/08/encryption-operating-modes-ecb-vs-cbc/
[11] OpenWrt Wiki, "OpenSSL Benchmarks", Available from: https://wiki.openwrt.org/inbox/benchmark.openssl
[12] Khalil-Hani, Mohamed, Vishnu P. Nambiar, and Muhammad Nadzir Marsono. "Hardware Acceleration of OpenSSL cryptographic functions for high-performance Internet Security." Intelligent Systems, Modelling and Simulation (ISMS), 2010 International Conference on. IEEE, 2010.
[13] Goldschlag, David, Michael Reed, and Paul Syverson. "Onion routing."Communications of the ACM 42.2 (1999): 39-41.
[14] Reed, Michael G., Paul F. Syverson, and David M. Goldschlag. "Anonymous connections and onion routing." Selected Areas in Communications, IEEE Journal on 16.4 (1998): 482-494.
[15] Citation for PF RING and MMAP, Available from:https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt
[16] A comparison of throughput between PF_PACKET and PF_RING, Available from: http://www.embedded.com/print/4008809
[17] O2 Compiler Optimization Flags, url: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

IX. Appendix

A. Packet Headers

a. Routing Header

TYPE	NAME	PURPOSE
u_int16_t	*saddr*	Source address
u_int16_t	*daddr*	Destination address
u_int16_t	*size*	Entire packet length
u_int8_t	*protocol*	Custom protocol to decide the type of service

b. Reliable Header

TYPE	NAME	PURPOSE
u_int8_t	*port*	Application instance port
u_int16_t	*info*	Necessary additional info
u_int32_t	seq	packet sequence number

c. Key Header

TYPE	NAME	PURPOSE
u_int16_t	size	size of public key

B. Sample Implementation of OpenSSL

- Key generation using DH

```
#include <openssl/dh.h>
#include <openssl/bn.h>
#include <openssl/pem.h>
…….

DH * dhparam = = DH_new();
dhparam = PEM_read_DHparams( , , , );
DH_generate_key(dhparam);                // generate secret and public key

send dhparam->pub_key over the network
receive the public key of connected host

DH_compute_key( , , );                   // compute symmetric key
write symmetric key to a file
end
```

- Encryption/Decryption using DES-CBC

```
#include <openssl/des.h>
….

set the Initialization Vector
DES_set_key_checked(key, schedule);      // create the key schedule
DES_cbc_encrypt(pt, ct,,, 1);            // 1 for encryption
DES_cbc_encrypt(ct, pt,,, 0);            // 0 for decryption
end
```

C. Sample Implementation of Packet Memory Map

a. Raw Socket

```
/* Use mmap to accelerate packet capturing */
[setup]     socket() -------> creation of the capture socket
            setsockopt() ---> allocation of the circular buffer (ring)
                              option: PACKET_RX_RING
setsockopt(fd, SOL_PACKET, PACKET_TX_RING, (void *) &req, sizeof(req))

            mmap() ---------> mapping of the allocated buffer to the
                              user process

[capture]   poll() ---------> to wait for incoming packets

[shutdown]  close() --------> destruction of the capture socket and
                              deallocation of all associated
                              resources.
```

```
/* Use mmap to accelerate packet transmission */
[setup]          socket() -------> creation of the transmission socket
                 setsockopt() ---> allocation of the circular buffer (ring)
                                   option: PACKET_TX_RING
e.x. setsockopt(fd, SOL_PACKET, PACKET_TX_RING, (void *) &req, sizeof(req))

                 bind() ---------> bind transmission socket with a network
                                   interface
                 mmap() ---------> mapping of the allocated buffer to the
                                   user process

[transmission]   poll() ---------> wait for free packets (optional)
                 send() ---------> send all packets that are set as ready in
                                   the ring
                                   The flag MSG_DONTWAIT can be used to return
                                   before end of transfer.

[shutdown]       close() --------> destruction of the transmission socket and
                                    deallocation of all associated resources.
```

b. PCAP

```
[scan device]       pcap_findalldevs()  -----> find a list of available devices
[open device]       pcap_open_live()    -----> return a handler for packet
                                               sniffing
[sniff packet]      pcap_loop()         -----> use the handler sniffed to
                                               capture packets. it loops a
                                               function to process the packet
                                               sniffed
[send packet]       pcap_inject()       -----> use the handler to inject a
                                               packet to the ethernet interface
                                               that handler corresponds to
[shutdown]          pcap_close()        -----> shutdown and destruct the
handler
```

D. Sample Implementation of Parallelizing Encryption using OpenMP

```
#include <omp.h>
...
#pragma omp parallel num_threads(4) shared(packet, sched, size)
{
       #pragma omp sections nowait
       {
              #pragma omp section
              {
                     // encryption for section 1
              }
              #pragma omp section
              {
                     // encryption for section 2
```

```
                }
                #pragma omp section
                {
                        // encryption for section 3
                }
                #pragma omp section
                {
                        // encryption for section 4
                }
        }
}
```

E. Compiler Optimization Flags[17]

Optimization Flag	**Description**
-fthread-jumps	*Perform optimizations that check to see if a jump branches to a location where another comparison subsumed by the first is found.*
-falign-functions	*Align the start of functions to the next power-of-two greater than n, skipping up to n bytes.*
-falign-jumps	*Align branch targets to a power-of-two boundary, for branch targets where the targets can only be reached by jumping, skipping up to n bytes like -falign-functions.*
-falign-loops	*Align loops to a power-of-two boundary, skipping up to n bytes like -falign-functions.*
-falign-labels	*Align all branch targets to a power-of-two boundary, skipping up to n bytes like -falign-functions.*
-fcaller-saves	*Enable allocation of values to registers that are clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls.*
-fcompare-elim	*After register allocation and post-register allocation instruction splitting, identify arithmetic instructions that compute processor flags similar to a comparison operation based on that arithmetic.*
-fcprop-registers	*After register allocation and post-register allocation instruction splitting, perform a copy-propagation pass to try to reduce scheduling dependencies and occasionally eliminate the copy.*

-fcrossjumping	*Perform cross-jumping transformation. This transformation unifies equivalent code and saves code size.*
-fcse-follow-jumps	*In common subexpression elimination (CSE), scan through jump instructions when the target of the jump is not reached by any other path.*
-fcse-skip-blocks	*This is similar to -fcse-follow-jumps, but causes CSE to follow jumps that conditionally skip over blocks. When CSE encounters a simple if statement with no else clause, -fcse-skip-blocks causes CSE to follow the jump around the body of the if.*
-fdelayed-branch	*Attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.*
-fdelete-null-pointer-checks	*Assume that programs cannot safely dereference null pointers, and that no code or data element resides at address zero.*
-fdevirtualize	*Attempt to convert calls to virtual functions to direct calls.*
-fdevirtualize-speculatively	*Attempt to convert calls to virtual functions to speculative direct calls.*
-fexpensive-optimizations	*Perform a number of minor optimizations that are relatively expensive.*
-fgcse	*Perform a global common subexpression elimination pass. This pass also performs global constant and copy propagation.*
-fgcse-lm	*When -fgcse-lm is enabled, global common subexpression elimination attempts to move loads that are only killed by stores into themselves.*
-fhoist-adjacent-loads	*Speculatively hoist loads from both branches of an if-then-else if the loads are from adjacent locations in the same structure and the target architecture has a conditional move instruction.*
-fif-conversion	*Attempt to transform conditional jumps into branch-less equivalents.*
-fif-conversion2	*Use conditional execution to transform conditional jumps into branch-less equivalents.*
-finline-functions-called-once	*Consider all static functions called once for inlining into their caller even if they are not marked inline.*

-finline-small-functions	*Integrate functions into their callers when their body is smaller than expected function call code (so overall size of program gets smaller).*
-findirect-inlining	*Inline also indirect calls that are discovered to be known at compile time thanks to previous inlining.*
-fipa-cp	*Perform interprocedural constant propagation.*
-fipa-cp-alignment	*When enabled, this optimization propagates alignment of function parameters to support better vectorization and string operations.*
fipa-pure-const	*Discover which functions are pure or constant.*
-fipa-profile	*Perform interprocedural profile propagation.*
-fipa-reference	*Discover which static variables do not escape the compilation unit.*
-fipa-sra	*Perform interprocedural scalar replacement of aggregates, removal of unused parameters and replacement of parameters passed by reference by parameters passed by value.*
-fipa-icf	*Perform Identical Code Folding for functions and read-only variables.*
-fisolate-erroneous-paths-dereference	*Detect paths that trigger erroneous or undefined behavior due to dereferencing a null pointer. Isolate those paths from the main control flow and turn the statement with erroneous or undefined behavior into a trap.*
-flra-remat	*Enable CFG-sensitive rematerialization in LRA. Instead of loading values of spilled pseudos, LRA tries to rematerialize (recalculate) values if it is profitable.*
-fmerge-all-constants	*Attempt to merge identical constants and identical variables.*
-fno-branch-count-reg	*Generate a sequence of instructions that decrement a register, compare it against zero, then branch based upon the result.*
-fno-guess-branch-probability	*Do not guess branch probabilities using heuristics.*
-fomit-frame-pointer	*This avoids the instructions to save, set up and restore frame pointers*

-foptimize-sibling-calls	*Optimize sibling and tail recursive calls.*
-foptimize-strlen	*Optimize various standard C string functions (e.g. strlen, strchr or strcpy) and their _FORTIFY_SOURCE counterparts into faster alternatives.*
-fpartial-inlining	*Inline parts of functions. This option has any effect only when inlining itself is turned on by the -finline-functions or -finline-small-functions options.*
-fpeephole2	*Disable any machine-specific peephole optimizations.*
-free	*Attempt to remove redundant extension instructions.*
-freorder-blocks	*Reorder basic blocks in the compiled function in order to reduce number of taken branches and improve code locality.*
-freorder-blocks-algorithm=stc	*Use the specified algorithm for basic block reordering.*
-freorder-blocks-and-partition	*In addition to reordering basic blocks in the compiled function, in order to reduce number of taken branches, partitions hot and cold basic blocks into separate sections of the assembly.*
-freorder-functions	*Reorder functions in the object file in order to improve code locality.*
-frerun-cse-after-loop	*Re-run common subexpression elimination after loop optimizations are performed.*
-fsched-group-heuristic	*Enable the group heuristic in the scheduler.*
-fsched-spec-load	*Allow speculative motion of some load instructions. This only makes sense when scheduling before register allocation, i.e. with -fschedule-insns or at -O2 or higher.*
-fschedule-fusion	*Performs a target dependent pass over the instruction stream to schedule instructions of same type together.*
-fschedule-insns	*If supported for the target machine, attempt to reorder instructions to eliminate execution stalls due to required data being unavailable.*
-fschedule-insns2	*Similar to -fschedule-insns, but requests an additional pass of instruction scheduling after register allocation has been done.*

-fsplit-paths	*Split paths leading to loop backedges. This can improve dead code elimination and common subexpression elimination.*
-fsplit-wide-types	*Split the registers apart and allocate them independently.*
-fstrict-aliasing	*Allow the compiler to assume the strictest aliasing rules applicable to the language being compiled.*
-fstrict-overflow	*Allow the compiler to assume strict signed overflow rules, depending on the language being compiled.*
-fssa-backprop	*Propagate information about uses of a value up the definition chain in order to simplify the definitions.*
-fssa-phiopt	*Perform pattern matching on SSA PHI nodes to optimize conditional code.*
-ftree-builtin-call-dce	*Perform conditional dead code elimination (DCE) for calls to built-in functions that may set errno but are otherwise side-effect free.*
-ftree-bit-ccp	*Perform sparse conditional bit constant propagation on trees and propagate pointer alignment information.*
-ftree-ccp	*Perform sparse conditional constant propagation (CCP) on trees.*
-ftree-ch	*Perform loop header copying on trees.*
-ftree-copy-prop	*Perform copy propagation on trees.*
-ftree-dce	*Perform dead code elimination (DCE) on trees.*
-ftree-dse	*Perform dead store elimination (DSE) on trees.*
-ftree-dominator-opts	*Perform a variety of simple scalar cleanups based on a dominator tree traversal.*
-ftree-forwprop	*Perform forward propagation on trees.*
-ftree-fre	*Perform full redundancy elimination (FRE) on trees.*
-ftree-loop-optimize	*Perform loop optimizations on trees.*
-ftree-phiprop	*Perform hoisting of loads from conditional pointers on trees.*
-ftree-switch-conversion	*Perform conversion of simple initializations in a switch to initializations from a scalar array.*

-ftree-tail-merge	*Look for identical code sequences. When found, replace one with a jump to the other. This optimization is known as tail merging or cross jumping.*
-ftree-pre	*Perform partial redundancy elimination (PRE) on trees.*
-ftree-pta	*Perform function-local points-to analysis on trees.*

F. ns File

Our custom network topology consists of 3 routers and 6 nodes, All 3 four port routers are fully connected to each other and two nodes are connected to each of router. The network simulation (ns) is as shown below:

```
set ns [new Simulator]
source tb_compat.tcl

set node1    [$ns node]
set node2    [$ns node]
set node3    [$ns node]
set node4    [$ns node]
set node5    [$ns node]
set node6    [$ns node]

set rtr1     [$ns node]
set rtr2     [$ns node]
set rtr3     [$ns node]

set link0 [$ns duplex-link $rtr1 $rtr2 1000Mb 5ms DropTail]
set link1 [$ns duplex-link $rtr1 $rtr3 1000Mb 5ms DropTail]
set link2 [$ns duplex-link $rtr2 $rtr3 1000Mb 5ms DropTail]

set link3 [$ns duplex-link $node1 $rtr1 1000Mb 5ms DropTail]
set link4 [$ns duplex-link $node2 $rtr1 1000Mb 5ms DropTail]

set link5 [$ns duplex-link $node3 $rtr2 1000Mb 5ms DropTail]
set link6 [$ns duplex-link $node4 $rtr2 1000Mb 5ms DropTail]

set link7 [$ns duplex-link $node5 $rtr3 1000Mb 5ms DropTail]
set link8 [$ns duplex-link $node6 $rtr3 1000Mb 5ms DropTail]

tb-set-link-loss $link0 0.01
tb-set-link-loss $link1 0.01
tb-set-link-loss $link2 0.01
tb-set-link-loss $link3 0.01
tb-set-link-loss $link4 0.01
tb-set-link-loss $link5 0.01
tb-set-link-loss $link6 0.01
tb-set-link-loss $link7 0.01
tb-set-link-loss $link8 0.01
```

```
tb-set-node-os $rtr1 Ubuntu1204-64-STD
tb-set-node-os $rtr2 Ubuntu1204-64-STD
tb-set-node-os $rtr3 Ubuntu1204-64-STD
tb-set-node-os $node1 Ubuntu1204-64-STD
tb-set-node-os $node2 Ubuntu1204-64-STD
tb-set-node-os $node3 Ubuntu1204-64-STD
tb-set-node-os $node4 Ubuntu1204-64-STD
tb-set-node-os $node5 Ubuntu1204-64-STD
tb-set-node-os $node6 Ubuntu1204-64-STD

tb-set-hardware $rtr1 MicroCloud
tb-set-hardware $rtr2 MicroCloud
tb-set-hardware $rtr3 MicroCloud
tb-set-hardware $node1 MicroCloud
tb-set-hardware $node2 MicroCloud
tb-set-hardware $node3 MicroCloud
tb-set-hardware $node4 MicroCloud
tb-set-hardware $node5 MicroCloud
tb-set-hardware $node6 MicroCloud

$ns rtproto Manual
$ns run
```

G. Time Calculation Script

This is the script we used to calculate file transfer throughput

```
#!/bin/sh
echo "===>start timestamp:"
A=$(date +%s%N)
echo $A
sudo ./pcap_sender 1 2
B=$(date +%s%N)
echo "===>end timestamp:"
echo $B
echo "===>Throughput"
echo "524288000 / ($B - $A) * 1000000000 * 8" | bc -l
echo "bits/second"
```