A Report on

Anonymous Multi-Hop Payment for Payment Channel Networks

CS540

CRYPTOCURRENCIES AND BLOCKCHAIN TECHNOLOGY

Lavi Jain
(2023CSM1005)

Wamique Zia
(2023CSM1020)

Shubham Thawait
(2023CSM1015)

Gulam Mustafa
(2023CSM1004)

Manpreet Kaur
(2022CSM1010)

1. Introduction

This report delves into the implementation and evaluation of an enhanced version of Anonymous Multi-Hop Lock (AMHL) called AMHL+ using Hashed Time-Locked Contracts (HTLC) for Anonymous Multi-Hop Payment in Payment Channel Networks. The project is divided into three distinct tasks, implementation of AMHL+, implementing Denial of Service Attack on the payment channel network, comparing the performance cost before attack as well as after attack.

1.1 Construction Of AMHL+

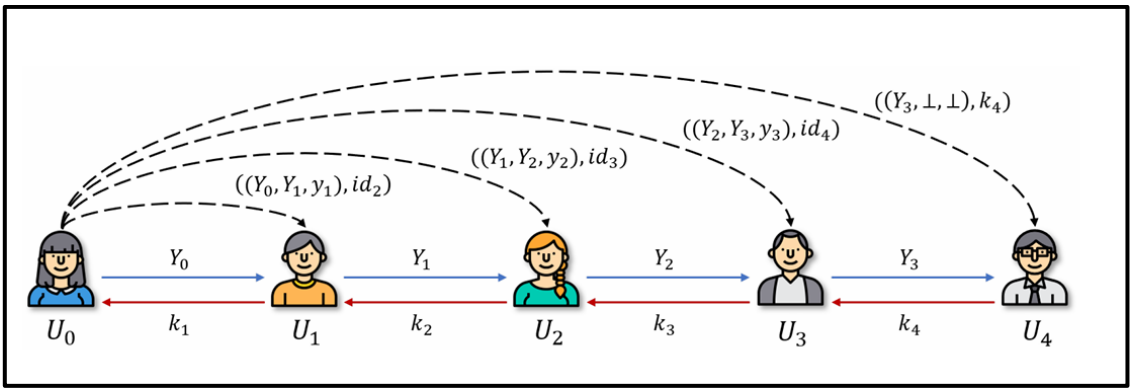


Figure 1.1.a:- Example of multi-hop AMHL+ phases between U0 and U4. The dotted black line indicates the messages U0 generated for each user. The Blue line indicates The Locks set between the users. The red line indicates the release phase.

We assume that there are n + 1 users $\{U_0,...,U_n\}$ on the payment path, and for each user U_i, $i \in [0,n]$, he/she has a public-private key pair $(pk_i, sk_i) \leftarrow$ HybridEnc.KeyGen. The system parameter is para =(F(),q), where F() is a homomorphic function and q is a large prime as that in AMHL. The details are as follows.

- **Setup**: As described in AMHL, the payer U0 samples n values (y0,...,yn−1) from Zq and computes $Y_0 = F(y_0)$, $Y_i = Y_{i-1} \cdot F(y_i)$ for $i \in [1,n-1]$ and $k_n =$ (sum from (n-1 to 0) y_i) mod q.
- **Lock**: The Lock phase starts from the payer U_0. He/she runs OE.Enc($\{pk_i\}^n_{i=1}$,{mi}n i=1) to get C1, where $m_i = Y_{i-1}||Y_i||y_i||id_{i+1}$ for $i \in [1,n-1]$ and $m_n = Y_{n-1}||k_n$, where id_{i+1} is the identity information of the user U_{i+1}. After that, U_0 sends $(Y_0,C1)$ to the user U_1. For U_i ($i \in [1,n-1]$), he/she runs HybridEnc.Dec(sk_i,C_i) to get $(Y_{i-1}||Y_i||y_i||id_{i+1}||C_{i+1})$. If either $Y_{i-1}' = Y_{i-1}$ or $Y_i = Y_{i-1} \cdot F(y_i)$ does not hold, U_i aborts. Otherwise, U_i accepts $L_{i-1} = Y_{i-1}$ and sets $Y_i = Y_i'$. After that, Ui sends (Y_i,C_{i+1}) to U_{i+1}, where $C_{i+1} = C_{i+1}||s_i$ and s_i is a random string with $|(C_{i+1}||s_i)| = |C_i|$.For U_n, he/she runs HybridEnc.Dec(sk_n,C_n) to get ($Y_{n-1}||k_n$). If either $Y_{n-1} = Y_{n-1}'$ or $Y_{n-1} = F(k_n)$ does not hold, U_n aborts. Otherwise, U_n accepts $L_{n-1} = Y_{n-1}$.
- **Release**: This phase starts from the payee U_n, who sends k_n to the user U_{n-1} to release L_{n-1} directly. For the inter mediate user U_i, $i \in [n-1,1]$, he/she checks the validity of k_{i+1} from the user U_{i+1} by using $Y_i = F(k_{i+1})$.If k_{i+1} is valid, the user U_i sets $k_i = k_{i+1} - y_i$ mod q and sends k_i to U_{i-1} to release the lock L_{i-1}. Otherwise, the user U_i aborts.

1.2 Denial Of Service Attack

Payment channel networks (PCNs) have become a viable solution for addressing scalability challenges in blockchain-based payment systems. The ability to conduct multi-hop payments within PCNs enables efficient and cost-effective value transfer across the network. However, like any other digital infrastructure, PCNs are susceptible to cyber threats, such as Denial-of-Service (DoS) attacks. These attacks can severely impact the performance and reliability of PCNs, jeopardizing multi-hop payment channels and the overall network's trustworthiness and usability. It is essential to understand the implications of DoS attacks on multi-hop payments in PCNs to devise effective defense mechanisms and ensure the resilience of these decentralized payment systems. This study thoroughly examines the potential impact of DoS attacks on multi-hop payments in PCNs, considering various scenario vectors, their impacts and costs.

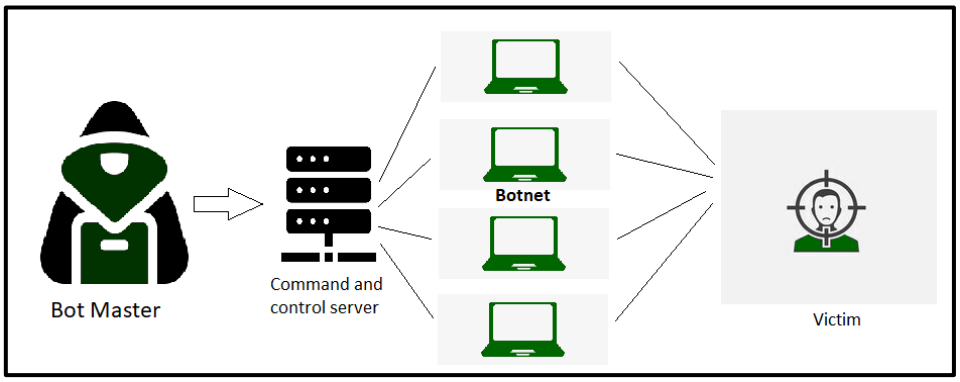


Figure 1.2.a:- Simple DOS attack

1.2.1 Availability Impact

DoS attacks can flood the payment channel network with excessive traffic or malicious requests, leading to node overload and subsequent unresponsiveness. This overload may cause delays in transaction processing or outright service interruptions. As a result, users may encounter difficulties in initiating or completing payments, thereby undermining the availability of the network. Such disruptions can erode user trust and confidence in the system's reliability.

1.2.2 Reliability Impact

In the event of a successful DoS attack, the network's ability to handle increased transaction volumes may be compromised. This can lead to delays, timeouts, or even dropped transactions, further exacerbating reliability issues. Users rely on the network to consistently facilitate their transactions in a timely and dependable manner. Any disruption in this regard can damage the perceived reliability of the system and undermine its effectiveness as a payment solution.

1.2.3 Performance Impact

DoS attacks can significantly degrade the performance of the payment channel network by increasing communication overhead and computational costs. The heightened traffic resulting from the attack can strain network resources, leading to increased latency and decreased throughput. Additionally, the

computational burden on network nodes may escalate as they contend with processing malicious requests and maintaining network stability. Consequently, the overall performance of AMHP for PCNs may suffer, impacting transaction efficiency and user experience.

In conclusion, the potential impact of DoS attacks on AMHP for PCNs affects availability, reliability, and performance aspects. Mitigation strategies must be implemented to safeguard against such attacks and maintain the integrity and functionality of the payment channel network. These strategies may include robust network monitoring, traffic filtering, resource allocation mechanisms, and contingency plans for rapid response and recovery in the event of an attack.

2. Implementation Specifications

We have implemented the AMHL+ algorithm in python. The simulation starts by user specifying n, which is the number of nodes in the payment channel network. We generate a random graph of n nodes using networkx library in python.

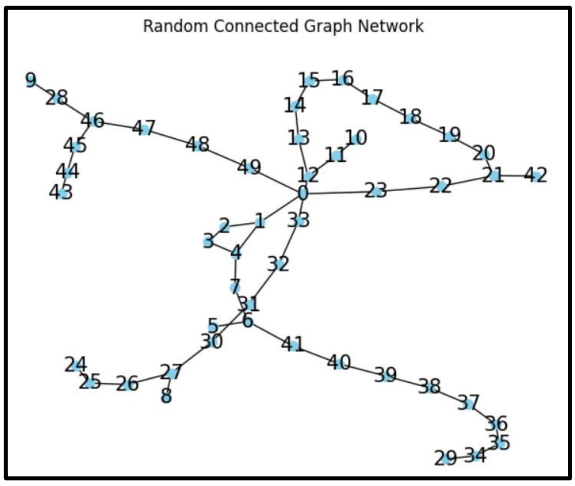


Figure 2.1:- Example of random generated graph using networkx

Then in this graph we define two more variables payer and payee, and find the payment_path from payer to payee using network.

```
{0: 7, 1: 4, 2: 1, 3: 0, 4: 23, 5: 22, 6: 21, 7: 20, 8: 19, 9: 18, 10: 17, 11: 16}

Number of nodes in path : 12
```

Figure 2.2:- The payment_path generated in the network, the map contains key to be userid and value to be the User name.

Then we define a faulty_map, which based on percentage_of_faulty_nodes, assigns 0 or 1 in the map indicating whether that node is impacted by dos or not.

```
Generated faulty map:
{0: 0, 1: 0, 2: 0, 3: 0, 4: 1, 5: 0, 6: 0, 7: 1, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 1, 15: 0, 16: 0, 17: 0, 18: 0, 19: 0, 20: 0,
```

Figure 2.3:- The faulty_map which tells whether the node is affected by DOS attack in the network or not

User_keys stores the public and private keys of all users.

Figure 2.4:- User_keys containing the various public and private keys of users

The homo-morphic function used is F(x) = g^x mod p, where g = 3 and p = a large prime integer.

Hybrid Algorithm (KeyGeneration , Encryption, Decryption):

In hybrid encryption, a symmetric encryption algorithm such as AES (Advanced Encryption Standard) is used to encrypt the actual data, while an asymmetric encryption algorithm such as RSA (Rivest-Shamir-Adleman) is used to securely transmit the symmetric encryption key.

Here's how hybrid encryption is implemented by us:

Key Generation:

```python
def generate_user_keys(user_list):
    user_keys = {}
    for user in user_list:
        # Generate private key for each user
        private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048
        )
        # Extract public key from private key
        public_key = private_key.public_key()
        # Store public-private key pair in the map
        user_keys[user] = (public_key, private_key)
    return user_keys
```

Figure 2.5:- Function to generate public, private keys

For RSA encryption, each party generates a public-private key pair. The public key is shared with others for encryption, while the private key is kept secret for decryption.

For AES encryption, a symmetric key is generated for each session or message.

Encryption Process:

```python
def hybrid_encrypt(plaintext, public_key):

    # Pad the plaintext
    pkcs7_padder = padding.PKCS7(AES.block_size).padder()
    padded_plaintext = pkcs7_padder.update(plaintext) + pkcs7_padder.finalize()

    # Generate new random AES-256 key
    key = os.urandom(256 // 8)

    # Generate new random 128 IV required for CBC mode
    iv = os.urandom(128 // 8)

    # AES CBC Cipher
    aes_cbc_cipher = Cipher(AES(key), CBC(iv))

    # Encrypt padded plaintext
    ciphertext = aes_cbc_cipher.encryptor().update(padded_plaintext)

    # Encrypt AES key
    oaep_padding = asymmetric_padding.OAEP(mgf=asymmetric_padding.MGF1(algorithm=SHA256()), algorithm=SHA256(), label=None)
    cipherkey = public_key.encrypt(key, oaep_padding)

    return {'iv': iv, 'ciphertext': ciphertext}, cipherkey
```

Figure 2.6:- Function for Hybrid Encryption using AES and RSA

The sender generates a random symmetric key (AES key) to encrypt the data.

The data is encrypted using AES with the symmetric key, resulting in ciphertext.

The symmetric key is then encrypted using the recipient's public RSA key. This step ensures that only the recipient, possessing the corresponding private key, can decrypt the symmetric key.

Transmission:

The encrypted symmetric key and the ciphertext are transmitted to the recipient over a potentially insecure channel.

Decryption Process:

```python
def hybrid_decrypt(ciphertext, cipherkey, private_key):

    # Decrypt AES key
    oaep_padding = asymmetric_padding.OAEP(mgf=asymmetric_padding.MGF1(algorithm=SHA256()), algorithm=SHA256(), label=None)
    recovered_key = private_key.decrypt(cipherkey, oaep_padding)

    # Decrypt padded plaintext
    aes_cbc_cipher = Cipher(AES(recovered_key), CBC(ciphertext['iv']))
    recovered_padded_plaintext = aes_cbc_cipher.decryptor().update(ciphertext['ciphertext'])

    # Remove padding
    pkcs7_unpadder = padding.PKCS7(AES.block_size).unpadder()
    recovered_plaintext = pkcs7_unpadder.update(recovered_padded_plaintext) + pkcs7_unpadder.finalize()

    return recovered_plaintext
```

Figure 2.7:- Function for Hybrid Decryption

The recipient uses their private RSA key to decrypt the symmetric key.

With the obtained symmetric key, the recipient decrypts the ciphertext using AES, revealing the original plaintext.

The messages list contains the messages for each user.

```
[   '654252368174946820903021717833:::721851354669124390817979323126:::724245471424294510904514187 67:::2',
    '721851354669124390817979323126:::300444396718743918285173588293:::534711114342470848902073175769:::3',
    '300444396718743918285173588293:::441299371968147669227358096622:::142640597316759700068913686789:::4',
    '441299371968147669227358096622:::533990911272505276877100788008:::153116286137499543061734351960:::5',
    '533990911272505276877100788008:::603415821294571328602136357940:::508487725539133860586941894835:::6',
    '603415821294571328602136357940:::310423796871386701080053290976:::288916307665560126589752013624:::7',
    '310423796871386701080053290976:::219265971833367912168592006198:::131125590249966942053420643285:::8',
    '219265971833367912168592006198:::214099717790218938598910576142:::541217453882969533156979298671:::9',
    '214099717790218938598910576142:::284796567919067266041012640339:::393708812133303338619440397422:::10',
    '284796567919067266041012640339:::635778102764116536264604526770:::604464733201167536415174258382:::11',
    '635778102764116536264604526770:::354757786387712459852229207860 1']
```

Figure 2.8:- messages generated for each user in payment path

Onion Encryption:

```
Cdashi, cipherkey =  hybrid_encrypt(messages[n-2].encode(),user_keys[f'U{n-1}'][0])
for i in range(n-3, -1, -1):
    newmessage = messages[i].encode() + b"|||"+ Cdashi['iv'] +b"|||"+ Cdashi['ciphertext'] + b"|||" + cipherkey
    Cdashi, cipherkey = hybrid_encrypt(newmessage,user_keys[f'U{i+1}'][0])
C1= Cdashi
```

Figure 2.9:- Function for Onion Encryption to compute C1

Onion Encryption of messages generated by first User in payment path, to compute C1.

Example Output:

User7(Payer) sends (Y0, C1) to User 4, In the Lock Phase we see that when User4 applies its private key to decrypt C1, he only sees his message and the rest of the content is encrypted. From the received message he gets the id of the next intended hop. In our example, the next hop user id is 2. From the mapping we see that userid:2 is for User 1. The User 4 uses Y0 and his message to check the Lock and accepts if it matches some preset conditions. This is how the message is transferred and Locks are applied.

```
User 4
Decrypted Message:-
b'654252368174946820903021717833:::721851354669124390817979323126:::724245471424294510904514187 67:::2|||b\xf9eC\x15\xcc#\xa86C\xb8\xd4hQ\xbai|||k\xbd\xaa\x06
Actual Message:-
['654252368174946820903021717833', '721851354669124390817979323126', '724245471424294510904514187 67', '2']
User 1 accepting lock 0


User 1
Decrypted Message:-
b'721851354669124390817979323126:::300444396718743918285173588293:::534711114342470848902073175769:::3|||}\xf2\x121\xca\x1f\x12\x11\xe6\x0f\xed0J@\xdf\xee|||
Actual Message:-
['721851354669124390817979323126', '300444396718743918285173588293', '534711114342470848902073175769', '3']
User 2 accepting lock 1


User 0
Decrypted Message:-
b'300444396718743918285173588293:::441299371968147669227358096622:::142640597316759700068913686789:::4|||s\xb0\xdd\x84\xb4v%\xde"a\x10\x05\xb2\xf0\xe9\n|||.\
Actual Message:-
['300444396718743918285173588293', '441299371968147669227358096622', '142640597316759700068913686789', '4']
User 3 accepting lock 2
```

Figure 2.10:- Example output of Lock Phase

Release Phase:

In the release phase the following condition is checked and Kn is updated if it is satisfied. The payment aborts if condition fails.

```
if int(Y_values[i]) == int(homomorphic_function(int(Kn))):
  Kn = int(Kn) - (int(random_values[i])%int(q))
```

Figure 2.11:- The checks for release phase and Ki computation

```
U16 sends K11 to U17 to release L10
U17 sends K10 to U18 to release L9
U18 sends K9 to U19 to release L8
U19 sends K8 to U20 to release L7
U20 sends K7 to U21 to release L6
U21 sends K6 to U22 to release L5
U22 sends K5 to U23 to release L4
U23 sends K4 to U0 to release L3
U0 sends K3 to U1 to release L2
U1 sends K2 to U4 to release L1
U4 sends K1 to U7 to release L0

=> AMHL+ payment execution successfull
```

Figure 2.12:- Example output of Release Phase

2.1 Libraries Used

2.1.1 pyCryptodome

PyCryptodome is a Python library for implementing cryptography with support for various algorithms and protocols.

2.1.2 networkX

NetworkX is a Python library for creating, manipulating, and analyzing graphs and complex networks.

2.1.3 random

The random module is a part of Python's standard library, providing functions for generating pseudo-random numbers.

2.1.4 cryptography:

Cryptography is a Python library offering cryptographic recipes and primitives for secure communication and data protection.

2.1.5 os

The os module is a core Python library for interacting with the operating system, providing utilities for file operations and system interactions.

3. EXPERIMENTS

3.1 SIMULATION - 1

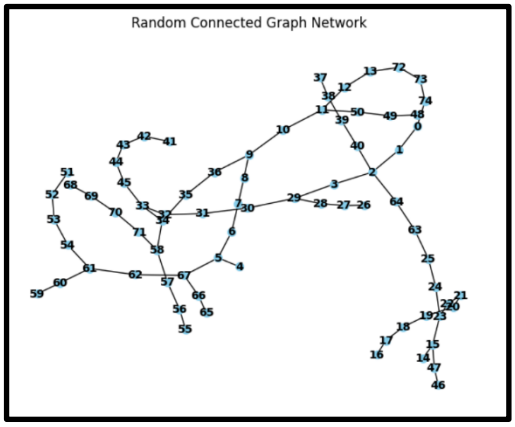
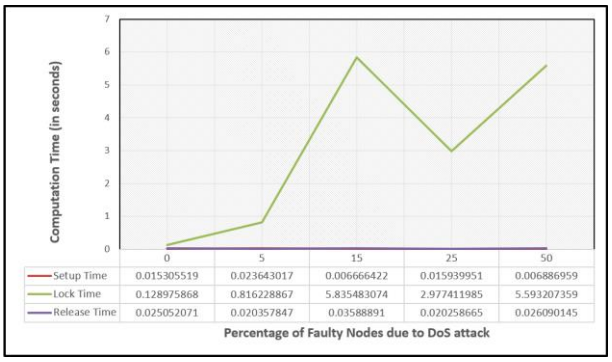
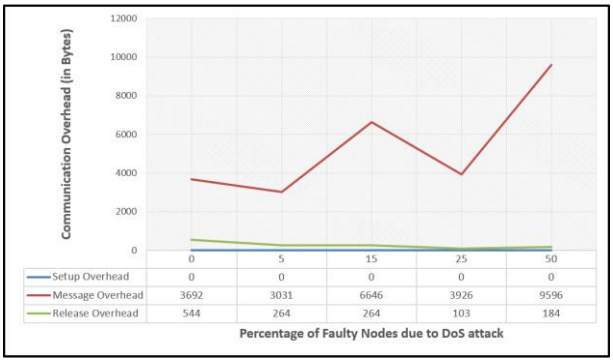


Fig. 3.1.a. Total number of nodes in the network is 75



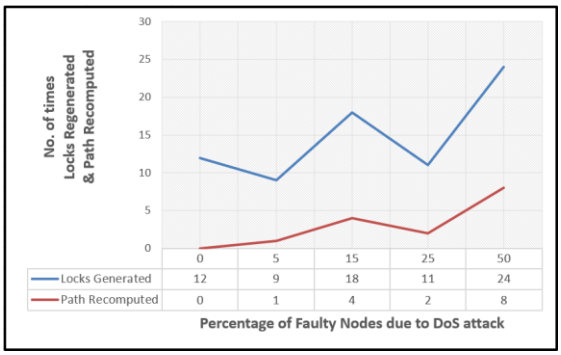
	0	5	15	25	50
Setup Time	0.015305519	0.023643017	0.006666422	0.015939951	0.006886959
Lock Time	0.128975868	0.816228867	5.835483074	2.977411985	5.593207359
Release Time	0.025052071	0.020357847	0.03588891	0.020258665	0.026090145

Fig. 3.1.b. Computation Time Comparison



	0	5	15	25	50
Setup Overhead	0	0	0	0	0
Message Overhead	3692	3031	6646	3926	9596
Release Overhead	544	264	264	103	184

Fig. 3.1.c. Communication Overhead Comparison



	0	5	15	25	50
Locks Generated	12	9	18	11	24
Path Recomputed	0	1	4	2	8

Fig. 3.1.d. Locks Regenerated & Path Recomputed

- Before DoS attack scenario is represented by 0% of faulty nodes.
- Generally, as the number of faulty nodes is increased, the computation time and communication overhead are increased as we can see in Fig.3.1.c.
- But it is actually dependent on the shortest path recomputed and due to that the number of locks regenerated.
- As we can see in Figure.3.1.c. the total of 25% nodes are faulty, but the locks generated after new path computation is less than as in 15% of faulty nodes.
- As we can see in Figure.3.1.d for the large dos attack the lock regeneration is also high. So, the communication time and communication overhead are inherently dependent on the factor of shortest path re-computation after encountering any faulty nodes and hence the locks regeneration.
- Lock Regeneration: When network paths are re-computed due to faulty nodes, the need for lock regeneration increases to maintain data integrity and security. This additional lock regeneration can cause delays and increase the overall computational load on the network.

3.2 SIMULATION – 2

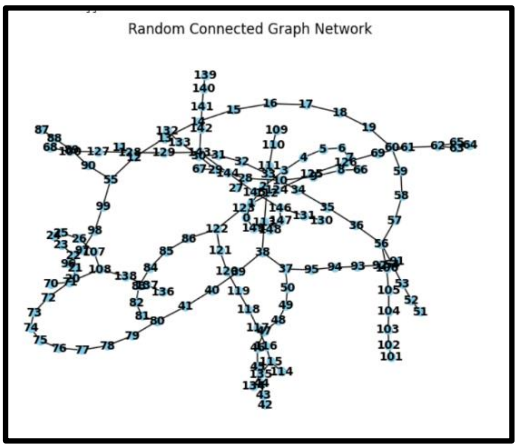


Fig. 3.2.a. Total number of nodes in the network is 150

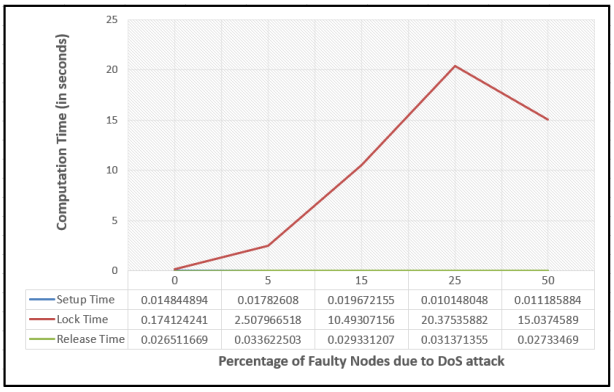


Fig. 3.2.b. Computation Time Comparison

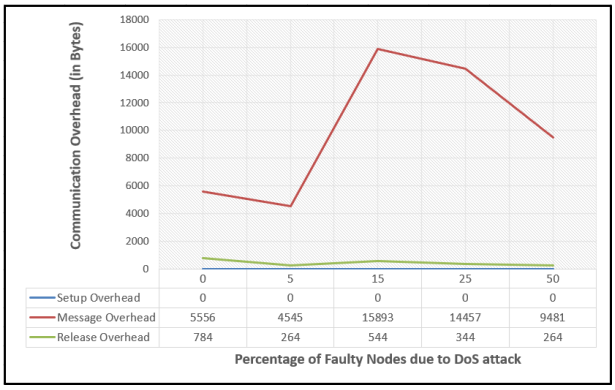


Fig. 3.2.c. Communication Overhead Comparison

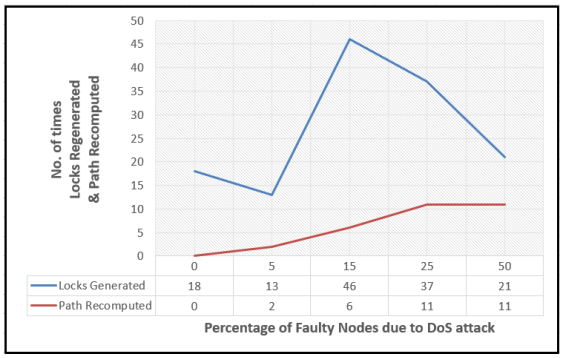


Fig. 3.2.d. Locks Regenerated & Path Recomputed

- Increasing faulty nodes leads to more computation and communication overhead: As more nodes in the network start malfunctioning, the system has to do more work to adjust, which leads to extra data transfer and processing. This can be seen in Figure 3.2.c.

- Shortest Path Re-computation: As we can see in Fig.3.2.b, the frequency of recomputing the shortest path is a key factor. When faulty nodes disrupt the original network topology, the system must determine new optimal paths. If the re-computation leads to more locks or additional network controls, this can introduce further overhead in terms of communication and processing time.

- As the number of faulty nodes grows, communication overhead can increase because of the extra messages needed to adjust network routes and reset security locks, but as we see in Fig.3.2.d locks generated decreases after 46 lock generation , hence Communication overhead is also decreasing in Fig.3.2.c

3.3 SIMULATION – 3

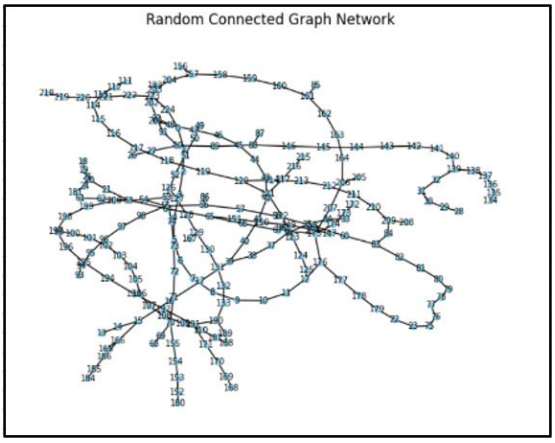


Fig. 3.3.a. Total number of nodes in the network is 225

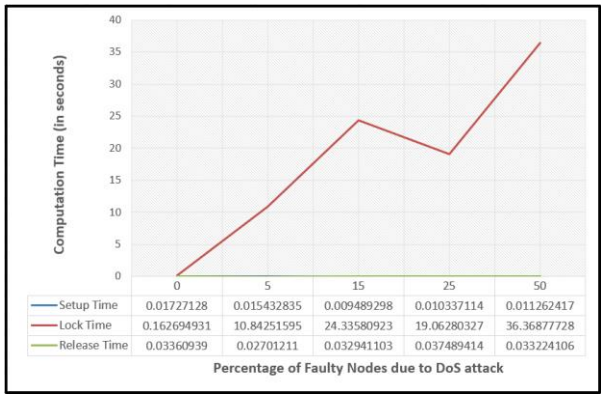


Fig. 3.3.b. Computation Time Comparison

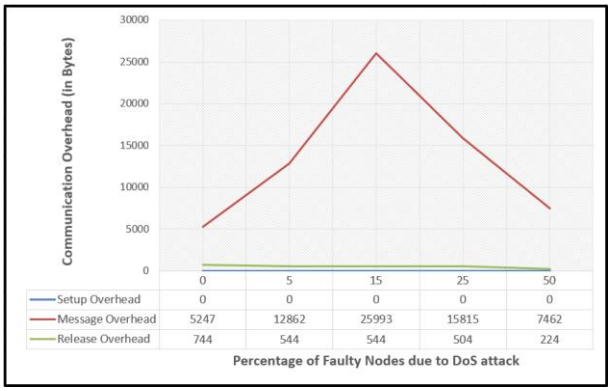


Fig. 3.3.c. Communication Overhead Comparison

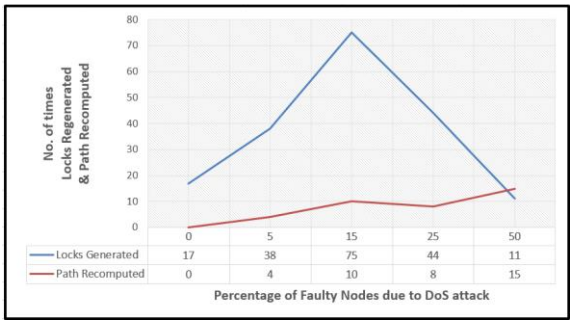


Fig. 3.3.d. Locks Regenerated & Path Recomputed

- Everything is working normally.As more nodes become faulty, computation time and communication overhead increase. As more nodes in the network fail, it takes longer to calculate new paths, as seen in Figure 3.3.b.
- Communication Overhead: Additional data transmission or communication required to maintain the network's functionality. This overhead generally rises with the increase in faulty nodes, here peaking at 15% as we can see in figure.3.3.c
- Release Overhead: Extra overhead during the release process. This is relatively stable but varies depending on the scenario.
- Locks Generated:Number of security locks generated during re-computation. This increases the when faulty nodes in the path increases as we can see in Fig3.3.d.

3.4 SIMULATION – 4:

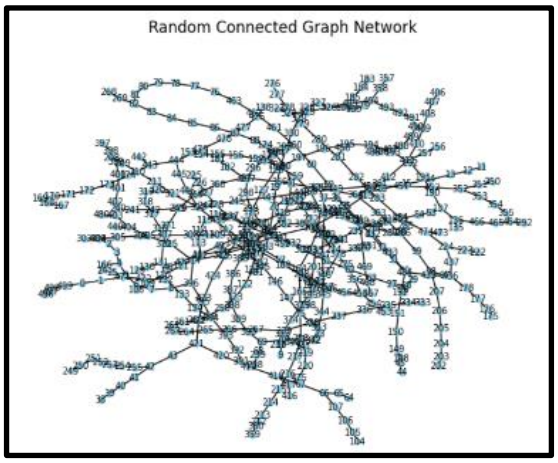


Fig. 3.4.a. Total number of nodes in the network is 500

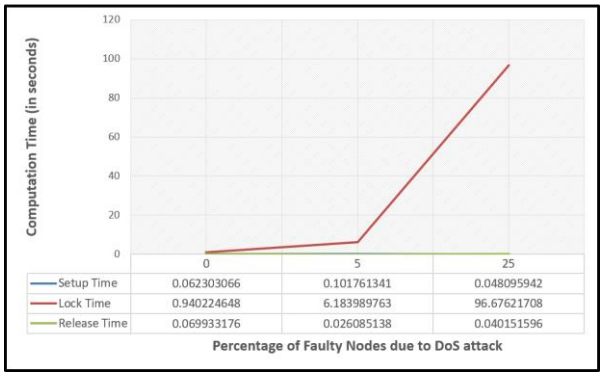


Fig. 3.4.b. Computation Time Comparison

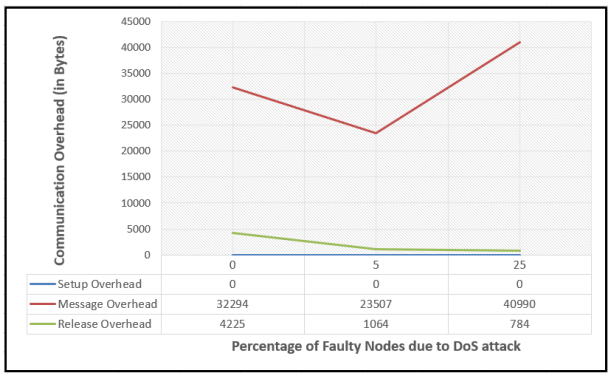


Fig. 3.4.c. Communication Overhead Comparison

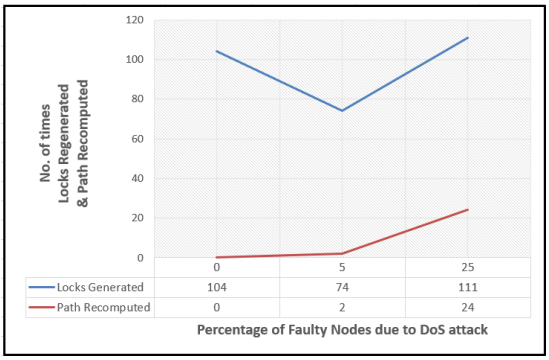


Fig. 3.3.d. Locks Regenerated & Path Recomputed

- In this simulation, when 15% of nodes, no path is found between the payer and payee node, hence no path is found.
- Same happened when 50% of nodes are faulty.
- Locks Generated: This refers to the number of security locks created when paths are recomputed. As the number of faulty nodes along a path increases, more locks are typically generated, as shown in Figure 3.3.d.
- The time of finding the shortest path in a network is crucial, as seen in Figure 3.2.b. If faulty nodes cause disruptions, the network must find new paths. If this process involves extra security measures like more locks or controls, it can slow things down and increase computation time.
- As more nodes in a network fail, it can lead to more communication overhead because the system sends extra messages to adjust routes and set up new security locks, as shown in Figure 3.3.d

4. Conclusion

- The computation time is highest in lock phase, then followed by release phase and setup phase.
- Communication overhead peaks during the lock phase due to intensive message exchange among users, with the release phase following closely behind.
- During the setup phase, there is no communication overhead as messages are solely generated without any transmission occurring.
- With the escalation of nodes affected by the DoS attack, we notice a proportional increase in both path recomputation and lock regeneration processes. Consequently, this uptick in activity directly impacts the computational workload across all phases.
- Upon detecting faulty nodes, the entire payment path necessitates reconstruction, essentially resetting the transaction process and resulting in elevated communication overhead.
- The main effect of DoS attack observed is that when the intermediate nodes are affected, the complete path needs to be reconstructed, resulting in increased overhead in Setup phase. It also leads to redundant locks in the network which will expire only after the time lock expires, hence increased computations in lock phase and decrease in efficiency of the payment channel network.