Implementing Flow Correlation Attacks in Mix Networks

1 Introduction

For the purposes of this coursework, we focus and implement attacks in Mix networks presented in [1]. The attack is implemented in Mix networks, a type of anonymity network that are intended to safeguard users' privacy by rerouting their traffic before it reaches its final destination. During such an attack, an adversary examines the patterns of the network's traffic in an effort to achieve a correlation between the sender and the receiver of the message. This is shown to be accomplished by the authors by monitoring the traffic on the network and then analysing the timing of the messages in order to compare them with the messages that were sent by the person who is the object of the investigation. The attack pipeline uses math-based algorithms that allow an adversary to extract useful information and thus launch a flow correlation attack.

The link for the VM containing the code for the attack is the following: Virtual Machine Link

2 Implementation of Mix Networks

In this paper, the authors mention 7 different mix network types:

- **Timed mix:** Within a specified time, this mix collects packets and once the time is reached, the packets are then sent out to the corresponding output nodes.
- **Threshold mix:** If the number of collected packets in the mix network exceeds a specified threshold, the mix then sends out the packets to the output nodes.
- **Threshold Or Timed mix:** This mix uses both principles of the timed and threshold mix whereby if the threshold or the specified timeout is reached, the mix sends out the packets.
- **Threshold And Timed mix:** Similarly, if the threshold and the specified timeout is reached, the mix sends out the batch of packets.
- **Threshold Pool mix:** Similar to the threshold, if the number of packets received exceeds the threshold plus a value, send out randomly chosen packets up to the number of the threshold
- **Timed Pool mix:** If the timer runs out, and the number of packets is greater than an allocated value, then send out, randomly, (number of packets - allocated value) packets.
- **Timed Dynamic Pool mix:** If the timer runs out, and the number of packets is greater than the threshold pool then send out $\max(1, p \cdot (n - f))$ randomly chosen packets, where p is a fraction.

2.1 Packet Generator

Before discussing the implemented forms of mix networks, we will explore how packets are generated to facilitate the simulation of mix networks. Our packet generator is responsible for generating packets with random source and destination nodes and random messages, as seen in Figure 1. In addition, Python was utilized to do this. Our method generates based on the variable n; the number of packets generated. Even though our method for creating packets was straightforward, it was designed to illustrate the most crucial aspects of network message traffic. The randomly generated source and destination nodes, for instance, simulated the notion that messages can originate from anywhere in the network and be sent to any node. In addition, the randomly produced messages helped replicate the concept that messages might be of many types and sizes, hence increasing the complexity of network traffic.

```python
def message_maker(n_nodes, config, random_time, fixed_prob):
    """
    Function used to generate a random packet with the following fields:
    - source IP
    - destination IP
    - message content
    - exact time the message was sent
    """
    # unpack the config of the src and dest nodes
    (in_nodes_ip, out_nodes_ip, dests) = config
    # define the characters that can be used in the message content
    letters = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
    # randomize the scr IP
    src = in_nodes_ip[random.randint(0, n_nodes - 1)]
    # choose a (0,1) value
    prob = random.random()
    # communicating with favorite
    if prob < fixed_prob:
        dest = dests[src]
    # communicating with any other
    else:
        dest = out_nodes_ip[random.randint(0, n_nodes - 1)]

    # randomize the message
    msg = ''.join(random.choice(letters) for i in range(10))
    # randomize the time that the message was sent
    now = datetime.now() + timedelta(seconds=random.randint(1, random_time))
    msg_time = now.strftime('%Y-%m-%d %H:%M:%S')
    # create the packet as a tuple of the above info and return it
    packet = (src, dest, msg_time, msg)
    return packet
```

Figure 1: A snippet of our packet generator.

2.2 Timed Mix

In this type of mix network, packets are gathered within a certain amount of time and then sent to the output nodes that they belong to. We used the parameter t, which is the time threshold in seconds, in our implementation. In terms of implementation, we need to have a timer that starts when the first packet arrives and stops when the time interval is reached. During this time, packets are collected in a buffer, and once the timer stops, the buffer is sent to the corresponding output nodes.

2.3 Threshold Mix

In this type of mix network, packets are collected until the number of packets in the mix reaches a specified threshold, and then they are sent out to the corresponding output nodes. The threshold is the main parameter within this algorithm, where once $n = m$, then packets are sent; m being the threshold to control the packet sending. To implement this mix type, we need to have a counter that counts the number of packets in the mix. Once the counter reaches the threshold, the packets are sent to the correct output nodes. However, this generally requires more resources than the timed mix type.

2.4 Threshold Or Timed Mix

The threshold or timed mix is a hybrid mix network that combines the principles of the timed mix and the threshold mix. The mix puts packets in a buffer until either the timer interval runs out or the buffer has the number of packets specified. Once either of these conditions is met, the mix sends out the packets in the buffer to the output nodes. The algorithm within this mix can be illustrated as follows: *if* t *times out,* n packets are sent; *else if* $n = m$, n packets are sent.

2.5 Threshold And Timed Mix

The threshold and timed mix is another hybrid mix network that combines the principles of the timed mix and the threshold mix. The mix collects packets in a buffer until the timer interval has expired and the buffer contains the specified number of packets. Once both of these conditions are met, the mix sends out the packets in the buffer to the output nodes. The algorithm used in this implementation is *if* (t) *times out, and* $(n \geq m)$, n packets are sent.

2.6 Timed Pool Mix

The Timed Pool Mix is a type of mix network that collects a pool of messages and sends them out only when a specified timeout is reached, and the number of messages in the pool exceeds a threshold value. The mix network starts by initializing a pool. The incoming messages are checked to see if they fall within the time threshold, and if they do, they are added to the pool. If the pool is filled up, a timer starts. If the timer reaches the specified timeout value, and the number of messages in the pool exceeds a specified threshold value, the mix network sends out the batch of packets. We can illustrate the algorithm deployed as follows: *if (t) times out, and* $(n > f)$, *then* $(n - f)$ randomly chosen packets are sent. The variable f represents the minimum number of packets left in the pool.

2.7 Threshold Pool Mix

The Threshold Pool Mix is similar to the Timed Pool Mix in that it collects a pool of messages and sends them out once a threshold value is reached. However, in this case, there is no time limit for collecting messages. Instead, the mix network waits until the number of messages in the pool exceeds a specified threshold value, at which point it sends out a batch of messages. The algorithm used can be represented as, $if\ n = m + f, then\ m$ randomly chosen packets are sent.

2.8 Timed Dynamic Pool Mix

The Timed Dynamic Pool Mix is a hybrid mix network that combines the principles of both the Timed Pool Mix and the Threshold Pool Mix. This mix network collects a pool of messages and waits for a specified timeout until the number of messages in the pool exceeds a threshold value. Once one of these conditions is met, the mix network sends out a batch of messages. However, in this mix network the threshold value is not fixed. Instead, it is determined by a dynamic function that takes into account the number of messages in the pool and the fraction of messages to be sent out, as can be seen in Figure 2. Thus, we introduced a new variable, p for this algorithm, in which p represents a fraction. Therefore, the algorithm can be expressed as *if (t) times out, and* $(n \geq m + f)$ then, send $max(1, [p(n - f)])$.

```
if len(pool) == m:
    # start timer
    timer = datetime.now() + timedelta(seconds=t)
    while True:
        # check if timer has timed out
        if time >= timer:
            # send packets only if pool has more than f packets
            if len(pool) > f:
                # calculate number of packets to send
                packets_to_send = max(1, int(p * (n - f)))
                # send packets randomly chosen from pool
                random.shuffle(pool)
                msgs = pool[:packets_to_send]
                pool = pool[packets_to_send:]
                break
            else:
                # reset timer and wait for more packets to arrive in pool
                timer = datetime.now() + timedelta(seconds=t)
                continue
```

Figure 2: The algorithm implemented in our Timed Dynamic Pool mix network

3 Adversarial Model

Before understanding our implementation of the attack, it is important to understand the threat model of the adversary and how our simulation does not break that model.

The adversary, described in the threat model, has full viewership of the network and can thus see the input and output links, and the time of sending/arrival of each one. However, due to the setup of mix networks whereby the packets final destination is hidden to provide anonymity, the adversary cannot view to whom the packet is sent, as well as its contents. In the attack's threat model, the adversary is a passive one, whereby they only monitor and do not look to alter, plant or prevent packets from being sent. Therefore, the only data the adversary can see is the inter-arrival times of packets at the mix network from each input node and arrival time at the output node.

Having understood the perspective of the attacker, we were able to simplify the mix network using a simulation. In our implementation, the correct packet flow is recorded in the *output.txt* file as seen in Figure 3. In the simulation, the packets are generated and sorted based on the mix network used and then written to the output file. This is a simulation of the mix network, as, in reality, the packets are not sent at the same time but instead are slowly sent to the mix, and the mix determines when each packet is sent out to the output node. But as we have written the attack based on the attackers perspective, who only collects the "public" packet information and runs the attack once the data has been collected, it does not a break the assumptions.

An other important aspect of our attack surface is to clarify why our implementation choices are honest when it comes to launching the attack. As we can see both in the paper and in our code, the attacker's functions, namely all the functions in the attack Python file, access the input and output flows as different dictionaries, thus no "cheating" is allowed, as the attacker is never able to extract information by comparing the two flows. Moreover, we do not write the packets in the file in a "first in, first out" manner, as we shuffle the packets before printing each batch. This of course does not affect our attack, but also does not allow an adversary (the viewer of the output file) to determine the original sequence of the messages. Therefore, our implementation does not break the threat model of the adversary.

```
251.70.43.5     113.182.0.4     2023-03-13 20:37:31     y1A3zQF6rQ      2023-03-13 20:37:56     thresholdpool
83.108.182.127  14.39.245.201   2023-03-13 20:37:47     AWumSxVdzg      2023-03-13 20:37:56     thresholdpool
123.57.252.88   86.245.152.45   2023-03-13 20:37:50     e0etRcksAp      2023-03-13 20:37:56     thresholdpool
110.88.137.215  74.88.89.230    2023-03-13 20:37:56     p7GiiwbqCC      2023-03-13 20:37:56     thresholdpool
146.6.91.130    83.25.39.135    2023-03-13 20:37:51     rRk33yGuGW      2023-03-13 20:37:56     thresholdpool
3.18.58.201     229.73.43.228   2023-03-13 20:37:43     GrqVMD0Jac      2023-03-13 20:37:56     thresholdpool
83.108.182.127  84.120.181.88   2023-03-13 20:37:42     4e847rGEFe      2023-03-13 20:37:56     thresholdpool
3.18.58.201     205.62.36.153   2023-03-13 20:37:43     fsRANj422U      2023-03-13 20:37:56     thresholdpool
182.109.179.101 150.185.86.175  2023-03-13 20:37:47     lV3iAouh1w      2023-03-13 20:37:56     thresholdpool
251.70.43.5     74.88.89.230    2023-03-13 20:37:36     AhbyiE7sHZ      2023-03-13 20:37:56     thresholdpool

3.18.58.201     74.88.89.230    2023-03-13 20:37:57     NXYIzJnI7n      2023-03-13 20:38:17     thresholdpool
18.244.178.198  98.159.118.192  2023-03-13 20:38:08     d9czEsrHJJ      2023-03-13 20:38:17     thresholdpool
18.244.178.198  18.51.228.201   2023-03-13 20:38:16     VQEaFqWOQ4      2023-03-13 20:38:17     thresholdpool
141.119.71.244  128.214.225.1   2023-03-13 20:37:34     LVJJ3tW5eN      2023-03-13 20:38:17     thresholdpool
83.108.182.127  19.204.202.97   2023-03-13 20:37:30     QT6WWD7yd3      2023-03-13 20:38:17     thresholdpool
18.244.178.198  229.73.43.228   2023-03-13 20:38:17     cqTCIcS7O9      2023-03-13 20:38:17     thresholdpool
138.220.111.214 128.214.225.1   2023-03-13 20:38:13     hR6jddHCKB      2023-03-13 20:38:17     thresholdpool
122.253.148.165 84.120.181.88   2023-03-13 20:37:44     jURBBSssum      2023-03-13 20:38:17     thresholdpool
110.88.137.215  170.108.164.50  2023-03-13 20:38:08     K4D5Guw3hQ      2023-03-13 20:38:17     thresholdpool
141.119.71.244  18.51.228.201   2023-03-13 20:37:55     CtJ6wlqaSx      2023-03-13 20:38:17     thresholdpool

3.18.58.201     138.172.74.32   2023-03-13 20:38:36     hsm4DP4ha7      2023-03-13 20:38:40     thresholdpool
96.164.35.166   86.245.152.45   2023-03-13 20:37:57     8NDm7GZJGB      2023-03-13 20:38:40     thresholdpool
```

Figure 3: An example of *output.txt*

4 The Attack

4.1 Attack Description

The attack takes place over 4 different stages. The first stage, data collection, is the gathering and sorting, by time and thus batch, all the packets based on input and output nodes, such that the packets at input node i:

$$A_i = (a_{i1}, a_{i2}, \ldots, a_{in}) \tag{1}$$

And the packets at output node j:

$$B_j = (b_{j1}, b_{j2}, \ldots, b_{jn}) \tag{2}$$

As the time of the packet sent is not encrypted, as it is observed, this means we can sort the packets based on time, batch and input/output node. The next stage is to extract the flow pattern vector from this data. It is different for timed and threshold mixes. For each input/output node and batch of a threshold mix is determined by:

$$X(input\ vector)_{ik} = \frac{(Num.\ of\ packets\ in\ batch\ k)}{(Ending\ time\ of\ batch\ k) - (Ending\ time\ of\ batch\ k-1)} \tag{3}$$

The input/output vector of a threshold mix determined by:

$$Y(output\ vector)_{jk} = \frac{(Num.\ of\ packets\ in\ k^{th}\ timeout\ interval)}{Timeout} \tag{4}$$

Once the input and output vectors are determined, it is necessary to calculate the distance between each vector. It, therefore, should be stated that the shorter the distance between two nodes, the higher the likelihood that the nodes are communicating. For all types of mixes, a distance function based on Mutual Information can work. Additionally, there exists a distance function that works for timed mixes as well, that is based on Frequency Analysis.

- **Mutual Information:** Mutual Information analyses how similar two vectors are. Using the formula below, the distance between two input/ output nodes can be determined.

$$d(X_i, Y_j) = \frac{1}{I(X_i, Y_i)} \tag{5}$$

- **Frequency Analysis:** Using the Discrete Fourier Transformation (FFT), it is possible to obtain the frequency spectrum of each input/output node vector. Using the matched filter method to correlate the spectrums, as shown in the formula below, where X_i^F Y_j^F are the frequency spectrums of each input/output node.

$$d(X_i, Y_j) = \frac{1}{M(X_i^F, Y_j^F)} \tag{6}$$

  Based upon the outputs of these distance functions, it is possible to analyse whether an input node and an output node are communicating.

4.2 Attack Implementation

In this section we describe how we implemented the attack. In the paper, the authors identify two different batching strategies: Mix and Link based batching. Mix-based batching creates one output queue for the mix, whereby each packet is sent irrelevant to the output node. However, link based batching creates separate queues for each output node. We chose to implement the mix-based batching for this attack. Our implementation is as follows:

1. **Data Collection:** By reading through *output.txt* we gather the times of the packets being sent and received by the corresponding nodes. Based on the way we simulated the mix network, we classified the input and output nodes based on the "source" and "destination" IPs. These values are arbitrary, meaning that they are not IPs, and instead act as classifiers for simplicity. For the reasoning as to why this does not break the assumptions of a mix network see section 3.

2. **Flow Pattern Vector Extraction:** The corresponding flow vectors for the input/output nodes, were created by iterating through the A and B sets, and for each node and batch creating a separate *numpy* array.

3. **Distance Functions:** We iterate through the pattern vectors to determine the distance between input and output nodes. Using library *sklearn*, we iterate through each pattern vector to determine the distance between nodes using mutual information. For frequency analysis, we had to first determine the frequency spectrum by calculating the FFT of the input vectors using *numpy* FFT, then had to determine the matched filter correlation between the nodes. To do this, we first determined the dot product of the input and output X,Y frequency spectrums, then determined the square root of the dot product of the Y frequency spectrums. We entered divided the values, according to the formula, and determined the distances using both of the distance functions.

4. **Flow Correlation:** Inside the loops that determine the distances, we created a *min value* variable which takes the value of the lowest distance calculated. This means that for each input node, we calculate the distance to each output node, and set the output node with the lowest distance to the corresponding input node in the set *Similar Nodes*. Having compared each input node, to each output node, we therefore, create the flow correlation between the nodes and return that result.

5 Results and Metrics

The metric used to determine the effectiveness of the attack is the **Detection Rate**. In the paper, they define the detection rate as the ratio of the number of correct decisions to the number of the attempts. In our case, we have simulated the attack, and thus, instead, define the detection rate as the ratio of the number of correct detections to the number of detections. If all the detections made by the attack are correct, we therefore have a detection rate of 1, and 0 if none are correct.

5.1 Results

To showcase the robustness of the attack, an increasing amount of packets are sampled to be used in the attack. The number of packets sampled begin at 600 and increase by doubling the number of samples, in the paper, to 153600. We used the same methodology to determine the detection rate, whereby we ran the attack from 600 until 49995 iterations, to receive the final results faster.

In the paper, as can be seen in Figure 4, for both distance functions, when supplied with enough sample size, manage to get a detection rate of 1, whereby the attack proves, with 100% detection rate, that two nodes are related. This is proven in our implementation as well, whereby initially we had a detection rate of under 40%, but when the sample size was raised to 17000, we also got 100% detection rate. Most interestingly, the freq. analysis distance function in the paper managed to have a faster climb to a detection rate of 1, which as seen in the graph, we have also proven.

Therefore, the flow correlation attack described in the paper signifies that despite the layers of encryption for the communication between the mix network and the nodes, a simple analysis of the timing of each packet has leaked the exact information mix networks are built to anonymize.

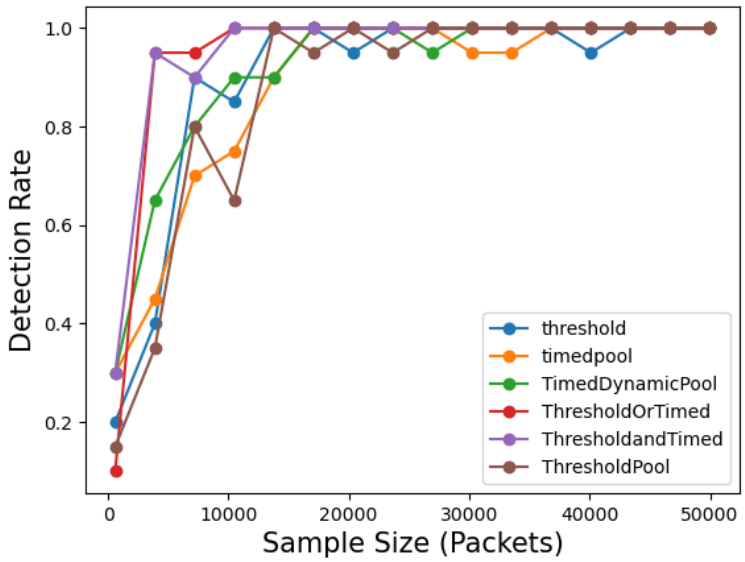

Figure 4: Final graphs after running each mix for up to 50000 iterations

6 Defences

6.1 Paper Suggestions

The article points out that traditional batching and reordering strategies are not enough to effectively counter flow correlation attacks. To defend against these attacks, a class of possible countermeasures can be developed based on making all the output flows of a mix look identical, which can be achieved by inserting dummy packets to make all the output flows behave similarly. However, the challenge is to insert a minimum number of dummy packets to maintain the network's usability.

The article presents an output-control algorithm that can ensure a maximum delay on each packet and guarantees that neither queue will overflow. The method can be extended and optimized for more complicated cases, and the number of virtual output links of a mix can be very large, and the overhead is limited. However, there is a need for a lower bound of the number of virtual queues required for each mix to maintain anonymity.

This paper is built on a relatively strong (for its time) threat model, as it disallows another aspect of traffic analysis that can be easily hidden: the size of the packets. Thus, all the proposed defenses against size-related analysis are already taken into consideration.

6.2 Relevant Literature Suggestions

Since Mix networks were believed to be the way to achieve anonymity on the internet, there has been plenty of similar work done to prevent time analysis attacks, a super-set of flow-correlation attacks. One of those, similar to the one proposed in this paper, is for all exit nodes to receive a constant rate of traffic, to be indistinguishable from each other, as proposed in [2]. Another interesting defense, proposed in [3] was "defensive dropping", that involved dummy packets to be inserted to a multi-layer mix. The innovation was that those packets which were there to obfuscate the traffic, are eventually dropped. However, at the same time [4] proposes a more powerful timing attack, that most of the defenses failed to protect against.

6.3 Alternative Solution: Onion Routing

With the idea of Onion routing being around, and with Tor being introduced [5], the scientific community then shifted away from Mix networks and the anonymization problems that they pose. After all, as mentioned in [6], anonymization heavily relies on the choices and habits of the "crowd" in which users are hidden. Mix networks are no exception, something that led researchers to halt their research on how to defend against those attacks and made them focus on investigating Tor and its characteristics.

7 Conclusion

In this report, we describe in a high-level how mix networks work to establish anonymity between communicating parties, as well as an attack on such networks whereby the timing of each packet being sent and received is used to establish which nodes are communicating. We replicated this attack by simulating an adversaries perspective of traffic analysis of a mix network such that they do not see any contents of the packets but only the times they are being sent and received. The attack is done by analyzing these times, and determining the distance between input and output nodes, **providing a 100% detection rate**, given enough packets are tracked. This is highly damaging for mix networks, and thus we have outlined the defences given in the paper, as well as in other similar pieces of work.

References

[1] Y. Zhu, X. Fu, B. Graham, R. Bettati, and W. Zhao, "On Flow Correlation Attacks and Countermeasures in Mix Networks," in *Privacy Enhancing Technologies*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, D. Martin, and A. Serjantov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, vol. 3424, pp. 207–225, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/11423409_13

[2] A. Pfitzmann, B. Pfitzmann, and M. Waidner, "ISDN-Mixes: Untraceable Communication with Very Small Bandwidth Overhead," in *Kommunikation in verteilten Systemen*, W. Brauer, W. Effelsberg, H. W. Meuer, and G. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, vol. 267, pp. 451–463, series Title: Informatik-Fachberichte. [Online]. Available: http://link.springer.com/10.1007/978-3-642-76462-2_32

[3] B. Levine, M. Reiter, C. Wang, and M. Wright, "Timing Attacks in Low-Latency Mix Systems," Feb. 2004.

[4] G. Danezis, "The Traffic Analysis of Continuous-Time Mixes," in *Privacy Enhancing Technologies*, ser. Lecture Notes in Computer Science, D. Martin and A. Serjantov, Eds. Berlin, Heidelberg: Springer, 2005, pp. 35–50.

[5] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: the second-generation onion router," in *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04. USA: USENIX Association, Aug. 2004, p. 21.

[6] R. Dingledine and N. Mathewson, "Anonymity loves company: Usability and the network effect," Jan. 2006.