Undermining LN privacy via gossiping, probing and network traffic timing

Utz Nisslmüller

Introduction

Privacy within the Lightning Network (as in the inability to correlate payments with their respective senders and recipients) is achieved via omitting the weights both parties have invested in a channel from the messages propagated by the gossip protocol. However, this comes along with a performance trade-off[1], as finding a route consists of iteratively probing channel paths, which have a sufficient total balance to satisfy a payment, for routes which have a sufficient total balance in the desired direction.

This short outline proposes two attacks on the privacy premise of the Lightning Network. The first attack makes use of the probing process mentioned above in order to correlate transaction flows between two Lightning network participants. The second attack times the arrival of specific Lightning Network messages[2] in order to determine the distance (in hops) to the payment recipient and make an educated guess about the route the payment has taken. Due to the onion routing properties, the transaction amount remains hidden under this second attack.

The attacks will be implemented by connecting to a local node on Ubuntu 18.04 and using the Python RPC interface.

Scenario 1: Monitoring node balances via continuous probing

The goal of this attack is to check for occurring payments between two Lightning Network nodes A and B. In theory, any party running a LN node should be able to do this by invoking standard RPC commands on its local node. The attack exploits the fact that routing a payment is not only possible between the querying node itself and a specified destination, but in fact it is possible to request a route between any two public nodes on the network. Let the A and B be connected by a set of Hops H = {Hop 1, Hop 2, …, Hop N} with each connecting channel holding a balance of 0.5 BTC, and we arrive at the following topology:

[1] https://eprint.iacr.org/2019/328.pdf
[2] https://github.com/lightningnetwork/lightning-rfc/blob/master/02-peer-protocol.md

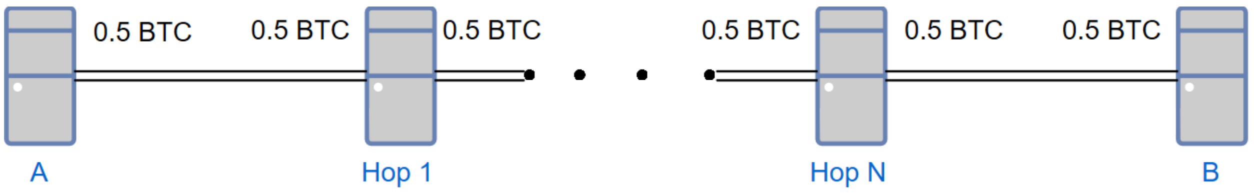

Figure 1

This, however, is not the view of an external node (e.g. a node connected to A which is not a part of H, such as A1 in Figure 3), trying to route a payment between A and B. This node will only see the total channel balances of 1.0 BTC per channel along the route. For the simplistic topology above, however, the maximum amount in millisatoshis which can be sent from A (src) to B (dest) can be determined as follows:

```python
def find_max_amount_for_route(rpc_object, src, dest, sensitivity=2.0):
    """Find an approximation of the maximum amount of funds (in millisatoshi)
      which can be transferred from src to dest. """
    amount_msat = 1000
    initial_loop = True

    # Geometric (fast) increase
    while initial_loop:
        try:
            test_amount_msat = int(amount_msat * sensitivity)
            route = rpc_object.getroute(fromid=src, node_id=dest,
                    msatoshi=test_amount_msat, riskfactor=0)
            amount_msat = test_amount_msat
        except RpcError as r:
            sensitivity = sqrt(sensitivity)
            if sensitivity == 1:
                initial_loop = False

    # Linear (slow) increase
    increment_amount = 1000.0
    while True:
        try:
            test_amount_msat = int(amount_msat + increment_amount)
            route = rpc_object.getroute(fromid=src, node_id=dest,
                    msatoshi=test_amount_msat, riskfactor=0, fuzzpercent=0)
            amount_msat = test_amount_msat
        except RpcError as r:
            increment_amount = increment_amount / 10
            if increment_amount < 1:
                return int(amount_msat - increment_amount)
```

This method will try to route a payment from src to dest with ever increasing payloads, until the limit for the given direction is reached. The return value for A and B will thus be ≤ 500,000,000,000 (millisatoshi), depending on the fees along the chosen route.

If a transaction containing 0.1 BTC occurs from A to B, the updated channel balances are as follows:

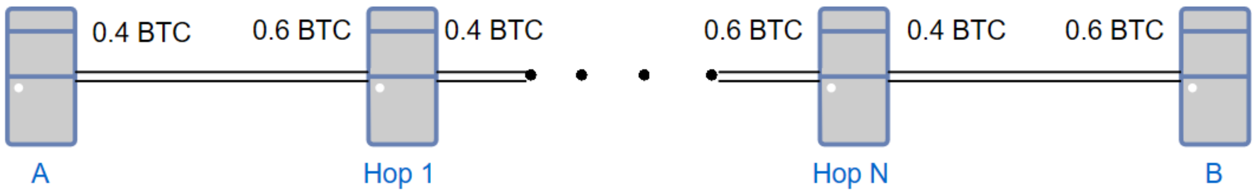

Figure 2

Following the transaction, the function shown above will also return an updated value for the maximum cash flow from A to B (now 0.4 BTC in the example above) and thus implying, that a transaction has taken place. However, this simplified network view omits the possibility of A and B being part of a larger network, as shown in Figure 3.

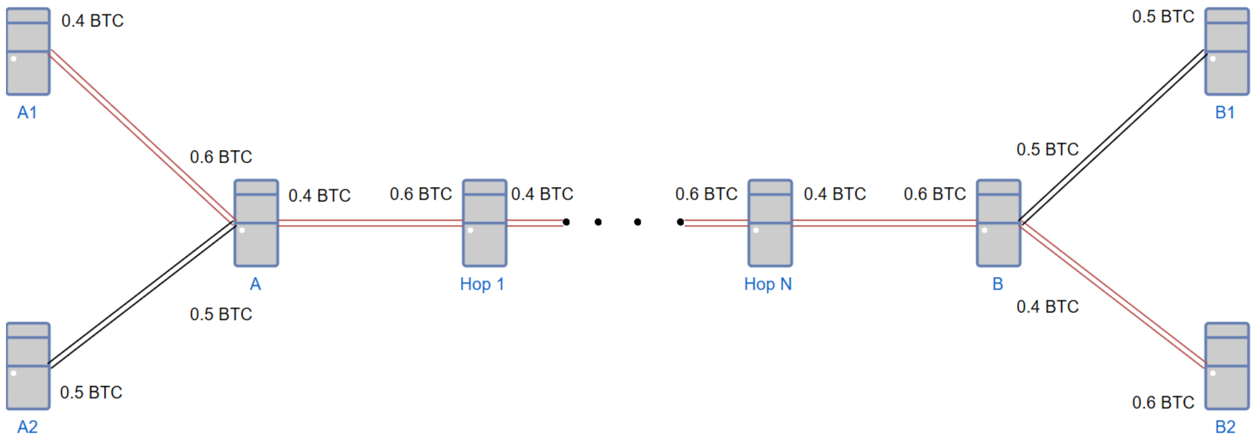

Figure 3

By simply monitoring the route between A and B as before, we arrive at the same view as in Figure 2, when in reality, a transaction has occurred between A1 and B2, using {A, H, B} to connect them. In order to deal with this complexity, it is necessary to also monitor the remaining edges (channels) at both A or B for a similar balance shift. Choosing node A as the node of interest in the example above, it is thus also necessary to run *find_max_amount_for_route()* between A1-A and A2-A as well as between B-B1 and B-B2. Assuming that prior to the transaction, channel balance was 1.0 BTC globally (weighted 50-50), trying to get a route for 500,000,000,000 millisatoshi will not only fail for A and B, but also for A1 and A as well as B and B2. It can thus be safely assumed that A and B were only hops themselves rather than the recipients.

Assumptions

This simplified scenario makes several assumptions, which don't necessary hold in the general Lightning Network:

- Nodes don’t demand fees for forwarding a payment. Accounting for fees adds a certain fuzziness and uncertainty to the tracking process, as the initial probing between A and B (at the time of Figure 1) will return slightly less than 500,000,000,000 millisatoshi. Depending on the fee policies of nodes A and B, a decrease in routable funds between A and B is not necessarily equally reflected by a tantamount decrease between A1-A and B-B2 (in regard to Figure 3).

- H is the only (feasible) route between A and B. Obviously, this is unlikely to be the case in the actual Lightning Network as both A and B might have several more channels with other nodes, which might have many more channels themselves, providing a multitude of possible routes between A and B.

- *find_max_amount_for_route()* returns valid results. With the function as it stands only being a first prototype, it remains to be seen whether it accurately returns the maximum routable amount between two nodes of interest.

Apart from examining the example above in further detail, the final paper will address these issues and discuss the impact of disregarding one or several of these assumptions. Another point worthy of further examination is how the use of riskfactor and fuzzpercent parameters[3] will affect the results.

Scenario 2: Passively listening for transactions by timing responses

The goal of this attack is to correlate payment flows by timing response times. The malicious node would be part of the route (or one possible route) between source and destination (therefore acting as a [non-altering] man-in-the middle node). Assuming a similar infrastructure to Scenario 1 (with the exception that Hop 1 is now connected to Hop N via a direct payment channel instead of several other hops), we arrive at Figure 4, with Hop N, marked in red, acting as the malicious node. The following example will omit transaction fees and payment channels have been provided with short channel IDs (scids) for demonstration purposes.

[3] https://lightning.readthedocs.io/lightning-getroute.7.html

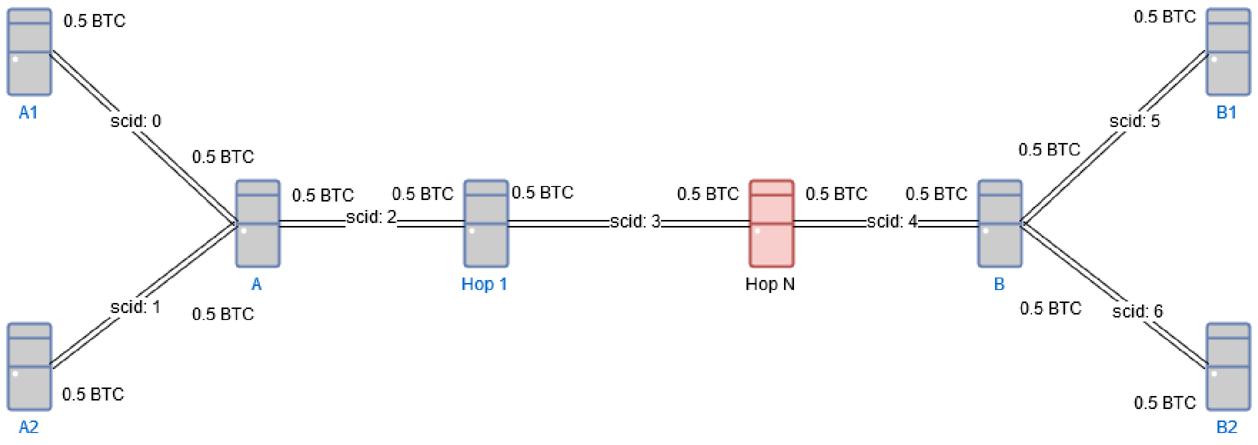

Figure 4

If, say, node A1 were to pay 0.1 BTC to node B, it first needs to find a viable route to B (here, R = {A, Hop 1, Hop N, B}). If a successful route has been found, an update_add_htlc message[4] is passed to the first node along the route, node A, upon which both nodes negotiate a HTLC:

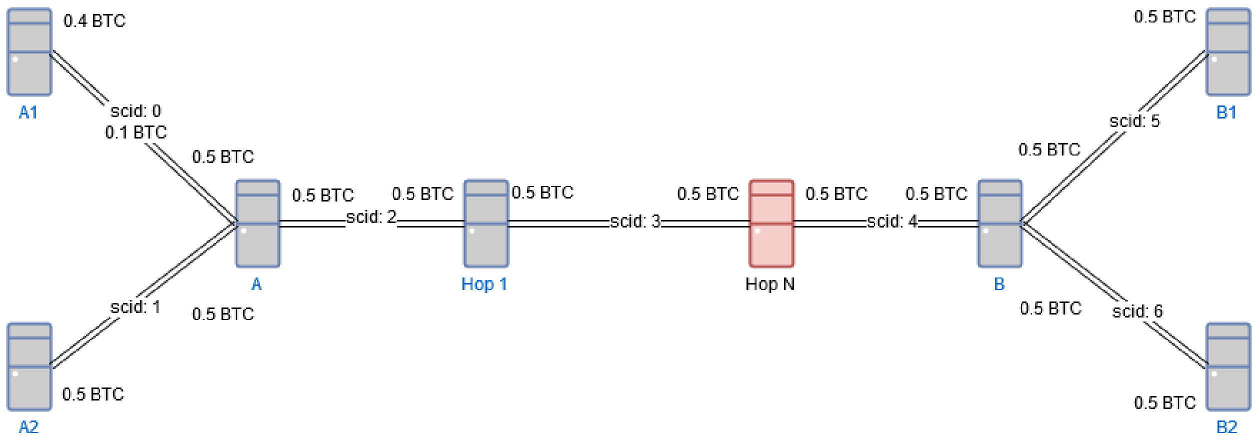

Figure 5

In similar fashion, HTLCs are set up between nodes A and Hop 1, as well as between Hop 1 and Hop N (the malicious node):

[4] https://github.com/lightningnetwork/lightning-rfc/blob/master/02-peer-protocol.md#adding-an-htlc-update_add_htlc

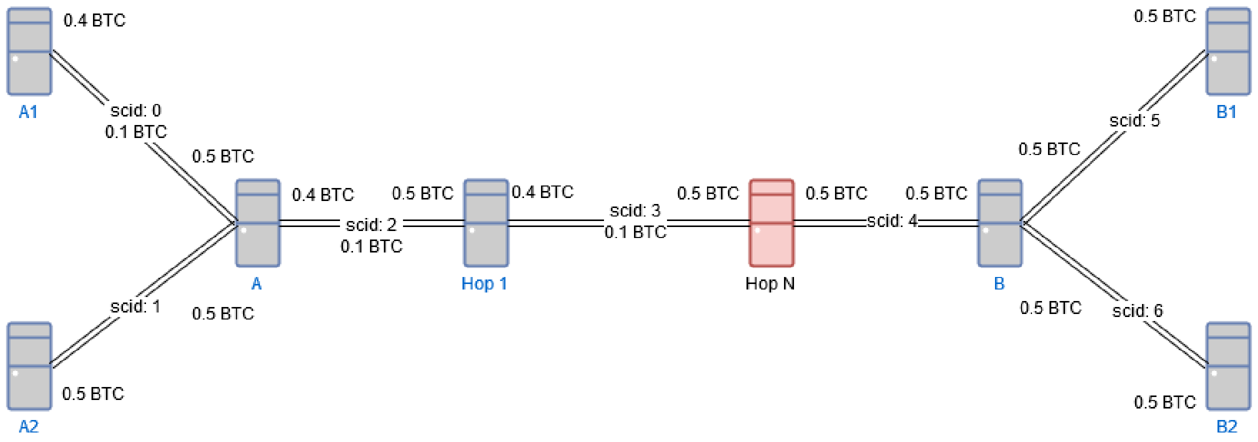

Figure 6

The next HTLC is decisive for this attack. Hop N routes the payment via another update_add_htlc message to node B, and the final HTLC is established on scid 4. However, since B is the final payment destination and hence is in possession of the payment preimage, it sends an update_fulfill_htlc[5] message to Hop N in order to in initiate the settlement of chained HTLCs along R (Figure 7). As this response from B is instantaneous, Hop N can time messages sends and receipts on scid 4 and therefore infer whether a payment has occurred to B, even though the padded Onion Routing mechanism[6] should prevent this.

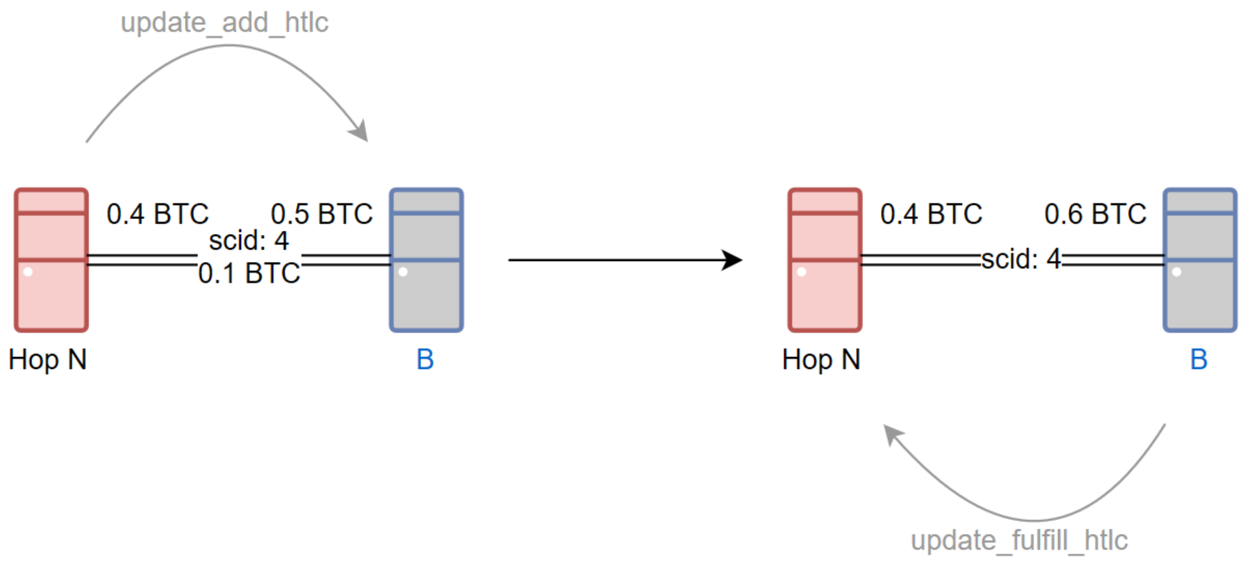

Figure 7

[5] https://github.com/lightningnetwork/lightning-rfc/blob/master/02-peer-protocol.md#removing-an-htlc-update_fulfill_htlc-update_fail_htlc-and-update_fail_malformed_htlc
[6] https://github.com/lightningnetwork/lightning-rfc/blob/master/04-onion-routing.md

Depending on the uniformity of network latency, it should be possible to determine whether a transaction has been sent to nodes connected to B (such as B1 or B2 in Figure 4). One aim of the final paper will be to determine the effective distance for which this attack provides reliable results. As in Scenario 1, it remains to be seen how effective this technique is in the actual Lightning Network topology and whether any potential issues can be circumvented by adapting the method.