Congestion Attacks in Payment Channel Networks

Ayelet Mizrahi and Aviv Zohar

The Hebrew University of Jerusalem
{ayelem02,avivz}@cs.huji.ac.il

Abstract. Payment channel networks provide a fast and scalable solution to relay funds, acting as a second layer to slower and less scalable blockchain protocols. In this paper, we present an accessible, low-cost attack in which the attacker paralyzes multiple payment network channels for several days. The attack is based on overloading channels with requests that are kept unresolved until their expiration time. Reaching the maximum allowed unresolved requests (`HTLCs`) locks the channel for new payments. The attack is in fact inherent to the way off-chain networks are constructed, since limits on the number of unresolved payments are derived from limits on the blockchain. We consider three versions of the attack: one in which the attacker attempts to block as many high liquidity channels as possible, one in which it disconnects as many pairs of nodes as it can, and one in which it tries to isolate individual nodes from the network. We evaluate the costs of these attacks on Bitcoin's Lightning Network and compare how changes in the network have affected the cost of attack. Specifically, we consider how recent changes to default parameters in each of the main Lightning implementations contribute to the attacks. Finally, we suggest mitigation techniques that make these attacks much harder to carry out.

Keywords: Lightning Network · Payment Channel Networks · Network Security · HTLC

1 Introduction

Payment channel networks such as the Lightning Network [26] are a second layer off-chain solution to the scalability problems of blockchains. They require participants to lock funds into channels, which then allows them to send payments to others over several hops. Altogether, they allow both a higher number of transactions as well as faster transaction resolution compared to the underlying blockchain.

In this paper we describe and evaluate a novel attack that locks funds in channels between honest participants that are potentially far away from the attacker, giving the attacker the ability to disrupt the transfer of payments throughout the network. In contrast to previously known attacks that locked liquidity in channels [29], the method we present here requires *lower* costs as it

requires the attacker to lock a smaller amount of liquidity. We evaluate these costs in the Lightning Network, where we show that spending less than half a bitcoin, the attacker can indefinitely lock up channels holding the majority of the funds currently assigned to the network. To summarize, our contributions are as follows:

- We leverage a known limitation of payment channels (their max HTLC limit) to form two different attacks on the Lightning Network (a network wide DoS attack and a more localized node isolation attack).
- We provide statistics on several aspects of the Lightning Network that are relevant to the attack which may be of independent interest.
- We provide a thorough evaluation of the amount of resources needed to conduct each attack and of the level of harm an attacker can cause.
- We report on small proof of concept experiments using actual lightning nodes (on a local test network) that confirm the feasibility of the attack.
- We propose and discuss several short-term mitigation approaches that make the attack more difficult to carry out and reduce its efficiency.

Our attack is based on the inner workings of the main mechanism that makes payment channel networks possible: Hashed Time-Locked Contracts (HTLC). Essentially, as payments are set up to move along some path in the network, all channels along the path reserve some funds for the transfer that is about to take place. The number of simultaneously reserved and unresolved payments per path is limited. Our attack thus simply opens many small payment requests along extremely long paths and keeps them unresolved for as long as possible. In this way, all channels along the path are unable to relay other transfers.

The vulnerability can be attributed to three fundamental properties of off-chain payment networks.

1. Payments are executed in a trustless manner. Payments are executed using conditional payment contracts (in the form of transactions with HTLCs) that are exchanged between parties and are only sent to the blockchain if disputes arise. These contracts grow in size as more conditional payments are pending, and so the total number of pending payments is limited by transactions sizes that can be placed on the blockchain. Bitcoin's Lightning Network is limited to at most 483 concurrent HTLCs [32], while Raiden [24], Ethereum's network, is limited to at most 160 due to gas costs [23].

2. Expiration times are long. To allow nodes to recover their funds if a malicious partner closes a channel that is part of a pending payment, HTLC expiration times have been set to allow nodes sufficient time to appeal such closures. In Bitcoin's Lightning Network things are even more severe: due to lower expressiveness of its scripting language, HTLC expiration times accumulate over the length of the path, reaching up to 2016 blocks – which typically take the Bitcoin network two weeks to produce.

3. The privacy of payments. Payment Channel Networks utilize onion routing that does not allow intermediate nodes on the path to recognize where payments originate and where they are going, allowing the attacker to act with impunity.

Payment privacy essentially prevents us from attributing blame to potential attackers and add mechanisms that effectively detect the attack.

A description of the attack. In order to paralyze channels, the attacker first adds a new node to the payment network. It then identifies a route suitable to attack, considering some restrictions on the path (maximum route length, locktime of intermediate nodes, remaining HTLC capacity) and maximizing the attack benefit (to lock channels with a large amount of funds or high betweenness value). It opens channels with the source and target of the route, and requests many tiny payments through this path, exhausting the number of simultaneously open HTLCs. Since the attacker is both the source and destination of this payment, it can choose to delay the final execution of the payment which would remove all pending HTLCs from the path. The path is then locked for long periods of time (up to several days). Just before expiration, the attacker sends an *update_failure* message to the previous node, which cancels the payment and reverts the state, avoiding a forced closure of the attacker's channel. This allows the attacker to re-run the attack once again and lock the same path for an additional period of time.

To successfully carry out the attack, the attacker needs liquidity on its outgoing channel as well as liquidity on its incoming channel. While it is easy to open a channel and invest liquidity, this liquidity is on the attacker's side of the channel initially which is suitable for the outgoing channel. The incoming channel's liquidity needs to allow for payments toward the attacker. Liquidity can be shifted in that direction by sending an outgoing payment from the attacker, e.g., to deposit funds in some exchange or purchase goods.

We stress that simple mitigation attempts such as increasing the number of HTLCs allowed per channel are not very effective. First, more allowed HTLCs will imply larger settlement transactions in Lightning (larger than the current block size). Second, the attacker can easily create enough payment requests to lock many more HTLCs (each message locks an HTLC for days and requires little effort).

In this paper we evaluate the attack specifically on the Lightning Network, which is the most prominently used payment channel network. We evaluate three main attack scenarios: First, we consider an attack on the entire network, which attempts to lock as many channels as possible and focuses on channels holding most of the funds in the network. This sort of disruption would severely hinder the volume of payments that can be sent on the Lightning Network. The second attack scenario we consider is one that disconnects as many pairs of nodes as possible and breaks the network into separate components. The main complexity in carrying out these attacks is picking routes in a way that respects limits on the maximal delay incurred along the path, and still targets the channels with the highest connectivity and liquidity. Finally, the third attack variant that we evaluate targets single nodes and paralyzes all channels that connect them to the network.

As far as we are aware, while exhaustion of the HTLCs of a channel is known to paralyze the channel, the attack that we describe has never been evaluated

for its effects on the network, or on individual nodes. In particular, there are no available estimates of the cost to attackers from executing either version of the attacks we propose.[1]

Due to ethical concerns, we did not attack the live network. Instead, we worked with two main complementary approaches. In the first, we use the code of actual lightning node implementations to set small local networks to test the basic mechanics underlying the attacks. We validate the behavior of nodes in a series of experiments reported in Appendix B. In the second, we perform the attacks on a simulation of the actual network based on topology data we extracted from a live node. We provide the full description and evaluation of the attacks in its different modes in Sections 4 - 5. In Section 7 we discuss prior work, including other DoS attacks on the Lightning Network. Our attack differs in that it requires fewer resources, repeating it indefinitely does not waste fees, and it does not require a direct connection to the victim node.

2   Background on the Lightning Network

The Lightning Network is the most widely used payment channel network to date. As of October 2020, it has more than 14k nodes and 37k channels and holds a total capacity of around 1100 BTC. We introduce some of the basic properties of the Lightning Network.

Hashed Time Locked Contracts (`HTLCs`*)* - conditional payments which promise an intermediate node on the channel that it can receive funds if it submits a cryptographic proof (pre-image of a hash) within a given timeframe (specified as a specific chain height). Each transaction that occurs in the Lightning Network is first set up by adding an additional `HTLC` output to every channel on its path. Once these are set up, the payment is executed by propagating the pre-image from the payment's recipient back along the path towards the sender. Once the pre-image arrives at some intermediate node, it can essentially guarantee that it can receive the funds (if it posts the transaction with the pre-image to the blockchain). The conditional payment is then removed from the channel and is replaced by a non-conditional reallocation of the funds.

The main problem with the approach above is that if several payments are being set up, the number of `HTLCs` on a channel grows. This implies that the transaction that will eventually be posted to the blockchain will be large – setting a natural limit on the number of `HTLCs` that can be simultaneously open on a channel.

HTLC Timeouts - Usually, channels are set up quickly and do not wait long for the pre-image to propagate. An *update_failure* message may sometime be

---

[1] We were able to find public record describing the basic idea of the attack, on a single channel [8,30]. We note that no full translation of this vulnerability to the entire network was previously considered. Due to the public nature of these posts, we did not perform a disclosure of the vulnerability to the devs.

returned instead of the pre-image if one of the intermediate nodes cannot or will not relay the payment. However, malicious nodes may withhold the pre-image and not propagate it back (or alternatively not complete the channel set up with `HTLCs`). In such cases, `HTLCs` are designed to expire. This is done using a CheckLockTimeVerify (`CLTV`) instruction, which essentially does not allow the `HTLC` to be redeemed after a certain block height. In order to ensure that intermediate nodes do not lose funds, outgoing `HTLCs` must expire before incoming `HTLCs` do. Each node specifies a parameter `cltv_expiry_delta`, which specifies the difference in timeouts it is willing to tolerate. The timeout of payments is therefore the accumulation of the `cltv_expiry_deltas` from the end of the route towards its beginning (the last node's timeout is limited by a parameter named `min_final_cltv_expiry` instead of `cltv_expiry_delta`). As `cltv_expiry_deltas` are typically either 40 blocks or 144 blocks, the timeouts of `HTLCs` can accumulate and often take days. Nodes impose a limit on the maximal timeout `locktime_max`, which is set to 2016 blocks (equivalent to 2 weeks). This high timeout makes the attack extremely potent.

Privacy - One of the goals of the Lightning protocol is to preserve the privacy of users – a fact that eventually aids our attack. For example, routing payments is done via Onion Routing which helps disguise the attacker. Additionally, the expiration of `HTLCs` is also conveyed along payment paths and to preserve privacy, senders are allowed to add arbitrary values to the initial delay. We exploit this fact to add to the expiration delay of `HTLCs` (up to the allowed maximum of two weeks).

3   Lightning Network Analysis

We begin our exploration of the current state of the Lightning Network by listing the default values for various parameters in the main implementations of the Lightning protocol. These are of interest since, as we show later below, most nodes use the defaults, and thus these heavily influence the state of the Lightning Network and its vulnerability to our attack.

3.1   Default Parameter Values

The BOLT (Basis of Lightning Technology) [32] specifications detail the protocol of Lightning Networks. In our work, we focus on the main three implementations: LND [18], C-Lightning [5], and Eclair [7]

Each of the implementations uses slightly different default values for parameters of interest. These are depicted in the table below, along with ranges or values specified in the BOLT.[2]

---

[2] We give the defaults used in mainnet. Testnet behavior differs slightly.

	LND	C-Lightning	Eclair	BOLT
cltv_expiry_delta	40	14	144	-
min_final_cltv_expiry	40	10	9	9
locktime_max	2016	2016	2016	$<5 \cdot 10^8$
max_concurrent_htlcs	483	30	30	≤ 483
dust_limit_satoshis	573	546	546	-
htlc_minimum_msat	1000	1000	1	-
fee_base_msat	1000	1000	1000	-
fee_proportional_millionths	1	10	100	-

Recent changes to the defaults have in fact made our attack easier to carry out: LND changed their cltv_expiry_delta default from 144 to 40 blocks (on Mar 12th, 2019) [27], which allows chaining more nodes in each path without reaching the locktime_max limit. Nodes running an old version may still hold the 144 default that was used prior to that.

Additionally, a locktime_max of 2016 was agreed upon by Lightning developers, in the 2018 Adelaide meeting to set the BOLT 1.1 specs [6]. This is an increase of previous values used in some implementations. Again, this allows for longer routes and longer expiration delays that make the attack more damaging and easier to carry out.

3.2 Network Statistics

We introduce some statistics on the parameters announced by nodes in channels on the Lightning Network.[3] In order to perform the calculations, we took snapshots of the Lightning Network mainnet. The information was obtained from a continuously running LND node. Our results correspond to a network snapshot taken on September 21st, 2020. We include additional analysis with snapshots taken over a period of 18 months for comparison.
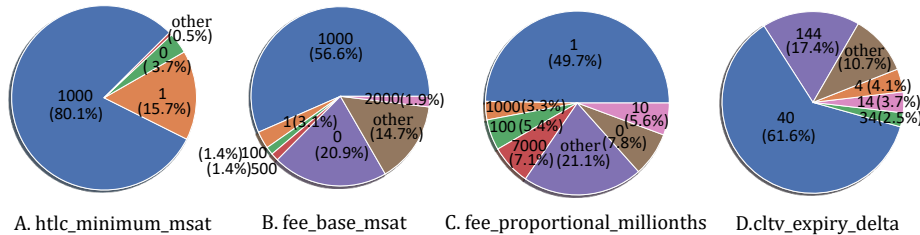

Fig. 1: Statistics on parameters announced by nodes in channels on the Lightning Network

In Figure 1, we present the most common values of four of the parameters announced by nodes. It is clear that very few values are used. The remaining values appeared less than 3% each (which we grouped together as "other").

---

[3] We ignore disabled channels and channels with nodes that do not reveal their policies.

Figures 1A,1B,1C show the distribution of `htlc_minimum_msat`, `fee_base_msat` and `fee_proportional_millionths`. These represent the minimal amounts nodes are willing to transfer, the flat fee for each transfer, and the fee that grows with the transferred amount. These values are small relative to the default configured dust limit, which sets a threshold below which `HTLCs` would not be added by nodes. Therefore, these parameters have a lower impact on the cost of the attack. We elaborate more on costs in Section 4.

Finally, we examine the distribution of `cltv_expiry_delta` - the minimum difference in `HTLC` timeouts the forwarding node will accept. We recall from the table in Section 3.1 that 144, 40, and 14 are the defaults that correspond to the different implementations mentioned previously. In Figure 1D, we see that the defaults constitute 82.7% of the total.

How do values change over time? In our attack, the route length we can compose is often limited by the values of `cltv_expiry_delta`. Figure 2 shows the changes in `cltv_expiry_delta` over an 18 month period[4]. We show only the most common values. The choice of presented dates was according to available information from our node and was slightly affected by downtime. Since channels are open for a long period of time, the exact day chosen does not impact the topology.
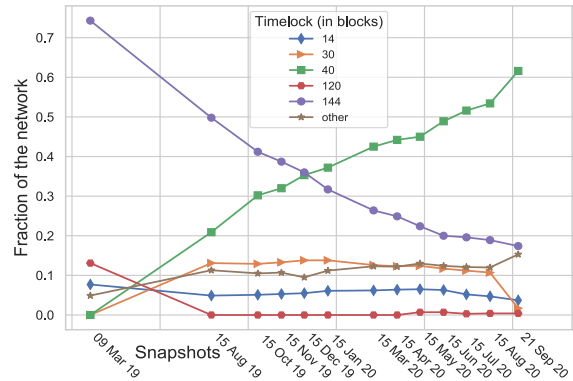

Fig. 2: `cltv_expiry_deltas` in different snapshots

The main change over this period is the decreased use of the value 144 for expiry time and the increase in the use of the value 40. We attribute this to the fact that LND changed their default `cltv_expiry_delta` from 144 to 40 in Mar 2019 [27]. In Appendix A, we show that LND nodes are both the most common nodes (we estimate that they constitute 91.3% of the network), and also the ones that hold most of the liquidity in the network.

---

[4] The snapshot from Mar 9th, 2019 was taken from [29,28].

4   Attacking the Entire Network

In this section, we consider a malicious node that wishes to disrupt the entire network's operation. Initially, it connects to other nodes in the network by opening channels with them, allowing it to learn the topology of the network and launch the attack. Then, the attacker uses a greedy algorithm in order to pick routes and paralyze as much liquidity or to disconnect as many pairs of nodes as possible. For each route, the attacker will initiate `max_concurrent_htlcs` payments, and withhold the response, turning all channels along the path unavailable for new requests. Just before expiration, the attacker will announce a failure to complete the payment. This step is repeated for multiple disjoint routes making the network less and less connected.

Examining several heuristics and path selection choices we found that a greedy choice of routes which we fully paralyze one by one makes the most out of every attacker's outgoing channel and achieves approximately optimal results for the attacker as we will present in the following evaluation. The main challenge faced by the attacker in this heuristic is to use routes composed of channels with similar `max_concurrent_htlcs` so that we do not leave parts of the path unlocked, and to fit as many high-liquidity channels within the limits of 20 hops and `locktime_max` total delay. Hence, we divide the network to subgraphs with similar `max_concurrent_htlcs`, and use a greedy algorithm to select routes. The algorithm that we utilized is parametrized by G (a subgraph of the network) and a parameter τ_{min} that denotes the minimal time (measured in blocks) that we would like paths to be locked for. We assign two types of weights to the edges (used in selecting the routes to attack) to support two different modes of attack. The greedy algorithm selects routes one by one, constructing each by consecutively picking high weighted edges. In the first mode we seek to freeze as much liquidity as possible and use the channel capacity as the weight. In the second mode, we seek to disconnect as many pairs of nodes and use the unweighted betweenness centrality measurement of edges as the weight (taking inspiration from the Girvan-Newman Algorithm [10]). A concise description of the algorithm follows (Appendix C provides a detailed description):

1. Pick a channel of maximal weight (capacity/betweenness).
2. Extend it to a route by repeatedly choosing an adjacent channel of maximal weight which meet the constraints of having similar `max_concurrent_htlcs` value and maintaining route validity (maximum route length and desired route locktime τ_{min}).
3. Remove the route channels from the graph.
4. Repeat until all channels are exhausted.

The result is a partition of G's channels into disjoint routes that can be paralyzed for at least τ_{min} blocks. Note that routes produced by the algorithm are circular (from the attacker to itself) and require two attacker channels: to begin and end each route.

For many channels in the network, the value set for `cltv_expiry_delta` is different depending on the direction we traverse the channel (this is because

nodes may have set different values for this parameter). Our greedy approach excelled at picking directions with lower `cltv_expiry_delta` values naturally, which allows it to form longer routes that paralyze more channels simultaneously. Other approaches that we explored, such as iterating over a single channel back and forth to form a long path, resulted in slightly worse performance.

The greedy algorithm does not optimize over the `htlc_minimum_msat` values when picking routes (Algorithm 1). The values set for this parameter are extremely low and their impact on the total cost is minor.

4.1   Evaluation

We run the attack locking channels for at least 3 days (τ_{min} = 432). We begin by attempting to freeze up a large amount of liquidity (setting the weight in Algorithm 1 to the channel's capacity).

We infer for each node, which implementation of the protocol it runs (See Appendix A), and then partition the network into two sub-graphs:

1. The network graph reduced to LND nodes. Which has `max_concurrent_htlcs` defaults that are 483.
2. The complementary graph consisting of all channels with at least one Eclair or C-Lightning node. These use a default `max_concurrent_htlcs` of 30.

In the implementation inference process we assume most users use the default values for the `max_concurrent_htlcs` parameter. This assumption is reinforced by Section 3.2, which presents distributions of other parameter values displaying high correspondence to their default values.

We visualize the results in Figure 3a, presenting the fraction of the network's capacity that the attacker succeeds in locking as a function of the resources it invests (the number of channels it opened). We find for example that the attacker can lock 20% of the network's capacity using only 68 channels, and can lock 90% using 1030 channels. We notice that the greedy algorithm is almost optimal on our graph. To do so we compare to an unachievable upper bound which is calculated as follows: The maximum allowed route length is 20. The attacker uses 2 channels to attack any route, hence it can attack at most 18 channels per route. We sort the channels by their capacities and use the highest capacity edges first, disregarding the constraint that paths are connected correctly.

We estimate the attack's costs, by considering two types of costs:

1. The cost of opening channels. The attacker pays the fee required to place channel funding transactions on the blockchain. We estimated the cost of opening a channel to be 2.2 USD (the average transaction fees observed on the date of the snapshot which the evaluation was performed on) [4].
2. The cost of provisioning channels with liquidity. Attackers must lock enough liquidity for payments they will later request. Locked funds are not spent and will return to the attacker once it completes the attack.

(a) Fraction of attacked network capacity   (b) Cost of the attack
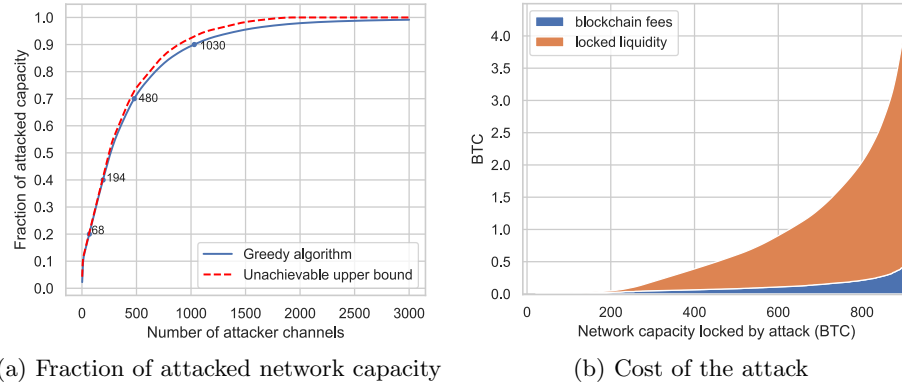
Fig. 3: Evaluation and cost of the attack

Figure 3b displays our evaluation of the costs. It clearly separates the two types of costs mentioned above (non-refundable blockchain fees and locked liquidity). Our results show that the attacker can paralyze most of the liquidity in the Lightning Network for 3 days spending less than half a Bitcoin. We take into account the dust limit configured in the main Lightning implementations which sets a threshold on the payment size below which `HTLCs` would not be added by nodes. The locked liquidity cost is mainly affected by the dust limit, the rest of the parameters (min `HTLC`, base fee, minimal channel capacity) have less of an impact because of their small values. The costs we estimate above can be further lowered by opening multiple channels with a single on-chain transaction. Once channels are established, the attack may be repeated again and again with no additional cost to the attacker.

To be able to block a route there needs to be sufficient balance in each channel to allow for a minimum payment (otherwise nodes along the path will reject the payment request). The required balance relies mainly on the dust limit and the `max_concurrent_htlcs` values configured along the route.

We show more details on the attack results in Figure 4. The figure shows that the attacker succeeds in attacking long routes (exploiting maximum route length), and that most of the routes are locked for more than the 3 days that were set as the minimal lock time.

In Figure 5a, we run the attack changing the number of days that channels remain locked for. The results indicate that the number of attacker channels required to lock paths for different periods (from 1 to 6 days) differs only slightly. This can be explained by the relation between the large `locktime_max` (2016 blocks) value, the small `cltv_expiry_deltas`, and the 20-hop route length constraint. In other words, most of the liquidity of the network can be attacked using routes that consist of small `cltv_expiry_deltas`, allowing the attacker to high timeouts and withhold the payments for a long period.

Figure 5b explores how the attack would work on the Lightning Network at different times. We use snapshots taken over several months. The results gener-
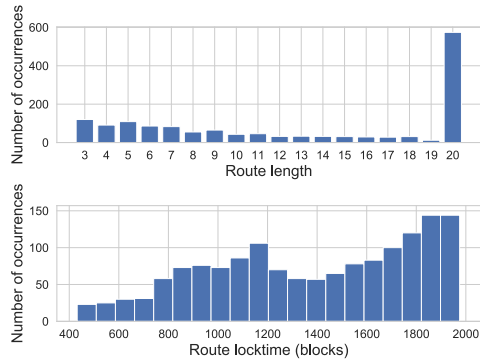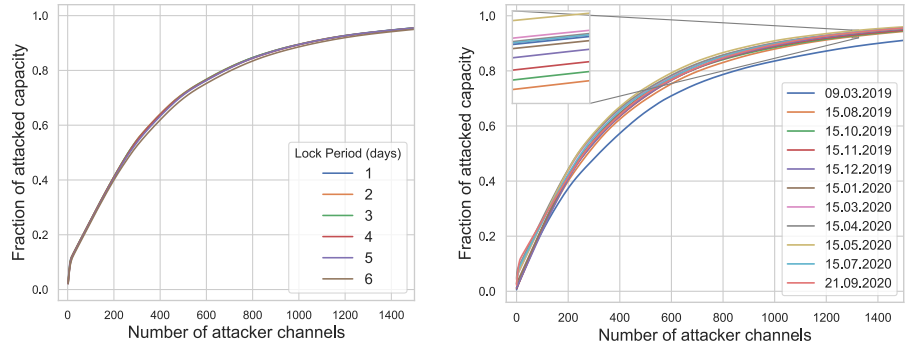
Fig. 4: Histogram of route lengths (including attacker's edges) and route lock times

(a) Fraction of attacked network capacity for different lock periods

(b) Fraction of attacked network capacity in different snapshots

Fig. 5: Evaluation of the attack

ally show that the attack gets *easier* as time passes (there is a slight improvement from May 2020). This can be explained by the changes made to default parameters – increasing `locktime_max` to 2016 in all implementations and decreasing `cltv_expiry_delta` from 144 to 40 in LND. Both changes make it easier to construct long routes with high timeouts.

In figure 6, we show how the attack affects connectivity between nodes in the network. We explored several algorithms to select the attacked routes and present them in Figure 6. The algorithms we explored are: Using Algorithm 1 – a greedy algorithm which picks channels with high betweenness centrality. The second approach utilizes spectral clustering to repeatedly cut the large connected component using an eigenvector corresponding to the second smallest eigenvalue (Fiedler vector) of the Laplacian matrix of the largest connected component. The sign of the coordinates partitions the vertices of the graph into 2, defining a
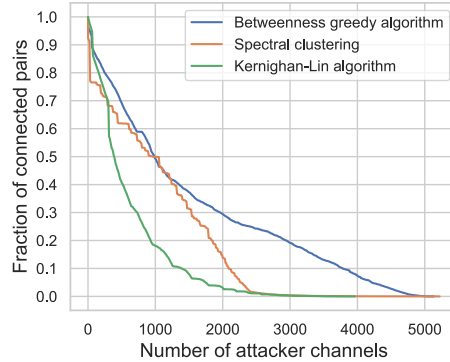
Fig. 6: Fraction of connected pairs of nodes in the network

cut [9]. Finally, we used a simplified version of Kernighan-Lin algorithm [15] that starts with an arbitrary partition that separates 1/4 of the nodes and greedily swaps nodes across the cut to minimize the cut. This yielded the best results.

While before the attack almost all pairs of nodes (>97%) are connected, using only 32 attacker channels we disconnect 23% of the pairs in the network, while with 385 channels we disconnect 50% of the pairs. We stress that right now different Lightning implementations try only a small handful of paths [35], so even a large fraction of nodes that we noted as connected will not be able to route payments between them.

5   Attacking Hubs - Attack on a Single Node

In this section we consider an attack aimed at disconnecting a single node from the network for an extended period of time. Here, the adversary connects to the victim and paralyzes its adjacent channels one by one using the following steps:

1. The adversary connects to the victim with a new channel.
2. It then initiates a payment to itself via a route that begins with its connection to the victim, and then traverses a single target channel back and forth multiple times, before returning to the attacker. Surprisingly, such paths that traverse channels back and forth are indeed possible (see Appendix B).
3. The attacker makes multiple payment requests over this path until the target channel reaches `max_concurrent_htlcs`. In this case, the attacker's own channel is usually not maxed out, and can be used to attack again.

We note that the attack is still possible to carry out if the victim does not accept direct connections (but at a somewhat lower efficiency). In this case, we would connect to neighbors of the victim.

Once the target channel is paralyzed, we move to the next one and apply the same method. We will need to open a new channel between the adversary and

the target node every time that the former reaches its `max_concurrent_htlcs`. Yet, at each payment we withhold only two `HTLCs` on the adversary's channel while it is possible to reach up to 18 `HTLCs` in the target channel at the same time. In other words, in order to attack all of the victim's channels, the adversary needs to open a small number of channels relative to the victim's degree.

5.1   Evaluation

We evaluate the attack on prominent nodes in the network. The following table summarizes our results:

Alias	% of Network Liquidity	Node's Degree	Attacker Channels
ACINQ	10.8%	774	151
Bitfinex [lnd1]	6.4%	169	19
OpenNode	4.2%	648	88
Bitrefill	3.8%	229	39
CoinGate	3.1%	609	68
LNBIG (25 nodes)	22.2%	3835	405

The names of nodes were taken from our snapshot data directly. The last entry in the table relates to an attack on LNBIG [17], a single entity that controls 25 nodes which are extremely central to the network, holding a significant share of the network's capacity in multiple channels. We isolate all 25 nodes, without paralyzing links between the nodes themselves. Paths were set so that all links are paralyzed for at least 3 days in each iteration.

We evaluated the cost of attack on *all* nodes in the network using a snapshot from September 21st, 2020, isolating each node for 3 days. Figure 7 presents a histogram of the degree of nodes and shows the relation between the degree and the number of channels attackers needed to perform the attack on each node. Each node is represented by a point in the graph. The number of channels is not directly determined by the degree, because different nodes have set up different values of `cltv_expiry_delta`. We see that most nodes have a very low degree and are extremely easy to isolate. Even nodes with high degree, require far fewer channels than the degree to attack.

In an additional evaluation which is described in Appendix D, we show that of the 3 main implementations of the Lightning Network, LND (the most common implementation) nodes are the easiest to attack.

6   Mitigation Techniques

The attack and vulnerabilities described in our work continue to be relevant and have been discussed by the Lightning community [3,1]. In this section we discuss several proposed adjustments to payment channel network protocols that may help mitigate the attack. Specifically, we discuss some ideas that were raised in the Lightning-dev mailing list [8,30], as well as our own suggestions. We discuss weaknesses and strengths of each such suggestion.
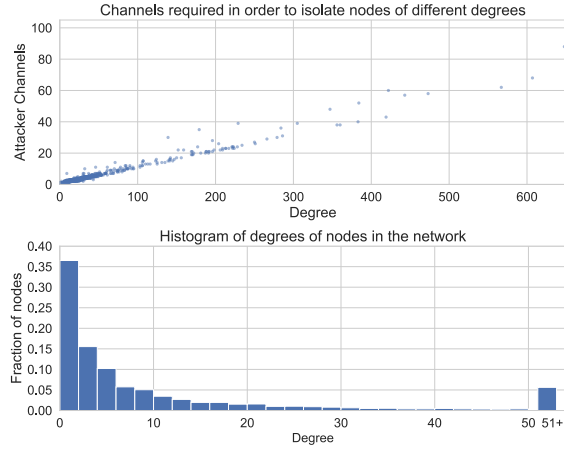
Fig. 7: Degree analysis

Enforcing fast HTLC resolution - This is our most drastic suggestion: While HTLC expiration times allow nodes to remain secure and provide sufficient time to publish transactions to the network, we propose the addition of another timeout mechanism. Specifically, if HTLC secrets are not propagated fast enough from one's neighbor the channel with this neighbor should be closed.

Each node should announce to its successor in the path its own deadline for resolving the HTLC. The node would then be able to communicate an earlier deadline for HTLC resolution to its next hop. If the timeout arrives, and the HTLC was not fulfilled or canceled, the node will wait for the HTLC to naturally expire but will close the channel with its neighbor.

To avoid having all channels along the path closed due to a failure to complete the HTLC in time, and specifically to avoid closing channels between compliant nodes, the last node in the path will provide proof of the channel closure to its predecessors (this can be done using a zero-knowledge proof for example).

We stress that this proposed mechanism does not replace the HTLC timeouts that still ensure the safety with regards to the *current* payment. Our mechanism is a way to disconnect misbehaving peers from the network in order to prevent them from repeating the attack many times at no cost. We note that it is risky to add behavior that automatically closes channels, and so this proposal warrants further evaluation. We leave this to future work.

Reducing route length - We suggest lowering the maximum allowed route length (currently 20 hops), as suggested in previous work [25]. The network graph is a small world network [29] - it is highly connected, and a smaller number of hops should still suffice. We point out that shortest paths between nodes in the network have an average of less than 3 hops and that the network diameter is ~ 6 [29,31], which are significantly lower than the 20 allowed hops. In Figure 8, we show the fraction of successfully attacked capacity (with respect to the attack

described in Section 4), assuming that different max route lengths are allowed. The figure shows that attackers need many more channels to attack if they are forced to use shorter route lengths.
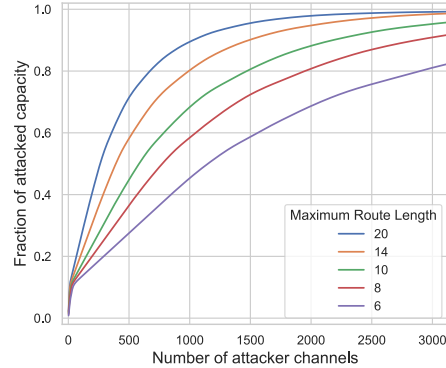

Fig. 8: Fraction of capacity attacked for different max route lengths

Setting number of max concurrent payments based on trust level - Currently, each node configures `max_concurrent_htlcs` to bound the maximum transfers it is willing to hold concurrently. Most nodes use the default value configured by the implementation they run. In all cases this value should not exceed the number 483 which is derived from the blockchain's limitations. We suggest changing the way nodes configure this parameter, adjusting the value according to the level of trust they have in particular peers. Setting a high `max_concurrent_htlcs` for some peer effectively allows it to route many concurrent payments through your node and to do more damage if it is malicious. Therefore, newly created channels with unknown and untrusted nodes should default to a low `max_concurrent_htlcs`. A new pull request has been opened recently (Aug 2020) promoting the basis of our proposal, allowing nodes to limit their exposure to the maximum number of concurrent `HTLCs` [1].

Loop Avoidance - As our experiments show (Appendix B), it is possible to construct paths that visit the same node several times, including traversals of the same channel back and forth. It is relatively simple for nodes to disallow such paths. Since `HTLCs` that belong to the same path use the same hash, they can be easily recognized and rejected. This will make our specific technique to isolate individual nodes harder to carry out, but may not solve the issue entirely.

7   Related Work

A DDoS attack on the Lightning Network occurred in March of 2018. Many nodes were flooded with traffic and around 200 Lightning nodes were taken offline [36].

Several studies explore sophisticated attacks on the Lightning Network. Some focus on privacy issues [14,33], and others on isolating nodes [29,25] or disrupting the network in other ways. Rohrer et al. [29] explored an attack that disrupts the liquidity balance of channels. The attacker initiates payments that move all the liquidity to one side, effectively blocking payments in that direction (payments in the other direction are still possible). Our attack differs from this attack [29], as they require direct connections to the victim node, as well as locked liquidity in high amounts (up to the liquidity the victim has), in addition to the payment of fees for large transactions.

A similar attack uses payment griefing but avoids paying the fees [25]. In this variant of the attack, the attacker still sends a payment in one direction that unbalances the channel in order to isolate a node. This time it withholds the `HTLC` pre-image in order to lock the amount, and never really executes the payment. Unlike our attack, this still requires amounts of locked funds that match the amounts being locked in the victim channel, but does indeed avoid paying most of the fees (channel establishment is still needed).

Tochner et al. [35] presents a denial-of-service attack based on route hijacking within the Lightning Network. They show how connecting with few channels to the network offering low fees draws most of the routes which yields a potent attack. Our work does not rely on the routing strategy of nodes to attack the network.

In a work parallel to ours, Tikhomirov et al. [34] quantify the effect of several attacks on the Lightning Network. They additionally discuss the limitation on the number of concurrent `HTLCs` and describe the attack vector and its effect on the network's scalability. We describe how to leverage the attack and provide deeper analysis.

The privacy of payments in the Lightning Network is known to be relatively weak. Discovering the current liquidity balance of a channel can be accomplished using techniques from Herrera-Joancomarti et al. research [14]. Tang et al. [33] explore the tradeoff between privacy and utility in PCNs, considering adding noise to channels as well (which adds privacy, but lowers efficiency).

The structural properties of the Lightning Network and its topology in the context of the network's robustness have been studied in several studies [31,16].

A technique to lower the delay of `HTLC` expiration is described by Miller et al. [22] and would make our attack less severe in this context. This sort of technique is not applicable to the Bitcoin blockchain, due to its more limited scripting language.

Several protocols improving upon privacy issues in off-chain payment channels are suggested in previous studies [11,13,19], as well as advances in payment channel networks like the addition of watchtowers [20,2].

McCorry et al. [21] presents a technical overview of Bitcoin's payment channel networks. Additional work on off-chain protocols can be found in SoK survey [12].

8   Conclusions and Future Work

In this paper we discussed a fundamental vulnerability that arises in payment channel networks as part of the construction of trust-less multi-hop payments. We presented three types of attacks: the first aims to lock as many high liquidity channels as possible for an extended period, the second disconnects as many pairs of nodes as possible in the network, and the third isolates hubs from the rest of the network. We evaluated these attacks over the Lightning Network. We examined the network's properties and different parameters set by the three main implementations of the Lightning Network. We showed how recent changes in default parameters agreed upon by Lightning Devs have made the attack easier to carry out. Our results show that it is possible to disrupt the Lightning Network at a relatively low cost.

Further work must be conducted in order to mitigate this type of attack. We suggested several solutions to reduce the success rate of these attacks, but such mitigation is generally harder due to the nature of the attack: it relies on several fundamental properties of payment channel networks, and the blockchain.

9   Acknowledgments

References

1. fundingmanager: configurable remote max htlcs [lnd pull request #4527] (Aug 2020), `https://github.com/lightningnetwork/lnd/pull/4527`
2. Avarikioti, G., Laufenberg, F., Sliwinski, J., Wang, Y., Wattenhofer, R.: Towards secure and efficient payment channels. arXiv preprint arXiv:1811.12740 (2018)
3. Bastien Teinturier, A.R., Jager, J.: Spamming the lightning network (Oct 2020), `https://github.com/t-bast/lightning-docs/blob/master/spam-prevention.md`
4. BitInfoCharts: Bitcoin avg. transaction fee historical chart (2020), `https://bitinfocharts.com/comparison/bitcoin-transactionfees.html#3m`
5. C-Lightning: A lightning network implementation in c (2020), `https://github.com/ElementsProject/lightning`
6. Dziemian, C.: Summary of the second lightning development summit. `https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-November/001595.html` (2018), [Lightning-dev]
7. Eclair: A scala implementation of the lightning network (2020), `https://github.com/ACINQ/eclair`
8. EmelyanenkoK: lightning-rfc issue #182: Payment channel congestion via spam-attack. `https://github.com/lightningnetwork/lightning-rfc/issues/182` (May 2017)

9. Fiedler, M.: Laplacian of graphs and algebraic connectivity. Banach Center Publications **25**(1), 57–70 (1989)

10. Girvan, M., Newman, M.E.: Community structure in social and biological networks. Proceedings of the national academy of sciences **99**(12), 7821–7826 (2002)

11. Green, M., Miers, I.: Bolt: Anonymous payment channels for decentralized currencies. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 473–489. ACM, Dallas, TX, USA (2017)

12. Gudgeon, L., Moreno-Sanchez, P., Roos, S., McCorry, P., Gervais, A.: Sok: Off the chain transactions. IACR Cryptology ePrint Archive **2019**, 360 (2019)

13. Heilman, E., Alshenibr, L., Baldimtsi, F., Scafuro, A., Goldberg, S.: Tumblebit: An untrusted bitcoin-compatible anonymous payment hub. In: Network and Distributed System Security Symposium. NDSS (2017)

14. Herrera-Joancomarti, J., Navarro-Arribas, G., Pedrosa, A.R., Cristina, P.S., Garcia-Alfaro, J.: On the difficulty of hiding the balance of lightning network channels. Ph.D. thesis, Dépt. Réseaux et Service de Télécom (Institut Mines-Télécom-Télécom SudParis . . . (2019)

15. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. The Bell system technical journal **49**(2), 291–307 (1970)

16. Lee, S., Kim, H.: On the robustness of lightning network in bitcoin. Pervasive and Mobile Computing **61**, 101108 (2020)

17. LNBIG: Lnbig lightning nodes (2018-2019), `https://lnbig.com/#/our-nodes`

18. LND: The lightning network daemon (2020), `https://github.com/lightningnetwork/lnd`

19. Malavolta, G., Moreno-Sanchez, P., Kate, A., Maffei, M., Ravi, S.: Concurrency and privacy with payment-channel networks. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 455–471. ACM, New York, United States (2017)

20. McCorry, P., Bakshi, S., Bentov, I., Meiklejohn, S., Miller, A.: Pisa: Arbitration outsourcing for state channels. In: Proceedings of the 1st ACM Conference on Advances in Financial Technologies. pp. 16–30. AFT, Zurich Switzerland (2019)

21. McCorry, P., Möser, M., Shahandasti, S.F., Hao, F.: Towards bitcoin payment networks. In: Australasian Conference on Information Security and Privacy. pp. 57–76. Springer (2016)

22. Miller, A., Bentov, I., Bakshi, S., Kumaresan, R., McCorry, P.: Sprites and state channels: Payment networks that go faster than lightning. In: International Conference on Financial Cryptography and Data Security. pp. 508–526. Springer (2019)

23. Network, R.: Setting the number of pending transfers keeping the gas limit (July 2018), `https://github.com/raiden-network/raiden/commit/107b3c3700a7d6cac3eae8634f945c1b6095f91c`

24. Network, T.R.: An off-chain scaling solution (2020), `https://github.com/raiden-network/raiden`

25. Pérez-Sola, C., Ranchal-Pedrosa, A., Herrera-Joancomartí, J., Navarro-Arribas, G., Garcia-Alfaro, J.: Lockdown: Balance availability attack against lightning network channels (2019)

26. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments (2016)

27. (Roasbeef), O.O.: Git Commit: "lnd: lower default CLTV delta from 144 to 40". `https://github.com/lightningnetwork/lnd/commit/c302f1ea3a91ccfa382d56851d23f4c73656208c#diff-356ddb2e7efca712327c3b2d94d3afd3` (Mar 2019)

28. Rohrer, E.: Lightning network snapshots (2018-2019), `https://gitlab.tu-berlin.de/rohrer/discharged-pc-data/tree/master/snapshots`

29. Rohrer, E., Malliaris, J., Tschorsch, F.: Discharged payment channels: Quantifying the lightning network's resilience to topology-based attacks. arXiv preprint arXiv:1904.10253 (2019)

30. Russell, R.: Loop attack with onion routing. `https://lists.linuxfoundation.org/pipermail/lightning-dev/2015-August/000135.html` (Aug 2015), [Lightning-dev]

31. Seres, I.A., Gulyás, L., Nagy, D.A., Burcsi, P.: Topological analysis of bitcoin's lightning network. arXiv preprint arXiv:1901.04972 (2019)

32. Specifications, L.N.: Basis of lightning technology (BOLTs) (2020), `https://github.com/lightningnetwork/lightning-rfc`

33. Tang, W., Wang, W., Fanti, G., Oh, S.: Privacy-utility tradeoffs in routing cryptocurrency over payment channel networks. arXiv preprint arXiv:1909.02717 (2019)

34. Tikhomirov, S., Moreno-Sanchez, P., Maffei, M.: A quantitative analysis of security, anonymity and scalability for the lightning network. IACR Cryptol. ePrint Arch. **2020**, 303 (2020)

35. Tochner, S., Schmid, S., Zohar, A.: Hijacking routes in payment channel networks: A predictability tradeoff. arXiv preprint arXiv:1909.06890 (2019)

36. Trustnodes: Lightning network ddos sends 20% of nodes down (2018), `https://www.trustnodes.com/2018/03/21/lightning-network-ddos-sends-20-nodes`

A   Tagging Nodes by Implementation

Our attack is based on overloading channels to their maximum `HTLC` capacity, which is determined by the minimum `max_concurrent_htlcs` of the peers in the channel. The `max_concurrent_htlcs` is a configured parameter set by the node's owner. The default `max_concurrent_htlcs` in the mainnet configuration for different implementations is 483 for LND nodes, and 30 for C-Lightning and Eclair nodes.

   The information of which client a node runs and what value of `max_concurre-nt_htlcs` it uses is not accessible publicly. Hence, we use the data nodes do publish via `channel_update` messages in order to infer which implementation they run and deduce the `max_concurrent_htlcs` defaults. Here, we rely strongly on the assumption that most users do not change default values too much. This assumption is supported by the statistics we show in Section 3.2.

   We perform classification using defaults from the table in Section 3.1 in order to infer the implementation of each node. Where nodes deviated from defaults, we use a score that is weighted according to the parameter that was changed in order to infer the most likely original implementation. The weights we have used are:

- `cltv_expiry_delta` : 0.75
- `htlc_minimum_msat` : 0.2
- `fee_proportional_millionths` : 0.05

   For each node in the network, we check the compatibility of these parameters in the policies of its channels with the implementation defaults (using the weights) and decide the label. [5]
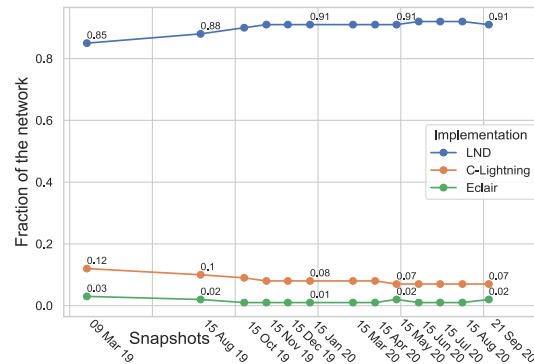

Fig. 9: Deduced implementation distribution

---

[5] The results were robust to the use of different classification methods that we tried. We found that 46% of nodes did not change *any* of their channels default values and that 57% of all *channels* were created with default values by at least one of their peers.

The results of the implementation inference at different times are presented in Figure 9. We see that our analysis resulted in tagging approximately 91.3% of the nodes as LND. When we then reduce the network graph to nodes labeled LND alone, we find that we remain with 73% of the network's original capacity (in BTC). Hence, LND nodes are both the most common nodes, and also the ones that hold most of the liquidity in the network.

B   Proof of Concept Implementation

To demonstrate the viability of the attacks, we conducted several experiments using C-Lightning nodes that were running over a separate Bitcoin network (in regtest mode). We detail the main experiments conducted—additional ones appear in code. The source code of our implementation including all experiments is available at github[6].

We begin with an experiment that shows that we can maintain a single `HTLC` "live" for an extended period, and that we can then revert the state without having any of our channels closed.

Experiment 1 (Maintaining a Single `HTLC`) *We set up a simple network Attacker1 ↔ Alice ↔ Bob ↔ Attacker2. Then Attacker1 initiates a single payment to Attacker2. Attacker2 does not respond with the `HTLC` secret immediately, but instead waits. We consider two different waiting periods:*

1. *Waiting past the `HTLC` timeout. In this case Bob closes the channel with Attacker2.*
2. *Waiting one block before the `HTLC` timeout, and sending an `update_fail_htlc` message. In this case the payment is canceled, the corresponding `HTLCs` along the entire path fail, but all channels remain open.*

The following experiment that we conducted demonstrates our ability to block payments over a channel between victims Alice and Bob.

Experiment 2 (Blocking the Victim's Channel) *We repeat the setup of Experiment1: Attacker1 ↔ Alice ↔ Bob ↔ Attacker2. We create 483 different payments from Attacker1 to Attacker2. Again, Attacker2 does not instantly respond with the secret.*

We now try to establish one more additional payment from Alice to Bob. The payment fails. Just before `HTLC` expiration, Attacker2 responds with `update_fail-_htlc` messages for all payments and now, an additional payment does succeed.

The above experiment also blocks the payment from Alice to Bob if some of the 483 payments are in the reverse direction (from Attacker2 to Attacker1), demonstrating that the `max_concurrent_htlcs` limit applies to `HTLCs` in either direction.

We additionally tried the experiment above in paths that contained loops (even including back and forth traversals of a single channel). These are all

---

[6] https://github.com/ayeletmz/Lightning-Network-Congestion-Attacks

allowed by nodes. We used this in a proof of concept experiment to attack a single hub.

Experiment 3 (Back and forth attack on a single hub) *We set up the network Attacker1 ↔ Hub ↔ Node ↔ Attacker2. Attacker1 initiates payments to itself in the following route, and does not respond with the secrets:*

1. *Attacker1 → Hub.*
2. *9 times back and forth on Hub ↔ Node.*
3. *Hub → Attacker1.*

After 26 such payments, Hub ↔ Node holds $26 \cdot 18 = 468$ unresolved payments. Sending an additional payment will fail ($27 \cdot 18 = 486 > 483$). Hence, Attacker1 sends 2 additional payments that are meant to fill up the remaining HTLC *quota:*

– *Similarly to the previous paths, only going back and forth 7 times on the channel connecting Hub ↔ Node*
– *Finally, a single payment to Attacker2: Attacker1 → Hub → Node → Attacker2*

We now try to establish a payment from Hub to Node, which fails, confirming that we did paralyze this channel. A payment from Attacker1 to Hub succeeds. In fact, Attacker1 succeeds sending 428 ($483 - 27 \cdot 2 - 1$) more payments to Hub while Hub ↔ Node is blocked. These "free" 428 payments may be used to attack other channels connected to this Hub.

In the next experiment we tried paths with varying `max_concurrent_htlcs` values, and verified that the minimal value constrains such paths. We further checked that only the edge with the minimal value is fully locked.

Experiment 4 (Varying `max_concurrent_htlcs`) *We set up the network Attacker1 ↔ Alice ↔ Bob ↔ Carol ↔ Attacker2. Attacker2 does not respond with the* HTLC *secrets. Attacker1, Alice, Carol and Attacker2 have* `max_concurrent_htlcs` *configured as 483, while Bob configured* `max_concurrent_htlcs` *to be 30. We create 30 different payments from Attacker1 to Attacker2. An additional payment from Attacker1 to Attacker2 fails. An additional payment including Bob in the route, fails. An additional payment from Attacker1 to Alice, or from Attacker2 to Carol succeeds. 453 additional payments from Carol to Attacker2 are accepted and wait for Attacker2 to respond.*

In the additional experiments, we also verified that paths are indeed limited to 20 hops and to 2016 block lock-time in total (aggregated over the entire path).

C   Greedy Algorithm Attack

We describe in Algorithm 1 the greedy algorithm that we utilize. The algorithm receives as input:

– **G** - a subgraph of the network.

Function ChooseRoutes(G, τ_{min})**:**

   $\mathcal{R}_G \leftarrow \varnothing$ # disjoint routes list
   $\mathcal{C}_G \leftarrow E(G)$ $channels$
   while $\mathcal{C}_G \neq \varnothing$ **:**
      $(n_{start}, n_{next}) = \mathcal{C}_G.pop\_max\_weight()$
      if $cltv\_expiry\_delta(n_{next}) < cltv\_expiry\_delta(n_{start})$ **:**
         $swap(n_{next}, n_{start})$
      $route \leftarrow (n_{start}, n_{next})$
      do
         $n_{cur} = n_{next}$
         $\mathcal{N} = Neighbors(n_{cur}, \mathcal{C}_G)$
         $\mathcal{L} = \{(n_{cur}, n) | n \in \mathcal{N}, CanExtendRoute(n, route, \tau_{min})\}$
         if $\mathcal{L} \neq \varnothing$ **:**
            $(n_{cur}, n_{next}) = \mathcal{L}.pop\_max\_weight()$
            $route \leftarrow n_{next}$
            $\mathcal{C}_G.remove((n_{cur}, n_{next}))$
      while $\mathcal{L} \neq \varnothing$ and $route.length < 19$
      $\mathcal{R}_G \leftarrow route$
   return $\mathcal{R}_G$

 # checks if adding the node keeps the route locked for at least
 τ_{min} blocks
Function CanExtendRoute(n, $route$, τ_{min})**:**

   $node\_delta = cltv\_expiry\_delta(n)$
   $route\_timeout = \sum\limits_{n_i \in\ route} cltv\_expiry\_delta(n_i)$
   if $node\_delta \leq 2016 - \tau_{min} - route\_timeout$ **:**
      return True
   return False

Algorithm 1: Splits G into disjoint routes that can be locked for at-least τ_{min} blocks

– τ_{min} - the minimal time in blocks that we would like paths to be locked for.

and outputs $\mathcal{R}_G$ — a partition of G's channels into disjoint routes that can be paralyzed for at least τ_{min} blocks.

The algorithm utilizes a "weight" function that is used to select the channels that are attacked. In some of our experiments we set this to be channel capacities (seeking to freeze as much liquidity as possible) and in some we set the weight function to be the unweighted betweenness centrality of the edges (seeking to disconnect as many pairs of nodes as we can). In the latter, we take inspiration from the Girvan-Newman Algorithm [10].

The **ChooseRoutes** method splits G into disjoint routes that can be locked for at least τ_{min} blocks. Each route is constructed by extracting a high weight channel, picking the direction which requires smaller `cltv_expiry_delta` for forwarding payments, and completing it to a circular route by repeatedly adding high weight channels (where channels had equal weight, we preferred ones with low `cltv_expiry_delta`). Only channels that keep the route locked for at least τ_{min} blocks are considered.

The **CanExtendRoute** method is used to check if an adjacent channel is suitable for extending the route by checking if with it, the route can still be locked for at least τ_{min} blocks.

We apply Algorithm 1 separately to subgraphs with similar `max_concurrent_htlcs`. We unify the outputs $\mathcal{R} = \bigcup_G \mathcal{R}_G$ and sort the set by routes weights in decreasing order as we want to attack routes with higher weight first.

D   Additional Node Isolation Evaluation

In Section 5 we evaluate the attack of disconnecting a single node from the network for an extended period of time.
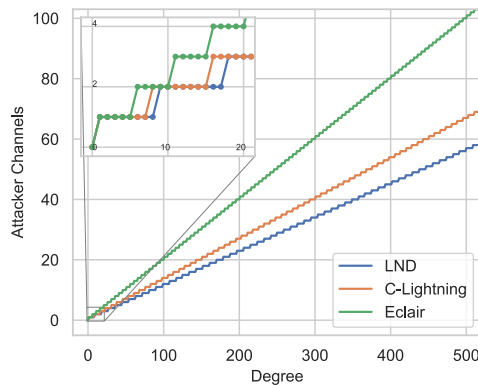

Fig. 10: Implementation analysis - number of channels required in order to isolate nodes of different degrees

We present an additional evaluation, estimating the cost of isolating nodes running one of the major implementations, assuming default values are used by it and its neighbors. We calculate the number of channels the attacker needs to open in order to isolate a node for 3 days for different degrees. We present the results in Figure 10. Notice that implementations differ due to their different default values. We recall from Appendix A that ~ 91.3% of nodes run LND. Figure 10 shows that these are the easiest to attack.

Eclair is hardest to attack due to its higher default `cltv_expiry_delta` value. LND's value of `cltv_expiry_delta` is higher than C-Lightning (40 vs 14), but it is still easier to attack due to its different `locktime_max` value (CLTV values were low enough so that they did not form a constraint—the number of hops was the main restriction on path length).