Outpost: A Responsive Lightweight Watchtower

Majid Khabbazian*
mkhabbazian@ualberta.ca
University of Alberta

Tejaswi Nadahalli*
tejaswin@ethz.ch
ETH Zürich

Roger Wattenhofer*
wattenhofer@ethz.ch
ETH Zürich

ABSTRACT

In the context of second layer payments in Bitcoin, and specifically the Lightning Network, we propose a design for a lightweight watchtower that does not need to store signed justice transactions. We alter the structure of the opening and commitment transactions in Lightning channels to encode justice transactions as part of the commitment transactions. With that, a watchtower just needs to watch for specific cheating commitment transaction IDs on the blockchain and can extract signed justice transactions directly from these commitment transactions that appear on the blockchain. Our construction saves an order of magnitude in storage over existing watchtower designs. In addition, we let the watchtower prove to each channel that it has access to all the data required to do its job, and can therefore be paid-per-update.

KEYWORDS

bitcoin, payment channels, lightning network, watchtower

ACM Reference Format:
Majid Khabbazian, Tejaswi Nadahalli, and Roger Wattenhofer. 2019. Outpost: A Responsive Lightweight Watchtower. In *AFT '19: Conference on Advances in Financial Technologies, October 21–23, 2019, Zürich, Switzerland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3318041.3355464

1 INTRODUCTION

Bitcoin [10] created a viable digital peer to peer payment system, and has been running for many years with no major problems. With a cap of 1MB on the size of each block, Bitcoin inherently limits the number of transactions that can fit into a block. The average size of a transaction is 300 bytes; with a block about every 10 minutes, the throughput is bounded to about 6 transactions per second. One may increase the block size and/or decrease the time between two blocks to achieve a higher throughput. However, these are consensus rule changes, and as such not easy to implement. Changing these parameters also adversely affect other security aspects of the Bitcoin network [8].

Duplex Micropayment Channels [6] and the Lightning Network [12] propose one type of solution to the scaling problem, allowing for higher throughput without changing Bitcoin's consensus rules. The idea of both these protocols is to handle most transactions outside the blockchain, in so-called channels. Bitcoin users would build a network of channels between them, and most transactions are handled in these channels. The Bitcoin blockchain would only be needed to setup and close these channels, and in this meta role, it handles far less transactions.

The Lightning Network in particular has seen implementations from multiple teams of developers and researchers (LND [4], Eclair [3], C-Lightning [2], LIT [9]), all implementing the same specifications [1]. All of these implementations build node software that helps form a peer to peer network of payment channels where value denominated in Bitcoin can flow from node to node.

However, there is still a major problem: Lightning channel payments can be received safely only if the receiving node stays online. Payment receivers risk losing payments if they go offline without closing channels that sent these payments, since a payment issuer can try to close a channel using an outdated earlier channel state. However, opening and closing channels are expensive blockchain transactions, which nodes want to avoid. To keep channels open *and* be able to go offline, nodes need the services of so-called watchtowers [11] to watch the Bitcoin blockchain and prevent fraudulent channel closures.

Watchtowers are *always-online* services run by impartial parties, either for an altruistic motive (to see the Lightning Network succeed), or for a business motive (to get paid by the participant(s) of a channel). If we consider the business motive, watchtower services can be paid either per fraud detected, or simply for watching. Given that the Lightning protocol punishes frauds, we posit that it is better to pay watchtowers based on their actual cost of watching, which is directly proportional to the amount of data they have

*Authors are listed alphabetically.

AFT '19, October 21–23, 2019, Zürich, Switzerland

ACM ISBN 978-1-4503-6732-5/19/10…$15.00
https://doi.org/10.1145/3318041.3355464

to store. This, in turn, depends on the number of channels they watch and the number of transactions that each of these channels have. On the other side, given that frauds are rare, we also need a mechanism which allows a watchtower to prove to channels that it is indeed doing its job. In our construction, after each transaction, the channel (one or both of its parties) sends the watchtower some data and fees. The watchtower, at any point in the future, can be asked for a proof by the channel that the watchtower was online as new Bitcoin blocks were mined and had access to this channel transaction data. This proof-scheme makes a pay-per-transaction scheme palatable to channel operators.

The watchtower's ability to perform its service depends on being able to watch the Bitcoin blockchain for a large number of transactions, with specific transaction IDs (or prefixes of IDs). Watchtower implementations need to have access to (and possibly store) this vast set of transaction IDs (*txids*) and accompanying data per *txid* that contain information on what to do when a specific *txid* shows up on the blockchain. LND's proof of concept implementation of watchtowers [5] requires around 300-350 additional bytes of storage per *txid*. A single watchtower could watch millions of channels and each channel could have billions of these micro-transactions. This places a large storage cost on the watchtower, as it has to store this 350 byte blob per transaction, for every transaction it knows about.

We propose Outpost, a construction that reduces this overhead to 16 bytes of additional storage. We do this using a novel lightning channel structure that changes the commitment transactions. In particular, we encode the information of a possible future transaction T_f in a present transaction T_p, so that T_f spends the output of T_p itself. This is non-trivial given how Bitcoin constructs and uses its *txids*. Outpost's reduction in storage costs will directly translate to the reduction in the operational cost of maintaining watchtowers. We believe that this will prompt more developers to run watchtowers, and thereby help the Lightning Network succeed.

2 BACKGROUND

2.1 Lightning Network

The Lightning Network is a peer to peer network of nodes running the Lightning node software. Each peer is connected to other peers through a specific construct called a payment channel. A payment channel is opened with a Bitcoin transaction that commits UTXOs (Unspent Transaction Outputs) controlled by two parties into a single output that is now controlled by a multisig that both parties have to sign to be able to spend in a future transaction. This is called the opening transaction (*topen*). Once the payment channel is opened, the two parties exchange signed Bitcoin transactions between each other. In these signed transactions, the total value of *topen* is allotted to each party depending on how the parties want value to flow between them. For example, if the payment channel was opened with 5 BTC from Alice and 10 BTC from Bob, a subsequent state might split the total 15 BTC of the channel so that Alice gets 7 BTC and Bob gets 8 BTC. This new split indicates a 2 BTC value flow from Bob to Alice, possibly for some goods or service that Bob received from Alice. This new division of *topen's* balance is established by Alice and Bob by exchanging partially signed commitment transactions (*ctxs*) with each other that they can sign themselves and broadcast later. At this point, the payment channel can also be closed with a closing transaction if both Alice and Bob agree to it. This is done by signing the multisig UTXO created by *topen* and sending 7 BTC to Alice and 8 BTC to Bob.

Typically, a channel is kept open by exchanging further *ctxs* that change the division of the balance between Alice and Bob as more goods and services go from Alice to Bob or vice versa. Note that at any time, if either party goes permanently offline, the counterparty can sign and broadcast their latest *ctx* to "commit" the latest state of the channel to the blockchain. The ability to unilaterally close the channel in case the other party goes offline makes this construction trustless. As a penalty for unilaterally closing the channel, the broadcasting party is made to wait for a timelock, whereas the counterparty (the one who might have gone offline) gets to spend their share of the channel instantly. This setup can be argued to be fair, because if a party broadcasts their *ctx* even if the counterparty is online, they get their share of the balance, but have to wait to spend it. The counterparty does not have to wait in this case.

Importantly, a party can try to unilaterally close a channel with a *ctx* (say, *previous_ctx*) that is not the latest agreed upon *ctx* (say, *latest_ctx*). Every party potentially has many such *previous_ctxs* in their storage going back all the way to the channel opening. This allows the dishonest party to cheat the honest counterparty by picking an old, more favorable *previous_ctx* from the past and broadcasting it. Lightning channels handle this cheating possibility by allowing *latest_ctx* to be exchanged only if they are also accompanied by ways of revoking the immediate *previous_ctx*. This revocation is handled through a revocation key that can allot the entire channel balance to the victim's control. This gives both parties a strong incentive to be honest. In case Alice tries to cheat by publishing a *previous_ctx*, Alice does not get her share of the channel balance immediately because it is timelocked, thereby giving Bob a time window to penalize this cheating *previous_ctx*. Bob looks up its corresponding revocation key that he got from Alice earlier, and uses it to construct the so-called justice transaction (*jtx*) to penalize this

previous_ctx. To be able to detect cheating, Bob has to monitor the blockchain for all *previous_ctxs* so that he can then construct the corresponding *jtx* and broadcast it. This is possible only if Bob is online whenever a new Bitcoin block is mined. If Bob is offline, Alice can cheat Bob by broadcasting a *previous_ctx* that is more favorable to her than the current channel balance reflected in the *latest_ctx*.

2.2 Watchtowers

To be able to go offline, Bob enlists the help of a watchtower that is always online, and can monitor the blockchain for cheating *ctxs*. Bob gives the watchtower the first 16 bytes of every *ctx's* transaction ID (*ctx_txid*), and encrypts the signed *jtx* to get an encrypted blob (*ejtx*) using the other 16 bytes of *ctx_txid* as the encryption key. Bob gives the pair [*ctx_txid_prefix*, *ejtx*] to the watchtower every time a new channel update is agreed upon.

The watchtower stores a map, where keys are *ctx_txid_prefixes* and values are *ejtxs*. It then watches the Bitcoin blockchain for any transaction whose *txid_prefix* matches any of the keys in its own map. If the watchtower finds a match, it extracts the *txid_suffix* from the blockchain *txid*, and uses this *txid_suffix* to decrypt the corresponding *ejtx* from its map to get the raw *jtx*, which is already signed by the corresponding Bob of that channel. The watchtower then broadcasts this *jtx* on the network to penalize the corresponding Alice of that channel. In implementations such as Lightning Network's *lnd* [4], the watchtower is not made to store the entire signed *jtx*, but an encoded struct that has addresses, signatures, and other metadata to be able to construct the *jtx*. This encoded struct is smaller than the corresponding raw bitcoin transaction, but is still around 300-350 bytes. The *ctx_txid_prefix* is constant at 16 bytes.

A single watchtower could be watching multiple channels and yet be oblivious to it. The watchtower just sees a stream of [*txid_prefix*, *ejtx*] pairs that it has to store, possibly forever. Such a watchtower cannot identify channels from such a stream as there is no channel identifier in each pair. This design preserves channel privacy in the sense that a watchtower cannot identify how many channel updates any particular channel has had. As a side note, if a channel has been closed, channel participants have no standardized way of informing the watchtower that a set of [*txid_prefix*, *ejtx*] pairs can be deleted from the watchtower's global map. One possible way is for the watchtower to allocate a limit on storage per user, and use a FIFO order to delete older items from its storage.

3 OUTPOST

In this paper, we propose Outpost, a watchtower construction where it is possible to store the *ejtx* inside the *ctx* that is exchanged between Alice and Bob as a part of the channel update. In other words, we can store the "future" justice transaction in the "present" commitment transaction. If this is possible, all the watchtower has to store is a map where keys are prefixes of *ctx_txids* and values are decryption keys for the corresponding *ejtx*. The *ejtx* itself does not need to be stored by the watchtower as it is now available in the cheating *ctx* that appears on the blockchain. With the Outpost construction, when a watchtower sees a cheating *ctx* on the blockchain, the following happens:

(1) The watchtower looks up the transaction in its global map, and finds the corresponding decryption key.
(2) The watchtower extracts the *ejtx* from the *ctx* which was seen on the blockchain.
(3) The watchtower uses the decryption key found in its map and decrypts *ejtx* to get the pre-signed *jtx*.
(4) The watchtower broadcasts this pre-signed *jtx*.

3.1 Why is this not possible in classic Lightning?

For Alice and Bob to have a signed *jtx* for a corresponding *ctx*, they need to first build the *jtx* with its inputs and outputs. The outputs are straightforward. For the *ctx* broadcast-able by Alice, the corresponding *jtx* will send all of Alice's timelocked balance to Bob without any timelocks. The inputs are not so straightforward. To refer to *ctx's* outputs as the inputs for *jtx*, we need to have *ctx_txid*. Let us say we have *ctx_txid*, and use it construct a *jtx*, and get Alice and Bob to sign it. Alice then uses some encryption key and encrypts *jtx* to get *ejtx*. We "encode" *ejtx* in *ctx* by using the OP_RETURN technique and make it the 3rd output of *ctx*. Now, we have a self-loop problem, given that the *ctx_txid* is constructed by double sha256 hashing the entire transaction, with its inputs and outputs. The moment we add a 3rd output, the *ctx_txid* present in *ejtx* is not the real *ctx_txid*. Given the way Bitcoin *txids* are constructed, there is no obvious way to encode a *jtx* that spends *ctx*, and still encode this *jtx* in the same *ctx*.

3.2 Two other constructions that do not work

3.2.1 Data Drop Method. One well known way of encoding arbitrary data in a Bitcoin transaction is to use the so-called Data-Drop method using P2SH transactions as elaborated in Sward et al [14]. To encode *jtx* inside *ctx*, we split *ctx* into two: ctx_1 and ctx_2. ctx_1'*s* output can be locked with scriptPubKey:

```
OP_HASH160 <hash(redeem_script)> OP_EQUAL
```

This will allow us to have a followup ctx_2 whose scriptSig has a redeem script of the type:

```
OP_DROP 2 <alice_pubkey> <bob_pubkey>
2 OP_CHECKMULTSIG
```

And scriptSig of the type:

```
0 <alice_sig> <bob_sig> <ejtx> <redeem_script>
```

With this setup, we can include arbitrary data in the scriptSig between the signatures and the actual redeem script, and encode *jtx* in this data (shown as *ejtx* above). The problem with this approach is that anyone can tamper with the scriptSig in such a way that this arbitrary data is changed, and the redeem script is still valid (scriptSig malleability). There is no guarantee that ctx_2 will make it to the blockchain in such a way that *ejtx* can be read off of it. Any intercepting forwarding full node, or even the miner who mines the relevant block can change the transaction to drop this extra data.

3.2.2 Data Hash Method. Another way of encoding arbitrary data in a Bitcoin transaction that is immune to scriptSig malleability is using the so-called Data-Hash method using P2SH transactions; also elaborated in Sward et al[14]. Here, like with the data-drop method, *ctx* is split into ctx_1 and ctx_2, and have $ctx_1's$ output locked in the same way as before. We prevent the subsequent $ctx_2's$ scriptSig from being tampered with, by encoding *ejtx* in the redeem script, and then using this redeem script's hash in ctx_1. Say, $ctx_2's$ redeem script looks like this:

```
OP_HASH160 <hash(ejtx)> OP_EQUALVERIFY
2 <alice_pubkey> <bob_pubkey> 2 OP_CHECKMULTSIG
```

$ctx_2's$ scriptSig will encode *ejtx* in the same way as before. This will enforce that the scriptSig cannot be tampered with while still keeping ctx_2 valid. If any tampering of $ctx_2's$ scriptSig (which contains *ejtx*) happens en route to a mined block, ctx_2 is not valid anymore as it cannot spend what ctx_1 locks. The data hash method solves the scriptSig malleability issue.

There is a subtler self-loop problem though: we include *ejtx* in the redeem script of ctx_2, and the redeem script's hash in ctx_1. This changes the *txid* of ctx_1 and we have to spend ctx_1 (through ctx_2) in *ejtx*. We are back to the self-loop problem again, where we have to spend a UTXO in the future, but encode this spending transaction in the present. The moment we do the encoding, the future UTXO's input *txid* changes, thereby invalidating the encoding. With a linear chain of transactions with the child spending the parents' output, we run into this self-loop problem. In the following section, we show how to encode the future in the present by doing it in a separate transaction path, and then merging these paths later.

3.3 Split Commitment Transaction Construction

In Outpost, we have 3 commitment transactions that represent a single channel state, as opposed to just 1 commitment transaction in classic Lightning. This is in addition to the opening transaction and the justice transaction. In this section, *topen* is common to both parties, Alice and Bob. Without loss of generality, the other 3 commitment transactions and 1 justice transaction are assumed to be Alice's to broadcast. Similar to classic Lightning, symmetrically opposite transactions are held by Bob, which Bob can broadcast in the same way as Alice. Another convention in the following listings is that *pubkey_i* can be signed by *sig_i*. In Section 4, we define these transactions more precisely with respect to signing, holding (not *hodling*), and broadcasting.

3.3.1 Opening Transaction. topen is exactly the same as in classic Lightning, in the sense that it has to spend two UTXOs, one of which is owned by Alice and one by Bob.

Listing 1: Opening Transaction in Bitcoin Script-like pseudocode

```
TOPEN: {
 txid: TOPEN_TXID
 vin: [{
   txid: source TXN that pays Alice
   scriptSig: <Alice sig_0>
  },{
   txid: source TXN that pays Bob
   scriptSig: <Bob sig_0>
 }]
 vout: [{
   value: <value of the channel>
   scriptPubKey:
        2
        <Alice pubkey_1>
        <Bob pubkey_1>
        2 OP_CHECKMULTISIG
 }]}
```

3.3.2 Commitment Transaction 1. The output of *topen* is spendable by ctx_1 (see Listing 2), which is similar to classic Lightning's commitment transaction, but differs in a few important ways.

(1) Output at index 0: Alice's balance is *not* spendable by just Alice after a timelock (as in classic Lightning). It is spendable with a multisig that both Alice and Bob need to sign. This allows us to "fork" this output into either the justice transaction or a followup commitment transaction (ctx_2). This ctx_2 gives Alice's share to Alice, but guards it with a timelock. This way, we realize classic Lightning's key idea that the broadcaster's balance needs to be timelocked to allow the counterparty to react in time.

(2) Output at index 1: As in classic Lightning, Bob's part is immediately spendable (no timelock) by Bob with just his signature.
(3) Output at index 2: The *auxiliary* output, which can be spent with a signature by both Alice and Bob, but has a value that is insignificant - say, just enough to be spendable with minimal fees. The sole purpose of the auxiliary output is to be a part of a subsequent auxiliary transaction (*aux_ctx*) which encodes *ejtx*. Its need will become more clear in the subsequent paragraphs.

Listing 2: Commitment Transaction 1 in Bitcoin Script-like pseudocode

```
CTX_1: {
 txid: CTX_1_TXID
 vin: [{
   txid: TOPEN_TXID
   index: 0
   scriptSig:
        0 <Alice sig_1> <Bob sig_1>
  }]
 vout: [{
   value: <Alice balance>
   scriptPubKey:
            2
            <Alice pubkey_2>
            <Bob pubkey_2>
            2 OP_CHECKMULTISIG
  }, {
   value: <Bob balance>
   scriptPubKey:
        <Bob pubkey_3> OP_CHECKSIG
  }, {
   value: INSIGNIFICANT_VALUE (ϵ)
   scriptPubKey:
        2
        <Alice pubkey_4>
        <Bob pubkey_4>
        2 OP_CHECKMULTISIG
  }]}
```

3.3.3 Justice Transaction. The *jtx* (see Listing 3) spends the multisig output of ctx_1 and gives all of Alice's share to Bob. Note that Alice will sign this transaction only after Bob signs it, and will not hand it over to Bob in raw format. Alice encrypts this *jtx* with a key of her choice to derive *ejtx* and hands over *ejtx* to Bob by encoding it in *aux_ctx*.

Listing 3: Justice Transaction in Bitcoin Script-like pseudocode

```
JTX: {
 txid: JTX_TXID
 vin: [{
   txid: CTX_1_TXID
   index : 0
   scriptSig:
        0 <Alice sig_2> <Bob sig_2>
  }],
 vout: [{
   value: <Alice balance>
   scriptPubKey:
        <Bob pubkey_5> OP_CHECKSIG
  }]}
```

3.3.4 Auxiliary Commitment Transaction. The purpose of *aux_ctx* (see Listing 4) is to be a vehicle to encode the encrypted justice transaction (*ejtx*) as its "non-monetary" OP_RETURN output. We inject *aux_ctx* in the channel by making it spend the small insignificant value from ctx_1, and make the final transaction ctx_2 spend from $aux\_ctx's$ "monetary" output. This way, the channel cannot be closed unilaterally without broadcasting *aux_ctx*. Once it is broadcast, *ejtx* is visible on the blockchain and anyone with a key to decrypt it can get the signed raw *jtx* and can broadcast it.

Listing 4: Auxiliary CTX in Bitcoin Script-like pseudocode

```
AUX_CTX: {
 txid: AUX_CTX_TXID
 vin: [{
   txid: CTX_1_TXID
   index: 2
   scriptSig:
        0 <Alice sig_4> <Bob sig_4>
  }]
 vout: [{
   value: INSIGNIFICANT_VALUE (ϵ)
   scriptPubKey:
        2
        <Alice pubkey_6> <Bob pubkey_6>
        2
        OP_CHECKMULTISIG
  }, {
   value: 0
   scriptPubKey: OP_RETURN EJTX
  }]}
```

3.3.5 Commitment Transaction 2. The final piece of the puzzle is ctx_2 (see Listing 5). It spends outputs of both ctx_1 (the actual channel balance carrying commitment) and *aux_ctx* (the *ejtx* carrying commitment). These outputs make up the inputs of ctx_2, and are timelocked using BIP68 [7] sequence numbers. In Listing 5, we use a delay of 144

blocks (1 day), which is represented as 0x00000090. The consensus rules of Bitcoin do not let this transaction through till 144 blocks have passed since both ctx_1 and *aux_ctx* are confirmed. This gives the watchtower enough time to look for a cheating *aux_ctx* on the blockchain, and decrypt *ejtx* which is visible *aux_ctx*. As a part of the protocol (refer 4), Alice would have shared with Bob the decryption key for *ejtx* in a followup state. Subsequently, Bob would have given this key, along with ctx_1*_txid* to the watchtower. This ensures that the watchtower has everything it needs to construct *jtx* and broadcast it without Bob ever having to be online.

Note that if *jtx* is confirmed on the Bitcoin blockchain, ctx_2 becomes invalid as one of its inputs (ctx_1'*s* output number #0) has now been consumed.

Listing 5: Commitment Transaction 2 in Bitcoin Script-like pseudocode

```
CTX_2: {
 txid: CTX_2_TXID
 vin: [{
   txid: CTX_1_TXID
   index: 0
   scriptSig:
        0 <Alice sig_2> <Bob sig_2>
        OP_TRUE
   sequence: 0x00000090
  },{
   txid: AUX_CTX_TXID
   index: 0
   scriptSig:
        0 <Alice sig_6> <Bob sig_6>
   sequence: 0x00000090
  }]
 vout: [{
   value: <Alice balance>
   scriptPubKey:
        <Alice pubkey_7> OP_CHECKSIG
  }]}
```

3.3.6 Cooperative Closure. If Alice and Bob are done using their channel, and want to get their current balances out, they create a classic Lightning like closure transaction that spends *topen* and allocates balances to Alice and Bob based on the latest state of the channel. With respect to opening and closing a channel, Outpost's transactions on the blockchain will look exactly the same as in classic Lightning. Listing 6 shows a cooperative closure.

Listing 6: Cooperative closure in Bitcoin Script-like pseudocode

```
CLOSURE: {
 txid: CLOSURE_TXID
 vin: [{
   txid: TOPEN_TXID
   index: 0
   scriptSig:
        0 <Alice sig_1> <Bob sig_1>
  }]
 vout: [{
   value: <Alice balance>
   scriptPubKey:
        <Alice pubkey_8> OP_CHECKSIG
  }, {
   value: <Bob balance>
   scriptPubKey:
        <Bob pubkey_8> OP_CHECKSIG
  }]}
```

3.3.7 Cheating. If Alice cheats by broadcasting an earlier state in the form of ctx_1, *aux_ctx*, and ctx_2 to the network, ctx_1 and *aux_ctx* can be mined and confirmed immediately, but ctx_2 is timelocked. In the timelocked time, the watchtower can watch for *aux_ctx* on the Bitcoin blockchain, extract and decrypt *ejtx* from it to get *jtx* and broadcast *jtx*, thereby invalidating ctx_2, and also giving Bob the entire channel balance.

3.3.8 Unilateral Closure. If Bob goes offline, and Alice wants to unilaterally close the channel (not cheating), she broadcasts the current state in the form of ctx_1, *aux_ctx*, and ctx_2. She knows that the decryption key for *ejtx* from the current state has not been shared with Bob, and hence, not with any watchtower either. This makes Alice get back her side of the balance, but after a timelock. This tradeoff of the unilateral channel closure having to wait for timelocks to expire is the key design insight of classic Lightning, and we preserve the same principle, but through BIP68 input locks in ctx_2.

3.3.9 Griefing. Griefing refers to a class of attacks where the attacker does not want to make a profit, but puts the counterparty at a disadvantage by deviating from the protocol. In our protocol, the watchtower can only help if both ctx_1 and *aux_ctx* are on the blockchain. Alice can initiate a griefing attack when Bob is offline by broadcasting a ctx_1 from an older state and *not* broadcasting the corresponding *aux_ctx* and ctx_2. Note that Alice is not cashing out here, but the watchtower cannot help Bob either - because *aux_ctx* is not published on the blockchain.

This attack is possible because *topen* has been spent by ctx_1, and has created a UTXO that cannot be spent by Bob alone. We can mitigate this by adding a flow to ctx_1 such that the UTXO it creates can be spent by Bob after a timelock unless it is spent by *aux_ctx*. This incentivizes Alice to not publish just ctx_1. If she does that, and does not publish the

corresponding *aux_ctx* and ctx_2, the entire channel balance can be swept by Bob after this timelock expires. Note that this timelock has to be larger than the timelock in ctx_2.

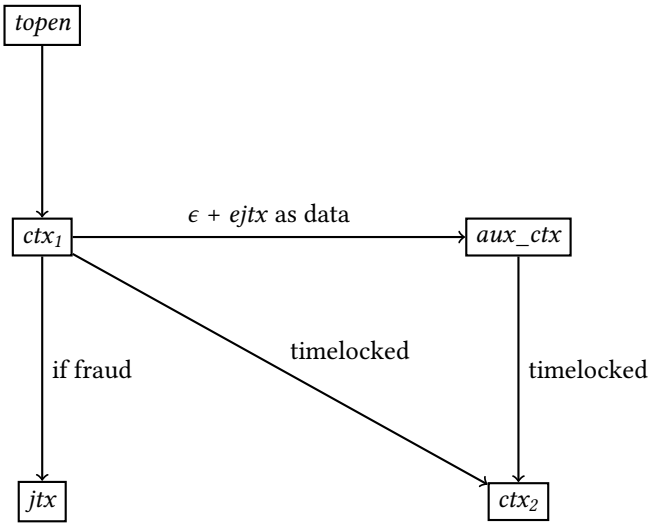

Figure 1: Transaction Flow

4 PROTOCOL

In this section, we elaborate on how, at each stage of the protocol, transactions are signed and exchanged by Alice and Bob. Decryption keys of encrypted blobs are also exchanged as a part of the protocol.

During channel opening, as with classic Lightning, Alice and Bob construct a single *topen* using their own inputs, and make it spendable by a multisig that they both need to sign. Note that between both Alice and Bob, there is only one *topen*. Alice and Bob do not sign this *topen* before the follow-up transactions of the next step (ctx_1, *aux_ctx*, and ctx_2) have been exchanged. This ensures that if either party disappears after the signed *topen* has been broadcast, the other party is not held in limbo and can close the channel unilaterally. In the following protocol description, we use "alice" and "bob" superscripts to denote the transactions that are held by Alice and Bob. In Lightning channels, both parties keep symmetric transactions to represent the collective state.

4.1 Opening Transaction

4.1.1 Bob → Alice.

- A UTXO that Bob controls.
- Pubkeys that are required for followup transactions.

4.1.2 Alice → Bob.

- *topen* with Alice's own UTXO filled in. This *topen* has its multisig output filled in with one of Bob's pubkeys and one of Alice's own pubkeys. This *topen* is not signed by either Alice or Bob yet.
- Two versions of ctx_1 (ctx_1^{alice}, ctx_1^{bob}) which share the same input transaction *topen*. These two ctx_1*s* have three outputs each. In ctx_1^{alice}, the 2nd singlesig output is sent to Bob's pubkey. In ctx_1^{bob}, the 2nd singlesig output is sent to Alice's pubkey. Note that Alice can construct both ctx_1*s* at this stage, and can even sign for her part of the input (*topen*) of each ctx_1.

4.1.3 Bob → Alice.

- ctx_1^{alice} signed by Bob. Bob signs ctx_1^{bob} and keeps it for himself.
- jtx^{bob} with Alice's balance sent to Bob's pubkey. Bob signs jtx^{bob} without worry because its output is being sent to him.

4.1.4 Alice → Bob.

- jtx^{alice} with Bob's balance sent to Alice's pubkey. Alice signs jtx^{alice} without worry because its output is being sent to her.
- aux_ctx^{bob}: Alice constructs the full signed jtx^{bob} with her own signature, and encrypts it with a random key to derive $ejtx^{bob}$. She then constructs aux_ctx^{bob} with two outputs, one of which is the OP_RETURN prefixed $ejtx^{bob}$. Alice signs aux_ctx^{bob} and sends it to Bob. After getting aux_ctx^{bob} signed by Alice, Bob signs it as well, but keeps it for himself.
- ctx_2^{bob}: Alice also constructs the signed ctx_2^{bob}, which needs her signatures for both its inputs: ctx_1^{bob} and aux_ctx^{bob}. When Bob gets ctx_2^{bob} signed by Alice, he signs it and keeps it for himself.

4.1.5 Bob → Alice.

- aux_ctx^{alice}: Bob constructs the fully signed jtx^{alice} with his own signature, and encrypts it with a random key to derive $ejtx^{alice}$. He then constructs aux_ctx^{alice} with two outputs, one of which is the OP_RETURN prefixed $ejtx^{alice}$. Bob signs aux_ctx^{alice} and sends it to Alice. After getting aux_ctx^{alice} signed by Bob, Alice signs it as well, but keeps it for herself.
- ctx_2^{alice}: Bob also constructs the signed ctx_2^{alice}, which needs his signatures for both its inputs: ctx_1^{alice} and aux_ctx^{alice}. When Alice gets ctx_2^{alice}, she signs it and keeps it for herself.
- *topen*: At this point, Bob has all the followup transactions signed by Alice with him, and can safely sign *topen*.

4.1.6 Alice → Blockchain.

- *topen*: At this point, Alice has all the followup transactions signed by Bob with her, and can safely sign *topen* and broadcast it on the Bitcoin network.

4.2 State Update

4.2.1 Bob → Alice.

- Same as from the Opening Transaction, except for the UTXO part.

4.2.2 Alice → Bob.

- Same as from Opening Transaction, except for the *topen* part.

4.2.3 Bob → Alice.

- Same as from Opening Transaction.

4.2.4 Alice → Bob.

- Same as from Opening Transaction.

4.2.5 Bob → Alice.

- Same as from Opening Transaction.
- Decryption Key: At this point, Bob has all the followup transactions signed by Alice with him and has effectively moved to the next state. He can now let Alice decrypt the previous state's $ejtx^{bob}$ if it is ever seen on the blockchain (through the confirmation of aux_ctx^{bob}). To do that, Bob sends Alice the key that can decrypt $ejtx^{bob}$ from the previous state. Now, Alice can send aux_ctx^{alice}'s *txid* and this decryption key to the watchtower.

4.2.6 Alice → Bob.

- Decryption Key: At this point, Alice has all the followup transactions signed by Bob with her and has effectively moved to the next state. She can now let Bob decrypt the previous state's $ejtx^{alice}$ if it is ever seen on the blockchain (through the confirmation of aux_ctx^{alice}). To do that, Alice sends Bob the key that can decrypt $ejtx^{alice}$ from the previous state. Now, Bob can send aux_ctx^{bob}'s *txid* and this decryption key to a possibly different watchtower.

5 LIMITATIONS

5.1 OP_RETURN size limit

One key limitation of the Outpost construction is the size constraint on the OP_RETURN output in *aux_ctx*. This size limitation is enforced by the IsStandard function of Bitcoin Core's reference implementation, which drops any transaction that has an OP_RETURN output of more than 80 bytes. This rule is not enforced by the Bitcoin consensus mechanism, in the sense that transactions with such outputs are considered valid, but not standard. Miners who see these transactions can still add them to their block template and generate valid blocks with them. So, *aux_ctx* can have the OP_RETURN output we want and can be handed to the miners directly to be included in their block template without violating Bitcoin's consensus rules.

Another way to circumvent this size limit is to use the data-hash method from [14] to encode arbitrary data in a standard Bitcoin transaction. In our case, we have to split *aux_ctx* into two transactions, say aux_ctx_1 and aux_ctx_2. In aux_ctx_1, there will be a hash of a specific redeem script (thereby making aux_ctx_1 a P2SH transaction). The actual redeem script will be in aux_ctx_2 and will enable the scriptSig of aux_ctx_2 to have the encrypted payload. The payload in our case is a typical *jtx* that spends using a multisig and pays to a P2PKH address. With signatures, these transactions are typically ~350 bytes long. They can be encrypted with AES-128 and we get an *ejtx* of size ~360 bytes. This can be encoded in the scriptSig of aux_ctx_2 quite easily as the maximum script element size in Bitcoin is 520 bytes.

5.2 Transaction Bloat and Complexity

In the Outpost construction, instead of one *ctx* per party to handle the channel update, like in classic Lightning, we have 3 transactions per party per state. This is not a true limitation, in that we are not increasing storage cumulatively. Each party needs to just keep their latest state in storage, and can discard all previous states. So, storing one transaction in classic Lightning vs three transactions with Outpost should not matter a lot. In classic Lightning, each party has to store the *ctx_txid* of each *ctx* that its counter-party can broadcast, to watch for cheating transactions. Along side the *ctx_txid*, the party has to also store the revocation key needed to construct the *jtx* for a cheating *ctx*. In Outpost, each party has to store the ctx_1_txid of each ctx_1 that its counter-party has to broadcast to cheat. Along side the ctx_1_txid, the party has to also store the decryption key for the encoded *ejtx* inside the *aux_ctx*. At the node level, this extra storage requirement is the same in Outpost as in classic Lightning. But at the watchtower level, it leads to considerable savings, which we will explore in the Analysis section.

6 OPTIMIZATION

Each party can derive their *jtx* encryption keys independently of each other, forcing the counter-party to store these decryption keys independently. We can optimize some of this storage away by deriving encryption keys using a hash-chain or an encrypted-key-chain. Say, Alice wants to generate 1000 encryption keys such that they can be used in a payment channel with Bob - with one key being used for each state update. As state updates happen, Alice

will give Bob these keys one by one, and Bob has to store all of them, along with the ctx_1_txid for each key. This can be made more efficient if Bob can just store the most recent key he received from Alice, but can compute the other keys based on this latest key.

There are multiple schemes that Alice could use to generate encryption keys such that if K_i and K_{i+1} are two keys with timestamps i and $i+1$, then:

- It is easy for Alice to generate either key from the other.
- It is hard for Bob to generate K_{i+1} from K_i, but easy to generate K_i from K_{i+1}.

We briefly outline 2 such schemes:

- Alice pre-generates these keys by starting with one random key, and generating subsequent keys by hashing the previous key, say using SHA256 - thereby forming a hash-chain. She starts her channel with Bob by using the last such generated key, and at each followup state, uses the hash preimage (which is also a hash of its own preimage) as the next key. Bob can now discard old keys as new keys come along, as he can always reconstruct them using the commonly known one-way hash function if he knows the current key and the index number of what key he wants to reconstruct. This scheme needs Alice to pre-compute hashes and store them on the "forward chain", thereby incurring both computation and storage costs. Going on the "backwards chain" is a matter of trivial lookup. A more advanced version of this scheme is found in [13].
- Alice creates an RSA key pair of sufficient length (say, modulus of size 2048 bits), keeps the private key to herself, and shares the public key with Bob. Say, e and n are the exponent and the modulus components of the public key, Alice can start the key chain with a large random number in the range $[2, n-1)$ and decrypt it using the private key to generate the next key in the sequence. Note that every number in the range $[2, n-1)$ has a valid RSA decryption. A secure one-way hash function can be used as a key derivation function on this large number to generate the (smaller) symmetric key required to encrypt *jtx* to get *ejtx*. Bob can always go back the chain and find older keys by encrypting the latest key using the public key that he knows, but cannot create newer keys, as it requires decrypting the latest key.

We can even optimize away the need to store ctx_1_txid for each state update. We can embed a channel ID in either ctx_1 or *aux_ctx* which we can then watch for on the blockchain. We also need to track the index of the state to be able to derive the right decryption key to construct the necessary *jtx*. The channel ID and the index together can be stored as the 4^{th} output of ctx_1 in an OP_RETURN instruction. This gives us constant storage per channel with respect to what we have to watch for on the blockhain. All we need to store per channel is the channel ID and seed of the hash-chain.

Using either of the schemes above, Bob's storage savings can also be realized at the watchtower level, if Bob is willing to let the watchtower know that all of his state updates are from the same channel by providing a channel ID in each of his watchtower requests. The watchtower then watches the blockchain for this ID, and can reconstruct all it needs from the transactions that appear on the blockchain that contain this ID. In case of a cooperative closure of a channel, Bob can get the watchtower to free up storage allocated to this channel ID.

7 ANALYSIS

We study watchtower storage costs for Outpost vs. Classic Lightning under two conditions.

- Known channel: Watchtower has access to a channel identifier in its state update stream.
- Unknown channel: Watchtower is oblivious to channel identities.

Classic	
Known Channel	$N \cdot$ size(ejtx) + $1 \cdot$ size(txid)
Unknown Channel	$N \cdot$ (size(txid) + size(ejtx))
Outpost	
Known Channel	$1 \cdot$ (size(txid) + size(key))
Unknown Channel	$N \cdot$ (size(txid) + size(key))

In Classic Lightning, if the watchtower knows which state updates belong to which channel, the watchtower still has to store the encrypted blobs corresponding to justice transactions (as these blobs have the victim's signature). Storage is proportional to how many updates the channel has seen (denoted by N) times the size of *ejtx*. As per LND's implementation of watchtowers [5], *ejtx* need not contain the full transaction, but just the relevant addresses, signatures, and other metadata. Our estimate is that *ejtx* will be around 300-350 bytes. Note that if the watchtower does not know the identity of the channels, the cost is the same because it still has to store all the *ctx_txids*.

In Outpost, if the watchtower does not know the channel identifiers per state update, it has to store the full decryption key per ctx_1_txid. This puts the storage cost at N times the size of the decryption key, which can be as low as 16 bytes for a symmetric encryption scheme like AES-128. If the watchtower does knows channel identifiers per state update, we use the hash-chain trick to reduce the storage requirements to the constant size of the channel ID and size of the hash-chain seed. We achieve this constant storage by offloading all the storage to the blockchain itself. Note that

we are not bloating the blockchain here. These transactions appear in the blockchain only when one of the parties attempts to cheat or grief their counterparty. We believe that given the incentives of Lightning (and thus, Outpost), this is not common. In the preferred case, the commitment, auxiliary commitment, or justice transactions do not appear on the blockchain, and we only see the cooperative closure transaction.

With Outpost, across billions of state updates per channel, we have the option of constant storage per channel. Or if we want stricter privacy with respect to the watchtower, we get storage savings of 16 bytes vs 350 bytes per state update.

8 RESPONSIVE WATCHTOWER DESIGN

We posit that the best way to compensate a watchtower is to pay-per-update. In this scenario, every time a channel sends an update to the watchtower it pays a small fee to the watchtower. In this case, it is reasonable for a channel to expect the following from the watchtower.

- The watchtower has access to the state updates that have been sent to it so far.
- The watchtower was online when the latest Bitcoin block was seen.

We present a scheme here with which a watchtower can prove these two conditions to the channels it serves.

If we are using the non-hash-chain version of Outpost, the watchtower needs to prove to every channel that it has access to the set of all pairs $[ctx_1\_txid, key]$ that a channel has sent it. Let us call $[ctx_1\_txid, key]$ as the *datarow* that the watchtower has to store. The watchtower can build a Merkle tree out of this set of *datarows*, and commit the Merkle root to the next Bitcoin block in an OP_RETURN data-transaction. The channel can now ask the watchtower for proof of a specific *datarow* in the set it has sent it, and the watchtower has to respond with a Merkle proof for the specific *datarow* that also comforms to the Merkle root that was committed in the Bitcoin block. To ensure that the same proof that worked for Block B_i does not work for Block B_{i+1} we can append the blockhash of Block B_i to each *datarow* so that the proof is unique per *datarow* per block.

This proof scheme can be generalized to many channels by building a Merkle tree out of the Merkle root of each channel's Merkle tree, and committing this "global" Merkle root to the Bitcoin blockchain. Each channel can then get a proof about any specific *datarow* that is has sent the watchtower. This should make the idea of paying the watchtower per update more palatable to a channel.

9 CONCLUSION

Watchtowers typically monitor tens of thousands of channels, and can potentially handle billions of updates per channel. Getting an order of magnitude storage savings will go a long way in making it attractive for developers to implement and host watchtower services for channels to use. We believe that the additional option of having constant storage per channel makes Outpost even more appealing. Additionally, we have shown a novel way to encode a "future" transaction that spends a "present" transaction's outputs in the same "present" transaction using parallel transaction flows, which might have other novel applications in the Bitcoin ecosystem.

ACKNOWLEDGMENTS

The authors would like to thank Patrick McCorry for comments on an earlier draft of the paper and StackExchange user *fgrieu* for their insights on RSA.

REFERENCES

[1] Bolt Authors. [n. d.]. Lightning Network Specifications. https://github.com/lightningnetwork/lightning-rfc. ([n. d.]). [Accessed: 2019-05-24].
[2] C-Lightning authors. [n. d.]. c-lightning - a Lightning Network implementation in C. https://github.com/ElementsProject/lightning. ([n. d.]). [Accessed: 2019-05-24].
[3] Eclair authors. [n. d.]. A scala implementation of the Lightning Network. https://github.com/ACINQ/eclair. ([n. d.]). [Accessed: 2019-05-24].
[4] LND authors. [n. d.]. LND: The Lightning Network Daemon. https://github.com/lightningnetwork/lnd. ([n. d.]). [Accessed: 2019-05-24].
[5] LND Authors. [n. d.]. LND: The Lightning Network Daemon, Watchtowers. https://github.com/lightningnetwork/lnd/blob/master/watchtower/blob/justice_kit.go. ([n. d.]). [Accessed: 2019-05-24].
[6] Christian Decker and Roger Wattenhofer. 2015. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*. Springer, 3–18.
[7] Mark Friedenbach, BtcDrak, Nicholoas Dorier, and kinoshitajona. [n. d.]. BIP68: Relative lock-time using consensus-enforced sequence numbers. ([n. d.]). https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki [Accessed: 2019-05-24].
[8] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. 2016. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM.
[9] MIT Digital Currency Initiative LIT authors. [n. d.]. Lightning Network node software. https://github.com/mit-dci/lit. ([n. d.]). [Accessed: 2019-05-24].
[10] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).
[11] Olaoluwa Osuntokun. 2018. Hardening Lightning, Stanford Cyber Initiative. (2018).
[12] Joseph Poon and Thaddeus Dryja. 2016. The bitcoin lightning network: Scalable off-chain instant payments. (2016).
[13] Rusty Russel. 2016. Efficient Chains Of Unpredictable Numbers. https://github.com/rustyrussell/ccan/blob/master/ccan/crypto/shachain/design.txt. (2016).
[14] Andrew Sward, Ivy Vecna, and Forrest Stonedahl. 2018. Data Insertion in Bitcoin's Blockchain. *Ledger* 3 (2018).