Battlement: A Quorum Based Design for Lightning Network Key Management

Omer Shlomovits

ZenGo X

Abstract

The lightning network is a payment channel network on top of bitcoin. It defines a set of protocols and specifications for securely establishing channels between lightning nodes and transmit payments between them without touching the Bitcoin blockchain. From a cryptographic stand point, a channel can be seen as a collection of two-party protocols, drawing their security from the orchestration of private keys. In this work we therefore treat channels as a key management problem. We model the adversary as an outsider attacker that gains either full or partial access to the different types of keys during the different stages of a channel life cycle.

In the context of the lightning network, a watchtower is a semi-trusted, always online third party. The watchtower gets a limited permission from a party to act on its behalf in the scenario where the counter party is trying to cheat while the party is offline. We build our key management solution based on the watchtower concept: we require a quorum of watchtowers, called a *Battlement*, to which a user delegates key management responsibilities. Under a threshold assumption, we show how our design eliminates the existing attack surface for attackers. We additionally show how a Battlement can mitigate a bribing attack existing in current lightning network design.

1 Introduction

Battlement: `A battlement in defensive architecture, such as that of city walls or castles, comprises a parapet, in which gaps or indentations, which are often rectangular, occur at intervals to allow for the launch of arrows or other projectiles from within the defences`[1]

The lightning network [13] is an instantiation of a payment channel network [9] compatible with the bitcoin blockchain. Lightning network provides high speed, low fee bitcoin transactions, enabling scalability to the bitcoin blockchain. The basic element in the network is the channel. Using a graph analogy, lightning nodes are the vertices and channels connecting them are the edges. Any two parties can open a channel between their nodes and join the network. Once a channel is open, it can be updated without interacting with the blockchain. A channel update can be a result of a direct payment from one party to its counter party in the channel, or it can be a part of a multihop payment that involves other nodes. With each new update the parties must invalidate the previous state of the channel. If a party decides to close the channel one-sided using an outdated state, the counter party is capable of

[1] `https://en.wikipedia.org/wiki/Battlement`

"punishing" that misbehaviour, and withdraw *all* the funds from the channel. However, this requires constant monitoring over the bitcoin blockchain which allows all participants of the lightning network only a minimal offline time and requires read/write access to the blockchain. To solve this issue, a third party, semi trusted entity, called a watchtower is used. The watchtower accepts from the user a "justice" transaction, or more precisely the means to compose such, after each channel update. The watchtower is guaranteed to be online and to operate a bitcoin node. In the case that a fraudulent transaction hits the blockchain, the watchtower will publish the justice transaction on behalf of the user. In this sense, the watchtower is "hired" by the user to enable the user to go offline.

1.1 Our Contributions

The different stages of a lightning channel are defined by a set of protocols and messages between two users, which we will often also call parties. In this work we choose to interpret channels by way of their cryptographic strength. We put forth a natural attack mode that was not discussed before in the context of the lightning network. Specifically, We focus on an attacker that is an outsider to the channel or even to the lightning network. To motivate our model, think of an attacker who manages to get elevated access to a machine running a lightning node. It might be write-only access which allows for certain types of attack (e.g. deleting keys) or read-only access which enables other types of attacks (key theft). This attacker does not have to be a participant in the lightning network, just as an attacker getting access to a bitcoin node does not have to own a wallet but can still be able to steal funds. We argue that the same is not true for a lightning node attacker, or that in fact it is much more complex and unclear what an attacker can hope to achieve in terms of profit. .

After we identify the potential attack vectors and how they are related to the different keys used during the life cycle of a channel we describe a new key management system, to mitigate the attacks. Our solution uses the already in-place watchtowers. We make the assumption that the watchtowers are connected to one another in quorums of size n such that no more than $t < n/2$ watchtowers are corrupted. This is also known as a honest majority threshold assumption, which is a common setting in multiparty computation. We discuss ideas for protocols under this new design that mitigate the risks imposed by an outsider attacker. We finally present a simple bribing attack existing in the currently implemented lightning network, specifically, a party in a channel can bribe a watchtower to not publish a justice transaction in return for a share from the profit. We provide an intuition as to why this is not a trivial problem and how our proposed Battlement structure is able to provide a new toolbox to handle this kind of attack.

1.2 Related Work

While no previous work addressed directly the outsider attacker model, the watchtowers' inefficiencies and security concerns did receive some attention and different designs were suggested. Outpost [8] focuses on improving the storage requirements of existing watchtowers. It requires an additional alteration for the structure of the transactions in the lightning network. Our design do not improve on the storage parameters, on the contrary - we require more storage from the Battlement. On the other hand, we do not change the base lightning protocol messages like done in Outpost but only add additional messages between the user and the Battlement. Brick [1] is an alternative construction for payment channels, based on a quorum of watchtowers. Brick uses a consistent broadcast to guarantee liveness

and safety in an asynchronous network. Our Battlement is also making use of a quorum of watchtowers. We do not address the synchronicity requirement, however, all our protocols are non-interactive or can be made such. Most improtantly, Brick is not compatible with the bitcoin blockchain, let alone with the existing lightning network channels while Battlement can be implemented on top of the existing infrastructure and specs.

Another contribution of Brick is being incentive-compatible. Their incentive design includes both rewards and punishment that are binded to a specific channel, improving over Pisa [12], which uses a global incentive design. A follow up work, Cerberus Channels [2] took this idea and developed an incentive system for watchtowers that is compatible with the bitcoin blockchain, however not compatible with existing lightning network channels. In this work we do not make a claim to solve the incentives problem but merely point to some added value that our design may bring in addressing this problem via threshold cryptography with access to a public ledger. Our main focus remains solving key management.

Hubs: A *hub* in the lightning network is a node with many healthy connections. A user connected to the hub saves cost and time of creating new channels. A custodial hub has full control over its users funds. This facilitates even more efficiency because the user do not need large amount of liquidity to participate in the network. While we overlay the Battlement on the existing watchtowers infrastructure, the same principles we used will work for quorum of hubs (threshold hub) as well. Hubs are not yet part of the lightning specifications and therefore we postpone treating them for future work.

2 Background

We provide here a short summary of how channels operates in the lightning network. We refer the reader to the lightning-rfc for more details and comprehensive understanding [2].

We will use Alice and Bob for the two parties on the two sides of a channel. The first procedure is opening a channel. This is done using a 2 out of 2 multisig between Alice and Bob to lock the funds on the bitcoin blockchain. To establish a channel the parties replace signatures and some other auxiliary data, publish the funding transaction on chain and wait for enough confirmations. The channel opening already includes a commitment transaction so that Alice or Bob will be able to close the channel and not get their funds locked forever if the counter party becomes unresponsive. A commitment transaction is sent from Alice to Bob and another one from Bob to Alice each time they want to update the channel. The transaction is sent directly, and not published on chain. The commitment transaction from Bob to Alice is a bitcoin transaction structured as follows. The input is the funding transaction signed by Bob. There are two outputs:

1. The amount that is owned by Alice: can be spent by Alice after a timelock or by Bob given Alice revocation key

2. The amount that is owned by Bob: can be spent by Bob unconditionally.

The commitment transaction from Alice to Bob is a mirror of the commitment transaction from Bob to Alice. As we can observe, Bob has a way of withdrawing all the funds from the channel, including Alice‘s, if he knows the revocation key. As part of each update to a channel the parties reveal to one another their respective revocation keys of the previous commitment transaction. By doing so they

[2] `https://github.com/lightningnetwork/lightning-rfc`

guarantee that any one sided closure of the channel using a commitment transaction that is not from the latest state will result in the counter party getting all the funds in the channel. In more details: In the happy flow a channel is closed gracefully: the parties simply exchange shutdown messages and signatures to agree on the fee. However, closing can be done also one sided. It is a mechanism required for the case the counter party becomes inactive. A cheating Bob can decide to publish an old commitment transaction. To catch this fraud and punish, Alice must monitor the blockchain for such transactions, which means she must stay online. To overcome this requirement, Alice can hire a watchtower. The watchtower runs a bitcoin full node or holds a stable connection to one. It will receive an update from Alice with each new state change and if a fraudulent transaction will ever appear on chain the watchtower will respond on behalf of Alice. In the state of the art design, the watchtower is holding an encrypted "justice" transaction, which will become decryptable and publishable conditioned on the publication of the fraudalant transaction by the counter party.

Another type of transactions sent via a channel are HTLC (Hashed TimeLock Contract) related transactions. They are being used to facilitate multihop payments which involve nodes that are not connected directly via a channel but rather via a route of several channels. There are three roles in a multihop payment: the payer (Bob), the payee (Alice) and nodes that are neither (Charlie). At first, HTLC transactions are added to each channel in the multi-channel route, at a later stage when the payment is being made each channel will redeem its pending HTLC transaction. For all Charlie nodes in between, there should be an incoming HTLC transaction from a node closer to the payer and an outgoing HTLC transaction to a node closer to the payee. We still must maintain the rule of revoking old states therefore HTLC transactions will contain outputs with revocation keys as well. To make a successful payment, Alice generates a hash value of which she keeps the preimage secret. The hash value is shared with the payer Bob, who initiates an HTLC transaction to the next node in the route. The HTLC transaction contains an output script that can be spent only given the knowledge of the preimage. After all the channels in the route are updated accordingly with the same hash preimage condition, the last node is Alice, who generated the preimage. Alice will reveal the preimage to redeem the HTLC transaction, spending it to a standard commitment transaction with her counter party. Each party in the route will use the preimage information to redeem the HTLC transaction in its channel until finally the channel with Bob is updated, completing the multihop payment from Bob to Alice.

3 The Attacker Model

In this work we discuss the security of lightning network channels with respect to key management. We follow a real world attacker point of view: seizing opportunities to elevate access and permissions into a machine or device which happens to be connected to the lightning network. Using this access, the attacker will try to corrupt a channel, optimizing for profit.

Motivation: With increasing adoption of the lighting network by different types of entites, i.e. enterprise, retail and so on, a more sophisticated node management solutions are being explored. Key management takes a big part of any such system. Fine-grained corruptions may occur as it is often the case in modern cyber security. For example, in a machine running a lightning node different users have different permissions, attacker can takeover a user that has only one of the permissions: read access or write access change completly the attack surface. As another example, if keys are in ROM they cannot be digitally deleted, but can be leaked. In many other areas of cryptography fine-grained corruptions

are studied and there are solutions which still guarantee security when some of the keys are stolen. For instance, a key exchange protocols where security is required as long as either the long term key or the session key is not compromised.

The parties taking part in the lightning network are assumed to be honest and rational, i.e. following the protocol instructions, but the attacker can convince a subset of them to run a different protocol if that alternate protocol introduces higher profit, or keeps them from losing funds. We assume that all cryptographic material is generated from a uniform distribution and follows a use-once policy.

In this work we focus on key management and not on general data availability problem, therefore we assume that commitment transactions data are public and can be accessed by anyone at anytime.

Finally, we assume secure communication between participants of the lightning network.

3.1 Threshold security and other cryptographic assumptions

Throughout the paper we rely heavily on threshold security. It is assumed that in a given quorum with n participants, no attacker can corrupt more than t participants. The corrupted parties are assumed to be controlled by a single adversary and they have complete freedom to deviate from protocols they participate in (malicious adversary). We make the assumption that the majority of the quorum is honest ($t < n/2$).

For simplicity, We assume all participants in a quorum got access to a Ledger, although this is not a strict requirement and implementation dependent. We assume PKI, which can be instantiated using the ledger.

4 Battlement

This section is the main body of our work. It is divided into three parts. We start by providing exposition to lightning network channels from the perspective of the various keys used and the cryptographic operations they derive. We move on to discuss concrete attacks enabled under our attacker model §3. The attacks we describe unveil an attack surface: what happens if each of the keys gets deleted or stolen by an attacker throughout the life cycle of a channel. In the third part, we present our solution, *Battlement*, which demonstrates how a quorum of semi-trusted third parties (watchtowers), interacting with the user under attack, can run threshold cryptography based protocols to mitigate the proposed attack surface.

4.1 Overview of the Different Keys Involved in a Lightning Channel Life Cycle

In this part we detail the data structures that must be kept protected in the lifespan of a Lightning channel.

- **secret key**: The secret key is used to open and close a channel. We assume one secret key per channel per party. Each state update for the channel requires a signature using each of the secret keys. The secret key must therefore exist for the entire lifetime of the channel and should be kept secret throughout. In the lightning network, the secret key is a secp256k1 secret key, same as used in the Bitcoin blockchain. We justify our assumption for one key per channel per user as it is common practice to generate bitcoin keys in a tree-like deterministic structure from a fixed min-entropy such that each interaction with the chain is done with a different key.

- **Revocation key**: For a channel to justify its existence, micro-payments must switch sides - the more off-chain transactions in a given channel the better its utilization. The revocation key generation is coupled with the commitment transaction generation: Each round of updates to the channel, involves the production of a new pair of commitment transactions followed by an exchange of revocation keys to invalidate the last state of the channel. The lifetime of a revocation key is until the channel is closed. The lifespan is divided in two parts:
  - *Phase1*: the revocation key is kept secret by its owner
  - *Phase2*: once the revocation key is sent to the counter party it must be kept by both parties until the channel is closed.
- **Multihop key**: As it is often the case, when there is no direct channel between a payer and a payee, a payment must hop between channels. This operation introduces a new key on top of the previous two mentioned. We call this key multihop key. In the lightning network, multihop keys are implemented as part of HTLCs, using hash preimage. The key is generated by the payee and all the transactions in the route between the payer and the payee must be conditioned on this key for successful transfer. This condition is embedded in each multihop transaction, on top of the existing structures that use revocation and secret keys. For the multihop payment to be complete the multihop key must be revealed. This makes this key short term: we can divide here again the lifespan of the key to two phases, just like the revocation key: in the first phase the key is known only to the payee who generated it. In the second phase the same key must be kept by all parties on the route from payer to payee. We make the simplifying assumption that once a multihop key is shared in the first hop it becomes public. In other words, once a channel in the route removes the HTLC, we no longer consider the multihop key a secret. Therefore, the effective lifespan of multihop key is until the HTLC is removed, which may happen long before the channel is closed. We leave treatment of the more transient phases of multihop keys to future work [3].

4.2 New Attack Vectors

We describe attacks that apply to the general framework of lightning network key-management and were overlooked so far. Our goal is to point to the existence of behaviours that force deviation from the intended lightning-rfc specification . Our methodology is simple: We analyze for each one of the keys from the previous section what happens if an attacker is allowed to manipulate it during the different phases of the channel lifetime.

4.2.1 Secret key deletion

If an attacker deletes a secret key from memory, a channel can no longer get state updates as one of the parties is unable to sign the new commitment transaction. This will force the channel to close. One point to notice is that the control of *when* to close is now completely at the hands of the counter party as the party who lost her secret key can no longer sign to close the channel.

[3]We acknowledge that some of the most famous attacks on lightning network protocol and implementation exist when the multihop key is in motion. See preimage extraction: `https://lists.linuxfoundation.org/pipermail/lightning-dev/2020-October/002857.html` and Warmhole attack [10]. These attacks however comes from "within" the lightning network while we focus on an outsider attacker model

4.2.2 Secret key theft

We assume the attacker is a third party, i.e. the attacker learns Alice's secret key in her channel with Bob (if the attacker controls Bob, the attacker will have full control over the channel and the attack is trivial). In this case, the attacker will "sell" the key to the counter party. Using this key, the counter party can sign on a revoked transaction on behalf of the attacked party and collect all the funds in the channel, i.e. the attacker can sign on Alice behalf on a commitment transaction to which Bob knows the revocation key, as it was sent to him by Alice sometime in the past. Bob will be able to publish and immediately spend the two outputs of this transaction, scoring all the funds in the channel. Even in the case where the channel was just opened and no previous states with revocation keys exist, Bob can first mutually update the state in Alice's favour and then once he learns a revocation key, do a one side closing of the channel.

Using techniques like Juggling Swap [14] or other fair exchange of private keys protocols, the attacker and the counter party (Bob) can securely conduct the "selling" of Alice's key.

4.2.3 Revocation key deletion

The revocation key of the latest commitment transaction is known only to the generator of the commitment transaction. If the revocation key is deleted from memory before a new state is agreed upon and finalized (releasing the old revocation key to the counter party) then there is no way for the party to reconstruct the revocation key and send it to the counter party as part of updating the channel. This means the counter party will not be able to propagate to the new state and the channel is stuck. The only viable option is for the parties to close the channel. This results in a kind of DoS attack where an attacker that can corrupt the revocation key can force-close a user channel over and over again.

For completeness, we note that once the revocation keys have been exchanged, marked before as *phase2*, the counter party is now motivated to keep the revocation key. This is where the watchtower functionality enters the loop and serves as hot backup (we skip the details about justice transaction and privacy here) thus avoids this attack vector.

4.2.4 Revocation key theft

Focusing again on *phase1*. A channel can be closed by one of the sides if that party publishes its latest commitment transaction. The revocation key of that particular commitment transaction must not be revealed to the counter party until the party‘s timelock has passed. In the case the revocation key is known to an attacker, the attacker can collude with the counter party to steal the party‘s funds. Concretely, say that in a channel with total of 10BTC the latest state is that Alice owns 2BTC and Bob owns 8BTC. Assuming the attacker learns Alice‘s revocation key, Bob will stop responding, making Alice trigger unilaterally channel close at some point. when Alice chooses to close the channel she publishes a commitment transaction to which her revocation key was never revealed, but in our case it is known to the attacker. With this commitment transaction, Bob can immediately spend his 8BTC while Alice must wait for a timelock to expire to spend her 2BTC. However, the 2BTC can also be spent immediately given Bob signature and knowledge of the revocation key. Bob and the attacker can jointly spend this output since Bob can provide his signature and the attacker knows the revocation key. This is a similar situation to multisig contract which means that Bob and the attacker will "sign" only on spending transaction that divides the 2BTC output equally between them.

4.2.5 Multihop key deletion

When a multihop key gets corrupted between HTLC creation and HTLC closure, there is no consequence that can lead to channel corruption as in the case of revocation key. However, it may lead to breaking the chain between the payee and the payer, potentially resulting in lost funds for the attacked node and failed HTLC. We note that while revocation keys must maintain integrity between payments, multihop keys integrity must be kept only within a payment.

4.2.6 Multihop key theft

We discuss only the case in which the attacker attacks the payee, who generated the multihop key. We first note that multihop key deletion, under our attacker model, is handled elegantly in the BOLT specifications. Indeed, if the payee fails to provide the key, which acts as a proof of payment, the HTLC will fail according to the spec. On the other hand, if the payee provided the correct multihop key, under our assumption, the HTLC is considered removed. The more interesting scenario is when the attacker gets access to the multihop key before the payee is required to release it. We will assume that the multihop key is a preimage of a hash function. The attacker can approach any node in the route between the payee and payer. Once that node gets an incoming HTLC transaction and before it generates an outgoing HTLC transaction for the same amount - the node can use the preimage received from the attacker to remove the incoming HTLC, profiting the entire amount. The rest of the nodes up the chain to the payer will continue according to protocol, until the payment is taken from the payer. This also means that from the payer view, the payment proof is verified, meaning the payee got its payment [4]. We give a concrete example: Let us say that Dave is supposed to pay Alice 10BTC via Bob and Charlie. Alice generates a preimage and send the hash of it to Dave. At this point the attacker extracts the preimage from Alice. The attacker contacts Charlie and provides him with the preimage. Once Dave offers Charlie HTLC commitment transaction, Charlie does not propagate the HTLC offer to Bob but instead removes the HTLC by providing Dave with the preimage and profiting 10BTC.

This attack is trickier to pull off because of the required knowledge of the exact route and the timing. There are ways around it, but for the sake of simplicity we assume that the attacker blindly approaches a set of large nodes connected to the payee (Alice). Finally, to achieve guaranteed profit for the attacker, we suggest the following: The node will produce an outgoing HTLC commitment transaction to the attacker, using the same hash value but for half the amount. The attacker will release the preimage to the node by removing the HTLC, and the node will propagate the preimage getting the full amount. This way the attacker and the node split the profit equally between them. In our example, The attacker will "replace" Bob and will receive from Charlie HTLC for 5BTC conditioned on the same hash value. Once the attacker and Charlie remove the HTLC between them, Charlie can now move on and remove the HTLC for 10BTC with Dave, profiting 5BTC.

4.3 Protection Mechanisms

In this work we present the concept of a *Battlement*. The two main ideas we put forth is to extend the functionality of the existing watchtower architecture and second, rely on a threshold assumption. Combining these two ideas we define a Battlement as a set of n connected watchtowers. Out of

[4] The implications of this are outside the scope of this work

which, it is assumed that no more than $t < n/2$ watchtowers are corrupted. Corruption here is in the cryptographic sense, i.e. t watchtowers may deviate from the protocols and might collude to break any security guarantee promised by a given protocol. This is often called security against malicious adversaries. On top of the allowed cryptographic corruption, we still assume that watchtowers are selfish (rational) and will take advantage of any strategy that maximize their gains. We show how a user can delegate responsibilities to the Battlement, improving on existing attacks described before but without compromising on security.

4.3.1 Battlement for key management

A Battlement adds a new layer of proactive defense to the lightning network. We now present how each of the keys used in a channel life cycle can be managed by a Battlement under threshold assumption and using tools from threshold cryptography.

Protecting the secret key: The secret key is being used to sign when a user opens, closes, and updates a channel. Once a secret key has been designated by the user, the user will play the role of a dealer in a $\{t, n\}$ linear secret sharing scheme to distribute shares of his secret key to the n watchtowers. Immediately after, the user will delete its secret key from memory. When the user requires to open the channel, the user will ask the watchtowers to run a threshold signature protocol and output the required signature. There will be separate requests for opening, closing and updating the channel. By running threshold signatures we are guaranteed that the secret key will never be reconstructed and therefore exposed. Furthermore, before the channel is created and after it is closed, the secret key has no power and it is isolated from other secret keys owned by the user. The watchtowers quorum can be made robust by instantiating the threshold signature protocol with a construction that supports Identifiable Abort, e.g. [5] for threshold ECDSA.

It is now clear that an attacker cannot corrupt/delete a secret key because for this the attacker must corrupt $> t$ watchtowers which violates the threshold assumption. For the same reason, once the secret key has been distributed and locally deleted from the user memory, it cannot be learnt by an attacker without breaking the threshold assumption. We note that the user still needs to authenticate to the Battlement. In a sense we translated the managing of multiple signing keys by the user to an authentication problem. It is possible to apply here any existing authentication scheme that works in a classic client-server setting.

Protecting the revocation keys: Once the revocation key is generated, locally by the party, it will be secret shared and distributed to the n watchtowers and be deleted from local memory of the party. This way, an attacker will have to compromise $t + 1$ watchtowers to learn or corrupt the revocation key. While simple, there are few issues with this techniques that we have to address:

- Reconstruction: under normal circumstances, the revocation key of a commitment transaction must be shared with the counter party in the channel once a new commitment transaction has been signed and exchanged. If we let the revocation key generating party to issue reconstruction requests with no other safety mechanism in place, an attacker can impersonate the party and request reconstruction. There are several ways to mitigate such an attack. Here we suggest only one: A reconstruction message must contain a signature using the party‘s secret key for that channel. During distribution the party will identify itself via the public key corresponding to its secret key. If we assume that the secret key management is done using the Battlement as

described before, it means that the public key must already be registered with the watchtower as part of opening the channel. A valid reconstruction request is therefore one that includes a signature produced by the Battlement itself. Finally, since a signature is already required as part of updating the channel we can simply bind the two processes together: for each threshold signature on a channel update the watchtowers will also perform reconstruction of previous revocation key.

- In the current implementation of the lightning network, the revocation key r is a preimage of a double hash function, such that $s = \text{RIPEMD160}(\text{SHA256}(r))$. This is following the convention used in Bitcoin to prevent length extension attacks. However in our context of revocation this is not relevant. To protect against a malicious distributor, we offer the following protocol for the party:

  1. Sample r randomly
  2. Compute $s_1 = \text{SHA256}(r)$
  3. Use Feldman verifiable secret sharing [7], to secret share r, such that each party $0 < i \leq n$ receives a point (i, r_i), on some polynomial $p(\cdot)$ whose coefficients $a_0 = r, \{a_j\}_{j=1}^{t-1}$ are kept hidden. Let $\mathbf{z}$ be the commitment vector to $\{a_j\}_{j=0}^{t-1}$.
  4. Compute a zk proof with witness $\omega = r$ and statement $\delta = \{s_1, z_0\}$, proving $s_1 = \text{SHA256}(decom(z_0))$
  5. Output $s = \text{RIPEMD160}(s_1)$

  where the zero knowledge proof can be based on a mixed algebraic and non-algebraic zero knowledge proof such as [6]. Alternatively to the above protocol, the watchtowers can generate the preimage in a distributed way among themselves, however due to non-algebraic structure of the hash function this is a costly operation.

- An accountability may also exist for $t < n/2$, as watchtowers in a group can oversee each other and in case a minority of the watchtowers are failing to deliver outputs to the user, the honest majority of the watchtowers can together decide to slash the faulty watchtower deposits. Using a simple proactive secret sharing (e.g. [11]), the honest majority can perform "healing" to the watchtowers committee, replacing the faulty watchtowers.

Protecting the multihop keys: To protect the multihop keys we need to assure that the hash preimage will not be available before all hops from the payer to the payee are completed, i.e. each party in the route added an HTLC transaction. The main requirement is thus that watchtowers should be aware to the route that the payment must take. In fact, we argue that it is enough for the watchtowers to be aware only for the direct channels connecting the payee to the network. Once a valid incoming HTLC arrives via one of these channels, it is safe to reconstruct the HTLC preimage. We keep as an open question the exact protocol to eliminate this privacy issue fully and note that we already assume that the direct channels are known to the watchtowers protecting the revocation keys. To distribute the preimages to the watchtowers we let the user define a batch parameter γ, e.g. $\gamma = 1000$, and produce a γ times nested SHA256: $y = ...(\text{SHA256}(\text{SHA256}(x)))$. The user publishes y and distributes $\gamma \cdot n$ verifiable secret shares of the nested preimages. Finally the user deletes all preimages from its memory. Each time the user accepts a payment via HTLC, it uses y_i, the latest public hash value and

request reconstruction from the Battlement for x_i ($y_i = \text{SHA256}(x_i)$, $y_{i-1} = x_i$). The Battlement will be triggered by a valid incoming HTLC transaction from one of the user's neighbors for hash value of y_i. After γ hash values have been exhausted the user will generate γ more. With this design, the payee can make sure of the integrity of the preimage and punish misbehaving watchtowers. It is assumed that the preimage generation protocol is done with honest distributor as unlike revocation keys the preimages are not purposed yet. There is also no incentive for an attacker at this stage. A more involved protocol that removes this assumption is feasible but out of scope.

The above protocol solves immediately the multihop key theft problem. To assist with multihop key deletion for any node in the route, we enable nodes to request the Battlement for reconstruction for the preimage indexed by y_i. We trade-off privacy for liveness: The attacked node must provide the Battlement with a completed HTLC, for y_i, revealing some information about its existing channels. This works because the payee already got its payment (completed the HTLC for y_i with its neighbour node), from that point, any node in the route can get access to the preimage x_i via the battlement and the order in which nodes learn the preimage is not critical, under our model of multihop key, see §4.1.

5 Using a Battlement to mitigate Bribery Attacks

The existing watchtower design is addressing only a single threat, in our terminology: the safe keeping and availability of the revocation keys during their second phase. In this section we describe a simple bribing attack that applies for rational watchtowers. We follow up explaining how the Battlement structure introduced in §4.3 mitigates such an attack.

5.1 Bribing the Watchtower

We make use of an ideal functionality to abstract and simplify the specification of a watchtower:

> This functionality works in a hybrid model with ideal global ledger blockchain functionality $\mathcal{G}_L$[3] and a predicate V to detect cheating and extract a justice transaction from memory.
>
> - Upon receiving a command `Store(payload)` from party P_i, store `{i, payload}`
> - Upon reading a state from $\mathcal{G}_L$, run V(`payload, state`) for all stored payloads. If V returns (1,`justice`), Send `Fraud(i)` message to adversary $\mathcal{A}$, Upon receiving `Punish` from $\mathcal{A}$, send a `Submit` command to $\mathcal{G}_L$ with `tx=justice`.

Watchtower ideal functionality $\mathcal{F}_w$

The interesting part to note is that the watchtower will require the adversary permission before publishing a justice transaction. Indeed, watchtowers have no guaranteed output delivery, as stated in the latest revision of BOLT13 [5] : *...the hiring protocol does not guarantee the transaction inclusion..* This is true even for accountable watchtowers since the watchtower has plausible deniability: it can always claim that it suffered some DoS or eclipse attack during the time, blame it on force majeure

[5] `https://github.com/sr-gi/bolt13/blob/master/13-watchtowers.md`

such as network glitch and so on. As a matter of fact, an attacker can indeed eclipse the watchtower node or DoS its network, but since these are expensive attacks we offer here a cheaper alternative.

We assume that adversary, Bob, wants to publish an outdated commitment transaction. Now he must wait for a timelock to expire, while Alice, assumed offline, counts on her watchtower to catch the malicious commitment transaction and publish a justice transaction on chain. We assume the watchtower is selfish, i.e. optimized for gain, same as with miners. Alice pays her watchtower a small fee (we defer the discussion about the exact fee model). Bob however, can approach the watchtower operator with a much more attractive offer: Bob will double the watchtower fee, if the watchtower will ignore the malicious commitment transaction.

In current watchtower implementations there is a form of privacy guarantee, hiding details about the channel, but this is only until fraudulent outdated commitment transaction is published. From the published transaction and the data at hand, the watchtower is able to defer the necessary information, e.g. that the transaction came from a channel between Alice and Bob.

The natural mitigation is for Alice to hire few watchtowers. However, under our model Bob can simply bribe all the watchtowers in existence as in theory the bribery money will go out of Alice pocket, funded from the outdated commitment transaction. For example, if Bob and Alice open a channel where Bob's input is 10BTC and Alice's input is 0: Bob will pay Alice 1BTC and then 8BTC for some service, updating the channel twice. Then Bob will publish the outdated commitment transaction from the state when he paid only 1BTC to Alice and now use 7BTC to bribe the watchtowers, lending him a profit of 1BTC.

To show the depth of this issue we give two more potential solutions that cannot work: (1) Alice can pay more, however this will lead to Bob paying more to the watchtower as well. The watchtower can even approach Alice and tell her that Bob paid more and she will have to outbid him. (2) We can put a mechanism to punish the watchtower, e.g. slashing each time it fails to publish a justice transaction. However, then Bob and Alice can collude to take the watchtower deposit.

5.2 Bribing a battlement is harder than bribing a watchtower

As we showed before, increasing the number of hired watchtowers, does not solve the problem since the attacker can always bribe more watchtowers as well, until we reach a point where Alice divides the total amount she has in the channel among all the watchtowers, which makes it not worth it for her to change the state in the channel. The reason a Battlement can get over this issue is because of the relation that exist between the watchtowers. In our model, watchtowers are no longer stand-alone entities but rather connected and dependent on one another. In the stand-alone watchtower, each watchtower faced two options - take Alice's fee or take Bob's bribe. Option two can always be more appealing. Now let us assume we have two connected watchtowers. This allows us to introduce a third option: if one watchtower does not publish the justice transaction, the watchtower that published the justice transaction can in addition present a publicly verifiable proof that the first watchtower participated in the protocol. A smart contract can verify this proof and slash the first watchtower deposit and give it to the second watchtower who behaved honestly. If the deposit that the watchtower can win is larger than Alice funds in the channel, which is the maximum amount of bribe Bob can offer, we got a classical Prisoner's Dilemma! For which we know from game theory that the watchtower strategy in one shot game will be to publish the justice transaction and try to take the deposit of the other watchtower. The same idea applies for a general $\{t, n\}$-threshold. In case that Alice uses changing sets of watchtowers

the strategy will be optimal.

Lastly, one important ingredient is how to generate the publicly verifiable proofs of participation and enforce financial penalties. Luckily, the MPC literature noticed this specific problem and concrete efficient protocols exist [4] .

6 Discussion

The lightning network key management problem is fascinating: The channel construction poses different restrictions on the time-frame and utility of each key. This allowed us to find non-trivial ways in which threshold cryptography can be used to address a specific attacker model in which the attacker is an outsider with an access to subset of the keys. In the common scenario of private key compromise, for example when attacker learns the master private key of a Bitcoin wallet, the attacker gets full control over the funds locked. However, as this work demonstrates, it is far less trivial for an attacker of a lightning network channel to translate the attack to profit. It depends on the type of access, the phase the channel is at and on specific attack protocols we developed in the work. The involved attack surface of lightning network, makes the problem of key management more interesting. In particular, the threshold assumption that dictates some honest majority or honest subset, can be combined with a "punish" functionality to resolve both attacks from the outside and bribing attacks from the inside. This idea of building on both game theory assuming rational behaviour and on cryptographic security of threshold cryptography should be further explored in our context.

Our methods to protect the various keys can be applied separately or together. It depends on the user preference on how much trust it is willing to delegate to the Battlement. There is some obvious shortcomings in our techniques: First and foremost they require to introduce a new cryptographic assumption about threshold security, not present currently in the lightning network. While we do describe a concrete bribing attack on the existing watchtower design that can be mitigated with the introduction of a threshold structure, it is not necessarily the only solution. Second, some of the protocols we use involve new cryptographic primitives such as the one required for threshold signing or for the mixed zero knowledge proof. Lastly, all our protocols requires additional interaction and communication over what is currently required within the already message packed lightning network. The overhead in communication might slow down the basic channel operations and we leave it for future work to run complexity analysis and determine if the added complexity affects the user experience.

7 Acknowledgements

We would like to thank Sergi Delgado Segura, Tal Be'ery, Varun Madathil, Doron Zarchy, Claudio Orlandi, Roy Sheinfeld, Ittay Eyal, Robin Linus and Bastien Teinturier

References

[1] Georgia Avarikioti, Eleftherios Kokoris Kogias, and Roger Wattenhofer. Brick: Asynchronous state channels. *arXiv preprint arXiv:1905.11360*, 2019.

[2] Zeta Avarikioti, Orfeas Stefanos Thyfronitis Litos, and Roger Wattenhofer. Cerberus channels: Incentivizing watchtowers for bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 346–366. Springer, 2020.

[3] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In *Annual International Cryptology Conference*, pages 324–356. Springer, 2017.

[4] Carsten Baum, Bernardo David, and Rafael Dowsley. Insured mpc: Efficient secure computation with financial penalties. *Financial Cryptography and Data Security (FC)*, 2020.

[5] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. Uc non-interactive, proactive, threshold ecdsa with identifiable aborts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1769–1787, 2020.

[6] Melissa Chase, Chaya Ganesh, and Payman Mohassel. Efficient zero-knowledge proof of algebraic and non-algebraic statements with applications to privacy preserving credentials. In *Annual International Cryptology Conference*, pages 499–530. Springer, 2016.

[7] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 427–438. IEEE, 1987.

[8] Majid Khabbazian, Tejaswi Nadahalli, and Roger Wattenhofer. Outpost: A responsive lightweight watchtower. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 31–40, 2019.

[9] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 455–471, 2017.

[10] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous multi-hop locks for blockchain scalability and interoperability. In *NDSS*, 2019.

[11] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. Churp: Dynamic-committee proactive secret sharing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2369–2386, 2019.

[12] Patrick McCorry, Surya Bakshi, Iddo Bentov, Sarah Meiklejohn, and Andrew Miller. Pisa: Arbitration outsourcing for state channels. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 16–30, 2019.

[13] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[14] Omer Shlomovits and Oded Leiba. Jugglingswap: Scriptless atomic cross-chain swaps. *arXiv preprint arXiv:2007.14423*, 2020.