Payment Trees: Low Collateral Payments for Payment Channel Networks

Maxim Jourenko[1](✉), Mario Larangeira[1,2], and Keisuke Tanaka[1]

[1] Department of Mathematical and Computing Sciences, School of Computing, Tokyo Institute of Technology, Tokyo 152-8550, Japan
jourenko.m.ab@m.titech.ac.jp, mario@c.titech.ac.jp, keisuke@is.titech.ac.jp

[2] Input Output Hong Kong, Hong Kong, China
mario.larangeira@iohk.io
http://iohk.io

Abstract. The security of blockchain based decentralized ledgers relies on consensus protocols executed between mutually distrustful parties. Such protocols incur delays which severely limit the throughput of such ledgers. Payment and state channels enable execution of offchain protocols that allow interaction between parties without involving the consensus protocol. Protocols such as Hashed Timelock Contracts (HTLC) and Sprites (FC'19) connect channels into Payment Channel Networks (PCN) allowing payments across a path of payment channels. Such a payment requires each party to lock away funds for an amount of time. The product of funds and locktime is the collateral of the party, i.e., their cost of opportunity to forward a payment. In the case of HTLC, the locktime is linear to the length of the path, making the total collateral invested across the path quadratic in size of its length. Sprites improved on this by reducing the locktime to a constant by utilizing smart contracts. Atomic Multi-Channel Updates (AMCU), published at CCS'19, introduced constant collateral payments without smart contracts. In this work we present the Channel Closure attack on AMCU that allows a malicious adversary to make honest parties lose funds. Furthermore, we propose the Payment Trees protocol that allows payments across a PCN with linear total collateral without the aid of smart contracts; a competitive performance similar to Sprites, and yet compatible to Bitcoin.

Keywords: Blockchain · Payment channel · HTLC · Collateral

1 Introduction

Blockchain based decentralized ledgers as introduced by Nakamoto [12] have enjoyed popularity and received interest from the research community and practitioners. Consensus protocols allow these ledgers to be operated by mutually

This work was supported by the Input Output Cryptocurrency Collaborative Research Chair funded by IOHK, JST CREST JPMJCR14D6, JST OPERA.

N. Borisov and C. Diaz (Eds.): FC 2021, LNCS 12675, pp. 189–208, 2021.
https://doi.org/10.1007/978-3-662-64331-0_10

distrustful parties at the cost of limited throughput. For example, Visa as a centralized system can process orders of magnitude more transactions within a given time frame than the most prominent blockchains as Bitcoin and Ethereum.

The main motivation for the development of offchain protocols is to close the gap in transaction throughput. The idea is to allow parties to interact with each other without interacting with the ledger, while still being able to use it to resolve disputes. Offchain protocols operate on *channels* that are created between two parties. Channels hold a state which can be enforced on the ledger. Payment channels [4,13,15] store the number of coins the two parties have locked inside that channel. Offchain protocols provide a means to alter this state arbitrarily often and thus improving the transaction throughput in the overall system.

Individual channels can be extended to channel networks, e.g. PCNs Lightning [15] and Raiden [1]. This is done using techniques, such as HTLC [2,15], that allow for payments of $b \in \mathbb{N}$ coins across a path of payment channels of length $n \in \mathbb{N}$. This is performed by executing the same payment on each channel within the payment path atomically. All parties on the payment path have to lock the payment amount for a duration of up to *locktime*. The opportunity cost a party has to invest is the *collateral* [10] which equals the payment amount b multiplied by the locktime. In turn, parties can impose fees to invest collateral. In the case of HTLC, a party's collateral equals $\mathcal{O}(nb\Delta)$ in the worst-case where Δ is a parameter of the underlying ledger and is the upper limit of the time it takes for a transaction to be included in the ledger.

High collateral investments can be exploited by malicious adversaries to perform *grieving* and *denial-of-service* attacks [11,14]. For example, an attacker might operate a channel to collect fees by forwarding payments. However, payments might be routed through competing channels instead. To sabotage the competitor, the attacker can route a payment through these channels without the intent of executing it, locking the competing channel's coins for the entirety of the locktime. These channels experience a denial-of-service scenario by being unable to forward any other payments, losing fees that the attacker can collect through their own channel. Performing this attack on a large scale can result in denial-of-service for the whole PCN. On a lower scale, a griever might force parties to lock away their funds for as long as possible by delaying their cooperation until the last moment. An alternative form of this attack involves routing multiple low value payments through a competing channel, up until a point where the channel cannot add any further HTLCs even though it contains enough coins. In the case of the Lightning network, these types of denial-of-service attacks can lock all of a channel's coins for up to around 2 weeks [11].[1]

For HTLC the total collateral locked over a whole payment path is $\mathcal{O}(n^2 b\Delta)$ and therefore quadratic in the payment paths length. Sprites [10] reduce the collateral of each party to $\mathcal{O}(b(n+\Delta))$ and the total collateral to $\mathcal{O}(bn(n+\Delta))$ by utilizing a smart contract. This is considered to be constant and linear respectively, since $n << \Delta$ such that $n + \Delta < 2\Delta$. Sprites mitigate the damage done by a possible attacker but its implementation is limited to ledgers with smart

[1] https://cointelegraph.com/news/developer-reveals-biggest-unsolvable-lightning-attack-vector.

contract capability. The Atomic Multi-Channel Updates (AMCU) protocol [7] is an attempt to close this gap and enable payments with constant collateral on ledgers without smart contract capabilities. However, even though AMCU is formalized as a functionality within Canetti's UC Framework [3], the very last, but crucial step, of the updateState function *does not seem to be presented* in the description of the AMCU protocol, and neither addressed by the simulator [7]. This gap results in a vulnerability that can be exploited by a malicious adversary to steal funds from honest parties.

Related Work. Payment channels [4,13,15] themselves allow only for offchain payments between two parties. Offchain protocols such as HTLCs [2,15] and Sprites [10] allow to perform payments across paths of channels allowing for the implementation of PCNs. Prominent examples are the Lightning Network [15] and Raiden [1]. Although offchain protocols exist that create new *virtual* channels out of two existing channels as Perun [5,6] and Lightweight Virtual Payment Channels [8], this work focuses on performing individual payments across a PCN. In the following we consider a payment of $b \in \mathbb{N}$ coins across a path of $n \in \mathbb{N}$ channels involving parties $\mathcal{P}_0, \dots, \mathcal{P}_n$.

The most prominent technique is based on HTLCs [2,15], which are scripts that perform conditional payments within a channel: The payer locks funds into the contract that are paid out if the payee can present a secret x such that $y = \mathcal{H}(x)$ where $\mathcal{H}$ is a cryptographic hash function. Otherwise, after time *locktime* the payment times-out and the payer can reclaim their funds. This contract is replicated along all channels within a payment path. The payment is performed as soon as $\mathcal{P}_n$ reveals x to their predecessor who then learns the value of x allowing them to claim the payment from their predecessor in turn. An attacker $\mathcal{P}_i, 0 < i \leq n$ might attempt to delay revelation of x to their predecessor until briefly before expiration of the *locktime*. To allow $\mathcal{P}_{i-1}$ to forward x in time, their locktime needs to be increased by at least Δ. This results in a locktime in $\mathcal{O}(n\Delta)$ and a total locktime in $\Theta(n^2\Delta)$.

Sprites [10] aim to reduce the locktime of a party up to a constant $\mathcal{O}(n + \Delta)$ where $n << \Delta$. This is done by setting up a smart contract entity called *PreimageManager*, s.t. submitting x to the PreimageManager allows to broadcast it to all nodes within a payment path in at most n communication rounds. The protocol requires creation of a smart contract, making it unavailable to script based ledgers as Bitcoin. AMCU [7] attempts to close this gap, i.e. compatibility with Bitcoin, by introducing an approach for constant locktime payments without the need of smart contracts. AMCU sets up payments on each channel within a payment path that are performed on the condition that an *Enable* transaction is created, upon which all payments are performed atomically. However, this Enable transactions results in several issues. For one, its size grows linearly in the payment path's length, making its implementation prohibitive for ledgers which have an upper limit for block size and transaction size. Moreover, no party has control over all of the Enable transaction's inputs. A malicious adversary can make two parties collaborate to double spend one of the Enable transaction's inputs, such that no party is able to enforce the payment on the ledger. If the double-spending is timed appropriately, this can lead to an attacker stealing funds from honest parties.

Jourenko et al. [8] proposed an offchain protocol that takes two channels γ_A and γ_B as input, one between $\mathcal{P}_A$ and $\mathcal{P}_I$ and one between $\mathcal{P}_I$ and $\mathcal{P}_B$ and creates a new channel γ^v between $\mathcal{P}_A$ and $\mathcal{P}_B$. As this approach is not optimized for individual payments, using it for this purpose would result in excessive collateral as parties would need to lock away more coins for a longer duration as in existing approaches. However, we re-use techniques from the lightweight virtual payment channel construction for the Payment Tree protocol.

Our Contributions. Our contributions are threefold. 1) We present an attack on AMCU performed by a malicious adversary. 2) We present *Payment Trees* that allow for payments across paths within a PCN without the need of smart contracts, requiring *only* logarithmic individual collateral $\mathcal{O}(b\Delta \log n)$ while requiring only linear total collateral $\mathcal{O}(nb\Delta)$ such that its performance is comparable to Sprites. 3) We provide efficiency and security analysis of Payment Trees, proving the properties *Balance Security* and *Liveness.*

Structure. In the remainder of this work, first, we provide background to this work in Sect. 2. We give an outline of the Channel Closure attack in Sect. 3. Next, we give an informal overview of the Payment Tree protocol in Sect. 4. Afterwards, we introduce the types of transactions used for our construction in Sect. 5 before introducing Payment Trees in Sect. 6 followed by efficiency and security analysis in Sect. 7. We conclude in Sect. 8.

2 Background

Notation. Throughout this work we make use of tuples and use short-hand notations as follows. Let $(a_1, a_2, \ldots, a_n)$ be a definition of a tuple of type A and let α be an instantiation of A. Then $\alpha.a_i$ equals the i-th entry of α.

The UTXO Paradigm. A UTXO is a tuple of the form $(b,\ \pi)$ where $b \in \mathbb{N}$ is an amount of coins and $\pi \in \{0,1\}^*$ is a script. The b coins of the UTXO are claimed by providing a witness $w \in \{0,1\}^*$ s.t. $\pi(w) = \mathsf{True}$. The state of the ledger is represented by a set of UTXO S_{utxo}, which can be changed by a transaction of the form $(U_{in},\ U_{out},\ t)$ where $t \in \mathbb{N}$ is the (absolute) timelock represented as a point in time, U_{out} is the list of unique UTXO for the *outputs* of the transaction, and U_{in} is the set of transaction *inputs* of the form $(\mathsf{ref}(u),\ w_u)$ where $\mathsf{ref}(u)$ is the pointer to the UTXO u, and w_u is the witness.

A transaction $(U_{in},\ U_{out},\ t)$ needs to fulfill the following conditions. (1) The locktime has passed, i.e. $t \leq \tau$ where τ is the current time, (2) all witnesses are valid, i.e. $\forall(\mathsf{ref}(u), w) \in U_{in} : u.\pi(w) = \mathsf{True}$ (3) the coins within the newly created UTXO are less or equal to those in the transaction's inputs, i.e. $\Sigma_{(\mathsf{ref}(u),w)\in U_{in}} u.b \geq \Sigma_{u \in U_{out}} u.b$, (4) all UTXOs in the transaction's inputs exist and have not yet been spent, i.e. $\forall(\mathsf{ref}(u), w) \in U_{in} : u \in S_{utxo}$. The transaction has the following effect on the ledger. All UTXOs referenced within U_{in}

are removed from S_{utxo} and all UTXOs defined in U_{out} are added to S_{utxo}. A transaction T is included in the ledger within a duration $\Delta \in \mathbb{N}$. Condition (4) implies that no UTXO can be claimed by two different transactions. After sending T to the ledger, if within time Δ another transaction T' claiming a subset of the same UTXOs as T is sent to the ledger, it would result in a race condition, in which it is non-deterministic whether T or T' will change the ledger's state. We note that while we use Δ as a ledger parameter in practice this value has to be estimated for real-world implementations. Special care has to be taken when selecting a value. A value that is too low breaks our assumptions and the protocol's security. A value too high increases the collateral and therefore the impact of attacks such as congestion and lockdown [11,14].

Transaction Graph. All transactions included in the ledger form a directed and acyclic graph. The set of all transactions form its vertices. An edge (T_0, T_1) from transaction T_0 to transaction T_1 exists, if T_1's inputs contain a pointer to one of T_0's outputs, i.e. $\exists u : u \in T_0.U_{out} \wedge (\mathsf{ref}(u), w) \in T_1.U_{in}$. Note that a transaction can only be included in a ledger if all of its ancestors have been included in the ledger before. In the remainder of this work we reference sets of transactions that are connected to form a sub-tree as *transaction trees*.

Scripting. Scripts in this work specify a UTXOs owner by requiring a signature of the transaction that spends the UTXO with the recipient's verification key. This is extended to 2-out-of-2 multisignatures that require verification keys of two parties $\mathcal{P}$ and $\mathcal{P}'$ effectively creating a shared wallet between both parties that can only be spent with consent of both parties. In the remainder of this work UTXOs requiring 2-out-of-2 multisignatures are termed Funding UTXO. Throughout this work we simplify scripts by only stating the set of parties which need to provide their signatures to spend the respective UTXO.

Channels. A channel γ between two parties consists of sub-protocols setup, closure and dispute. In setup both parties create a transaction Tr_f containing a Funding UTXO between each other which locks their funds into the channel. They create a transaction tree with the Funding UTXO as its ancestor that represents the channel which we reference in the remainder of this work as *channel-tree*. Only after the channel-tree is created and either party holds signatures of its transactions, both parties sign and commit Tr_f to the ledger while holding off commitment of transactions within the channel-tree. Both parties can perform closure of the channel by committing a transaction to the ledger that spends the Funding UTXO unlocking the channel's funds according to its most recent state. In case of a dispute, the dispute sub-protocol enforces the channel's state by committing the channel-tree's transactions onto the ledger.

Offchain Protocols perform a state transition of a channel by transforming its channel-tree. Any honest party must be able to enforce the new channel's state which might require an explicit invalidation step that disables commitment of an older version of the channel-tree or allows for punishment of a party that does so. An efficiency requirement of offchain protocols is that performing them $n \in \mathbb{N}$ times grows the channel-tree by at most $\mathcal{O}(1)$ transactions.

Invalidation by Timelock. Timelocks can be used to define at which point a transaction can be committed to the ledger. Assume there are two transactions that spend the same UTXO, but which have timelocks that are 1) in the future and 2) have a difference of at least Δ. In this case parties can enforce commitment of the transaction with the lower timelock to the ledger. The transaction with the lower timelock *invalidates* the transaction with the higher timelock.

Hashed Timelock Contracts. Let $\mathcal{P}_0, \mathcal{P}_1, \dots, \mathcal{P}_n, n \in \mathbb{N}$ be parties where parties $\mathcal{P}_{i-1}$ and $\mathcal{P}_i, i \in \{1, \dots, n\}$ control channel γ_i. HTLCs are used to perform payments of $b \in \mathbb{N}$ coins from $\mathcal{P}_0$ to $\mathcal{P}_n$ by replicating the payment on each channel γ_i within a payment path $\gamma_1, \dots, \gamma_n$ from $\mathcal{P}_0$ to $\mathcal{P}_n$. (1) On a channel γ_j, $j \in \{1, \dots, n\}$ the payment is performed by extending the channel-tree with a conditional payment: If the payee $\mathcal{P}_j$ can show the pre-image $x \in \mathbb{N}$ of a hashed value $y = H(x)$, where H is a cryptographic hash function, they will receive b coins from the payer $\mathcal{P}_{j-1}$. However, after expiration of a locktime t_j the payment expires and the payer $\mathcal{P}_{j-1}$ will have their coins refunded instead. (2) Only after the conditional payments are set up on all channels, the payment is executed atomically by having $\mathcal{P}_n$ show the pre-image x to $\mathcal{P}_{n-1}$, proving that they have the *capability* to claim the coins on the ledger through the conditional payment. In turn, $\mathcal{P}_{n-1}$ learns the pre-image x s.t. they can show it to party $\mathcal{P}_{n-2}$ reclaiming the coins they forwarded to $\mathcal{P}_n$. The information on x propagates through the whole payment path in this manner. (3) Lastly, to keep the payment offchain, the parties need to consolidate the payment on each channel respectively. This is done by updating the channel-tree. The conditional-payment is removed and the b coins that were locked into the channel are credited to the payee. At this point the channel-tree has the same form as before the payment, but with updated balance distribution to account for the payment. This ensures that the channel-tree does not grow in size with each payment, thus fulfilling the efficiency requirements of an offchain protocol. Note that, if the payer $\mathcal{P}_{j-1}$ does not cooperate with consolidation, payee $\mathcal{P}_i$ can reclaim their coins by resolving the conditional payment on the ledger instead. Due to this the timelock t_j has to be chosen s.t. $\mathcal{P}_i$ has enough time to do so before the conditional payment expires, even if they learn the pre-image from $\mathcal{P}_{i+1}$ at the last moment just shortly before expiration of timelock t_{j+1}. Thus the relation $t_i \geq t_{i+1} + \Delta$ has to hold, making the locktime grow linearly with the payment path's length. This results in a collateral cost of $bt_i \in \mathcal{O}(bn^2\Delta)$ which is quadratic in the path's length.

The Wormhole Attack. The HTLC protocol is vulnerable to the wormhole attack [9]. An adversary controlling two parties $\mathcal{P}_i$, $\mathcal{P}_j$, $1 \leq i \leq j + 2 \leq n - 1$ within a payment path can prevent intermediaries $k, i < k < j$ to participate at the payment and receive their fees by having $\mathcal{P}_i$ forward pre-image x to $\mathcal{P}_{i-1}$ after $\mathcal{P}_j$ learns it from $\mathcal{P}_{j+1}$ and without forwarding it to party $\mathcal{P}_{j-1}$.

Brief Description of AMCU. A payment within the AMCU protocol is performed by replicating the payment on each channel using *Consume* transactions. All Consume transactions share a common ancestor within the protocol's transaction tree which is the *Enable* transaction. The Enable transaction has UTXOs from each individual channel as input and thus requires signatures of all parties within the payment path. As soon as the parties exchange signatures for the Enable transaction, all Consume transaction could be committed on the ledger, thus performing the payment. If any party refuses to collaborate in the creation of the Enable transaction, all parties have their coins refunded using *Lock* transactions after the expiration of the specified locktime period. As we show in the next section, in contrast to HTLCs, AMCU cannot ensure that all parties have the capability to claim their coins on the ledger after the payment. In fact it takes only one pair of parties controlling a channel to spend one of the Enable transaction's inputs with a different transaction s.t. the Enable transaction and transitively the Consume transactions cannot be committed to the ledger. This could be remedied by performing a consolidation step on all channels atomically. Although the functionality PCN^+, that models AMCU, correctly performs this consolidation step, the AMCU protocol itself does not.

3 The Channel Closure Attack on AMCU

In the following we present the Channel Closure attack informally. A more detailed description of AMCU and a formal treatment of the attack are supplemented in the full version of the paper.

The Vulnerability. While the Enable transaction is the core of the AMCU construction, it also seems to be its vulnerability. While the Enable transaction receives inputs from each channel, no party has control over all channels within the payment path. At any time, two parties sharing a channel can maliciously spend a UTXO that is provided as input of the transaction, or as input to any of its ancestors within the transaction tree. When this happens, the Enable transaction cannot be committed to the ledger and all parties have their coins refunded through Lock transactions. Effectively, no party can enforce payment after execution of the AMCU protocol. On top of that, an adversary can take this further, performing a Channel Closure attack to steal funds from honest parties. We remark that PCN payments require a consolidation step in which a payment is included within the parties' individual channels. While the functionality PCN^+ modeling AMCU performs a consolidation step atomically on all channels, this step is omitted by the AMCU protocol. Second, performing the consolidation step atomically on all channels is highly non-trivial as atomic operations on multiple channels is exactly the problem statement that protocols such as HTLCs, Sprites and AMCU themselves attempt to solve.

The Channel Closure Attack is performed by abusing exactly these two observations. First, the adversary corrupts two parties within a payment path $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$. These parties cooperate in execution of the AMCU protocol right up until the consolidation step. Then, $\mathcal{P}_i$ performs the consolidation step with $\mathcal{P}_{i-1}$ on channel γ_{i-1} while $\mathcal{P}_{i+1}$ does not cooperate with $\mathcal{P}_{i+2}$ to consolidate the payment on channel γ_{i+1}. Next, $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$ close their channel γ_i such that the Enable transaction cannot be committed to the ledger. This allows $\mathcal{P}_{i+1}$ to reclaim coins from $\mathcal{P}_{i+2}$ using their shared Lock transaction. Effectively, $\mathcal{P}_i$ received the payment amount from $\mathcal{P}_{i-1}$ on γ_i through consolidation, while $\mathcal{P}_{i+1}$ did not forward the payment.

4 Protocol Overview

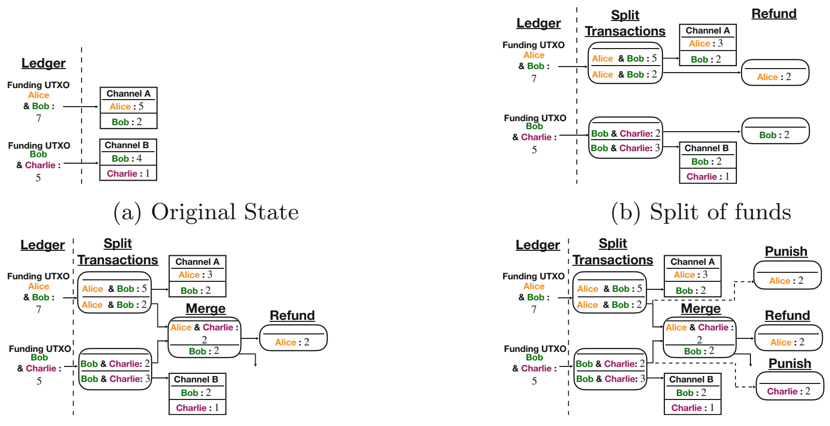
In the following, we define communication and adversarial models, before giving an overview of the protocol. Lastly we define the properties of our construction.

Communication Model. Communication between parties occurs in rounds. Any message sent within one round is available to the recipient at the beginning of the next round. The duration of any round has an upper limit.

Adversarial Model. We define an Adversary $\mathcal{A}$ consistent with related work [7,8,10]: At the beginning of protocol execution, the adversary can statically corrupt up to n of $n+1$ parties, receiving their internal state and having all communication to and from these parties be routed through the adversary. The adversary is malicious and can make any corrupted party deviate from the protocol. Moreover, within each communication round, the adversary can delay and re-order all messages sent.

We illustrate the life-cycle of the Payment Tree protocol for a payment of 2 coins from Alice to Charlie across two channels using Figs. 1 and 2. The protocol's approach is to take two channels, one between parties Alice and Bob, one between parties Bob and Charlie and construct a transaction tree that effectively creates a virtual channel [8] optimized for a one-time payment between Alice and Charlie. Our construction utilizes two approaches to perform updates to transaction trees atomically. On the one hand, we use these techniques to empower the intermediary Bob to ensure correctness of the protocol, while on the other hand, we incentivise Bob to actually do so by means of punishment. Our construction consists of multiple transaction tree updates. Updates are done using the *invalidation by timelock* technique, but for simplicity we leave the details to Sect. 6.

Constructing Transaction Trees Atomically. We observe that committing a transaction to the ledger requires that all of its ancestors are committed to the ledger beforehand. For a transaction to be able to be committed to the ledger it needs to contain all required witnesses, i.e. signatures. Therefore, (1) we can atomically create a transaction tree rooted in a transaction tr_{root} that is

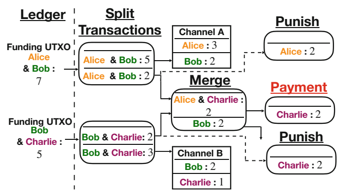


(a) Original State
(b) Split of funds
(c) Merge of trees
(d) Securing funds

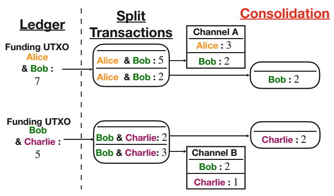
Fig. 1. Stepwise construction of a Payment Tree across two channels. Boxes with straight corners represent channel trees displaying their state. Boxes with round corners represent transactions displaying output UTXOs. Edges indicate which transactions spend the UTXO at their origin.

common ancestor to all other transactions. First, we add signatures to all transactions except tr_{root}. Afterwards, adding signatures to tr_{root} makes the whole transaction tree committable to the ledger at the same moment. (2) We assume two transactions, tr_0 and tr_1, that require signatures of Alice and Bob, as well as Bob and Charlie respectively. Bob can enforce that both transactions are created atomically by only providing his signature *after* he received signatures from Alice and Charlie. We note that techniques (1) and (2) can be used in tandem.

Payment Tree Construction. Figure 1 depicts construction of a Payment Tree between Alice, Bob and Charlie. Construction consists of three atomic transaction tree updates. We note that the balance distribution between the parties remains unchanged between the updates and no payment is executed. Alice and Bob as well as Bob and Charlie share a channel as depicted in Fig. 1a. (1) Then, as shown in Fig. 1b we update both trees by introducing a *Split* transaction that spends the channels' Funding UTXOs and creates two new Funding UTXOs each. One UTXO contains the payment amount and is funded by coins from Alice, who is payer, and Bob, who is intermediary, respectively. The other UTXO contains the remaining coins and is used as Funding UTXO to reopen both channels which can be used for further payments within the channels or further Payment Tree constructions. (2) Next as shown in Fig. 1c both separate transaction trees are combined using a Merge transaction. This transaction creates two UTXOs. One UTXO requires Bob's signature to be spent and contains his collateral. The other UTXO is a Funding UTXO requiring the signatures of Alice and Charlie and it contains Alice's payment to Charlie. At this point, the coins are given to Alice. (3) Lastly, as shown in Fig. 1d, before we can proceed with a payment,



(a) Payment



(b) Consolidation

Fig. 2. Payment and Consolidation using Payment Trees. Figure 2a modifies the Payment Tree to forward the funds in the Merge transaction's Funding UTXO to the payee. Figure 2b splits up the Payment Tree and distributes funds according to the Payment Tree's state in Fig. 2a.

the funds within the Merge transaction's Funding UTXOs need to be secured in case two parties, for example Bob and Charlie, collude to spend their Merge transaction's or Split transaction's input with a different transaction. This attack is similar to the Channel Closure attack described in Sect. 3 and would disable commitment of the Merge transaction. However, we observe that all UTXOs that can be spent for this attack require Bob's signature. Respectively, in this scenario we can uniquely identify Bob as malicious. In order to punish Bob and secure the funds of Alice and Charlie respectively we create *Punish* transactions. These transactions spend the same Funding UTXOs as the Merge transaction but have a timelock that is higher than that of the Merge transaction by at least Δ. Due to this, Bob can always avoid commitment of a Punish transaction by committing the Merge transaction to the ledger. However, in case Bob acted maliciously such that the Merge transaction cannot be committed to the ledger, Alice and Charlie can reclaim their coins from Bob through the Punish transactions.

Payment and Consolidation. Figure 2 depicts payment and consolidation using a Payment Tree. Assuming a fully constructed Payment Tree as shown in Fig. 1d a payment is executed by giving the coins within the Merge transaction's Funding UTXO to Charlie instead of Alice. As shown in Fig. 2a this changes the balance distribution represented by the transaction tree, reducing Alice's coins by 2 and adding those to Charlie's balance. A consolidation requires one atomic transaction tree update as shown in Fig. 2b. This update spends the UTXOs within the Merge transaction's inputs and gives the coins to Bob and Charlie respectively. Note that this step does not change the balance distribution between the parties. Bob needs to make sure that this update is done atomically s.t. he avoids commitment of a Punish transaction. At this point the transaction trees are separate and in control of each channels' members respectively. Both pair of parties can now perform a last transaction tree update that replaces the respective transaction tree with a channel as shown in Fig. 1a but that now represents the new balance distribution instead.

System Goals. In the following we define the desired properties of our protocol.

Theorem 1 (Balance Security). *Outside of performing the intended payment, the sum of a honest party's coins is not reduced by participation in the Payment Tree protocol.*

Theorem 2 (Liveness). *Eventually any honest party receives access to their coins through UTXOs spendable with a witness consisting of a signature corresponding to their verification key.*

5 Transactions

We use three types of transactions. Split transactions are used to split off coins from one channel, making them available to our construction in form of a Funding UTXO. Payout transactions take a Funding UTXO as input and pay the money to one of the two parties involved in it. Lastly, the Merge transaction is used to combine the Funding UTXOs that were split off two channels by taking them as input, paying out the intermediary's coins out as collateral and creating a Funding UTXO between the two remaining non-intermediary parties.

Split Transactions are of form $Tr_{\mathsf{split}} = (U_{\mathsf{in}}, U_{\mathsf{out}}, t)$ where $U_{\mathsf{in}} = \{\mathsf{ref}(f_\gamma)\}$ consist of one Funding UTXO provided by the channel-tree of γ, $U_{\mathsf{out}} = \{f_{\mathsf{change}}, f_{\mathsf{pay}}\}$ consists of two Funding UTXOs. It holds that $f_{\mathsf{change}}.b + f_{\mathsf{pay}}.b = f_\gamma$ and $f_{\mathsf{pay}}.b = b$. Moreover, $f_\gamma.\pi = f_{\mathsf{change}}.\pi = f_{\mathsf{pay}}.\pi$, i.e. all Funding UTXOs are shared between the same parties. The function call $\mathsf{SPLIT}(\gamma, b, t)$ creates a Split transaction as described above and returns f_{pay}. A function call to $\mathsf{UNSPLIT}(\gamma)$ consolidates the transaction into the channel by updating the channel's balance distribution with the split off balance. Additionally it sets up a channel between both parties by constructing a channel-tree with Funding UTXO f_{change} as root. *Split* transactions are used to take off b coins from each channel to be used for our construction. They are used to avoid that the existing channels are affected in case a corrupted intermediary misbehaves. Although we represent this by using a Split transactions as done with Virtual Channels and AMCU, it could be included similarly as conditional payments from HTLCs by placing a Funding UTXO instead of a HTLC contract.

Merge Transactions are of form $Tr_{\mathsf{merge}} = (U_{\mathsf{in}}, U_{\mathsf{out}}, t)$ where $U_{\mathsf{in}} = \{f_{\mathsf{pay},0}, f_{\mathsf{pay},1}\}$ and $U_{\mathsf{out}} = \{f_{\mathsf{pay}}, u_{collateral}\}$. The two Funding UTXOs that are provided as input $f_{\mathsf{pay},0}$ and $f_{\mathsf{pay},1}$ are shared between parties $\mathcal{P}_A$ and $\mathcal{P}_B$ as well as between parties $\mathcal{P}_B$ and $\mathcal{P}_C$ respectively. The newly created Funding UTXOs f_{pay} in the output is shared between parties $\mathcal{P}_A$ and $\mathcal{P}_C$. The other UTXO within the outputs is $u_{collateral}$ which pays out funds to $\mathcal{P}_B$. Lastly it holds that the coins in all UTXOs are equal, i.e. $f_{\mathsf{pay},0}.b = f_{\mathsf{pay},1}.b = f_{\mathsf{pay}}.b = u_{collateral}.b = b$. The function call $\mathsf{MERGE}(f_{\mathsf{pay},0}, f_{\mathsf{pay},1}, t)$ is a short-hand notation to construct a Merge transaction. We extend helper function $\mathsf{OUT\_UTXO}$ to accept a Merge transaction as input as well. In this case it returns UTXO f_{pay}. The helper function $\mathsf{IN\_UTXO}$ takes a Merge transaction as input and outputs the UTXOs that are used within its inputs, i.e. $f_{\mathsf{pay},0}, f_{\mathsf{pay},1}$. *Merge* transactions are used to combine transaction trees into one, essentially opening up a virtual channel between Alice and Charlie that can be used for a one-time payment.

Payout Transactions are of form $Tr_{\mathsf{payout}} = (U_{\mathsf{in}}, U_{\mathsf{out}}, t)$ where $U_{\mathsf{in}} = \{f\}$ is a Funding UTXO and $U_{\mathsf{out}} = \{u_{\mathsf{payout}}\}$. It holds that u_{payout} pays out funds to a party $\mathcal{P}$ and $f.b = u_{\mathsf{payout}}.b$. The function call $\mathsf{PAYOUT}(f, \mathcal{P}, t)$ constructs a Payout transaction as described above. We extend helper function $\mathsf{IN\_UTXO}$ to take a Payout transaction as input in which case it outputs the UTXO f. Payout transactions are used at several points within our construction to serve different roles as shown in Fig. 3. *Refund* transactions are used whenever Funding UTXOs are created. They are used to ensure that no funds are locked away within Funding UTXOs indefinitely even when any other party stops collaboration, which is essential to ensure the liveness property. *Punish* transactions are used to incentivise an intermediary to collaborate and ensure Merge transactions can be committed to the ledger. Without those, in case a Merge transaction is not committed to the ledger it could result in the loss of coins for Charlie in case the Refund transaction between Bob and Charlie is committed to the ledger instead and after the payment between Alice and Charlie has been performed. The *Payment* transaction is used to perform a change of the state, i.e. balance distribution, represented by the transaction tree, effectively performing a payment. Lastly, *Consolidation* transactions are used to deconstruct the transaction tree by applying the payment on both original transaction trees atomically. Without these, we cannot enforce the payment outside of committing the transaction tree to the ledger itself because of which the protocol would not fulfill the efficiency requirements for offchain protocols and thus not being classified as such. We note that the Refund and Punish transactions between Alice and Bob represent the same state s.t. the Punish transaction is redundant. However, for simplicity we opted to include both transactions making the construction symmetric. Whereas similarly the Consolidation and Punish transactions between Bob and Charlie do represent the same state in Fig. 3, it is not possible to remove any of the transactions in the case where fees are paid to Bob which would be included within the Consolidation but not the Punish transactions.

6 Our Payment Tree Construction

We describe the construction of a payment tree in respect to our running example. Let $\mathcal{P}_0, \mathcal{P}_1, \dots, \mathcal{P}_n, n \in \mathbb{N}$, be parties where parties $\mathcal{P}_{i-1}$ and $\mathcal{P}_i, i \in \{1, \dots, n\}$ control channel γ_i. The protocol performs a payment of $b \in \mathbb{N}$ coins from $\mathcal{P}_0$ to $\mathcal{P}_n$. The value $\tau \in \mathbb{N}$ represents the current time, whereas $\Delta \in \mathbb{N}$ is the maximum time it takes for a transactions to be included in the ledger after committing it. We illustrate our approach in Fig. 3 for a two-hop payment, i.e. for the case of $n = 2$. It is designed such that it can be extended to payment

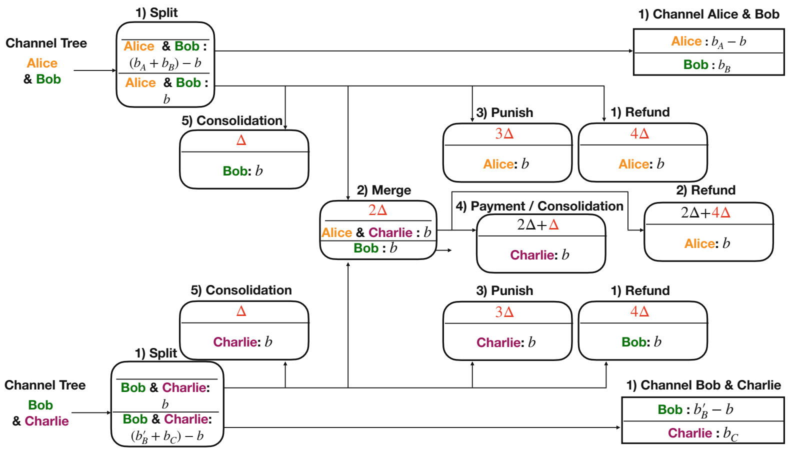


Fig. 3. Transaction tree of a payment of b coins across 2 hops. Beforehand, the respective balances are b_A and b_B for Alice and Bob, b'_B and b_C for Bob in Charlie within their channels. Transactions are boxes with round corners containing the UTXOs they create, whereas referenced UTXOs in inputs are indicated implicitly by arrows originating from the UTXO that is spent. Red numbers indicate timelocks. Numbers atop the transaction indicate order of construction whereas transactions with same numbers are constructed atomically. Channel trees are boxes with straight edges forming a black box. (Color figure online)

paths of arbitrary lengths. The construction is based on the overview given in Sect. 4. Numbers indicate the order in which transactions are created, whereas transactions with the same numbers are created atomically (Fig. 4).

The Payment Tree Protocol. The protocol for constructing a Payment Tree across a path of n channels is depicted in Algorithm 4. It makes use of Algorithm 1 that allows an intermediary to atomically create two transactions, Algorithm 2 that performs a construction step of the Payment Tree, and Algorithm 3 that performs a consolidation step of the Payment Tree.

Helper Functions. Function $\mathsf{SIGN}(Tr, P_S, P_R)$ is used to sign and exchange signatures of transactions. It takes a transaction Tr and two sets of parties P_S and P_R as input. Each party in P_S signs Tr and sends the signature to each party in P_R. This includes verification of signatures by the recipients. Function $\mathsf{PARTIES}$ takes a Funding UTXO as input and outputs a set containing the two parties of which a signature is required to spend the UTXO. Function $\mathsf{INTERMEDIARY}(f_0, f_1)$ takes two Funding UTXOs f_0, f_1 as input, if an intermediary exists, i.e. $|\mathsf{PARTIES}(f_0) \cap \mathsf{PARTIES}(f_1)| = 1$, then it returns the intermediary $\mathcal{P} \in \mathsf{PARTIES}(f_0) \cap \mathsf{PARTIES}(f_1)$. Otherwise it returns $\perp$. Function $\mathsf{COUNTERPARY}(f, \mathcal{P})$ takes a Funding UTXO and a party as input, if $\mathcal{P} \in \mathsf{PARTIES}(f)$, then it returns its counterparty $\mathcal{P}_C \in (\mathsf{PARTIES}(f)) \setminus \{\mathcal{P}\}$.

Algorithm 1 Atomically signing two Payout transaction

```
1: function ATOMIC_SIGN($Tr_0, Tr_1$)
Require: $Tr_0, Tr_1$ are Payout transactions between three parties.
2:     $f_0, f_1 \leftarrow \mathsf{FUTXO}(Tr_0), \mathsf{FUTXO}(Tr_1)$
3:     $\mathcal{P}_I \leftarrow \mathsf{INTERMEDIARY}(f_0, f_1)$
4:     $\mathcal{P}_A, \mathcal{P}_B \leftarrow \mathsf{COUNTERPARY}(f_0, \mathcal{P}_I), \mathsf{COUNTERPARY}(f_1, \mathcal{P}_I)$
5:     $\mathsf{SIGN}(Tr_0, \{\mathcal{P}_A\}, \{\mathcal{P}_I\})$, $\mathsf{SIGN}(Tr_1, \{\mathcal{P}_B\}, \{\mathcal{P}_I\})$
6:     $\mathsf{SIGN}(Tr_0, \{\mathcal{P}_I\}, \{\mathcal{P}_A\})$, $\mathsf{SIGN}(Tr_1, \{\mathcal{P}_I\}, \{\mathcal{P}_B\})$
7: end function
```

Fig. 4. Algorithm that takes two Payout transactions as input and allows the intermediary party to enforce that either both or no transactions are fully signed.

Atomic Signatures. We assume a setting with two channels between three parties. Protocol *ATOMIC_SIGN* is shown in Algorithm 1. It enables the intermediary party to enforce that two transactions – one on each channel – are created atomically. This is done by having the intermediary party provide signatures to both transactions only after they received all signatures from its counterparties (Fig. 5).

Merging Channels. Protocol *MERGE* as shown in Algorithm 2 takes two Funding UTXOs f_0, f_1, an amount of coins b and a time t as input where f_0 is shared between parties $\mathcal{P}_A$ and $\mathcal{P}_I$, f_1 is shared between parties $\mathcal{P}_I$ and $\mathcal{P}_B$ and it holds that $f_0.b = f_1.b = b$. It creates a Merge transactions with timelock $t_m = t + 2\Delta$ spending both Funding UTXOs, paying out b coins to $\mathcal{P}_I$ and containing a Funding UTXO holding b coins, which are paid out to $\mathcal{P}_A$ after time $t_m + 4\Delta$ by means of a Payout transaction. This transaction tree is created atomically as its root, which is the Merge transaction, is signed last. Only after each party holds a fully signed instance of the Merge transaction, two Punish transactions spending f_0 and f_1 and paying out b coins to $\mathcal{P}_A$ and $\mathcal{P}_B$ respectively are created atomically using *ATOMIC_SIGN*. These have timelocks equal to $t + 3\Delta$. Note that the creation of the Merge transaction must not re-distribute funds, i.e. the funds in f_0 are paid by $\mathcal{P}_A$ and the funds in f_1 are paid by $\mathcal{P}_I$. The Punish transactions are used to secure the funds within the Merge transaction by paying out funds to $\mathcal{P}_A$ and $\mathcal{P}_B$, if the Merge transaction cannot be committed to the ledger. Timelocks are selected to perform transformations on the existing transaction through the invalidation by timelock technique and also to allow the construction to be performed iteratively. Timelock t_m is selected s.t. a Consolidation transaction can be placed with timelock $t + \Delta$ during the protocol's consolidation phase. Timelocks of the Punish transactions are selected s.t. they are invalidated by the Merge transaction *conditionally*, i.e. only if the Merge transaction can be committed to the ledger, the Punish transactions are invalid. The Payout transaction acts as a Refund transaction for the new Merge transaction. Respectively we assign it

Algorithm 2 Construction Step of a Payment Tree

```
1: function MERGE($f_0, f_1, b, t$)
2:     $\mathcal{P}_I \leftarrow \mathsf{INTERMEDIARY}(f_0, f_1)$
3:     $\mathcal{P}_A, \mathcal{P}_B \leftarrow \mathsf{COUNTERPARY}(f_0, \mathcal{P}_I), \mathsf{COUNTERPARY}(f_1, \mathcal{P}_I)$
4:     $Tr_{mrg} \leftarrow \mathsf{MERGE}(f_0, f_1, t + 2\Delta)$
5:     $Tr_{refund} \leftarrow \mathsf{PAYOUT}(\mathsf{OUT\_UTXO}(Tr_{mrg}), \mathcal{P}_A, t + 6\Delta)$
6:     $Tr_{punish,A} \leftarrow \mathsf{PAYOUT}(f_0, \mathcal{P}_A, t + 3\Delta)$
7:     $Tr_{punish,B} \leftarrow \mathsf{PAYOUT}(f_1, \mathcal{P}_C, t + 3\Delta)$
8:     $\mathsf{SIGN}(Tr_{refund}, \{\mathcal{P}_A, \mathcal{P}_B\}, \{\mathcal{P}_A, \mathcal{P}_B\})$
9:     $\mathsf{SIGN}(Tr_{mrg}, \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_I\}, \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_I\})$
10:    ATOMIC_SIGN($Tr_{punish,A}, Tr_{punish,B}$) return $Tr_{mrg}$
11: end function
```

Fig. 5. Creation of a Funding UTXO between two counterparties. The intermediary can enforce atomic construction while Punish transactions provide incentive.

a timelock of $t_m + 4\Delta$ such that Consolidation, Merge and Punish transactions can be placed with timelocks $t_m + \Delta$, $t_m + 2\Delta$ and $t_m + 3\Delta$ respectively. Note that if the Merge transaction is on top of the Payment Tree s.t. it is not used for further channel merges, the Refund transaction's timelock can be reduced to $t_m + 2\Delta$. Lastly, if a transaction spends another transaction, its timelock needs to be larger by at least Δ to ensure that all transactions can be committed to the ledger as soon as their timelocks expire (Fig. 6).

Consolidation. Algorithm 3 takes a Merge transaction as input, invalidates it by creating two Payout transactions atomically using the ATOMIC_SIGN protocol that spend the Merge transaction's inputs. Both consolidation transactions perform a payment by giving the funds to the payee. Note that the protocol can be adjusted to cancel a payment by refunding the funds to the payer instead (Fig. 7).

Payment Trees. Algorithm 4 performs a payment from $\mathcal{P}_0$ to $\mathcal{P}_n$ by iteratively merging Funding UTXOs, s.t. the Merge transactions form the nodes of a balanced binary tree as illustrated in Fig. 8. The algorithm takes the following inputs: (1) The payment path $\gamma_1, \ldots, \gamma_n$, (2) the payment amount b, and (3) time t_{min}. The value t_{min} is negotiated by the parties and represents the maximum amount of time the parties have to execute the protocol. The dispute protocol starts if the protocol is not concluded until t_{min}. Note that even existing methods as HTLCs have to account for t_{min}.

In the following we refer to a certain depth within this binary tree as *level*, beginning with Split transactions on level 0. The algorithm maintains lists of Funding UTXOs F_UTXO_i for each level $i \geq 0$ of the binary tree, as well as lists of Merge transactions MRG_j for each level $j \geq 1$ of the binary tree. The

Algorithm 3 Deconstructing Step of a Payment Tree

```
1: function CONSOLIDATE($Tr_{mrg}$)
2:     $f_0, f_1 \leftarrow \mathsf{IN\_UTXO}(Tr_{mrg})$
3:     $\mathcal{P}_I \leftarrow \mathsf{INTERMEDIARY}(f_0, f_1)$
4:     $\mathcal{P}_A, \mathcal{P}_B \leftarrow \mathsf{COUNTERPARY}(f_0, \mathcal{P}_I), \mathsf{COUNTERPARY}(f_1, \mathcal{P}_I)$
5:     $Tr_A \leftarrow \mathsf{PAYOUT}(f_0, \mathcal{P}_B, t + \Delta)$
6:     $Tr_B \leftarrow \mathsf{PAYOUT}(f_1, \mathcal{P}_C, t + \Delta)$
7:     ATOMIC_SIGN($Tr_A, Tr_B$)
8: end function
```

Fig. 6. Invalidating a Merge transactions and atomically updating the state on the two original Funding UTXOs.

algorithm proceeds as follows. Add a Funding UTXO from each Split transaction to F_UTXO_0 in order (4–7) and create the Payment Tree by iterative use of the *MERGE* protocol level-by-level (8–18). The Merge transactions and Funding UTXOs created on level j are added to lists MRG_j and F_UTXO_j respectively and in order (12–13). Note that if there is an uneven amount of Funding UTXOs within a level, we leave the odd one to be used in the level above instead (15–17). The payment is executed after construction is concluded (19). Afterwards the payment tree is deconstructed in reverse order by executing the *CONSOLIDATE* protocol on each Merge transaction (20–24). Lastly the Split transactions are removed and consolidation within all original channels concludes (25–27).

Dispute. This protocol is executed at time t_{min} if the payment tree protocol has not come to conclusion in an orderly manner. Every honest party submits their transactions to the ledger as soon as their respective timelocks expire. This will result in commitment of the payment tree onto the ledger where transactions are committed in order of their priority. If a Merge transaction cannot be committed to the ledger, refunds and payments are done via Punish transactions.

Fees. Fees can be paid by the payer $\mathcal{P}_0$ and payee $\mathcal{P}_n$ or either of them alone to the intermediaries to compensate for their invested collateral. Our approach to handling fees is similar to the approach used for HTLCs, however, adapted to the binary tree structure of Payment Trees. Any party acting as intermediary when creating a Merge transaction receives cumulative fees from the other two parties participating in the Merge transaction's construction. The cumulative fee paid to the intermediary is composed of two parts. For one, it contains the fees paid to the intermediary themselves, and for another, it contains coins the party has to forward to the parties who act as intermediaries of Merge transactions on the lower levels of the Payment Tree. For simplicity, in the following we assume that the path's length is a power of 2, i.e. $n = 2^i, i \in \mathbb{N}$, the paid fees f are equal for

Algorithm 4 Payment Tree Construction

```
1: function PaymentTree($\gamma_1, \gamma_2, \ldots, \gamma_n$, $b$, $t_{min}$)
2:     $F\_UTXO_i \leftarrow [], 0 \leq i \leq \lceil (\log n) - 1 \rceil$
3:     $MRG_i \leftarrow [], 1 \leq i \leq \lceil \log n \rceil$
4:     for $1 \leq i \leq n$ do
5:         $f_i \leftarrow \mathsf{SPLIT}(\gamma_i, b, t_{min})$
6:         Append $f_i$ to $F\_UTXO_0$
7:     end for
8:     for $i = 0$ until $i = \lceil (\log n - 1) \rceil$ do
9:         for $0 \leq j \leq \lfloor |F\_UTXO_i|/2 \rfloor$ do
10:            Retrieve $f_{2j}$, $f_{2j+1}$ from $F\_UTXO_i$
11:            $Tr_{mrg,j} \leftarrow \text{MERGE}(f_{2j}, f_{2j+1}, b, t_{min} + 2i\Delta)$
12:            Append $\mathsf{OUT\_UTXO}(Tr_{mrg,j})$ to $F\_UTXO_{i+1}$
13:            Append $Tr_{mrg,j}$ to $MRG_{i+1}$
14:        end for
15:        if $|F\_UTXO_i| \mod 2 = 1$ then
16:            Remove last entry of $F\_UTXO_i$ and append to $F\_UTXO_{i+1}$
17:        end if
18:    end for
19:    $Tr_{Payment} \leftarrow \mathsf{PAYOUT}(\mathsf{OUT\_UTXO}(MRG_{\lceil \log n \rceil}[0]), \mathcal{P}_n, t_{min} + 2\Delta \log n + \Delta)$
20:    for $i = \lceil \log n \rceil$ until $i = 1$ do
21:        for $Tr_{mrg}$ in $MRG_i$ do
22:            CONSOLIDATE($Tr_{mrg}$)
23:        end for
24:    end for
25:    for $1 \leq i \leq n$ do
26:        $\mathsf{UNSPLIT}(\gamma_i)$
27:    end for
28: end function
```

Fig. 7. The full Payment Tree protocol from construction to consolidation.

each intermediary and all fees are shared between payer $\mathcal{P}_0$ and payee $\mathcal{P}_n$ equally. Then, a party that acts as intermediary of level i of the Payment Tree receives $f_i = f + 2\frac{f_{i-1}}{2} = f + f_{i-1}$ coins, where $f_1 = f$. The fee f_i is paid equally by the other two parties involved in the Merge transaction's construction. Payment of fees happens within Merge transactions by adding a fee to the collateral the intermediary receives. However, this raises the challenge that we have to ensure that all transactions receive sufficient funding: The coins within a transaction's inputs have to cover all coins within their outputs. Moreover, to ensure that the consolidation step can be performed, the collateral of the intermediary within a Merge transaction has to be at least as high as the coins within the Merge transaction's Funding UTXO [8]. Therefore, when performing the merge step on level i every party has to have an additional balance of $f_{i,cum} = \sum_{j=i}^{h} f_j$, whereas the collateral of a Merge transaction's intermediary equals $b + f_{i,cum} + f_i$ where f_i is paid equally from balances brought by the other two parties.

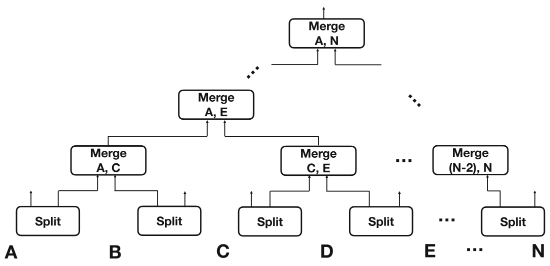


Fig. 8. Payment tree in the shape of a balanced binary tree.

7 Collateral Efficiency and Security Analysis

In this section we discuss properties of the Payment Tree construction.

Efficiency. Figure 9 depicts the efficiency properties of Payment Trees, comparing it to existing approaches. We compare two metrics: (1) The collateral, and (2) the number of transactions that have to be committed to the ledger in case of dispute. We do this for individual parties, as well as for the whole payment.

Commitment of each Merge transaction unlocks the collateral of one party. To commit a Merge transaction located on level i of the payment tree it needs to commit i transactions beforehand, i.e. $i-1$ Merge transaction as well as a Split transaction. This will happen at time $2\Delta i$. As the height of the Payment tree is limited by $\lceil \log n \rceil$ it follows that any party invests $b2\Delta i \in \mathcal{O}(b\Delta \log n)$ collateral and has to commit $i+1 \in \mathcal{O}(\log n)$ transactions. Regarding the total payment, we observe that there are $\frac{n}{2^i}$ Merge transactions on level i of the payment tree. It follows that the total collateral equals the sum $\sum_{i=1}^{\lceil \log n \rceil} b2\Delta i \frac{n}{2^i} = b2\Delta n \sum_{i=1}^{\lceil \log n \rceil} \frac{i}{2^i}$. As $\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$ and each part of the sum is positive, it follows that the total collateral $b2\Delta n \sum_{i=1}^{\lceil \log n \rceil} \frac{i}{2^i} < 4b\Delta n \in \mathcal{O}(b\Delta n)$ is linear in the length of the payment path n. The number of transactions can be computed in a similar fashion, however, an intuitive approach is to recall that the transactions form a balanced binary tree of height $1 + \lceil \log n \rceil$ which has at most $2^{1+\lceil \log n \rceil} \leq 2n \in \mathcal{O}(n)$ nodes. Although the collateral any individual party has to invest is logarithmic, therefore higher than Sprites but lower than HTLCs, the total collateral incurred over the whole payment is linear in the path's length. This is comparable to the performance of Sprites and is by a factor of n lower than the total collateral of HTLCs. A trade-off of Payment Trees is that an individual party might have to commit up to $\mathcal{O}(\log n)$ many transactions. Nevertheless the total number of transactions over the whole payment is comparable to both, HTLCs and Sprites. Payment Trees provide a performance comparable to Sprites without requiring a ledger with smart contract capability.

Method	pp Collateral	pp Tr.	Total Collateral	Total Tr.	Smart Contracts
HTLC [15,2]	$\mathcal{O}(b\Delta n)$	$\mathcal{O}(1)$	$\mathcal{O}(b\Delta n^2)$	$\mathcal{O}(n)$	No
Sprites [10]	$\mathcal{O}(b(n+\Delta))$	$\mathcal{O}(1)$	$\mathcal{O}(b(n+\Delta)n)$	$\mathcal{O}(n)$	Yes
Payment Tree	$\mathcal{O}(b\Delta\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(b\Delta n)$	$\mathcal{O}(n)$	No

Fig. 9. Comparison of the performance of Payment Trees across the whole payment (Total) and individually per party (pp).

Denial of Service Attacks. The Payment Tree protocol mitigates existing attacks such as the congestion and lockdown attacks [11,14] on HTLCs that aim to lock a channel's coins within unfulfilled HTLCs. This is done by reducing the total and individual collateral of payments. While a large scale DoS attack on multiple channels is difficult as the total collateral of Payment Trees is linear in the payment path's length, a specific intermediary can be targeted to act a intermediary on the highest level of the Payment Tree to pay a logarithmic collateral. Another aspect of the Lockdown attack is that a channel is blocked by saturating the number of HTLCs applicable to a channel which is limited by the maximum size of a transaction. The Payment Trees protocol mitigates this by using Split transactions. Each pending payment requires the construction of a Split transaction. This prevents that there is any transaction that increases in size depending on the number of pending transactions. However, a tradeoff to using Payment Trees is the increased number of transactions that would need to be committed to the ledger in case of a dispute.

Wormhole Attacks. The Payment Tree protocol pays coins to intermediaries of a Merge transaction and they include fees for all intermediaries on lower levels of respective sub-trees. An attack similar to the wormhole attack can be performed by a corrupted intermediary when creating a Merge transaction by replacing the Merge transaction's inputs with UTXOs they control. Doing this they could take all fees that were intended to be forwarded to other parties while preventing them to participate in the protocol. In contrast to the wormhole attack on HTLCs a wormhole-like attack on the Payment Trees protocol requires making changes to the transaction tree which in-turn can be detected and prevented. We assume that either $\mathcal{P}_0$ and $\mathcal{P}_n$ are honest. Otherwise, if both are corrupted the attack would only redistribute coins between corrupted parties resulting in no net gain to the adversary. During creation of the Payment Tree all intermediaries send their view of the protocol to $\mathcal{P}_0$ and $\mathcal{P}_n$, i.e. the Merge transactions they are involved in. Having this information $\mathcal{P}_0$ and $\mathcal{P}_n$ can verify correctness of the construction and abort the payment in the negative case.

Security Proofs. We provide proof of Theorems 1 and 2 in the full version of the paper.

8 Conclusion

Payment Trees provide competitive performance to state-of-the-art approaches as Sprites, while having fewer restrictions to its employability by not requiring smart contract capability. Thus providing the first secure alternative to HTLCs for the Lightning Network.

References

1. Raiden network. Accessed 03 Sept 2018
2. Bowe, S., Hopwood, D.: Hashed time-locked contract transactions (2017). https://github.com/bitcoin/bips/blob/master/bip-0199.mediawiki. Accessed 29 Aug 2020
3. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 61–85. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70936-7_4
4. Decker, C., Wattenhofer, R.: A fast and scalable payment network with bitcoin duplex micropayment channels. In: Pelc, A., Schwarzmann, A.A. (eds.) SSS 2015. LNCS, vol. 9212, pp. 3–18. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21741-3_1
5. Dziembowski, S., Eckey, L., Faust, S., Malinowski, D.: Perun: Virtual payment hubs over cryptocurrencies. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 106–123. IEEE (2019)
6. Dziembowski, S., Faust, S., Hostáková, K.: General state channel networks. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 949–966. ACM (2018)
7. Egger, C., Moreno-Sanchez, P., Maffei, M.: Atomic multi-channel updates with constant collateral in bitcoin-compatible payment-channel networks. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019, pp. 801–815. ACM Press, November 2019. https://doi.org/10.1145/3319535.3345666
8. Jourenko, M., Larangeira, M., Tanaka, K.: Lightweight virtual payment channels. Cryptology ePrint Archive, Report 2020/998 (2020). https://eprint.iacr.org/2020/998
9. Malavolta, G., Moreno-Sanchez, P., Schneidewind, C., Kate, A., Maffei, M.: Anonymous multi-hop locks for blockchain scalability and interoperability. In: Network and Distributed Systems Security Symposium (2019)
10. Miller, A., Bentov, I., Bakshi, S., Kumaresan, R., McCorry, P.: Sprites and state channels: payment networks that go faster than lightning. In: Goldberg, I., Moore, T. (eds.) FC 2019. LNCS, vol. 11598, pp. 508–526. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32101-7_30
11. Mizrahi, A., Zohar, A.: Congestion attacks in payment channel networks. arXiv preprint arXiv:2002.06564 (2020)
12. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
13. PDecker, C., Russel, R., Osuntokun, O.: Eltoo: a simple layer2 protocol for bitcoin (2017). https://blockstream.com/eltoo.pdf
14. Pérez-Solà, C., Ranchal-Pedrosa, A., Herrera-Joancomartí, J., Navarro-Arribas, G., Garcia-Alfaro, J.: LockDown: balance availability attack against lightning network channels. In: Bonneau, J., Heninger, N. (eds.) FC 2020. LNCS, vol. 12059, pp. 245–263. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51280-4_14
15. Poon, J., Dryja, T.: The bitcoin lightning network: scalable off-chain instant payments (2016). https://lightning.network/lightning-network-paper.pdf