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-to Meguro-ku Oookayama 2-12-1 W8-55, Japan.
{jourenko.m.ab@m, mario@c, keisuke@is}.titech.ac.jp
[2] Input Output Hong Kong.
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 [10] have enjoyed popularity and received interest from the research community and prac-

* This work was supported by the Input Output Cryptocurrency Collaborative Research Chair funded by IOHK, JST CREST JPMJCR14D6, JST OPERA, JSPS KAKENHI 16H01705, 17H01695.

titioners. Consensus protocols allow these ledgers to be operated by mutually 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 [4,11,12] 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 [12] and Raiden [1]. This is done using techniques, such as HTLC [12,2], 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* [9] 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. For example, an attacker might operate a channel to collect fees by forwarding payments. However, payments might be routed through competing channels instead. In order to sabotage the competing option, 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. During this time 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 very 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 they do have enough coins. In the case of the Lightning network, these types of denial-of-service attacks can lock up to 5 coins for up to 2 weeks. [3]

For HTLC the total collateral locked over a whole payment path is $\mathcal{O}(n^2b\Delta)$ and therefore quadratic in the payment paths length. Sprites [9] 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 respec-

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

tively, 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 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 in [7]. This gap results in a vulnerability that can be exploited by a malicious adversary to steal funds from honest parties.

This work addresses this gap and proposes a protocol that improves in the individual and total collateral values.

Related Work. Payment channels [4,11,12] themselves allow only for offchain payments between two parties. Offchain protocols such as HTLCs [12,2] and Sprites [9] allow to perform payments across paths of channels allowing for the implementation of PCNs. Prominent examples are the Lightning Network [12] 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 $\mathcal{P}_0, \ldots, \mathcal{P}_n$.

The most prominent technique is based on HTLCs [12,2], 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 [9] 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. Details are shown in Section 3.

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 Section 2. We give an outline of the Channel Closure attack in Section 3. Next we introduce Payment Trees in Section 4 followed by efficiency and security analysis in Section 5. We conclude in Section 6.

2 Background

Notation. Throughout this work we make frequent use of tuples. We use shorthand notations to reference entries 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, π) 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 UTXO referenced within U_{in} are removed from S_{utxo} and all UTXO 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.

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. The most common script included in UTXOs specifies its owner by requiring a signature of the transaction that spends the UTXO with the recipient's verification key. Additionally we make use of scripts specifying a 2-out-of-2 multisignature with verification keys of two parties $\mathcal{P}$ and $\mathcal{P}'$. This enforces that the respective UTXO can only be spent with the consent of both $\mathcal{P}$ and $\mathcal{P}'$ effectively creating a shared wallet between both parties. In the remainder of this work UTXOs requiring 2-out-of-2 multisignatures are termed Funding 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.

Payment Channel Networks. Let $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n, n \in \mathbb{N}$ be parties where parties $\mathcal{P}_{i-1}$ and $\mathcal{P}_i, i \in \{1, \ldots, n\}$ control channel γ_i. Individual channels can be connected to form a Payment Channel Network (PCN) by means of an offchain protocol that performs a payment 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 $\gamma_1, \ldots, \gamma_n$ from $\mathcal{P}_0$ to $\mathcal{P}_n$. The *collateral* investment of party $\mathcal{P}_j, 1 \leq j \leq n-1$ equals bT_j. Such a payment is performed by (1) extending each channel-tree with a (conditional) payment, (2) triggering all payments atomically, making them enforceable on the ledger and (3) *consolidating* the payment within each individual channel, such that they can be enforced without the previously created channel-tree extensions. Note that the *consolidation* step is necessary. Otherwise the channel-tree would grow depending on the number of payments performed which effectively would only delay, not avoid commitment of transactions to the ledger.

Hash Time Locked Contract can be used to perform payments within a PCN as a offchain protocol. Let $\mathcal{H}$ be a cryptographic hash function. A payment of b coins from $\mathcal{P}_0$ and $\mathcal{P}_n$ is performed by extending each channel γ_i on the path $\gamma_1, \ldots, \gamma_n$ with a HTLC performing a conditional payment from $\mathcal{P}_{i-1}$ to $\mathcal{P}_i$: If $\mathcal{P}_i$ reveals a secret x s.t. $y = \mathcal{H}(x)$ they can claim b coins, otherwise after expiration of time T_i $\mathcal{P}_{i-1}$ can reclaim the b coins they paid into the conditional payment contract. The payment is executed by having each party, starting with $\mathcal{P}_n$, reveal secret x to their predecessor, proving that they may claim the payment by committing the transaction tree of the channel to the ledger. Both parties can *consolidate* the payment by updating the channel-tree's state to reflect that payment instead. As soon as a party learns x they can forward it to their predecessor to reclaim the funds they forwarded to their successor s.t. the payment is performed atomically in all channels along the payment path. Timelocks T_i have to be picked such that, if $\mathcal{P}_i$ does not reveal x until the very last moment, $\mathcal{P}_{i-1}$ has to have enough time to claim the coins from their predecessor. Therefore it is required that $T_i \geq T_{t+1} + \Delta$. It follows that a party's collateral is $\mathcal{O}(nb\Delta)$ whereas the total collateral of all parties is $\sum_{i=0}^{n-1} b(n-i)\Delta = b\Delta\frac{n(n-1)}{2} \in \mathcal{O}(n^2 b\Delta)$.

Lightweight Virtual Payment Channel. 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 $\mathcal{P}_B$. This is done by splitting off coins from the original channels γ_A and γ_B. A Merge transaction spends the coins that were split off and creates a Funding UTXO for the new channel. Party $\mathcal{P}_I$ has to lock away the same amount of coins as both $\mathcal{P}_A$ and $\mathcal{P}_B$ in form of a collateral. The protocol allows $\mathcal{P}_I$ to enforce setup and

tear-down of the virtual channel atomically, while losing their collateral if they fail to do so. This, on the one hand, punishes $\mathcal{P}_I$ if they act maliciously and on the other hand secures funds for $\mathcal{P}_A$ and $\mathcal{P}_B$. The construction can be applied iteratively to create payment channel between parties that lie apart multiple hops within the underlying PCN.

However, using virtual channels is unpractical for individual payments. For one, $\mathcal{P}_I$ needs to lockup their funds during the full duration of the virtual channel which results in a long locktime and therefore a high collateral. Moreover, $\mathcal{P}_B$ and γ_I need to invest the same amount of funds into channels γ_B and γ_A respectively as $\mathcal{P}_A$. Therefore, not all payments that are possible with HTLCs can be performed with virtual channels: That is every party $\mathcal{P}_i$ needs to have a balance of b coins in both γ_i and γ_{i+1}, whereas for HTLCs they only need b coins in channel γ_{i+1}.

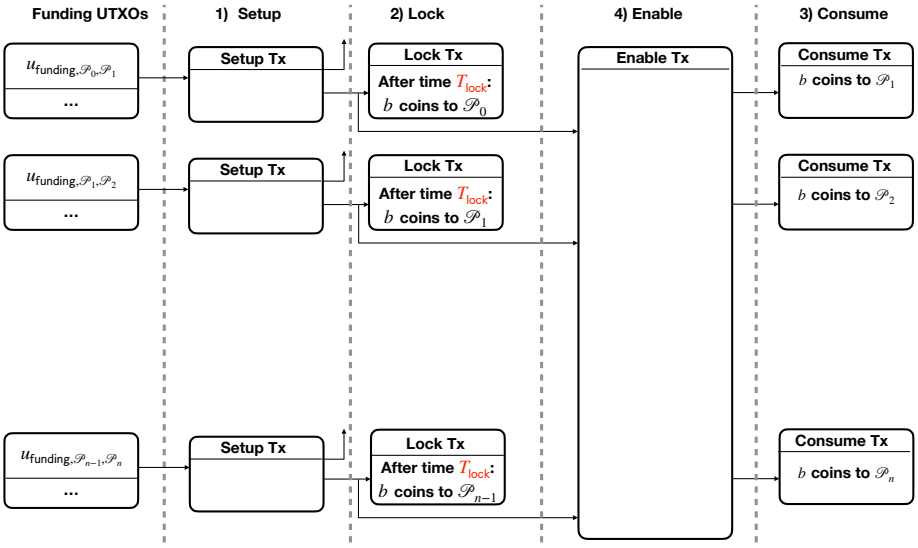

Fig. 1: Informal illustration of AMCU as a transaction-tree.

AMCU. Although the Sprites protocol reduces a party's collateral to $\mathcal{O}(b(n + \Delta))$, AMCU is the first proposal to reduce the collateral for UTXO based ledger that do not use smart contracts.

To perform an offchain payment AMCU operates in four phases in which the transaction tree shown in Figure 1 is created. 1) In a Setup phase b coins from $\mathcal{P}_{i-1}$'s balance are split up from the channel using the Setup transaction. 2) In the Lock phase the Lock transaction is created which spends $\mathcal{P}_{i-1}$'s b coins from the Setup transaction and pays out all coins back to $\mathcal{P}_{i-1}$ after expiration of time T_{lock}. 3) In the Consume phase the parties create a Consume transaction

paying the b coins to $\mathcal{P}_i$, however, instead of spending the Setup transaction it spends a not-as-of-yet created Enable transaction. 4) In the Finalize phase, first a Disable transaction is created that spends the Enable transaction after expiration of time T_{lock} and returns the b coins to $\mathcal{P}_{i-1}$ in the same manner as the Lock transaction does. Lastly the Enable transaction is created spending b coins of all Setup transactions on the payment path and creating the UTXOs that are spent by each channel's Consume and Disable transactions.

AMCU achieves atomic payment across the whole payment path by creating the Consume transactions to have the Enable transaction as common ancestor. As soon as it is signed, all Consume transactions can be committed to the ledger, thus rendering each payment on the payment path enforceable. However, this approach is impractical. The size of the Enable transaction grows as the number of its inputs and outputs increases and therefore its size grows linearly with the payment path's length n. Payments across long paths cannot be performed if the size of the Enable transaction exceeds the limits of a transaction's size.

The security of the AMCU protocol is attempted to be proven within Canettis UC Framework [3] by presenting a simulator that shows that the AMCU protocol realizes an ideal functionality PCN^+. However, while its updateState function, that is used to perform payments across a payment path, concludes with a *consolidation* step that atomically applies the payment on each individual channel within the payment path, this step is skipped within the AMCU protocol and not addressed by the simulator. Exactly this gap between ideal functionality and protocol is a vulnerability that allows a malicious adversary to have corrupted parties potentially steal funds from honest parties. We introduce the *Channel Closure Attack* formally in Section 3, in which a pair of intermediate parties within a payment path can steal funds from honest parties executing the AMCU protocol.

3 The Channel Closure Attack on AMCU

In the following let $\mathcal{P}_0, \ldots, \mathcal{P}_n$ be parties where parties $\mathcal{P}_{i-1}$ and $\mathcal{P}_i, i \in \{1, \ldots, n\}$ control channel γ_i. Let S_i be the setup transaction and L_i be the Lock transaction for channel γ_i respectively. The parties perform a payment of $b \in \mathbb{N}$ coins over the payment channel path $\gamma_1, \ldots, \gamma_n$ using the AMCU protocol. If the adversary can influence the order of channel consolidation then the attack can be performed with $n \geq 2$ where at most $n-1$ channels are consolidated atomically. Otherwise, we require $n \geq 3$ where at most $n-2$ channels are consolidated atomically.

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^+, that models AMCU, correctly performs this consolidation step, the AMCU protocol itself does not. 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.

Informally, an adversary can attack AMCU by corrupting two parties $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$ that share channel γ_i along a payment path. First, parties cooperate in execution of the protocol right until after creation of the Enable transaction at which point the protocol concludes. We observe that if the protocol is not followed up by a consolidation step as in the ideal functionality PCN^+, $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$ can close their channel γ_i maliciously, e.g. by double-spending the UTXO used as input into their Setup transaction. This prohibits commitment of the Setup transaction to the ledger and, as it is the ancestor of the Enable transaction which in-turn is common ancestor of all Consume transactions, no Consume transaction can be committed to the ledger, effectively reverting the payment. After the execution of the protocol, no party can enforce the payment by committing the Consume transactions. Performing the payment requires a final consolidation step, as defined in functionality PCN^+, allowing any party to enforce the payment on their ledger through the channels they participate in.

It is essential that the consolidation step is done atomically on all channels within a payment path, as otherwise this could lead to honest parties losing funds. However, this step is non-trivial as performing a state transition on multiple channels atomically is the very problem statement HTLCs, Sprites and AMCU approach to solve. In the following we present the *Channel Closure Attack* that allows a malicious adversary to have corrupted parties steal funds from honest parties as long as at most $n-2$ out of n channels are consolidated atomically.

The Adversary. The adversary is created according to AMCU's adversarial model. At beginning of the protocol, the adversary can corrupt $n-1$ parties s.t. it receives the party's internal state and all subsequent incoming and outgoing communication is routed through them instead. This corruption is static and the adversary cannot switch corrupted parties or corrupt any additional parties during execution of the protocol. The adversary is malicious and can deviate from the protocol arbitrarily, however, it is computationally polynomially bounded.

The adversary succeeds if the set of corrupted parties holds strictly more funds compared to when all Consume transactions are committed to the ledger.

The Approach. The corrupted parties steal coins by, first, executing the protocol correctly until the consolidation phase. They pick a party $\mathcal{P}_i, i \in \{1, \ldots, n\}$ where channel γ_i is consolidated before γ_{i+1}. After they receive coins through consolidation of γ_i, two parties γ_{j-1} and γ_j close channel j such that $\mathcal{P}_i$ has their money returned through the Lock transaction L_{i+1} instead of forwarding the coins. There are a few edge cases: (1) If $i = j + 1$ then channel γ_{i+1} is controlled by the corrupted parties, so we require γ_i is consolidated before γ_{i+2} instead. (2) If $i = n$ then $\mathcal{P}_i$ is already the payment's recipient. In this case, we require $\mathcal{P}_0$ to be corrupted as well, such that $\mathcal{P}_n$ receives their funds before $\mathcal{P}_0$ pays them out.

Channel Closure Attack. The adversary picks $i, j \in \{1, \ldots, n\}, i \neq j$ such that following conditions hold. (1) If $i \neq j - 1$, then γ_i is consolidated before $\gamma_{(i+1) \mod n}$, otherwise γ_i is consolidated before $\gamma_{(i+2) \mod n}$. (2) If $\gamma_{(j+1) \mod n}$ is consolidated before $\gamma_{(i+1) \mod n}$ then $\gamma_{(j-1) \mod n}$ is consolidated before the channel $\gamma_{(i+1) \mod n}$. The adversary corrupts $\mathcal{P}_i$, $\mathcal{P}_{j-1}$ and $\mathcal{P}_j$. If $i = n$ the adversary also corrupts $\mathcal{P}_0$. Upon starting the protocol, the adversary behaves honestly and collaborates with the execution of the AMCU protocol up until the Consolidation step. After $\mathcal{P}_i$ receives funds through consolidation of γ_i, they do not respond to any parties requesting consolidation of their channels, but instead the adversary orders $\mathcal{P}_{j-1}$ and $\mathcal{P}_j$ to close γ_j by spending the UTXO that is the input of their Setup transaction S_j.

Discussion. In the general case the adversary needs to corrupt at least 4 parties, thus the attack requires $n \geq 4$. However, if the adversary can influence the order in which channels are consolidated, in the case of $n \geq 3$ they can always pick $1 \leq i = j - 1 \leq n - 2$ and reduce the parties they need to corrupt to 2. Moreover, note that if the order in which channels are consolidated is not known a-priori, the adversary has to guess values for i and j. We assume the adversary picks values for i and j randomly out of a uniform distribution of all possible values, i.e. $1, \ldots, n$. The probability to guess one out of n parties for the value for i equals at least $1/n$. As $i \neq j$, the value of j has to be guessed out of $n - 1$ parties which equals a probability of at least $1/(n - 1)$. Thus the probability for the adversaries success is at least $1/((n - 1)n)$ which is not negligible.

4 Our Payment Tree Construction

We describe the construction of a payment tree in respect to our running example. Let $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n, n \in \mathbb{N}$ be parties where parties $\mathcal{P}_{i-1}$ and $\mathcal{P}_i, i \in \{1, \ldots, n\}$ 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.

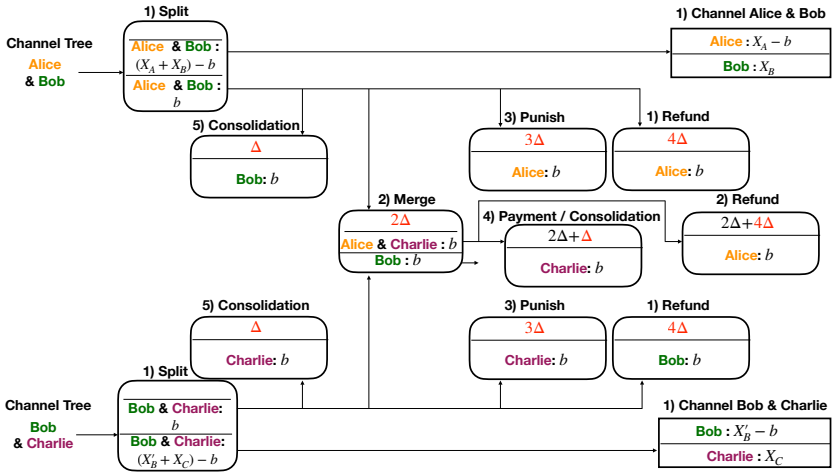

Fig. 2: Transaction Tree of a payment of b coins across 2 hops. Transactions are represented by boxes with round corners containing the UTXO they create, whereas referenced UTXOs in inputs are indicated implicitly by arrows originating from the UTXO that is spent. Red numbers indicate timelocks. Numbers atop the transaction indicate order of construction whereas transactions with the same number are constructed atomically. Payment channel sub-trees are represented as boxes with straight edges forming a black box.

Overview. We illustrate our approach in Figure 2 for a two-hop payment, i.e. for the case of $n = 2$. It is designed such that it can be extended to payment paths of arbitrary lengths. The overall approach is to take two channels, one between parties Alice and Bob, one between Bob and Charlie and construct a payment tree that creates a Funding UTXO between Alice and Charlie which both can use to perform an offchain payment. Within this construction, Bob has a special role as *intermediary*. As the intermediary Bob is the only party which has control over both channels and none of the channels can be closed before the end of construction without Bob's consent. The Payment Tree protocol enables Bob to *enforce* correctness of the construction. However, while empowering the intermediary Bob, if he fails to enforce correctness we can blame him as acting maliciously and hold him accountable. This is done by paying out Bob's collateral to the other parties as punishment, preventing loss of their funds.

Transaction Types. We use three types of transactions during our construction. Split transactions are used to split off coins from one channel, making them available to our construction in form of a Funding UTXO. The coins for the Funding UTXO are paid by the party with the lower index within the payment path. Payout transactions take a Funding UTXO as input and pay the money to one of the two parties involved in the Funding UTXO. 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.

Two Hop Payments. The payment is executed within 6 steps as shown in Figure 2 where the numbers next to the transaction indicate in which step a transaction is created. The transactions within each step are created atomically which is done using two techniques. 1) As any transaction can be committed to the ledger only when all of its ancestors can be committed to the ledger, we create transaction-trees atomically by, first, creating the whole transaction-tree and, second, signing its root afterwards. 2) We use the *ATOMIC_SIGN* protocol that allows an intermediary party to enforce creation of two transactions atomically. We use timelocks to assign priorities to transactions. A transaction that has a higher priority than another has a timelock of at least Δ lower such that any party can enforce commitment of the transaction that has the higher priority.

A payment proceeds within 6 steps. 1) Similar to AMCU [7] and Lightweight Virtual channels [8] we extend each channel-tree with an additional Funding UTXO via a Split transaction. The coins within the Funding UTXO's are paid by the parties with lower index within the payment path. The Refund payout transactions refund those coins after time 4Δ. 2) A Merge transaction spends these Funding UTXO. With a timelock of 2Δ the Merge transaction has a higher priority than the Refund transactions. The UTXO created by the Merge transaction is refunded after time $2\Delta + 4\Delta$. 3) Punish transactions are created that payout Bob's collateral. As they have a timelock of 3Δ they have a lower priority than the Merge transaction such that Bob is able to avoid payout by committing the Merge transaction as soon as its timelock expires. However, on the other side the Punish transactions are used to secure Alice's and Charlie's coins as they are next in priority in case the Merge transaction cannot be committed to the ledger. 4) Alice and Charlie perform a payment by spending their shared Funding UTXO on top of the Merge transaction with a Payment transaction that has a lower timelock and thus higher priority than the Refund transaction. 5) Consolidation payout transactions spend the Split transaction's Funding UTXOs replicating the payment within the two original channels. These have a lower timelock and thus higher priority than the Merge transaction. 6) Using the Consolidation transactions Bob can enforce payment from Alice, whereas Charlie can enforce payment from Bob. This allows both pairs of parties to safely consolidate the payment within their respective channels. Note that Refund and Payment transactions which spend the Merge transaction's Funding UTXOs have timelocks set symmetric to the Refund and Consolidation transaction that spend the Split transactions' Funding UTXOs. This allows replication of the construction using Funding UTXO created in Merge transactions instead of Split transactions, extending the approach to payment paths of arbitrary length as shown in Figure 7. The construction is applied in a balanced manner by forming a balanced binary tree of transactions and minimizing the transaction tree's height.

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. 3: Algorithm that takes two Payout transactions as input and allows the intermediary party to enforce that either both or no transaction is fully signed.

The Payment Tree Protocol. In the following, first we formally define Split, Payout and Merge transactions, before introducing algorithms 1 to 4 making up the Payment Tree protocol.

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 UTXO. 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 UTXO 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}. Analogously 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. Informally, it takes a channel and extends it with a Funding UTXO holding b coins which we can use in our construction. Although we represent this by using a Split transactions as it is 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 UTXO 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 UTXO 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 UTXO 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 as above. We extend the 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 IN_UTXO takes a Merge transaction as input and outputs the UTXOs that are used within its inputs, i.e. $f_{\mathsf{pay},0}, f_{\mathsf{pay},1}$. Informally the transaction takes the pay-Funding UTXOs of two transactions as input, creates a Funding UTXO between the two parties that did not share a Funding UTXO prior (here: $\mathcal{P}_A$ and $\mathcal{P}_B$), and pays out the collateral of the third party (here: $\mathcal{P}_C$).

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. Moreover we extend helper function 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 as Refund, Punish, Payment and Consolidation transactions depending on their purpose as shown in Figure 2.

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 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 UTXO 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 $\mathcal{P}_C \in (\mathsf{PARTIES}(f)) \setminus \{\mathcal{P}\}$.

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.

Merging Channels. Protocol *MERGE* as shown in Algorithm 2 takes two Funding UTXO 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_0 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 $t+2\Delta$ spending both Funding UTXO, 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 $t+2\Delta+4\Delta$ 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

Algorithm 2 Construction Step of a Payment Tree

function MERGE(f_0, f_1, b, t)
  $\mathcal{P}_I \leftarrow$ INTERMEDIARY(f_0, f_1)
  $\mathcal{P}_A, \mathcal{P}_B \leftarrow$ COUNTERPARY($f_0, \mathcal{P}_I$), COUNTERPARY($f_1, \mathcal{P}_I$)
  $Tr_{mrg} \leftarrow$ MERGE($f_0, f_1, t + 2\Delta$)
  $Tr_{refund} \leftarrow$ PAYOUT(OUT_UTXO(Tr_{mrg}), $\mathcal{P}_A, t + 6\Delta$)
  $Tr_{punish,A} \leftarrow$ PAYOUT($f_0, \mathcal{P}_A, t + 3\Delta$)
  $Tr_{punish,B} \leftarrow$ PAYOUT($f_1, \mathcal{P}_C, t + 3\Delta$)
  SIGN($Tr_{refund}, \{\mathcal{P}_A, \mathcal{P}_B\}, \{\mathcal{P}_A, \mathcal{P}_B\}$)
  SIGN($Tr_{mrg}, \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_I\}, \{\mathcal{P}_A, \mathcal{P}_B, \mathcal{P}_I\}$)
  ATOMIC_SIGN($Tr_{punish,A}, Tr_{punish,B}$) **return** Tr_{mrg}
end function

Fig. 4: Creation of a Funding UTXO between two counterparties. The intermediary Party can enforce atomic construction while Punish transactions are created to make it accountable if it fails to do so.

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.

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.

Payment Trees. Algorithm 4 performs a payment from $\mathcal{P}_0$ to $\mathcal{P}_n$ by iteratively merging Funding UTXO, s.t. the Merge transactions form the nodes of a balanced binary tree as illustrated in Figure 7. The algorithm takes 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 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

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. 5: Taking a Merge transactions, invalidating it and atomically updating the state on the two original Funding UTXO.

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 UTXO 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.

5 Collateral Efficiency and Security Analysis

In this section we discuss properties of the Payment Tree construction using the ongoing example of a payment of b coins across channels $\gamma_1, \ldots, \gamma_n$.

Efficiency. Figure 8 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 over the whole payment.

We observe that 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

Algorithm 4 Payment Tree Construction

function PAYMENTTREE($\gamma_1, \gamma_2, \ldots, \gamma_n$, b, t_{min})
  $F\_UTXO_i \leftarrow [], 0 \leq i \leq \lceil (\log n) - 1 \rceil$
  $MRG_i \leftarrow [], 1 \leq i \leq \lceil \log n \rceil$
  for $1 \leq i \leq n$ **do**
    $f_i \leftarrow \mathsf{SPLIT}(\gamma_i, b, t_{min})$
    Append f_i to F_UTXO_0
  end for
  for $i = 0$ until $i = \lceil (\log n - 1) \rceil$ **do**
    for $0 \leq j \leq \lfloor |F\_UTXO_i|/2 \rfloor$ **do**
      Retrieve f_{2j}, f_{2j+1} from F_UTXO_i
      $Tr_{mrg,j} \leftarrow \text{MERGE}(f_{2j}, f_{2j+1}, b, t_{min} + 2i\Delta)$
      Append $\mathsf{OUT\_UTXO}(Tr_{mrg,j})$ to F_UTXO_{i+1}
      Append $Tr_{mrg,j}$ to MRG_{i+1}
    end for
    if $|F\_UTXO_i| \mod 2 = 1$ **then**
      Remove last entry of F_UTXO_i and append to F_UTXO_{i+1}
    end if
  end for
  $Tr_{Payment} \leftarrow \mathsf{PAYOUT}(\mathsf{OUT\_UTXO}(MRG_{\lceil \log n \rceil}[0]), \mathcal{P}_n, t_{min} + 2\Delta \log n + \Delta)$
  for $i = \lceil \log n \rceil$ until $i = 1$ **do**
    for Tr_{mrg} in MRG_i **do**
      CONSOLIDATE(Tr_{mrg})
    end for
  end for
  for $1 \leq i \leq n$ **do**
    $\mathsf{UNSPLIT}(\gamma_i)$
  end for
end function

Fig. 6: Construction of a full payment tree in the shape of a balanced binary tree, execution of a payment and subsequent deconstruction.

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.

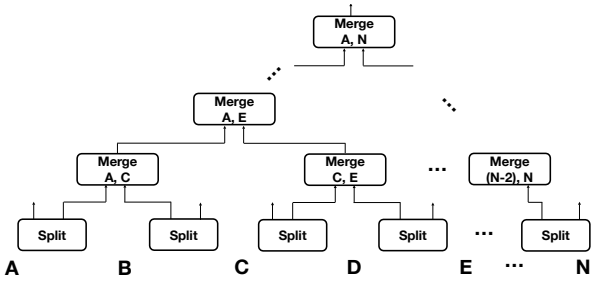

Fig. 7: Transaction tree forming a payment tree. The shape represents a balanced binary tree.

Method	pp Collateral	pp Tr.	Total Collateral	Total Tr.	Smart Contracts
HTLC [12,2]	$\mathcal{O}(b\Delta n)$	$\mathcal{O}(1)$	$\mathcal{O}(b\Delta n^2)$	$\mathcal{O}(n)$	No
Sprites [9]	$\mathcal{O}(b(n+\Delta))$	$\mathcal{O}(1)$	$\mathcal{O}(b(n+\Delta)n)$	$\mathcal{O}(n)$	Yes
Payment Tree	$\mathcal{O}(b\Delta \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(b\Delta n)$	$\mathcal{O}(n)$	No

Fig. 8: Comparison of the performance of Payment Trees across the whole payment (Total) and individually per party (pp).

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 which is the case if they act as intermediary of the Merge transaction on the top-most level of the payment tree. 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.

Security. First, we define our communication and adversarial models and thereafter define and show two properties of our construction which are *Balance Security* as defined in Theorem 1 and *Liveness* as defined in Theorem 2.

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 [8,7,9]: 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.

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.*

Sketch of Proof. First, we consider the case in which the adversary does not deviate from the protocol, but stops collaboration mid-way. We observe that due to the order in which transactions are (atomically) created, the funds accessible for any party within Merge- and Payment transaction is unchanged, except when executing the payment between $\mathcal{P}_0$ and $\mathcal{P}_1$ explicitly. Any party receives their Funds by having the transaction tree be committed to the ledger. Even if a corrupted party acts as intermediary and stops collaboration after receiving signatures and before providing signatures themselves. As only they risk losing funds due to Punish transactions, having them selectively commit and withhold transactions does not result in the loss of funds of their counterparties.

Next, we consider the case where the adversary corrupts two parties to double-spend a Funding UTXO that is the input of a Merge transaction. Assume $\mathcal{P} \in \{\mathcal{P}_1, \ldots, \mathcal{P}_{n-1}\}$ is neither payer or payee of the overall payment and is honest. Moreover, the adversary double spends a Funding UTXO that is the input of a Merge transaction $Tr_{mrg,\mathcal{A}}$ on level i. Note that for this to happen, the party that acts as intermediary of $Tr_{mrg,\mathcal{A}}$ must be corrupted as either Funding UTXO that is input of $Tr_{mrg,\mathcal{A}}$ requires its signature to spend it. If $\mathcal{P}$ is not part of a transaction that is descendant of $Tr_{mrg,\mathcal{A}}$ they are unaffected and do not lose funds. Otherwise, if they are part of $Tr_{mrg,\mathcal{A}}$ they are not the intermediary party as they are honest and they will receive b coins through a Punish transaction. If they are not part of $Tr_{mrg,\mathcal{A}}$, let $j, i < j \geq \log n$ be the lowest level on which they are part of a Merge transaction that has $Tr_{mrg,\mathcal{A}}$ as descendant. Then they must not be the intermediary, as otherwise, they would have a descendant of $Tr_{mrg,\mathcal{A}}$ on a lower level. Therefore they are not intermediary and receive b coins through a Punish transaction on that level. However, as they are neither $\mathcal{P}_0$ nor $\mathcal{P}_n$ they act as intermediary of a Merge transaction on level $k, j < k < \log n$ which is descendant of $Tr_{mrg,\mathcal{A}}$. On that level $\mathcal{P}$ has to pay out b coins through one Punish transaction. Note that $\mathcal{P}$ does not pay out b coins through two Punish transactions as otherwise they could commit the Merge transaction instead to avoid payout of any Punish transaction. Moreover, any party is intermediary of a Merge transaction only once within the transaction. Overall, $\mathcal{P}$'s balance equals $b - b = 0$ s.t. they do not lose funds. The reasoning for $\mathcal{P}_0$ and $\mathcal{P}_n$ is analogous. As they are on the top level of the Payment Tree, they have a Merge transaction that is descendant of $Tr_{mrg,\mathcal{A}}$ and therefore do receive b coins. However, they are never intermediary of a Merge transaction, s.t. their balance is b coins. Therefore, $\mathcal{P}_0$ and $\mathcal{P}_n$ do not lose coins independently of whether they performed the payment between each other or not. □

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.*

Sketch of Proof. All honest parties commit the transactions they are involved in as soon as their timelocks expire. First, we note that any transaction containing a Funding UTXO is created atomically with a Payout transaction that pays out the funds to a party that receives exclusive access to it. Therefore, the adversary cannot have funds being locked within a Funding UTXO they share control of indefinitely. Although all transactions have increasingly higher timelocks, all transactions can be committed to the ledger by time $t_{min} + 2\Delta \log n + 4\Delta$. By this time, no funds are locked within a Funding UTXO and any funds can be claimed by one party exclusively through Payout transactions. As *Balance Security* holds, no party loses funds when all Payout transactions are committed to the ledger s.t. for any party it holds that by time $t_{min} + 2\Delta \log n + 4\Delta$ they have exclusive access to all their funds. □

6 Conclusion

In this work we introduced Payment Trees, a protocol for scalable payments within a PCN with logarithmic individual collateral and linear total collateral. Although, as trade-off in the worst case a party needs to commit a logarithmic amount of transactions to the ledger in case of a Dispute, the total amount of transactions committed to the ledger is linear which is equal to related work, i.e. HTLCs and Sprites. 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 of its underlying ledger, thus providing the first alternative to HTLCs.

References

1. Raiden network. raiden.network, accessed: 2018-09-03
2. Bowe, S., Hopwood, D.: Hashed Time-Locked Contract transactions. https://github.com/bitcoin/bips/blob/master/bip-0199.mediawiki (2017), [Online; accessed 29-August-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 (Feb 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: Symposium on Self-Stabilizing Systems. pp. 3–18. Springer (2015)
5. Dziembowski, S., Eckey, L., Faust, S., Malinowski, D.: Perun: Virtual payment hubs over cryptocurrencies. In: Perun: Virtual Payment Hubs over Cryptocurrencies. IEEE (2017)
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 (Nov 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. 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, Heidelberg (Feb 2019). https://doi.org/10.1007/978-3-030-32101-7_30
10. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
11. PDecker, C., Russel, R., Osuntokun, O.: eltoo: A simple layer2 protocol for bitcoin. See https://blockstream.com/eltoo.pdf (2017)
12. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments. See https://lightning. network/lightning-network-paper. pdf (2016)