Ethna: Channel Network with Dynamic Internal Payment Splitting*

Stefan Dziembowski[1] and Paweł Kędzior[1]

University of Warsaw

Abstract *Off-chain channel networks* are one of the most promising technologies for dealing with blockchain scalability and delayed finality issues. Parties that are connected within such networks can send coins to each other without interacting with the blockchain. Moreover, these payments can be "routed" over the network. Thanks to this, even the parties that do not have a channel in common can perform payments between each other with a help of intermediaries.
In this paper, we present a new technique (that we call *Dynamic Internal Payment Splitting (DIPS)*) that allows the intermediaries in the network to split the payments into several sub-payments. This can be done recursively multiple times by subsequent intermediaries. Moreover, the resulting "payment receipts" can be aggregated by each intermediary into one short receipt that can be propagated back in the network. We present a protocol (that we call "ETHNA") that uses this technique. We provide a formal security definition of our protocol and we prove that ETHNA satisfies it. We also implement a simple variant of ETHNA in Solidity and provide some benchmarks.

1 Introduction

Blockchain technology [24] allows a large group of parties to reach consensus about contents of an (immutable) ledger, typically containing a list of transactions. In Blockchain's initial applications these transactions were simply describing transfers of *coins* between the parties. One of the very promising extensions of the original Bitcoin ledger, are blockchains that allow to register and execute the so-called *smart contracts*, i.e., formal agreements between the parties, written down in a programming language and having financial consequences. Probably the best-known example of such a system is *Ethereum* [32]. One of the main limitations of several blockchain-based systems is delayed finality, lack of scalability, and non-trivial transaction fees. For example, in Bitcoin it takes at least around 10 minutes to confirm a transaction, at most 7 transactions per second can be processed, and the average transaction fee is currently around 30 cents.

Off-chain channels [6, 28, 29] are a powerful approach for dealing with these issues. Let us start with describing the most basic variant of this technology, called the "*payment* channels". Informally, a payment channel between Alice and Bob is an abstract object in which both parties have some coins. A channel has a corresponding smart contract on the blockchain that can be used for resolving conflicts between the parties. The parties *open* a channel by putting some coins into it. They can later change the *balance* of the channel (i.e. information on how many of channel's coins belong to Alice and Bob, respectively) just by exchanging messages, and without interacting with the blockchain. The channel can be *closed* by Alice or Bob, in which case the last channel's balance is used to determine how many coins are transferred to each of them. Since updates do not require blockchain participation (the are done "off-chain"), each individual update is immediate (its time is determined by the network speed) and at essentially no cost. The only operations that involve blockchain are: "opening" and "closing" the channel. Hence, this approach also significantly improves scalability. All these advantages hold only if Alice and Bob are cooperating. In the "pessimistic" case (when one of them is malicious) there are no benefits of using this technology, and the only thing that is guaranteed is that the honest party does not loose her coins. This is ok, since in practice, it is expected that in a vast majority of cases the parties are cooperating (i.e. "behaving optimistically"). For more background on channels see Sec. 2.

* This work was partly supported by a grant *FY18-0023 PERUN* from the *Ethereum Foundation* and by the *TEAM/2016-1/4* grant from the *Foundation for Polish Science*.

2 Background and our contribution

Let us start (in Sec. 2.1) with providing an overview of the state of the art in this area. Our contribution is outlined in Sec. 2.2. For the purposes of this informal description suppose that the maximal blockchain reaction time is 1 hour, or more precisely: each party is guaranteed that every correct message that it sends to the blockchain will be accepted by it within 1 hour.

Let us start with some short preliminaries. We consider protocols run by a set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$. For standard definitions of cryptographic algorithms such as signature schemes or hash functions, see, e.g., [17]. When we say that a message is "signed by some party" we mean that it was signed using some fixed signature scheme that is existentially unforgeable under chosen-message attack. The only party that is explicitly signing messages in our paper is P_n. A message m signed by P_n will be denoted $[\![m]\!]$. Natural numbers are denoted with $\mathbb{N}$. Let A be some finite alphabet. Strings $\alpha \in A^*$ will be frequently denoted using angle brackets: $\alpha = \langle \alpha_1, \ldots, \alpha_m \rangle$. We use some standard notation for functions and string operations (such as concatenation or taking prefixes). For completeness it is presented in Appx. A.

2.1 Contracts and channels

2.1.1 Smart contracts. As already mentioned, smart contracts (or simply: "contracts") are formal agreements that are written down on the blockchain. They are expressed in some programming language. In Ethereum this language is called *Solidity*, see, e.g., [5]. A contract has its own variables in which it can store data. It can also own coins (i.e. it has an "account"). A contract is deployed by a blockchain user that also pre-loads it with some coins. The parties send messages to the contract by calling its functions. The state of the contract is public. Deploying and executing a contract costs *fees* (payed by the party who initiated this action), and it is not immediate (due to blockchain's delayed finality). For more on this topic see, e.g., [5, 8, 20].

2.1.2 Payment channels. As mentioned above, a payment channel is *opened* when Alice and Bob deploy a smart contract $\mathcal{C}^{\text{ledger}}$ on the ledger, and deposit some number of coins (say: x, and y, respectively) into it. The initial *balance* of this channel is: "x coins in Alice's account, y coins in Bob's account" (or $[\text{Alice} \mapsto x, \text{Bob} \mapsto y]$ for short). We model amounts of coins as non-negative integers. This can be done without loss of generality, since one can always take as the "unit" the smallest coin value in the system (e.g. "Satoshi" in Bitcoin) This balance can be *updated* (to some new balance $[\text{Alice} \mapsto x', \text{Bob} \mapsto y']$, such that $x + y = x' + y'$) by just exchanging messages between the parties. Hence, the parties can perform payments between each other very quickly and for free. The corresponding smart contract guarantees that each party can at any time *close* the channel and get the money that correspond to her latest balance. Only the opening and closing operations require interaction with the blockchain. Alice and Bob maintain a counter i, initially equal to 0, and increased by 1 after balance update. The update procedure is symmetric for both users. Suppose that Alice wants to update channel's balance from $\beta := [\text{Alice} \mapsto x, \text{Bob} \mapsto y]$ to $\beta' := [\text{Alice} \mapsto x', \text{Bob} \mapsto y']$. She then sends to Bob a pair (β, i) signed by her. Bob then replies with his signature on (β', i) and the balance is updated. This procedure permits an essentially unbounded number of balance updates. In the closing procedure the $\mathcal{C}^{\text{ledger}}$ contract compares version numbers of signed channel balances submitted by the parties, and distributes the coins according to the balance with the higher version number. Channel closing in total requires at most 3 rounds of interaction with the blockchain (see, e.g., [10]).

The update procedure is immediate in the optimistic case (since it amounts to a simple message exchange between the parties, without any blockchain interaction). The pessimistic case is a bit more tricky. Let us look again at the channel update scenario presented above. Observe that if the update is beneficial for Bob (i.e. $y' > y$) then Alice does not need to obtain Bob's signature, since she is also ok with channel being closed with the previous balance. Hence, if she does not obtain Bob's signature she can remain silent. The situation is different when $x' > x$. In this case Alice, in order to be sure that she was payed needs to contact the blockchain. Since Bob turned out to be unreliable the most reasonable thing for her is to simply close the channel. During the closing procedure it will become evident if Bob is closing the channel with balance

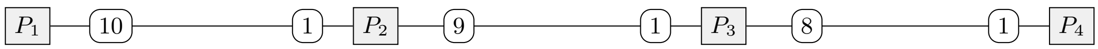
$\widehat{\beta}$ or β, i.e, if the update happened or not. It is therefore convenient to distinguish between *immediate* and *non-immediate* updates. The former in which the updating party does not need the confirmation from the other party. The latter are those where such confirmation is needed. For the payment channels the immediate ones are exactly those that are beneficial for the other party.

2.1.3 State channels. In a nutshell, a *state channel* is an off-chain channel that, in addition to the payments between Alice and Bob, allows them to internally execute smart contracts "inside of a channel" (i.e. without interacting with the blockchain). The corresponding smart contract $\mathcal{C}^{\text{ledger}}$ *deployed on the ledger* is responsible for executing the *internal* smart contract $\mathcal{C}^{\text{int}}$ in case the parties enter into a dispute. Normally, however, this is not needed, and the parties can perform this execution "peacefully", i.e., without asking the blockchain for the conflict resolution. Apart from the balance information, state channel contains data D that is interpreted as the state of the internal contract $\mathcal{C}^{\text{int}}$. Additionally, some coins that initially belong to the parties can be *blocked* for the purposes of $\mathcal{C}^{\text{int}}$ (i.e. they can "belong" to this contract). As long as the parties are cooperating they perform the "execution" of $\mathcal{C}^{\text{int}}$ just by updating D. This is done by exchanging signatures on the new "versions" of D, exactly as it is done in case of the payment channels. Resolving conflicts is a bit more tricky. Suppose the current state of $\mathcal{C}^{\text{int}}$ is D and Alice wants to call a function f of $\mathcal{C}^{\text{int}}$, which results in some new state D'. She sends signed D' (together with the version number) to Bob, who does *not* reply with his signature. Alice then asks $\mathcal{C}^{\text{ledger}}$ for help in execution. This procedure is slightly involved (for the details see, e.g., [8]), and can take up to 3 rounds of blockchain interaction. Fortunately, Alice's input to f (or: "message that she sends to the $\mathcal{C}^{\text{int}}$") becomes known to Bob already after 1 round of blockchain interaction (or, in our, terms: after 1 hour.) There are many details that we omitted in this short description, e.g., the real-life state channel systems need to allow parallel execution of several contracts withing a single channel. For more on state channels see, e.g., [1, 4, 8, 9, 10]).

Similarly to the payment channels, in the state channels we also distinguish between the immediate and the non-immediate updates, where the former ones do not require confirmation from the counterparty, while the latter ones do. Non-immediate updates are used only when the initiating party is ok with the counterparty not accepting the update (since she has "nothing to loose"). We show an example of such a situation in the next section. We say that *a party P sent a message m via a state channel $P \multimapboth P'$ to a party P'* when P makes an update to $P \multimapboth P'$ channel that results in m being recorded in D (when this message is not needed anymore it can be erased from D, we often do not mention this erasure when it is clear from the context). We say that m is *immediate* if the corresponding update is immediate. Message m can come together with a request to block some coins of the sender (i.e., transfer them to the account of $\mathcal{C}^{\text{int}}$).

2.1.4 Payment channel networks (PCNs). Suppose we are given a set of parties $P_1, \ldots, P_n$ and channels between them. These channels naturally form an (undirected) *channel graph*, which is a tuple $\mathcal{G} = (\mathcal{P}, \mathcal{E}, \Gamma)$ with the set of vertices $\mathcal{P}$ equal to $\{P_1, \ldots, P_n\}$ and set $\mathcal{E}$ of edges being a family of two-element subsets of $\mathcal{P}$. The elements of $\mathcal{P}$ will be typically denoted as "$P_i \multimapboth P_j$" (instead of $\{P_i, P_j\}$). Every $P_i \multimapboth P_j$ represents a channel between P_i and P_j, and the *cash function Γ* determines the amount of coins available for the parties in every channel. More precisely, every $\Gamma(P_i \multimapboth P_j)$ is a function f of a type $f : \{P_i, P_j\} \to \mathbb{Z}_{\geqslant 0}$. We will often write $\Gamma^{P_i \multimapboth P_j}$ to denote this function. The value $\Gamma^{P_i \multimapboth P_j}(P)$ denotes the amount of coins that P has in her *account* in channel $P_i \multimapboth P_j$. A *path* (in $\mathcal{G}$) is a sequence $P_{i_1} \multimap \cdots \multimap P_{i_t}$ such that for every j we have $P_{i_j} \multimapboth P_{i_{j+1}} \in \mathcal{E}$. The channel system is deployed with some initial value of Γ_0, which evolves over time, resulting in functions $\Gamma_1, \Gamma_2, \ldots$. For simplicity we assume that (a) once a channel system is established, no new channels are created (i.e., $\mathcal{E}$ remains fixed), and (b) no coins are added to the the existing channels, i.e., the total amount of coins available in every channel $e = P_i \multimapboth P_j$ never exceeds the total amount available in it initially. Channel graphs can serve for payment routing. We start by recalling how this works in the most popular payment channel networks, such as *Lightning* or *Raiden*. Our description is rather high-level (for the details, see, e.g., [28]). It also uses some non-standard terminology (in particular "pushing" and "acknowledging" payments) that will be useful in the next sections. We also note, that the basic concept of the PCNs that is presented below can been extended in several ways. We explain the basic payment routing by the following example. Suppose we have of four parties: P_1, P_2, P_3, and

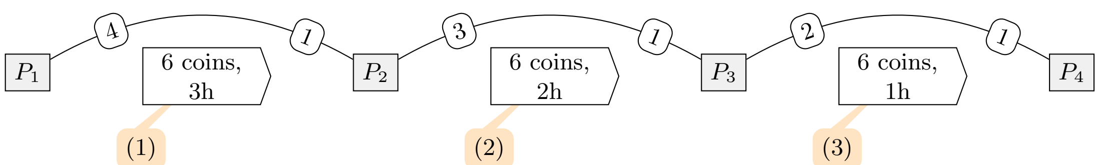
P_4, such that there exists a state channel between each P_i and P_{i+1}, in which P_i has $11 - i$ coins, and P_{i+1} has 1 coin. This is depicted below:



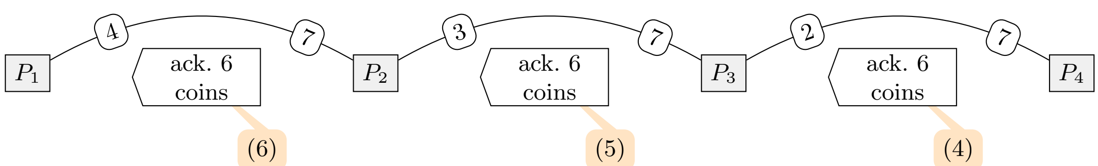
(1)

Now, suppose the *sender* P_1 wants to send 6 coins to the *receiver* P_4 over the path $P_1 \to P_2 \to P_3 \to P_4$, with P_2 and P_3 being *intermediaries* that *route* these coins. This is done as follows: (1) party P_1 asks P_2 to forward 6 coins in the direction of P_4 (we call such a request *pushing* coins from P_1 to P_2). The proof that P_4 received these coins has to be presented by P_2 within 3 hours (denote this proof with π — we will discuss how π looks like in a moment). If P_2 manages to do it by this deadline, then she gets these coins in her account in the channel $P_1 \circ\!\!-\!\!\circ P_2$. To guarantee that this will happen, P_1 initially blocks these coins in the channel $P_1 \circ\!\!-\!\!\circ P_2$. These coins can be claimed back by P_1 if the 3 hours have passed, and P_2 did not claim them.

Since P_2 also does not have a channel with P_4, she asks P_3 to route the coins. This is done in step (2) by pushing 6 coins the same way as before, except that the deadline is now set to 2 hours. Finally, (3) party P_3 pushes the coins to P_4, i.e., she offers P_4 to claim (by providing proof π) 6 coins in the channel channel $P_3 \circ\!\!-\!\!\circ P_4$ within 1 hour. Pictorially this looks as follows (labels (1), (2), and (3) indicate the steps described above).



Now, suppose that (4) party P_4 claims her 6 coins in channel $P_3 \circ\!\!-\!\!\circ P_4$. This can only be done by providing a proof π that she received these coins. We call this process *acknowledging* the payment. Parties P_3 and P_2 can now consecutively claim theirs coins in the channels $P_2 \circ\!\!-\!\!\circ P_3$ and $P_1 \circ\!\!-\!\!\circ P_2$ (respectively) by submitting an acknowledgment containing the proof π. This is done in steps (5), and (6), as indicated on the picture below.



Note that the "n hours" deadlines in the state channels $P_1 \circ\!\!-\!\!\circ P_2$ and $P_2 \circ\!\!-\!\!\circ P_3$ are chosen in such a way that each intermediary P_i can be sure that if she looses 6 coins in channel $P_i \circ\!\!-\!\!\circ P_{i+1}$ then she gets 6 coins in channel $P_{i-1} \circ\!\!-\!\!\circ P_i$. The "1 hour" in channel $P_3 \circ\!\!-\!\!\circ P_4$ is needed to ensure that P_3 does not claim her money back immediately after learning π (recall that the messages sent to the blockchain are not secret, and can be learned by malicious parties even before they appeared on it). Observe also that in the above example the amount of coins that can be pushed via a channel $P_i \circ\!\!-\!\!\circ P_{i+1}$ is upper-bounded by the amount of coins that P_i has in this channel. Therefore the maximal amount of coins that can be pushed over path $P_1 \to P_2 \to P_3 \to P_4$ is equal to the minimum of these values (which is equal to 8 in the situation depicted in Eq. (1)). We will call this value the *capacity* of a given path.

On the technical level, in the Lightning network the proof π is constructed using so-called *hash-locked transactions*: before the routing procedure starts party P_4 generates a random string X and sends $h := H(X)$ to P_1 (string X is long enough to make it infeasible to guess X based only on h). Then, the payment proof π

is simply equal to X. To guarantee security, "pushing" and "acknowledging" is done via the state channels. The push message additionally blocks the pushed coins. This message is immediate, since the party P who sent it, does not risk anything if P denies receiving it (since P can take her money back after the specified deadline). The "acknowledging" message is *not* immediate since the party who sent it needs to be sure that the right amount of coins was transfered to her. As explained in Sec. 2.1.3, in the pessimistic case this can take up to 3 hours, but luckily X becomes known to each P^i in at most 1 hour after it was sent by P^{i+1}. Hence P^i is guaranteed to learn X early enough to forward it to channel $P^{i-1} \multimapboth P^i$ before the timeout.

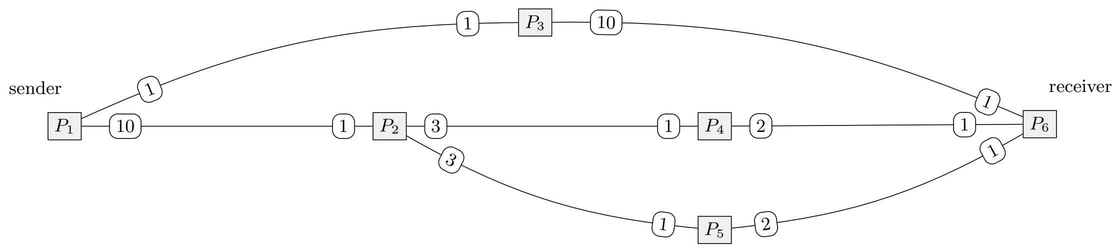
An interesting feature of this protocol is that proof $\pi(= X)$ serves not only for internal purposes of the routing algorithm, but can also be viewed as the output of the protocol which can be used by P_1 as a receipt that she transfered some coins to P_4. In other words: P_1 can use π to resolve disputes between with P_4, either in some smart contract (that was deployed earlier, and uses the given PCN for payments), or outside of the blockchain. Since P_1 and P_4 can potentially perform lots of different payments, in real life it is also useful to have a way to link a given payment with an "invoice" issued by P_4. The invoice should at contain the value of h and possibly some other data that describe the purpose of payment. It is signed by P_4 and passed to P_1 before the payment starts. For more on how invoices are implemented in Lightning see, e.g., [30].

The routing procedure described above works over arbitrary channel graphs. The path over which the payments in the channel graph are routed does not need to be fixed in advance. Party P_1 needs to decide only on the maximal length of the routing path (this is done by determining the timeout for the "push" transaction originating from P_1). All the later routing decisions can be made spontaneously. Consider, e.g, the channels as on Fig. 1 (a), and suppose the sender P_1 wants to transmit u coins to the receiver is P_6. There are three paths over which this payment can be routed: (a) $(P_1 \twoheadrightarrow P_3 \twoheadrightarrow P_6)$, (b) $(P_1 \twoheadrightarrow P_2 \twoheadrightarrow P_4 \twoheadrightarrow P_6)$, and (c) $(P_1 \twoheadrightarrow P_2 \twoheadrightarrow P_5 \twoheadrightarrow P_6)$. In Lightning it is possible to first push these coins to P_2 without initially deciding which of the two paths ((b) or (c)) will be chosen, and then let P_2 decide whether she wants to push the money further to P_4, or P_6. The advantage of this feature is that it allows the parties to react in an ad-hoc way to the changes in the capacities of paths that lead to the receiver. Note, however, that once P_1 decided to push x coins to P_2, it is impossible for P_2 to *split* them into two amounts v and w and send them via paths (b) and (c) separately. The ability to perform such "dynamic payment *splitting*" is the main novelty of the Ethna protocol that we introduce.

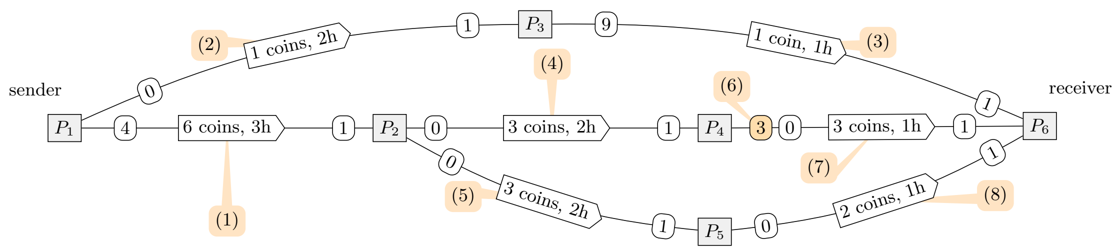
2.2 Our contribution and related work

One of the main problems with the existing PCNs is that routing a payment between two parties requires a path from the sender to the receiver that has sufficient capacity. This problem is amplified by the fact that capacity of potential paths can change dynamically, as several payments are executed in parallel. Although usually the payments are very fast, in the worst case they can be significantly delayed since each "hop" in the network can take as long as the pessimistic blockchain reaction time, which was 1 hour in the example in Sec. 2.1.4. Therefore it is hard to predict exactly what will be the capacity of a given path even in very close future. This is especially a problem if capacity of a given channel is close to being completely exhausted (i.e. it is close to zero, because of several ongoing payments). Some research suggests [7] that while Lightning is very efficient in transferring small amount of coins, transferring the larger ones is much harder, and in particular transfers of coins worth \$200 succeed with probability 1%. In these paper we show a protocol that addresses this problem. We present Ethna[1], a payment channel network protocol that allows the intermediaries to "split" the payments and later aggregate the receipts on the way back to the payment sender. This can be done ad hoc, in a reaction to dynamically changing capacity of the paths, or to the fees. For this reason, we call our technique *Dynamic Internal Payment Splitting (DIPS).* It is important to stress that the amount of data that is passed between two consecutive parties on the path does *not* depend on the number of sub-payments in which the payment is later divided. The same applies to the data that these two

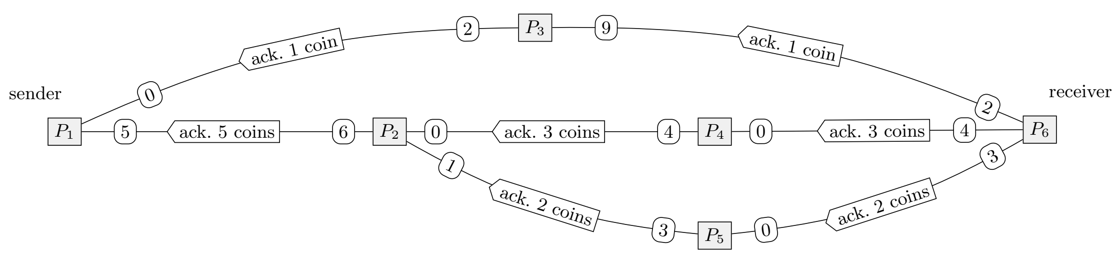
[1] Since the coin transfers in Ethna resemble a bit a lava flood (with large streams recessively bifurcating into small sub-streams), we call our protocol Ethna, in reference to Etna, one of the highest active volcanoes in Europe. The letter "h" is added so that the prefix "Eth-" is reminiscent of ETH, the symbol of Ether (the currency used in Ethereum).



(a) The channel graph with the initial coin distribution.



(b) The sender P_1 wants to send 7 coins to the receiver P_6. She splits these coins into two amounts: 6 coins pushed to P_2 and 1 coin is pushed to P_3. This is indicated with labels (1) and (2) respectively. Then (3) party P_3 simply pushes 1 coin further to P_6. Party P_2 splits 6 coins into $3+3$, and pushes 3 coins to both P_4 (4) and P_5 (5). Path $P_4 \rightarrow P_6$ initially had capacity 2 only (see Fig. (a) above), but luckily in the meanwhile 1 coin got unlocked (6) for P_4 in channel $P_4 \multimap\!\!\!\circ\, P_6$, and hence (7) party P_4 pushes all 3 coins to P_6. No coins got unlocked in channel $P_5 \multimap\!\!\!\circ\, P_6$, so P_5 pushes only 2 coins to P_6. Above, the "n hours" deadlines come from the fact that the maximal length of the paths leading to P_6 via a given party is n. The channel balances correspond to the situation *after* the coins are pushed (except of channel $P_4 \multimap\!\!\!\circ\, P_6$ where we also indicated the fact that 1 coin got unlocked (6)).



(c) Party P_2 acknowledges payment of 1 coin to P_3, which, in turn acknowledges it to P_3. Party P_6 also acknowledges payment of 3 coins to P_4 and 2 coins to P_5, who later acknowledge them to P_2. Once P_2 receives both acknowledgments she "aggregates" them into a single acknowledgment (for 5 coins) and sends it to P_1. As a result $5+1=6$ coins are transferred from P_1 to P_6. The channel balances correspond to the situation *after* the coins were acknowledged. Note that these actions can happen concurrently, e.g., acknowledgments along the path $P_6 \rightarrow P_3 \rightarrow P_1$ can be arbitrarily interleaved with what is done in the other parts of the graph (even before steps (4) and (5) on Fig. (b) above started).

Figure 1: An example of how Ethna operates. An edge "$\boxed{P_i}\!-\!(x)\!-\!-\!-\!(y)\!-\!\boxed{P_j}$" denotes the fact that there exists a channel between P_i and P_j, and the parties have x and y coins in it, respectively.

parties send to the blockchain in case there is a conflict between them. Perhaps the easiest way to describe Ethna is to look at the payment networks as tools for outsourcing payment delivery. For example, in the scenario from Sect. 2.1.4 party P_1 outsources to P_2 the task of delivering 6 coins to P_4, and gives P_2 three hours to complete it (then P_2 outsources this task to P_3 with a more restrictive deadline). The sender might not be interested in *how* this money is transferred, and the only thing that matters to her is that it is indeed delivered to the receiver, and that she gets the receipt. In particular, the sender may not care if the money gets split on the way to the receiver, i.e., if the coins that he sends are divided into smaller amounts that are transferred independently over different paths. In many cases the sender may also be ok with not all the money being transferred at once. More precisely, suppose that he intends to transfer u coins to the receiver. Then he can also accept the fact that $v < u$ coins were transferred (due to network capacity limitations), and try to transfer the remaining $u - v$ coins later (in another "installments"). Also, in many cases (e.g. file sharing) the goods that the seller delivers in exchange for the payment can be divided into very small units, and sent to the buyer depending on how many coins have been transferred so far. Observe also that such payment splitting can also in a reaction to different fees that the intermediaries ask (some routes may be cheaper than some other ones).

Ethna permits such recursive payment splitting into "sub-payments" and partial transfers of the coins. The "sub-receipts" for sub-payments are aggregated by the intermediaries into one short sub-receipt, so that their size does not grow with the number od aggregated sub-receipts. All this is achieved with very reasonable complexity. In particular, the gas and communication complexities for individual parties do not depend on the number of payments in which their payment is split by further intermediaries. We summarize the complexity of Ethna in Sec. 5.3.1. We also implemented Ethna in Solidity. We describe this implementations and provide some benchmarks in Sec. 6. in Ethna we avoid using advanced techniques such as non-interactive zero knowledge or homomorphic signature schemes. Instead, we rely on a technique called "fraud proofs" in which an honest behavior of parties is enforced by a punishing mechanism (this method was used before, e.g., in [8, 16, 27, 31]).

2.2.1 Other related work. Some of the related work was mentioned already before. Off-chain channels are a topic of intensive research, and there is no space here to describe all the recent exciting developments [8, 9, 10, 11, 12, 13, 18, 19, 21, 22, 23] in this area. The reader can also consult SoK papers on off-chain techniques [14, 15]. Splitting payments into multiple paths has been considered before in [2, 25, 26]. The fundamental difference between Ethna and all these works is that, unlike Ethna none of them allows the intermediaries to split the payments in an ad-hoc way, or to aggregate the sub-receipts. More concretely, Osuntokun [25] proposed so-called "atomic multipath payments" technique. The main idea of this approach is to divide a payment into several sub-payments and route them over different paths. "Atomicity" means that the sender is guaranteed that either all payments were processed or none of them. This is opposite to what we do, since we actually deliberately allow partial transfers of the payments. On the other hand, "atomicity" can be also obtained in Ethna by agreeing that if a total transfer of coins did not succeed within some deadline, then the coins are returned to the sender (this can be enforced by the receiver, since he holds a receipt for these coins). Partial coin transfers were considered by Piatkivskyi and Nowostawski in [26], but with no aggregation techniques and ad-hoc splitting. In a recent, very interesting paper Bagaria et al. [2] proposed a *Boomerang* system which allows to split the payments (by the sender) into multiple parts in a "redundantly" and tolerate the fact that only some of them succeed. The papers [2, 26] focus on routing techniques, which is a topic that we do not address in this paper (we leave designing routing algorithms for Ethna as a future research problem).

Organization of the rest of the paper In Sec. 3 we describe Ethna features (i.e. its informal specification, including the security properties). Then, in Sec. 4 we provide an informal description of Ethna's main design principles. In Sec. 5 we present formal security definition, protocol details, and security analysis (due to the lack of space part of the security proof is moved to Appx. **??**). An overview of our implementation is presented in Sec. 6.

3 Overview of Ethna features

Let us now explain informally the features of Ethna (for a formal definition see Sec. 5.2). As highlighted above, the main advantage of Ethna over the existing PCNs is that it allows ad-hoc payment splitting. Let P_1 be the *sender*, $P_2, \ldots, P_{n-1}$ be the *intermediaries*, and P_n be the *receiver*. Moreover, let v be the amount of coins that P_1 wants to send to P_n, and let t be the maximal time until when the transfer of coins should happen. Since in general P_1 can perform multiple payments to P_n, we assume that each payment comes with a nonce μ (taken form the set of natural numbers, say) that can be later used to identify this payment. Sometimes we will simply call it "payment μ". In this paper we present our protocol in a stand-alone way, i.e., we do not take into account possible parallel executions of Ethna (e.g., with P_2 being the sender and P_1 being one of the intermediaries) and other protocols. This is done purely for the sake of simplicity, and we believe that our protocol satisfies such stronger "composability" [3] properties. We leave formalizing this as open research direction.

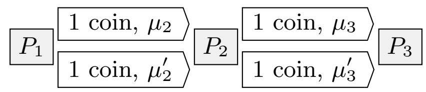
For simplicity, in this informal description we assume that all the parties are honest. The security properties (taking into account malicious behavior of the parties) are described at the end of this section, and formally defined in Sec. 5.2. For an example of Ethna routing see Fig. 1. Let us start by describing how the protocol looks like from the point of view of the sender P_1. Let $P_{i_1}, \ldots, P_{i_t}$ be the neighbors of P_1, i.e., parties with which P_1 has channels. Suppose the balance of each channel $P_1 \multimapboth P_{i_j}$ is $[P_1 \mapsto x_i, P_{i_j} \mapsto y_j]$ (meaning that P_1 and P_{i_j} have x_i and y_j coins in their respective accounts in this channel). Now P_1 chooses to push some amount v_j of coins (such that $v_j \leqslant \min(x_i, v)$) to P_n via some P_{i_j}, and set up a deadline t_j for this (we will also call v_j a *sub-payment* of payment μ). This results in: (a) the balance $[P_1 \mapsto x_i, P_{i_j} \mapsto y_j]$ changing to $[P_1 \mapsto x_i - v_j, P_{i_j} \mapsto y_j]$, (b) the amount of coins that P_1 still wants to transfer to P_n decreased as follows: $v := v - v_j$, and (c) P_{i_j} holding "v_j coins of P_1" that she should transfer to P_n within time t_j. It is also ok if P_{i_j} transfers only some part $v'_j < v_j$ of this amount (this can happen, e.g., if the paths that lead to P_n via P_j do not have sufficient capacity). In this case, P_1 has to be given back the remaining ("non-transferred") amount $r = v_j - v'_j$. More precisely, before time t_j comes, party P_{i_j} acknowledges the amount v'_j that she managed to transfer. This results in changing the balance of the channel $P_1 \multimapboth P_{i_j}$ by (a) crediting v'_j coins to P_{i_j}'s account in it, and (b) r coins to P_1's account. Moreover (c) P_1 adds back the non-transferred amount r to v, by letting $v := v + r$. Here (a) corresponds to the fact that P_{i_j} has to be given the coins that she transfered (and hence "lost" in the other channels"), and (b) comes from the fact that not all the coins were transfered (if P_{i_j} managed to transfer all the coins, then, of course, $r = 0$). Finally, (c) is used for P_1's "internal bookkeeping" purposes, i.e., P_1 simply writes down the fact that r coins "were returned" and still need to be transferred.

While party P_1 waits for P_{i_j} to complete the transfer that it requested, she can also contact some other neighbor P_{i_k} asking her to transfer some other amount v_k to P_n. This is done in exactly the same way as transferring coins via P_{i_j} (described above). In particular, the effects on the balance of the channel $P_1 \multimapboth P_{i_k}$ are as before (with subscript "j" replaced with "k"). In the example on Fig. 1 party P_1 splits 7 coins into 6 (that she pushes to P_2) plus 1 that she pushes to P_3. In more advanced cases several such transfers can be done in parallel with other neighbors of P_1. Moreover, P_1 can push several sub-payments (of payment μ) to one neighbor. For example, P_1 can push again some new amount to P_{i_j} hoping that maybe this time there will be more capacity available for routing payments via this party.

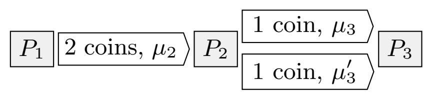
The main feature of Ethna is that this process can be repeated by the intermediaries. Let P be a party that holds some coins that were "pushed" to it by some P' (and that originate from P_1 a have to be delivered to P_n). Now, P can split them further, and moreover she can decide on its own how this splitting is done depending, e.g., on the current capacity of the possible paths leading to P_n. For instance, P_{i_j} can decide to split v_{i_j} further to between its neighbors in the same way as P_1 split u between its neighbors.

In the example on Fig. 1 party P_2 splits 6 coins into two halves, each of them pushed to one of her two neighbors (P_4 and P_5). Party P can also decide on its own about the timeout t of this payment. The only restriction is that t has to come at least 1 hour before the time she has to acknowledge that payment back to P'. This is because P needs this "safety margin" of 1 hour in case P' is malicious, and the acknowledgment has to be done "via the blockchain".

To make it as general as possible, Ethna permits also that several sub-payment of the same payment μ are routed via the same party independently. For example if there existed a path between P_3 and P_4 on Fig. 1, then P_3 could push its 1 coin to P_4. This also means that we allow using exactly the same path for more than one sub-payment of the same payment. For example on a path $P_1 \twoheadrightarrow P_2 \twoheadrightarrow P_3$ it is possible for P_1 send 2 coins to P_3 in two different ways depicted on Fig. 2 (for a moment ignore the μ's).



Way 1: P_1 splits 2 coins as $1+1$ and pushes each of them to P_2 that then separately pushes each of them to P_3.



Way 2: P_1 pushes 2 coins to P_2 that splits them as $1+1$ and pushes each of them separately pushes each of them to P_3.

Figure 2: Different ways in which a payment of 2 coins can be split along the path $P_1 \twoheadrightarrow P_2 \twoheadrightarrow P_3$.

The payment splitting can be done in an arbitrary way, except of two following restrictions. First of all, we do not allow are "loops" (i.e. paths that contain the same party more than once), as it is hard to imagine any application of such a feature. In the basic version of the protocol (described informally in Sec. 4.2 and then formally in Sec. 5.3) we assume that the number of times a given payment sub-payment is split by a single party P is bounded by a parameter $\alpha \in \mathbb{N}$, called *arity* of the Ethna (for example arity on Figs. 1 and 2 is at most 2). In Appx. B we present an improved protocol where α is unbounded (at a cost of a mild increase of the pessimistic number of rounds of interaction).

An important feature of Ethna is that it permits sub-receipt aggregation. In the process of acknowledging payments the receiver P_n signs several sub-receipts that are later sent to the sender P_1 via the paths over which they arrived to P_n. Each intermediary that received more than one sub-receipt can aggregate them into one short sub-receipt that she sends further to P_1. Finally, P_1 also produces one short receipt for the entire payment. This results in small communication complexity, and in particular, the pessimistic gas costs are low. We discuss this in more detail in Secs. 5.3.1 and 6.

Ethna *security properties* In the description in Sec. 3 we assumed that all the parties are behaving honestly. Like all the other PCNs, Ethna works also if the parties are malicious, and in particular, no honest party P can loose money, even if all the other parties are not following the protocol and are working against P. Formal security definition appears in Sec. 5.2 Let us now informally list the security requirements, which are quite standard, and hold for most PCNs (including Lightning).

The first property is called "fairness for the sender". To define it, note that as a result of payment μ (with timeout t), the total amount of coins that each party P has in the channel with other parties typically changes. Let $net_\mu(P)$ denote the amount of coins that P gained in all the channels. Of course $net_\mu(P)$ can be negative if P lost $-net_\mu(P)$ coins. We require that by the time t an honest P_i holds a receipt of a form

$$\text{"an amount } v \text{ of coins has been transfered from } P_1 \text{ to } P_n \text{ as a result of payment } \mu\text{"}, \tag{2}$$

with $v \leqslant u$. Moreover, under normal circumstances, i.e. when everybody is honest, v is equal to $-net_\mu(P_1)$ (i.e. the sum of the amounts that P_1 lost in the channels). In case some parties (other than P_1) are dishonest, the only thing that they can do is to behave irrationally, and let $v \geqslant -net_\mu(P_1)$, in which case P_1 holds a receipt for transferring *more* coins than she actually lost in the channels. Note that introducing receipts makes our model stronger than the models that have no receipts (e.g. [28]). This is because the "no receipts" settings makes sense only under the assumption that the sender and the receiver trust each other, and in particular the receiver is not corrupt (which is is a *stronger* security assumption that the one that we use in our paper). A receipt can be later used in another smart contract (e.g., a contract that delivers some digital goods whose amount depends on v). If the sender sends a large amount of coins in several installments. Finally, the receipts can serve as an evidence in front of the court of law. "Fairness for the receiver" is defined analogously, i.e.: if P_1 holds a receipt (2) then typically $v = net(P_n)$, and if some parties (other than P_n) are dishonest, then they can make $v \leqslant net_\mu(P_n)$. In other words, P_1 cannot get a receipt for an amount that is higher than what P_n actually received in the channels. Finally, we require that the following property called "balance neutrality for the intermediaries" holds: for every honest $P \in \{P_2, \ldots, P_{n-1}\}$ we have that $net_\mu(P_n) \geqslant 0$. Again: if everybody else is honest then we have equality instead of inequality.

4 Overview of Ethna protocol

After having described ETHNA's features, let us now present the main ideas behind the protocol itself (for a formal description of the construction see Sec. 5.3, and for an overview of the implementation see Sec. 6). Consider some payment μ. Each time after P_n receives some sub-payment v that reached it via some path $\Pi = P_1 \twoheadrightarrow P_{i_1} \twoheadrightarrow \cdots \twoheadrightarrow P_{i_t}$ it issues a "sub-receipt" and passes it to P_{n-1}. In order to keep the data and gas complexities low we need a mechanism to help the intermediaries to "aggregate" several such sub-receipts. One option for doing this would be to let the sub-receipt be signed using a homomorphic signature scheme, and then exploit this homomorphism to aggregate the sub-receipts. In this paper we use a simpler solution that can be efficiently and easily implemented in the current smart-contract platforms. Very informally speaking, we ask P_n to perform the "sub-payment" aggregation herself (this is done at the moment of signing a sub-receipt, and does not require any further interaction with P_n). Then, we just let the other parties verify that this aggregation was performed correctly. If any "cheating by P_n" is detected (i.e. some party discovers that P_n did not behave honestly) then a proof of this fact (called a "fraud proof") will count as a receipt that a full amount has been transferred to P_n. From the security point of view this is ok, since an honest P_n will never cheat (and hence, no "fraud proof" against him will ever be produced), and the security of a *dishonest* P_n is not of our concern. Thanks to this approach, we completely avoid using any expensive advanced cryptographic techniques (such as fully homomorphic signatures, or non-interactive proofs). Before we proceed to the description of ETHNA we need some more tools and definitions that we introduce in the next section.

4.1 Tools

4.1.1 Payment routes. Recall that ETHNA allows multiple transfers of the sub-payments of the same payment via the same party. It is convenient to distinguish different uses of the same path. This is done by adding nonces to the paths. Recall that the intermediaries can decide dynamically about the paths over which the sub-payments are routed and P_1 does not know how each path will be divided. To reflect this, we allow every party P_{i_j} on a path $P_1 \twoheadrightarrow P_{i_1} \twoheadrightarrow \cdots \twoheadrightarrow P_{i_t} \twoheadrightarrow P_n$ to contribute her own nonce to a sequence of nonces that identifies a particular sub-payment that goes along this path.

To capture this, we define payment *routes* that are similar to paths, except that they allow every party on a path to have its "own nonce" in it, and satisfy some additional simple constraints about the parties that appear on it. Formally, for a channel graph $\mathcal{G} = (\mathcal{P}, E, \Gamma)$ a string $\pi = \langle (P_{i_1}, \mu_1), \ldots, (P_{i_{|\pi|}}, \mu_{|\pi|}) \rangle$ is a *payment route over* $\mathcal{G}$ *for payment* μ if each μ_i is a nonce and $P_{i_1} \twoheadrightarrow \cdots \twoheadrightarrow P_{i_{|\pi|}}$ is a path in $\mathcal{G}$ that has at least two elements, its first element is equal to P_1, its last element is equal to P_n, and it has no loops (i.e. every element from $\mathcal{P}$ appears in π at most once). We assume that a payment route corresponding to a payment μ will always start with (P_1, μ) (hence $\mu_1 := \mu$). We say that P *appears on* π *(at position* j*)* if we have that $P = P_{i_j}$. A *payment route prefix (over $\mathcal{G}$)* is a string π' that is a prefix of some payment route over $\mathcal{G}$.

Let us look again at Fig. 2. The μ's on this figure indicate the nonces associated with the parties to which the arrows are pointing. The nonce of P_1 is missing there, since, as mentioned above it is equal to μ. For example the payment routes corresponding to "Way 1" on this figure are: $\langle (P_1, \mu), (P_2, \mu_2), (P_3, \mu_3) \rangle$ and $\langle (P_1, \mu), (P_2, \mu_2'), (P_3, \mu_3') \rangle$.

4.1.2 Payment trees.

Consider some fixed μ and $\mathcal{G}$. During the execution of ETHNA for $\mathcal{G}$ and μ, several sub-payments are delivered to P_n. Let $\pi^1, \ldots, \pi^t$ denote the consecutive paths over which these sub-payments go (of course they need to be distinct), and let $v^i \in \mathbb{Z}_{>0}$ be the amount of coins transmitted with each π^i. Let $\mathcal{R} := \{(\pi^i, v^i)\}_{i=1}^t$. We now show how $\mathcal{R}$ can be naturally compressed into a labeled tree that we call *payment tree* $\mathsf{tree}(\mathcal{R})$[2] (as we explain in a moment in ETHNA this compression is performed by P_n). Concretely, we define $\mathsf{tree}(\mathcal{R})$ as $(T, \mathcal{L})$, where T is the set of all prefixes of the π^i's, i.e., $T := \bigcup_i \mathsf{prefix}(\pi^i)$ (where $\mathsf{prefix}(\pi)$ denotes the set of prefixes of π). If ETHNA has arity α (see p. 9) then the arity of T in every node $\pi || (P, \mu)$ (for any *honest* P and any μ) is at most α. Then for every $\pi \in T$ we let $\mathcal{L}(\pi) := \sum_{i:\pi \in \mathsf{prefix}(\pi^i)} v^i$. In other words: every payment route prefix π gets labeled by the arithmetic sum of the value of the payments that were "passed through it". Obviously, the label $\mathcal{L}(\varepsilon)$ of the root node of $\mathsf{tree}(\mathcal{R})$ is equal to the sum of all v^i's, and hence it is equal to the total number of coins transferred by the sub-payments in $\mathcal{R}$. We also have that for every payment route prefix σ

$$\mathcal{L}(\sigma) = \sum_{\pi \text{ is a child of } \sigma} \mathcal{L}(\pi). \tag{3}$$

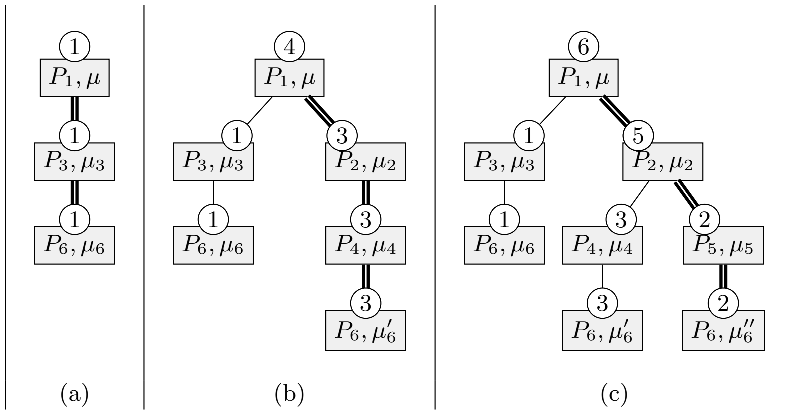
It is also easy to see that $\mathsf{tree}(\mathcal{R})$ can be constructed "dynamically" by processing elements of $\mathcal{R}$ one after another. More precisely, this is done as follows. We start with an empty tree Φ, and then iteratively apply the algorithm Add_Φ (see Alg. 1) for $(\pi^1, v^1), (\pi^2, v^2), \ldots$ (for a moment ignore the output of this algorithm). From the construction of the algorithm it follows immediately that if we start with Φ being an empty tree,

Algorithm 1: $\mathsf{Add}_\Phi(\pi, v)$

```
assumption: v ∈ Z_{>0} and π ∉ T
  This algorithm operates on a global state Φ = (T, L). Its side effect is a change of the global state.
for j = 1, ..., |π| do
    if π|_j ∈ T then
        let L(π|_j) := L(π|_j) + v
    else
        let T := T ∪ {π|_j} let L(π|_j) := v
output ⟨L(π[1]), ..., L(π_{|π|})⟩ (the labels on path π)
```

and then iteratively apply Add_Φ to (π^i, v^i)'s for $i = 1, \ldots, t$, then the final state of Φ is equal to $\mathsf{tree}(\mathcal{R})$. For example, if we apply this procedure to the situation on Fig. 1 (c) we get the trees depicted on Fig. 3. For a payment tree $\Phi = (T, \mathcal{L})$ and $\pi \in T$ define $\mathsf{labels}(\Phi, \pi)$ as the sequence (of length $|\pi|$) of all labels leading from the tree root to π, i.e., for every $i = 1, \ldots, |\pi|$ let $\mathsf{labels}(\Phi, \pi)[i] := \mathcal{L}(\pi|_i)$. For example on Fig. 3 we have: $\mathsf{labels}(\Phi, \pi^1) = \langle 1, 1, 1 \rangle$ and $\mathsf{labels}(\Phi, \pi^2) = \langle 4, 3, 3, 3 \rangle$ and c) $\mathsf{labels}(\Phi, \pi^3) = \langle 6, 5, 2, 2 \rangle$ (see the paths marked with double lines on this figure).

[2] In this paper we define trees as prefix-closed sets of words over some alphabet A. Formally, a *tree* is a subset T of A^* such that for every $\alpha \in T$ we have that any prefix of α is also in T. Any element of T is called a *node* of this tree. For two nodes $\alpha, \beta \in T$ such that $\beta = \alpha || a$ (for some a) we say that α is the *parent* of β, and β is a *child* of α. A *labeled tree over* A is a pair $(T, \mathcal{L})$, where T is a tree over A, and $\mathcal{L}$ is a function from T to some set of *labels*. For $\alpha \in T$ we say that $\mathcal{L}(\alpha)$ is the *label of* α.



First, (a) we apply algorithm Add_Φ to (π^1, v^1), where $\pi^1 = \langle (P_1, \mu), (P_3, \mu_3), (P_6, \mu_6) \rangle$ and $v^1 = 1$, then (b) we apply it to (π^2, v^2), where $\pi^2 = \langle (P_1, \mu), (P_2, \mu_2), (P_4, \mu_4), (P_6, \mu_6') \rangle$ and $v^2 = 3$, and finally (c) to (π^3, v^3), where $\pi^3 = \langle (P_1, \mu), (P_2, \mu_2), (P_5, \mu_5), (P_6, \mu_6'') \rangle$ and $v^3 = 2$. The last added path is indicated with a double line.

Figure 3: Illustration of the iterative application of Algorithm Add_Φ to the payment routes to from Fig. 1.

4.1.3 Sub-receipts, payment reports, and fraud proofs. For a graph $\mathcal{G}$ and a nonce μ a *sub-receipt (over $\mathcal{G}$, for payment μ)* is denoted $[\![\pi, \lambda]\!]$. It is a pair (π, λ) signed by P_n such that π is a payment route over $\mathcal{G}$ (for payment μ), and λ is a non-increasing sequence of positive integers, such that $|\lambda| = |\pi|$. A *payment report for* μ is a set $\mathcal{S}$ of sub-receipts for μ such that no two of them have the same π, i.e.: ($[\![\pi, \lambda]\!] \in \mathcal{S}$ and $[\![\pi, \lambda']\!] \in \mathcal{S}$) implies $\lambda = \lambda'$. For example, on Fig. 3 (c) set $\{[\![\pi^1, \langle 1,1,1 \rangle]\!], [\![\pi^2, \langle 4,3,3,3 \rangle]\!], [\![\pi^3, \langle 6,5,2,2 \rangle]\!]\}$ is a payment report. For a payment report $\mathcal{S}$ a sub-receipt $[\![(\pi || \sigma), \lambda]\!]$ is a *leader of* $\mathcal{S}$ *at* π if for every $[\![(\pi || \sigma'), \lambda']\!] \in \mathcal{S}$ we have that $\lambda[|\pi|] \geqslant \lambda'[|\pi|]$. In normal cases (i.e. if P_n is honest) the leader at π is always unique (and is equal to the *last* sub-receipt of a from $[\![(\pi || \sigma'), \lambda']\!]$ signed by P_n), however in general this does not need to be the case. When we talk about *the* leader of $\mathcal{S}$ at π we mean $[\![(\pi || \sigma), \lambda]\!]$ that is the smallest according to some fixed linear ordering. For example, on Fig. 3 (c) $(\pi^3, \langle 6,5,2,2 \rangle)$ is the leader of $\mathcal{S}$ at every prefix of π^3.

As already mentioned in Sec. 2.2 Ethna is constructed using "fraud proofs". We now define them, while postponing discussion on how they are used in the protocol until Sec. 4.2. A *fraud proof (for μ)* is a payment report $\mathcal{Q}$ for μ of a form $\mathcal{Q} = \{[\![(\sigma || \pi_i), \lambda_i]\!]\}_{i=1}^m$, all the $\pi_i[1]$'s are pairwise distinct[3], such that the following condition holds:

$$\max_{i:=1,\ldots,m} \lambda_i[|\sigma|] < \sum_{i:=1}^{m} \lambda_i[|\sigma| + 1] \tag{4}$$

(note that the left-hand side of Eq. (4) is equal to the value of $\lambda[|\sigma|]$, where $[\![\sigma || \pi, \lambda]\!]$ is the leader of $\mathcal{Q}$ at π). If Ethna has arity at most α (see Sec. 3) then we require that $m \leqslant \alpha$. Informally speaking Eq. (4) means simply that in $\mathcal{Q}$ the largest label of σ is at smaller than the sum of all labels of σ's children. If none of the subsets of a payment report $\mathcal{S}$ is a fraud proof then we say that $\mathcal{S}$ *is consistent.* We now have the following lemma.

Lemma 1. *Suppose a party P_n executes Add_Φ multiple times (for some payment μ, and starting from $\Phi = \emptyset$) and signs every output. Let $\mathcal{S}$ be the set of sub-receipts signed by party P_n during the execution of the Add_Φ algorithm. Then $\mathcal{S}$ is consistent.*

Proof. Take an arbitrary payment route prefix σ and an arbitrary set $\mathcal{Q} \subseteq \mathcal{S}$ that has a form $\mathcal{Q} = \{[\![(\sigma || \pi_i), \lambda_i]\!]\}_{i=1}^m$. Without loss of generality assume paths in $\mathcal{Q}$ are sorted according to the time by which

[3] In other words: the paths in $\mathcal{Q}$ form a tree with exactly one vertex π that has more than on child.

the paths in this set were signed (starting from the first). From the fact that in the Add algorithm the values in the labels can only increase we get that

$$\max_{i=1,\ldots,m} \lambda_i[|\sigma|] = \lambda_m[|\sigma|].$$

From (3) we know that the time when path $[\![(\sigma||\pi_m), \lambda_m]\!]$ was signed all the children on σ in the tree T were labeled by values that sum up to $\lambda_m[|\sigma|]$. The sum $\sum_{j:=1}^{m} \lambda_i[|\sigma|+1]$ is *at most* equal to this value. This is because (a) it is a *subset* of the set of all children of σ, and (b) these paths were signed *earlier* than when $[\![(\sigma||\pi_m), \lambda_m]\!]$ is signed (here we again use the fact that in the Add algorithm the values in the labels can only increase). Altogether we get that

$$\max_{i:=1,\ldots,m} \lambda_i[|\sigma|] \geqslant \sum_{i:=1}^{m} \lambda_i[|\sigma|+1],$$

and hence $\mathcal{Q}$ cannot be a fraud proof (cf. (4)). Therefore $\mathcal{S}$ does not have fraud proofs, and hence it is consistent. □

4.1.4 Size of the fraud proofs. Note that the description of set $\mathcal{Q}$ as defined above can be quite large (it is of size $O(\alpha \cdot (\ell + \kappa))$, where α is Ethna's arity, ℓ is the maximal length of payment routes, and κ is the security parameter (we need this to account for the signature size). Luckily, there is a simple way the "compress" it to $O(\alpha \cdot \kappa)$ (where κ is the security parameter) by exploiting the fact that the only values that are needed to prove cheating (cf. Eq. 4) are the positions on the indices $|\sigma|$ and $|\sigma| + 1$ of the λ's. Using the "bisection" method [16, 31], this can be reduced further to $O(\kappa)$ (hence: it can be made independent of the arity α) at a cost of $O(\log \alpha)$ rounds of interaction between the party that discovered cheating and P_n. We describe this in Appx. B.

4.2 The Ethna protocol

Let us now describe informally the Ethna protocol. In our description we make several simplifications, e.g., we ignore some special, but rare cases (like P_n not acknowledging some payments at all). For a more detailed description see Sec. 5.3 (and, in particular, Fig. 6 on p. 19). Let $\mathcal{G}$ be the channel graph. As already mentioned before, the main idea is to let the sender P_n perform the payment aggregation herself, and to "punish" her in case she tries to cheat. Cheating will be proven using the fraud proofs defined above. Of course, if P_n is honest then nobody can produce a valid fraud proof. Therefore the punishment for cheating can be arbitrarily severe. In our settings we simply let a fraud proof serve as a receipt (see Eq. (2)) that all the coins were transferred.

Going a bit more into the details, the protocol for every new payment μ starts with P_n declaring that she request a maximal transfer of u coins within this payment. This is done by P_n sending a signed pair $[\![\mu, u]\!]$ (called an *invoice*) to P_1. If later P_1 obtains a fraud proof $\mathcal{Q}$ for μ then $\mathcal{Q}$ will serve as a receipt that *all* the u coins were transferred in payment μ. This is ok, since (a) if P_n is honest then no fraud proof can be produced, and (b) the protocol is constructed in such a way that P_n never pushes more coins than u to her neighbors (within payment μ). Let us now provide some more information on how the coins are pushed and acknowledged.

4.2.1 Pushing payments. Initially no coins have been transfered within payment μ, so P_1 holds all u of them. Pushing payments is done in a recursive way. Suppose P holds some number v of coins that were pushed to P via some path π (in case $P = P_1$ this path is simply $\langle(P_1, \mu)\rangle$). Let t be the deadline until this payment has to be completed. Party P initiates a variable $\mathcal{S}^\pi$ that she will use for bookkeeping purposes. Variable $\mathcal{S}^\pi$ contains a payment report and is initially empty. Party P can now push some part v' of her v coins to a neighbor P' of hers. This is done by sending a push message in the state channel $P \multimap\!\!\!\circ\, P'$ and blocking v' coins of P in it. This message comes with a parameter $(\pi||(P', \mu'))$ (where μ' is some fresh nonce) and a deadline $t' < t - \Delta$ until when this payment has to be completed. As in the case of Lightning (see Sec. 2.1.4) this message is immediate since it imposes no commitments on P'. Before describing how the payments are acknowledged by the intermediaries, and how the final receipt is produced by P_1 let us present the procedure for the receiver P_n.

4.2.2 Payment acknowledgment by P_n. For every payment μ party P_n maintains a payment tree Φ^μ that is initially empty. Party P_n waits for push requests. Each such a message arrives from one of P_n's neighbors in $\mathcal{G}$ and is transmitted via some state channel $P \multimapboth P_n$ of $\mathcal{G}$. They all come with parameters π, v, and t, where π is a payment route (starting with (P_1, μ) and with P_n appearing as its last element), v is the number of pushed coins, and t is a timeout for this sub-payment. The receiver now decides on the number $v' \leqslant v$ of coins that she is willing to accept from this sub-payment. She then runs $\mathsf{Add}_{\Phi^\mu}(\pi, v')$. Recall that this results in updating state Φ^μ and producing an output λ (equal to the labels on path π *after* updating the state). Party P_n acknowledges the sub-receipt of v' coins by sending a signed pair $[\![\pi, \lambda]\!]$ back to the state channel $P \multimapboth P_n$, and claims v' coins from the amount locked in $P \multimapboth P_n$ by P. As in Lightning, this message is not immediate. Party P learns $[\![\pi, \lambda]\!]$ within 1 hour. Observe that from Eq. (3) we get that $\lambda[1]$ is equal to the sum of all the coins that were so far transmitted to P_n within payment μ. Note also that from Lemma 1 we get that as long as P_n is honest Φ^μ is consistent, and hence no fraud proof can be produced.

4.2.3 Payment acknowledgment by the intermediaries. Let us now go back to party P that pushed some coins to P' via channel $P \multimapboth P'$ and waits receive acknowledgment from P' (via the same channel). For a moment suppose the P is an intermediary, i.e. $P \in \{P_2, \ldots, P_{n-1}\}$ (the protocol for $P = P_1$ is described below). Let P'' be the party that earlier pushed v coins to P. In the most likely case P receives some $[\![\phi, \lambda]\!]$ (with π being a prefix of ϕ). In this case she adds $[\![\phi, \lambda]\!]$ to $\mathcal{S}^\pi$. The state channel is constructed in such a way that $\phi[|\pi| + 1]$ coins (from those that were locked by P) are transfered to P', while the rest goes back to P. Once P wants to acknowledge payment π (this can only happen if there are no open push request for sub-payments of π) she looks at $\mathcal{S}^\pi$. If it is consistent then she finds the leader of at this set at π (see Sec. 4.1.3). Let $[\![(\pi||\widehat{\sigma}), \widehat{\lambda}]\!]$ be this leader. Party P acknowledges π by sending back $[\![(\pi||\widehat{\sigma}), \widehat{\lambda}]\!]$ to $P'' \multimapboth P$. At the same time she claims $\lambda[|\pi|]$ coins from the coins locked in this channel. Observe that since $\mathcal{S}^\pi$ is consistent, thus $\lambda[|\pi|]$ is at least as large as the sum $\sum_{[\![(\pi||\sigma),\lambda]\!] \in \mathcal{S}^\pi} \lambda[|\pi| + 1]$, and this sum is exactly equal to the total number of coins that P "payed" to the parties to which she pushed this payment. Hence she never looses money.

The second option is that $\mathcal{S}^\pi$ is inconsistent. Let w be the fraud proof. Party P simply sends w back to P'' (over the channel $P'' \multimapboth P$). Think of it as "throwing an exception" in recursive application of "pushing" procedure. In some sense w is a "wild card" that allows to claim *all* the v coins that were pushed to a party that presents it. Since it works "universally" no honest party looses money. In particular, although P'' has to accept that all the coins were "transferred" to P, she can later use the same w to claim all the coins that were blocked by the party that pushed this payment to her.

4.2.4 Receipt by the sender. For the sender P_1 the protocol works similarly, except that P_1 does not "push" messages back, but simply outputs them as a receipt. More precisely if $\mathcal{S}^{(P_1,\mu)}$ is consistent and no fraud proof have been received, then let $[\![\widehat{\phi}, \widehat{\lambda}]\!]$ be the leader of $\mathcal{S}^{(P_1,\mu)}$. In this case case party P_1 concludes that $\widehat{\lambda}[1]$ coins were transferred, and $[\![\widehat{\phi}, \widehat{\lambda}]\!]$ is the receipt. Otherwise let w be the fraud proof. Then P_1 concludes that all u coins were transferred and a pair $(w, [\![\mu, u]\!])$ is the receipt (the "$[\![\mu, u]\!]$" component is needed to demonstrate what was the maximal transmitted value that P_n agreed for).

5 Formal details

In this section we present the formal details of our construction. We start with describing the model (in Sec. 5.1). Then (in Sec. 5.2) we state the security definition. Finally, in Sec. 5.3 we show our protocol and outline the security analysis.

5.1 Our Model

We assume that the protocol is attacked by a polynomial-time adversary Adv who can *corrupt* some parties (when a party is corrupt Adv learns all its secrets and takes a full control over it). The adversary can also send messages to the honest parties that influence their behavior in the protocol, and receive messages from them. A party that has not been corrupt is called *honest*. As highlighted in the introduction some pairs of parties are connected by state channels. Let $\mathcal{G} = (\mathcal{P}, \mathcal{E}, \Gamma)$ be a channel graph. We assume that every edge $e = P_i \multimapboth P_j \in \mathcal{E}$ has a corresponding "*state channel*" on a blockchain. This state channel is modeled by a machine denoted C^e. We assume that C^e has two special registers denoted $C^e.\mathsf{cash}(P_i)$ and $C^e.\mathsf{cash}(P_j)$. The values in these registers are non-negative integers, and $C^e.\mathsf{cash}(P)$ denotes the amount of coins that $P \in e$ has in her *account in* C^e. We stress that $C^e.\mathsf{cash}$ will also be viewed as a function $C^e.\mathsf{cash} : e \to \mathbb{Z}_{\geqslant 0}$. At the beginning of the protocol Adv chooses the set of edges $\mathcal{E}$, and for every $e \in \mathcal{E}$ she preloads it with some coins (i.e. determines the values of the function $C^e.\mathsf{cash}$). Such modeling of state channels is made only for the sake of simplicity. In every real-life implementation state channels are not "machines", but are replaced by protocols (see, e.g., [8]) that emulate such machines.

We assume a synchronous communication network, i.e., the execution of the protocol happens in rounds. The notion of rounds is just an abstraction which simplifies our model, and was used frequently in this area in the past (see, e.g., [8, 10]). Whenever we say that some operation (e.g. sending a message or simply staying in idle state) *takes at most* $\tau \in \mathbb{N} \cup \{\infty\}$ *rounds* we mean that it is up to the adversary to decide how long this operation takes (as long as it takes at most τ rounds). We assume that every machine is activated in each round. The communication between each two parties P_i and P_j takes 1 round. Sending the "immediate messages" (see Sec. 2.1) sent by P's to a state channel $P \multimapboth P'$ also take 1 round. The "non-immediate" messages take two rounds if both P and P' are honest (since P' needs to send back her signature to P). If one of them is dishonest, then this can take up to Δ rounds, where $\Delta \gg 1$ is a constant that depends on the blockchain finality.

5.2 Security definition of DIPS

Let us now present the formal security definition. A *Dynamic Internal Payment Splitting (DIPS) protocol* Π *for a channel graph* $\mathcal{G} = (\mathcal{P}, \mathcal{E}, \Gamma)$ consists of: *party machines* $P_1, \ldots, P_n$, the *state channel machines* C^e (for every $e \in \mathcal{E}$), and *receipt verification machine* RVM. The role of RVM is to model the fact that the receipts produced by P_1 need to be publicly-verifiable (so, e.g., they can be used later in another smart contract, see Sec. 3). We stress that the RVM has very limited interaction with the other machines. In fact, the only interaction that happens is: P_1 sends a message to RVM, and RVM decides if it is a valid receipt and outputs information on how many coins were transferred within a given payment. Hence, we can think of RVM as an efficiently computable ("non-interactive") function. We assume that RVM has memory and outputs a receipt for given payment μ at most once. This is a purely syntactic choice that makes our security definition slightly simpler. To model the fact that the parties can make internal decisions about the protocol actions we use a concept of an *environment* [3] that is responsible for "orchestrating" the execution. We model it by a poly-time interactive machine $\mathcal{Z}$. The party machines interact with the environment $\mathcal{Z}$ via messages starting with "env-" prefix. The environment sends the following messages to the parties: "env-send" and "env-receive" — sent simultaneously to P_1 and P_n (respectively) and used to initiate a payment μ, messages "env-push" — to push sub-payments further, and messages "env-acknowledge" — to acknowledge a payment. The parties respond with messages "env-pushed" — to signal that a sub-payment was pushed.[4]For reference, these messages and their syntax are summarized on a cheat sheet on Fig. 5 (see p. 17).

[4] Under normal circumstances, if $\mathcal{Z}$ asked to P to push a sub-payment to P' then in the next round she will receive an "env-pushed" message from P'. Of course, this does not need to be the case when P or P' are corrupt. Interestingly, it is even possible that "env-pushed" was sent to $\mathcal{Z}$ even though no corresponding "env-push" message was sent. For example nothing prevents a corrupt P to behave "irrationally" and "push" to P' some sub-payment that does not correspond to any payment μ. In this case P' has no way to discover that this fact, and may continue pushing sub-payments further. Our modeling takes such irrational behavior into account, i.e., we allow $\mathcal{Z}$ to push such "fake sub-payments" further, and our security proof guarantees that no coins of honest parties are lost.

$\mathcal{Z}$ takes as input a channel graph $\mathcal{G} = (\mathcal{P}, \mathcal{E}, \Gamma)$, where Γ will be treated as a variable that will be changing throughout the execution of $\mathcal{Z}$ (the values $\mathcal{P}, \mathcal{E}$ will remain constant). It also defines the following variables:

- Ω — a set of payments routes (initially empty) called the *open* push *requests*. When we say that we *open a* push *request* π, we mean that we add π to Ω. When we say that we *close a* push *request* π we mean that we remove π from Ω.
- $\mathit{sent}, \mathit{value}, \mathit{timeout}$ — functions of a type $\Omega \to \mathbb{Z}_{\geqslant 0}$.

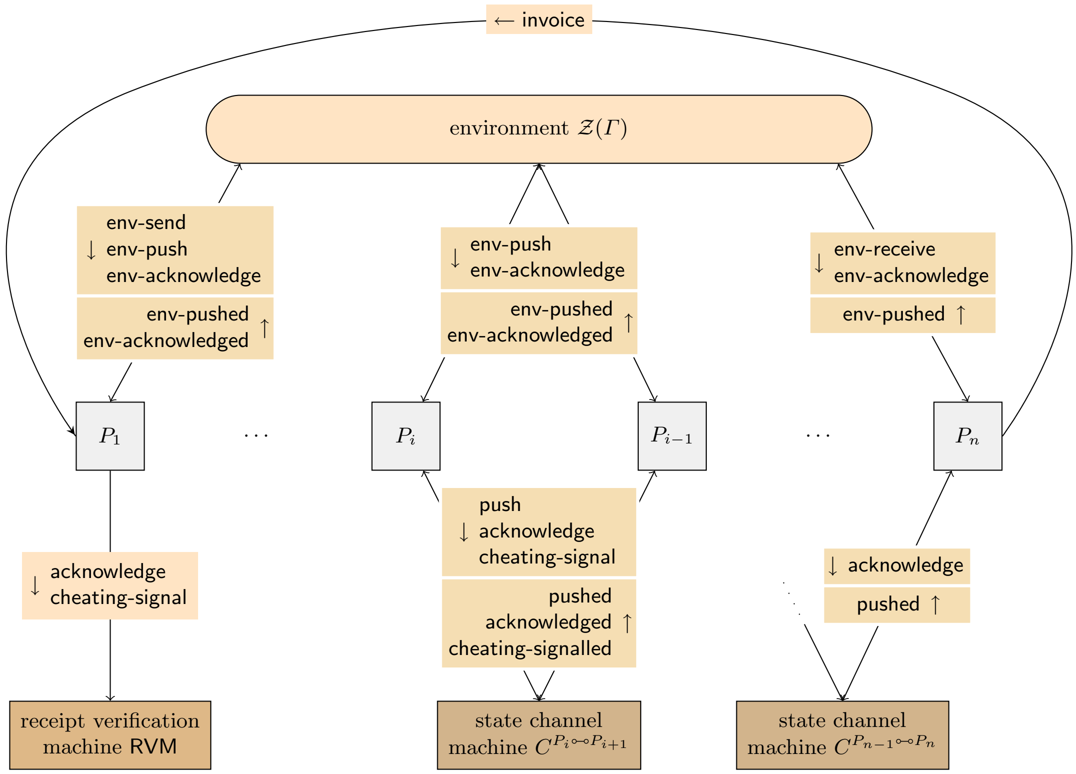
The environment $\mathcal{Z}$ interacts with the parties in an arbitrary way, as long as certain restrictions are satisfied. There are no restrictions on the messages that $\mathcal{Z}$ receives. Instead we specify "conditions" on when a message is valid (if they are not met, then the message is ignored). Both the outgoing and incoming messages can result in modifications of the variables (we call these modifications the "side effects").

- $\mathcal{Z}$ sends a message $(\textsf{env-send}, v, \mu, t)$ to P_1 and a message $(\textsf{env-receive}, v, \mu, t)$ to P_n.
  Restrictions: (a) these messages have to be sent simultaneously, (b) the nonce μ has not been used before in the env-send and env-receive messages.
- $\mathcal{Z}$ receives a message $(\textsf{env-pushed}, (\pi||(P, \mu)), v, t)$ from a party P.
  Restrictions: $t > \tau$, where τ is the current time.
  Side effects: open a push request $(\pi||(P, \mu))$ and let $\mathit{sent}((\pi||(P, \mu))) := 0$ and $\mathit{value}((\pi||(P, \mu))) := v$ and $\mathit{timeout}((\pi||(P, \mu))) := t$.
  We require that in time t the latest $\mathcal{Z}$ sends a message $(\textsf{env-acknowledge}, (\pi||(P, \mu)))$ (see below) to party P.
- $\mathcal{Z}$ receives a message $(\textsf{env-acknowledged}, (\pi||(P, \mu)||(P', \mu')), v)$ from a party P.
  Condition: a push request $(\pi||(P, \mu)||(P', \mu'))$ is open.
  Side effects: let $\mathit{sent}((\pi||(P, \mu))) := \mathit{sent}(\pi||(P, \mu)) + v$ and $\Gamma^{P \multimap\!\!\circ P'}(P) := \Gamma^{P \multimap\!\!\circ P'}(P) + v' - v$ and $\Gamma^{P \multimap\!\!\circ P'}(P') := \Gamma^{P \multimap\!\!\circ P'}(P') + v$, where $v' := \mathit{value}((\pi||(P, \mu)||(P', \mu'))$.
- $\mathcal{Z}$ sends a message $(\textsf{env-push}, (\pi||(P, \mu)||(P', \mu')), v, t)$ to a party P.
  Restrictions: (a) no env-push request with the same argument $(\pi||(P, \mu)||(P', \mu'))$ has been sent before, (b) a push request $(\pi||(P, \mu))$ is open, (c) at most $\alpha - 1$ requests env-push with the argument $(\pi||(P, \mu))$ have been sent before, (d) $t \leqslant \mathit{timeout}(\pi||(P, \mu)) - \Delta$, (e) $v \leqslant \mathit{value}(\pi||(P, \mu)) - \mathit{sent}(\pi||(P, \mu))$, and (f) $v \leqslant \Gamma^{P \multimap\!\!\circ P'}(P)$.
  Side effects: let $\Gamma^{P \multimap\!\!\circ P'}(P) := \Gamma^{P \multimap\!\!\circ P'}(P) - v$.
- $\mathcal{Z}$ sends a message $(\textsf{env-acknowledge}, (\pi||(P, \mu)||(P', \mu')))$ to a party P'.
  Restriction: a push request $(\pi||(P, \mu)||(P', \mu'))$ is open. No push request $(\pi||(P, \mu)||(P', \mu')||\pi')$ (for $\pi' \neq \epsilon$) is open.
  Side effects: close this push request.

Figure 4: Admissible $\mathcal{Z}$ for ETHNA with arity α.

We assume that the environment gets the channel graph $\mathcal{G}$ as input. Each state channel machine C^e (where $e = \left(P^i \multimap\!\!\circ P^j\right) = \{P^i, P^j\}$ — recall that we think of edges as two-element sets) gets her cash function Γ^e pre-loaded at startup, i.e., its special registers are set to $C^e.\mathsf{cash}(P) := \Gamma^e(P)$ (for both $P \in \{P^i, P^j\}$). The environment $\mathcal{Z}$ is called *admissible* if it satisfies certain criteria presented on Fig. 4. It maintains a set Ω of "open push requests" (see Fig. 4) and functions $\mathit{sent}, \mathit{value}$, and $\mathit{timeout}$ that are used to store information about these requests. We say that a party P has an *open* push request if there exists $\pi \in \Omega$ such that P appears on π. The main idea behind the "admissible environment" is that it restricts us to the environments that satisfy some natural correctness requirements, such as "do not push more coins than you hold in a given sub-payment". These conditions are called "restrictions". Most of the actions result in some modification of the internal variables of $\mathcal{Z}$. These restrictions are called "side effects". The environment is also responsible for terminating the protocol. More concretely, we say that the protocol *terminated* is $\mathcal{Z}$ stops. The environment can only do it if there are no open push requests. The notion of termination is useful when we define security, since it allows us to capture the fact that we only care about the final balance of the parties (it is ok for a party to temporarily "loose" some coin as long as eventually she gets them back).

The protocol is attacked by a poly-time adversary Adv and operates in a communication model described in Sec. 5.1. We assume that the adversary can actively corrupt are the party machines $P_1, \ldots, P_n$, and no other parties. Suppose some execution was performed and terminated. Let $\widehat{\Gamma}$ be a cash function describing the



Message syntax

Types of variables

- v — a positive integer denoting amounts of coins,
- μ — a nonce,
- π — payment path prefix over $\mathcal{G}$, and
- t — time.

Messages sent and received by $\mathcal{Z}$

The environment $\mathcal{Z}$ sends the following messages to the parties:

- $(\mathsf{env\text{-}send}, v, \mu, t)$ (this message is sent only to P_1),
- $(\mathsf{env\text{-}receive}, v, \mu, t)$,
- $(\mathsf{env\text{-}push}, \pi, v, t)$, and
- $(\mathsf{env\text{-}acknowledge}, \pi)$.

The environment $\mathcal{Z}$ also receives the following messages from the parties:

- $(\mathsf{env\text{-}pushed}, \pi, v, t)$, and
- $(\mathsf{env\text{-}acknowledged}, \pi, v)$.

Messages exchanged between the parties

Party P_n sends to party P_1 a message:

- $(\mathsf{invoice}, [\![\mu, u]\!], t)$.

Messages exchanged between the parties and the state channel machines

The parties send the following messages to the state channel machines:

- $(\mathsf{push}, \pi, v, t)$,
- $(\mathsf{acknowledge}, R)$, where R is either equal to (π, empty) (where "empty" is a keyword) or it is equal to $[\![\psi, \lambda]\!]$, where (ψ, λ) is a sub-receipt over $\mathcal{G}$, and
- $(\mathsf{cheating\text{-}signal}, w)$, where w is an fraud proof,

The state channel machines send the following messages to the parties:

- $(\mathsf{acknowledged}, R)$, where R is as above, and
- $(\mathsf{cheating\text{-}signalled}, w)$, where w is an fraud proof.

Messages send by P_1 to RVM

Party P_1 sends the following messages to the receipt verification machine RVM:

- $(\mathsf{acknowledged}, R)$, where R is as above, and
- $(\mathsf{cheating\text{-}signal}, w, [\![\mu, u]\!])$, where w is a fraud proof.

Figure 5: The flow of messages exchanged in the system, and their syntax.

amount of coins in the state channels after this execution, i.e, let every $\widehat{\Gamma}(e)$ be equal to a function $f : e \to \mathbb{Z}_{\geqslant 0}$ such that $f(P) := C^e.\mathsf{cash}(P)$. Now, look at this execution from a perspective of some party machine P. Let $\mathcal{U}$ be the set of all parties that have a channel with P, i.e., let $\mathcal{U} = \{P' : \text{ such that } (P \multimap P') \in \mathcal{E}\}$. The *net result of* P in this execution (so far) is defined as $net(P) := \sum_{P' \in \mathcal{U}} \widehat{\Gamma}^{P \multimap P'}(P) - \Gamma^{P \multimap P'}(P)$. This can be extended to the state channels, namely, the *net result of channel* e in this execution (so far) is defined as $net(e) := \sum_{P \in e} \widehat{\Gamma}^e(P) - \Gamma^e(P)$. Let us also define the *total transmitted* sum of coins until this moment as $\sum_{(\mu,v) \in \mathcal{W}} v$, where $\mathcal{W}$ is the set of outputs of RVM. We have the following *functionality* requirements that must hold for every DIPS protocol with overwhelming probability. The reason for having them is that we want to avoid trivial protocols (e.g. a protocol that does not perform any action) that would trivially satisfy the security requirements. *Guaranteed sending:* Suppose P_1 and P_n are honest, and they both simultaneously receive messages $(\mathsf{env\text{-}send}, v, \mu, t)$ and $(\mathsf{env\text{-}receive}, v, \mu, t)$ (respectively) from $\mathcal{Z}$. Then in the next round P_1 sends $(\mathsf{env\text{-}pushed}, (P_1, \mu), v, t)$ to $\mathcal{Z}$. *Guaranteed pushing:* Suppose P and P' are honest, and P receives a message $(\mathsf{env\text{-}push}, (\pi||(P, \mu)||(P', \mu')), v, t)$ from $\mathcal{Z}$ (for some π, v, t, μ, and μ'). Then in the next round P' sends a message $(\mathsf{env\text{-}pushed}, (\pi||(P, \mu)||(P', \mu')), v, t)$ to $\mathcal{Z}$. This is the only case when P' sends an env-pushed message to $\mathcal{Z}$ with this route prefix. *Guaranteed acknowledgment by $P' \in \{P_2, \ldots, P_n\}$:* Suppose P and P' are honest, and P' receives a message $(\mathsf{env\text{-}acknowledge}, (\pi||(P, \mu)||(P', \mu')))$ from $\mathcal{Z}$ (for some π, μ and μ'), and let $v := sent((\pi||(P, \mu)||(P', \mu'))$. Then in the next round P sends a message $(\mathsf{env\text{-}acknowledged}, (\pi||(P, \mu)||(P', \mu')), v)$ to $\mathcal{Z}$. This is the only case when P sends an env-pushed message to $\mathcal{Z}$ with this route prefix. *Guaranteed acknowledgment by P_1:* Suppose P_1 is honest and it receives a message $(\mathsf{env\text{-}acknowledge}, (P_1, \mu))$ from $\mathcal{Z}$ (for some μ). Let $v := sent((P_1, \mu))$. Then in the next round the receipt verification machine outputs (v, μ).

We have the following *security* requirements that must hold with overwhelming probability after the execution terminates. *Fairness for the sender P_1:* Suppose that P_1 is honest and has no open push request. Then $net(P_1) + v \geqslant 0$. *Fairness for the receiver P_n:* Suppose that P_n is honest. Then $net(P_n) - v \geqslant 0$. *Balance neutrality of the intermediaries:* Suppose that $P \in \{P_2, \ldots, P_{n-1}\}$ is honest and has no open push request, then $net(P) \geqslant 0$. *"No money printing" in the state channel machines:* For every channel $P \multimap P'$ we have that $net(P \multimap P') \leqslant 0$.

5.3 The protocol and its analysis

Ethna has been described informally in Sec. 4.2. Its formal description appears on Figs. 6 and 7, and it should be rather self-explanatory, given the description form Sec. 4.2. Let us only comment on the types of messages that are sent within the protocol (see also the cheat sheet on Fig. 5 on p. 17 in the appendix). The parties communicate with each other only via the state channels (except of the first "invoice" message sent from P_n to P_1). The messages that are used are: "push" to push a sub-payment (the corresponding message sent by the channel to the other party is "pushed"), "acknowledge" to acknowledge a sub-payment (the corresponding message is "acknowledged"), and "cheating-signal" to signal fraud (the corresponding message is "cheating-signalled"). The messages sent by P_1 to the RVM are either "acknowledge" (if everything went ok), or "cheating-signalled" (if cheating by P_n was detected). While presenting the protocol informally, we already argued about its security. To be more formal, we now state the following.

Lemma 2. *Assuming that the underlying signature scheme is existentially unforgeable under a chosen message attack, the* Ethna *is a Dynamic Internal Payment Splitting (DIPS) protocol.*

Proof. We need to show that the functionality and security requirements from Sec. 5.2 hold in presence of an arbitrary adversary Adv and any admissible $\mathcal{Z}$.

The functionality requirements follows easily from the construction of the protocol. Let us now argue about the security requirements. We start with showing the balance neutrality for the intermediaries. Suppose an honest party $P_i \in \{P_2, \ldots, P_{n-1}\}$ starts a $\mathsf{handle\text{-}route}(\pi, v, t)$ procedure (see Fig. 6 (a)), and let P be the last element of π. During this execution it initiates a number of handle-push procedures. Let us look at the execution of some $\mathsf{handle\text{-}push}((\pi||(P', \mu)), v', t')$. At the beginning P_i sends a message $(\mathsf{env\text{-}push}, (\pi||(P', \mu)), v', t')$ to $C^{P_i \multimap P'}$. As a result, $C^{P_i \multimap P'}$ removes v coins from P_i's account (see Fig. 7 (b)).

(a) The parties

Party P_1

Wait to receive messages ($\mathsf{env\text{-}send}, u, \mu, t$) from the environment $\mathcal{Z}$. Handle each such a message as follows.

If in the next round you receive a signed message ($\mathsf{invoice}, [\![\mu, u]\!], t$) from P_n then store it, and execute the *route handling procedure* procedure $\mathsf{handle\text{-}route}(\langle(P_1, \mu)\rangle, v, t)$ defined as follows (since this procedure is also used for for parties $P_{>1}$ we allow it to take slightly more general input):

$\mathsf{handle\text{-}route}(\pi, v, t)$

Let $\mathcal{S}^\pi$ be a variable containing a set of sub-receipts that initially is empty, send ($\mathsf{env\text{-}pushed}, \pi, v$) to $\mathcal{Z}$ and wait for the following messages from $\mathcal{Z}$:

- ($\mathsf{env\text{-}push}, (\pi||(P', \mu')), v', t'$) — handle each such a message by executing the following *push handling procedure*:

  $\mathsf{handle\text{-}push}((\pi||(P', \mu')), v', t')$

  Send a message ($\mathsf{push}, (\pi||(P', \mu')), v', t'$) to $C^{P_i \multimap\!\circ P'}$, and wait to receive one of the following messages from $C^{P_i \multimap\!\circ P'}$:
  - ($\mathsf{acknowledged}, (\pi||(P', \mu')), \mathsf{empty}$) — then send a message ($\mathsf{env\text{-}acknowledged}, (\pi||(P', \mu')), 0$) to $\mathcal{Z}$,
  - ($\mathsf{acknowledged}, [\![\psi, \lambda]\!]$), where ψ is such that $(\pi||(P', \mu'))$ is a prefix of ψ — then store $[\![\psi, \lambda]\!]$ in $\mathcal{S}^\pi$ by letting $\mathcal{S}^\pi := \mathcal{S}^\pi \cup \{[\![\psi, \lambda]\!]\}$. Let $\widehat{v} := \lambda[|\pi| + 1]$. Send ($\mathsf{env\text{-}acknowledged}, (\pi||(P', \mu')), \widehat{v}$) to $\mathcal{Z}$, and
  - ($\mathsf{cheating\text{-}signalled}, w$) — then store w and send a message ($\mathsf{env\text{-}acknowledged}, (\pi||(P', \mu')), v'$) to $\mathcal{Z}$.

  Then end the $\mathsf{handle\text{-}push}$ procedure.

- ($\mathsf{env\text{-}acknowledge}, \pi$) — do the following
  - If you stored ($\mathsf{cheating\text{-}signalled}, w$) (for some (P', μ')) or if $\mathcal{S}^\pi$ is inconsistent and w is the fraud proof — then output ($\mathsf{cheating\text{-}signal}, w$).
  - Otherwise: if $\mathcal{S}^\pi$ is empty then output ($\mathsf{acknowledge}, \pi, \mathsf{empty}$).
  - Otherwise let $[\![\psi, \lambda]\!]$ be the leader of $\mathcal{S}^\pi$ at π. Output ($\mathsf{acknowledge}, \pi, [\![\psi, \lambda]\!]$).

  After producing the output end the $\mathsf{handle\text{-}route}$ procedure.

If the output of $\mathsf{handle\text{-}route}(\langle(P_1, \mu)\rangle, v, t)$ is ($\mathsf{cheating\text{-}signal}, w$) then send ($\mathsf{cheating\text{-}signal}, w, [\![\mu, u]\!]$) to RVM. Otherwise (i.e. if it was a "$\mathsf{acknowledge}$" message) simply send this output to RVM.

Party $P_i \in \{P_2, \dots, P_{n-1}\}$

Wait to receive messages ($\mathsf{pushed}, \pi, v, t$) from some $C^{P \multimap\!\circ P_i}$. Handle each such a request by the route handling procedure $\mathsf{handle\text{-}route}(\pi, v, t)$ described above. After this procedure terminates send its output back to $C^{P \multimap\!\circ P_i}$.

Party P_n

Wait to receive messages ($\mathsf{env\text{-}receive}, u, \mu, t$) from the environment $\mathcal{Z}$. Handle each such a request as follows.

1. Send a message ($\mathsf{invoice}, [\![\mu, u]\!], t$) to P_1. Let $\mathcal{S}^\mu$ be a variable containing a payment report that initially is empty,
2. Wait to receive messages ($\mathsf{pushed}, \pi, v, t$) from some $C^{P \multimap\!\circ P_n}$. Handle each such a message as follows. Once you receive it send ($\mathsf{env\text{-}pushed}, \pi, v, t$) to $\mathcal{Z}$ and wait to receive ($\mathsf{env\text{-}acknowledge}, \pi, v'$) from $\mathcal{Z}$. Once this happens, execute $\mathsf{Add}_{\mathcal{S}^\mu}(\pi, v')$. Let $[\![\pi, \lambda]\!]$ be the output of this procedure. Send a message ($\mathsf{acknowledge}, [\![\pi, \lambda]\!]$) to $C^{P \multimap\!\circ P_n}$.

Figure 6: The ETHNA protocol (the algorithms of the parties)

(b) The state channel machine $C^{P_i \multimap\!\!\circ P_j}$

Recall that the values of registers $C^{P_i \multimap\!\!\circ P_j}.\mathsf{cash}(P_i)$ and $C^{P_i \multimap\!\!\circ P_j}.\mathsf{cash}(P_j)$ were pre-loaded before the execution started.
Wait for the messages $(\mathsf{push}, (\pi||(P,\mu)||(P',\mu')), v, t)$ from $P \in P_i \multimap\!\!\circ P_j$ (where $P' := \mathsf{other\text{-}party}(P)$) such that (a) $t \leqslant \tau + \Delta$ (where τ is the current time), (b) $(\pi||(P,\mu)||(P',\mu'))$ is a payment route prefix, (c) $v \leqslant C^{P_i \multimap\!\!\circ P_j}.\mathsf{cash}(P)$, and (d) you have not previously received a push request with the same parameters. Upon receiving such a message let $C^{P_i \multimap\!\!\circ P_j}.\mathsf{cash}(P) := C^{P_i \multimap\!\!\circ P_j}.\mathsf{cash}(P) - v$, and send $(\mathsf{pushed}, (\pi||(P,\mu)||(P',\mu')), v, t)$ to (P',μ'). Then wait until one of the following happens:

- you receive a message $(\mathsf{acknowledge}, [\![\psi, \lambda]\!])$ from (P',μ') where ψ is a route with a prefix $(\pi||(P,\mu)||(P',\mu'))$ — then let $\widehat{v} := \lambda[|\pi|+2]$. Let $C^{P_i \multimap\!\!\circ P_j}.\Gamma(P') := C^{P_i \multimap\!\!\circ P_j}.\Gamma(P') + \widehat{v}$, and $C^{P_i \multimap\!\!\circ P_j}.\Gamma(P) := C^{P_i \multimap\!\!\circ P_j}.\Gamma(P) + v - \widehat{v}$, then send $(\mathsf{acknowledged}, [\![\psi, \lambda]\!])$ to P,
- you receive a message $(\mathsf{cheating\text{-}signal}, w)$ from P' where w is an fraud proof — then let $C^{P_i \multimap\!\!\circ P_j}.\Gamma(P') := C^{P_i \multimap\!\!\circ P_j}.\Gamma(P') + v$ and forward this message to P,
- time t comes — then let $C^{P_i \multimap\!\!\circ P_j}.\mathsf{cash}(P) := C^{P_i \multimap\!\!\circ P_j}.\mathsf{cash}(P) + v$ and send a message $(\mathsf{acknowledged}, (\pi||(P,\mu)||(P',\mu')), \mathsf{empty})$ to P.

(c) The receipt verification machine RVM

Wait for one of the following messages from P_1:

- $(\mathsf{acknowledge}, ((P_1,\mu), \mathsf{empty}))$ — then output $(\mu, 0)$.
- $(\mathsf{acknowledge}, [\![((P_1,\mu)||\psi), (v||\lambda)]\!])$ — then output (μ, v).
- $(\mathsf{cheating\text{-}signal}, w, [\![\mu, u]\!])$, where w is a fraud proof for some route σ, and ν is the nonce in the first element of σ — then output (μ, u).

For a given μ output a pair that contains it only once (i.e. after outputting (μ, v) ignore all the future calls that would lead to outputting (μ, v') for some v').

Figure 7: The Ethna protocol (the algorithms for the state channel machine and th recepit verification machine)

From the construction of the state channel machine it is clear that in time $t' + \Delta$ the latest party P receives one of the following messages back from $C^{P_i \multimap\!\!\circ P'}$ (each of them results in transferring back to her account in $C^{P_i \multimap\!\!\circ P'}$ some amount z of coins):

- a message $(\mathsf{acknowledged}, (\pi||(P',\mu')), \mathsf{empty})$ — in this case $z = v$,
- a message $(\mathsf{acknowledged}, [\![\psi, \lambda]\!])$ (where π is a prefix of ψ) — in this case z is equal to the last element of λ.
- a message $(\mathsf{cheating\text{-}signalled}, w)$ — in this case $z = 0$.

Call $(v - z)$ the *coins gained by* P_i *in effect of the* $\mathsf{handle\text{-}push}$ *procedure* and denote it with $\mathit{gained}_{P_i}(\pi)$.

The $\mathsf{handle\text{-}route}(\pi, v, t)$ procedure ends when P_i receives a message $(\mathsf{env\text{-}acknowledge}, \pi)$ from $\mathcal{Z}$ (from the construction of $\mathcal{Z}$ it follows that this message must be sent by $\mathcal{Z}$ in time t the latest). Once this happens, party P_i sends one of the following messages to $C^{P \multimap\!\!\circ P_i}$ (each of them results in transferring to her account in $C^{P_i \multimap\!\!\circ P_i}$ some amount of y coins, see Fig. 7 (b)):

- a message $(\mathsf{cheating\text{-}signal}, w)$ — in this case $y = v$,
- a message $(\mathsf{acknowledge}, \pi, \mathsf{empty})$ — in this case $y = 0$, or
- a message $(\mathsf{acknowledged}, [\![\psi, \lambda]\!])$ — in this case $y = \widehat{v}$, where $\widehat{v}$ is equal to the last element of $\lambda[|\pi|+1]$.

We will call y the *coins lost by* P_i *in effect of the* $\mathsf{handle\text{-}push}$ *procedure* and denote it with $\mathit{lost}_{P_i}(\pi)$.

Claim. For every honest $P \in \{P_2, \ldots, P_{n-1}\}$ let π be some payment route such that a $\mathsf{handle\text{-}route}(\pi, v, t)$ procedure has been executed (for some v and t), and let Π be the set of all payment routes $(\pi||(P',\mu))$ such that $\mathsf{handle\text{-}push}((\pi||(P',\mu)), v', t')$ had been executed. Then we have

$$\mathit{gained}_{P_i}(\pi) \geqslant \sum_{\pi' \in \Pi} \mathit{lost}_{P_i}(\pi') \tag{5}$$

Proof. First, observe that if P_i sends to $C^{P \multimap\!\circ P_i}$ a message $(\mathsf{cheating\text{-}signal}, w)$ then Eq. (5) must hold, because in this case $\mathit{gained}_{P_i}(\pi) = v$, while $\sum_{\pi' \in \Pi} \mathit{lost}_{P_i}(\pi') \leqslant v$ (this follows from the fact that an admissible $\mathcal{Z}$ never asks P_i to push more coins in total than v, see Fig. 4). Hence, what remains is to consider the case when no cheating was detected by P_i and in particular $\mathcal{S}^\pi$ is consistent. Let $\mathcal{S} = \{[\![\phi_i, \lambda_i]\!]\}_{i=1}^m$. From the construction of the protocol we get that

$$\mathit{gained}_{P_i}(\pi) := \lambda(|\pi|),$$

where $[\![\psi, \lambda]\!]$ is the leader of $\mathcal{S}^\pi$. This, from the consistency of $\mathcal{S}^\pi$ is at least equal to

$$\sum_{j:=1}^{m} \lambda_i[|\sigma| + 1],$$

which, in turn is equal to $\mathit{lost}_{P_i}(\pi')$. This finishes the proof the claim.

It is also easy to see that for every $P \in \{P_1, \ldots, P_{n-1}\}$ we have that

$$\mathit{net}(P) = \sum_{\pi} \mathit{gained}_P(\pi) - \sum_{\sigma} \mathit{lost}_P(\sigma),$$

where the sums are taken over all π's such that $\mathsf{handle\text{-}route}((\pi||P), v, t)$ (for some v and t) has been executed, and all σ's such that $\mathsf{handle\text{-}push}((\pi||P||P'), v, t)$ (for some v, t, and P') has been executed. Hence, by applying Claim 5.3 we obtain that $\mathit{net}(P) \geqslant 0$, and the balance neutrality holds.

To show fairness for the sender observe that the procedure for P_1 is very similar to the procedure for the intermediaries. Essentially, the only differences are as follows. First of all P_1, instead of receiving an $(\mathsf{pushed}, \pi, v, t)$ message from a state channel machine, receives an $(\mathsf{env\text{-}send}, v, \mu, t)$ message from $\mathcal{Z}$ and (in the next round) a $(\mathsf{invoice}, [\![\mu, u]\!], t)$ message from P_n. Secondly, the $\mathsf{cheating\text{-}signal}$ message has a different syntax (see Fig. 6 (a)). Thirdly, RVM does *not* transfer any coins to P_1's account (in fact, there are not "accounts" in this machine). Instead RVM outputs (μ, y) (see Fig. 7 (b)). Despite of these differences, the proof is essentially the same as the one for the intermediaries. The main difference is that the gained_{P_1} is now defined with respect to the values output by the receipt verification machine RVM. Namely, once this machine outputs (v, μ) we let

$$\mathit{gained}_{P_1}((P_1, \mu)) := (\mu, v)$$

(while the definition of lost_{P_1} remains as for the other P_i's). We can show that for every μ the total sum of coins that P_1 looses as a result of executing $\mathsf{handle\text{-}route}((P_1, \mu), v, t)$ in his channels with other parties, is not greater than v', where (μ, v') is the value output by RVM. This, of course, implies that the *total* amount of coins that P_1 looses cannot be larger than the value of $\mathsf{transmitted}$. The proof goes along the same lines as above. In particular we use the fact that the P_1 cannot loose more coins that u (this follows the construction of $\mathcal{Z}$), and therefore if P_1 detects inconsistency, the fairness for P_1 is guaranteed to hold, as P_1 can always make RVM output (v, μ), by sending to it the inconsistency proof together with $(\mathsf{invoice}, [\![\mu, u]\!], t)$.

To show fairness for the receiver, consider some nonce μ such that P_n received a message $(\mathsf{env\text{-}receive}, v, \mu, t)$ from $\mathcal{Z}$ (for some u and t). Recall (see Fig. 6 (a)) that P_n constructs a payment tree Φ^μ by executing $\mathsf{Add}_{\mathcal{S}^\mu}(\pi, v')$ each time when it receives a message $(\mathsf{env\text{-}acknowledge}, \pi, v')$. By Lemma 1 Φ^μ is always consistent. Recall also that P_n sends a message $(\mathsf{acknowledge}, [\![\pi, \lambda]\!])$ to $C^{P \multimap\!\circ P_n}$ (for some P). We have that $\lambda[|\pi|] := v'$, and therefore P_n gains v' coins in the channel $C^{P \multimap\!\circ P_n}$. The following invariant has to holds. Let S^μ be equal to the total amount of coins that P_n gained this way, and let $[\![\widehat{\psi}, \widehat{\lambda}]\!]$ be the leader of Φ^μ. Then

$$S^\mu = \widehat{\lambda}[n].$$

Hence, no matter what a (potentially malicious) P_1 sends to the receipt verification machine RVM, this machine will never output (v, μ), with $v > S^\mu$. Hence, the fairness for the receiver holds.

Finally, it is also easy to see that the "no money printing" holds for every state channels machine $C^{P_i \multimap\!\circ P_j}$. This is because each such a machine will add at most v coins to the accounts of P_i and P_j, and this will happen only after deducing v coins from an account of one of them. $\square$

5.3.1 Efficiency analysis. Let us comment on the efficiency of Ethna. When analyzing security of the off-chain protocols one typically considers the *optimistic* scenario (when the parties are cooperating) and the *pessimistic* one when the malicious parties can try to slow down the execution. We analyze the efficiency of Ethna in both cases.

Time complexity. In the optimistic case the payments are almost immediate. It takes 1 for a payment to be pushed, and 2 rounds to be acknowledged (since for acknowledgment the messages sent to state channels are not immediate). Hence in the most optimistic case the time for executing a payment is $3 \cdot \ell$ (where ℓ is the depth of the payment tree, i.e., the maximal size of a path from P_1 to P_n over which a sub-payment goes). Of course, in reality even honest parties can add delays, e.g, waiting for more capacity available in a given path. In the pessimistic case pushing the payments also takes 1 round (since this message is immediate). During the acknowledgment every malicious party can delay the process by time at most Δ. Hence, the maximal pessimistic time is $(1 + \Delta) \cdot \ell$.

Blockchain costs. The second important measure are the blockchain costs, i.e., the fees that the parties need to pay. Below we provide a "theoretical" analysis of such costs (by this we mean that we abstract away from practical features of Ethereum). For results of concrete experiments see 6. Note that in the optimistic case these the only costs are channel opening and closing, and hence (in theory) they can be considered independent of the tree depth and of its arity. In the pessimistic case all the messages in state channels need to be sent "via the blockchain". This is especially unpleasant, since its not clear whose fault it was, and who should pay the fees (in other words: this fault is "non-uniquely attributable" and can lead to "griefing", see, e.g. [10, 11] for an explanation of these notions). Let us consider two cases. In the first case there is no fraud proof (e.g. because P_n is honest). Then, the only message that is sent via the blockchain is $\mathsf{acknowledge}(sign\phi, \lambda)$, which has size linear $O(\ell + \kappa)$ (where ℓ is as above, and κ is the security parameter, and corresponds to space needed to store a signature). The situation is a bit different when P_n is cheating. As remarked in Sec. 4.1.4 the size of a fraud proof is $O(\alpha \cdot (\ell + \kappa))$, where α is Ethna's arity, ℓ is the maximal length of payment routes, and κ is the security parameter. Note that the fraud proof is "propagated", i.e., even if a given intermediary decided to keep its arity small (i.e.: not to split her sub-payments in too many sub-payments), she may be forced to pay fees that depend on some (potentially larger) arity. This could result in griefing attacks and it is the reason why we introduced a global bound on the arity. There are many ways around this. First of all, we could modify the protocol in such a way that the fraud proofs by P_n are posted directly in a smart contract on a blockchain in such a way that all the other parties do not need to repost it, and can just refer to it. This would mean that the fees are payed only by the first party that discovers the fraud proof. She could then be compensated from a deposit put aside before the protocol starts. Note that we could even extend it so that a fraud proof from one payment μ can serve as a "wild card" for any other payment whose receiver is P_n. Thanks to this, one "universal" deposit by P_n would suffice for all the payments. Moreover, the proof size can be significantly reduced using techniques described in Appx. B.

6 Implementation

We implemented a simple version of Ethna in Solidity. Compared to the version described in this paper, this preliminary version lacks the ability to add nonces (hence,each party can route only one sub-payment of a given sub-payment π). It also does not have any optimization tricks described above or in Appx. B. The following table summarizes the execution costs *in terms of thousands of gas*, and depending on the arity and the maximal path length.

`arity / path length`	`constructor`	`close`	`addState`	`addCheating-Proof`	`addComple-tedTransa-ction`	`closeDisa-greement`
5/10	2,391	14	93	1,053	155	14
5/5	2,249	14	94	871	145	14
2/5	2,088	14	93	779	145	14
2/3	2,191	14	93	590	140	14

Above, `constructor` denotes da procedure for deploying a channel, `close` corresponds to closing a channel without disagreement, `addState` is used to register the balance in case of disagreement, `addCheatingProof` is used to add a fraud proof, `addCompletedTransaction` — to add a sub-receipt when no cheating was discovered, and `closeDisagreement` – to finally close a channel after disagreement. Assuming cost 1,000 gas = $0.00018 (according to `ethgasstation.info` this is the average rate as of Jan 21st, 2020) we get that the most expensive action (deploying a channel, `addCheatingProof`) costs $0.43.

References

[1] G. Avarikioti, E. Kokoris-Kogias, and R. Wattenhofer. "Brick: Asynchronous State Channels." In: *CoRR* abs/1905.11360 (2019). arXiv: 1905.11360. URL: http://arxiv.org/abs/1905.11360.

[2] V. K. Bagaria, J. Neu, and D. Tse. "Boomerang: Redundancy Improves Latency and Throughput in Payment Networks." In: *CoRR* abs/1910.01834 (2019). arXiv: 1910.01834. URL: http://arxiv.org/abs/1910.01834.

[3] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols." In: *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*. IEEE Computer Society, 2001, pp. 136–145. DOI: 10.1109/SFCS.2001.959888.

[4] J. Coleman, L. Horne, and L. Xuanji. *Counterfactual: Generalized State Channels*. https://l4.ventures/papers/statechannels.pdf. 2018.

[5] C. Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. 1st. USA: Apress, 2017. ISBN: 1484225341.

[6] C. Decker and R. Wattenhofer. "A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels." In: *Stabilization, Safety, and Security of Distributed Systems - 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings*. Ed. by A. Pelc and A. A. Schwarzmann. Vol. 9212. Lecture Notes in Computer Science. Springer, 2015, pp. 3–18. DOI: 10.1007/978-3-319-21741-3_1. URL: https://doi.org/10.1007/978-3-319-21741-3_1.

[7] Diar. *Lightning Strikes, But Select Hubs Dominate Network Funds*. https://diar.co/volume-2-issue-25/. (Accessed on 01/20/2020).

[8] S. Dziembowski, S. Faust, and K. Hostáková. "General State Channel Networks." In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by D. Lie, M. Mannan, M. Backes, and X. Wang. ACM, 2018, pp. 949–966. DOI: 10.1145/3243734.3243856.

[9] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková. "Multi-party Virtual State Channels." In: *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*. Ed. by Y. Ishai and V. Rijmen. Vol. 11476. Lecture Notes in Computer Science. Springer, 2019, pp. 625–656. DOI: 10.1007/978-3-030-17653-2_21. URL: https://doi.org/10.1007/978-3-030-17653-2_21.

[10] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski. "Perun: Virtual Payment Hubs over Cryptocurrencies." In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 106–123. DOI: 10.1109/SP.2019.00020.

[11] C. Egger, P. Moreno-Sanchez, and M. Maffei. "Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks." In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by L. Cavallaro, J. Kinder, X. Wang, and J. Katz. ACM, 2019, pp. 801–815. DOI: 10.1145/3319535.3345666.

[12] C. Egger, P. Moreno-Sanchez, and M. Maffei. "Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks." In: *IACR Cryptology ePrint Archive* 2019 (2019), p. 583. URL: https://eprint.iacr.org/2019/583.

[13] M. Green and I. Miers. "Bolt: Anonymous Payment Channels for Decentralized Currencies." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM, 2017, pp. 473–489. DOI: 10.1145/3133956.3134093.

[14] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais. "SoK: Off The Chain Transactions." In: *IACR Cryptology ePrint Archive* 2019 (2019), p. 360. URL: https://eprint.iacr.org/2019/360.

[15] M. Jourenko, K. Kurazumi, M. Larangeira, and K. Tanaka. "SoK: A Taxonomy for Layer-2 Scalability Related Protocols for Cryptocurrencies." In: *IACR Cryptology ePrint Archive* 2019 (2019), p. 352. URL: https://eprint.iacr.org/2019/352.

[16] H. A. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. "Arbitrum: Scalable, private smart contracts." In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by W. Enck and A. P. Felt. USENIX Association, 2018, pp. 1353–1370. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner.

[17] J. Katz and Y. Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007. ISBN: 1584885513.

[18] R. Khalil and A. Gervais. "Revive: Rebalancing Off-Blockchain Payment Networks." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM, 2017, pp. 439–453. DOI: 10.1145/3133956.3134033.

[19] A. Kiayias and O. S. T. Litos. "A Composable Security Treatment of the Lightning Network." In: *IACR Cryptology ePrint Archive* 2019 (2019), p. 778. URL: https://eprint.iacr.org/2019/778.

[20] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts." In: *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 839–858. DOI: 10.1109/SP.2016.55.

[21] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei. "Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability." In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. URL: https://www.ndss-symposium.org/ndss-paper/anonymous-multi-hop-locks-for-blockchain-scalability-and-interoperability/.

[22] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi. "Concurrency and Privacy with Payment-Channel Networks." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM, 2017, pp. 455–471. DOI: 10.1145/3133956.3134096.

[23] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry. "Sprites and State Channels: Payment Networks that Go Faster Than Lightning." In: *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*. Ed. by I. Goldberg and T. Moore. Vol. 11598. Lecture Notes in Computer Science. Springer, 2019, pp. 508–526. DOI: 10.1007/978-3-030-32101-7_30. URL: https://doi.org/10.1007/978-3-030-32101-7_30.

[24] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. http://bitcoin.org/bitcoin.pdf. 2009.

[25] O. Osuntokun. *[Lightning-dev] AMP: Atomic Multi-Path Payments over Lightning*. https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-February/000993.html. (Accessed on 01/19/2020). 2018.

[26] D. Piatkivskyi and M. Nowostawski. "Split Payments in Payment Networks." In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2018 International Workshops, DPM 2018 and CBT 2018, Barcelona, Spain, September 6-7, 2018, Proceedings*. Ed. by J. García-Alfaro, J. Herrera-Joancomartí, G. Livraga, and R. Rios. Vol. 11025. Lecture Notes in Computer Science.

Springer, 2018, pp. 67–75. DOI: 10.1007/978-3-030-00305-0_5. URL: https://doi.org/10.1007/978-3-030-00305-0_5.
[27] J. Poon and V. Buterin. *Plasma: Scalable Autonomous Smart Contracts.* Accessed: 2017-08-10. 2017. URL: https://plasma.io/plasma.pdf.
[28] J. Poon and T. Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments.* Draft version 0.5.9.2, available at https://lightning.network/lightning-network-paper.pdf. Jan. 2016.
[29] J. Spilman. *[Bitcoin-development] Anti DoS for tx replacement.* https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-April/002433.html. (Accessed on 01/20/2020). 2013.
[30] Suredbits. *Lightning 101: What is a Lightning Invoice?* 2019. URL: https://medium.com/suredbits/lightning-101-what-is-a-lightning-invoice-d527db1a77e6.
[31] J. Teutsch and C. Reitwießner. *A scalable verification solution for blockchains.* https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf. 2017.
[32] G. Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger.* http://gavwood.com/paper.pdf. 2014.

A Standard function and string notation

By $[a_i \mapsto x_1, \ldots, a_m \mapsto x_m]$ we mean a function $f : \{a_i, \ldots, a_m\} \to \{x_1, \ldots, x_m\}$ such that for every i we have $f(a_i) := x_i$. Let α be a string $\langle \alpha_1, \ldots, \alpha_n \rangle$. For $i = 1, \ldots, n$ let $\alpha[i]$ denote α_i. Let ε denote an empty string, and "$||$" denote concatenation of strings. We overload this symbol, and write $\alpha||a$ and $a||\alpha$ to denote $\alpha||\langle a \rangle$ and $\langle a \rangle||\alpha$, respectively (for $\alpha \in A^*$ and $a \in A$). For $k \leqslant n$ let $\alpha|_k$ denote α's prefix of length k.

B Reducing the size of the fraud proofs

Recall that a fraud proof is a payment report $\mathcal{Q}$ of a form $\mathcal{Q} = \{[\![(\sigma||\pi_i), \lambda_i]\!]\}_{i=1}^m$, all the $\pi_i[1]$'s are pairwise distinct, such that the following condition holds:

$$\max_{i:=1,\ldots,m} \lambda_i[|\sigma|] < \sum_{i:=1}^{m} \lambda_i[|\sigma| + 1]. \tag{6}$$

Hence, in the most straightforward implementation it is of length $\Omega(\alpha \cdot (\ell + \kappa))$, where α is Ethna's arity, ℓ is the maximal length of payment routes, and κ is the security parameter

B.1 Fraud proof of length independent from ℓ

We now show how to reduce this to $O(\alpha \cdot \kappa)$. We do it by designing an algorithm that signs the sub-receipts $[\![\phi, \lambda]\!]$ in a different way. Let H be a collision-resistant hash function, and let $(\mathsf{KGen}, \mathsf{Sig}, \mathsf{Vf})$ be a signature scheme. Suppose $(\mathsf{sk}, \mathsf{pk}) \leftarrow_{\$} \mathsf{KGen}(1^\kappa)$ is the key pair of P_n. To sign (ϕ, λ) we define a new signature scheme $(\mathsf{KGen}, \mathsf{Sig}, \mathsf{Vf})$ (i.e. we later let $[\![\phi, \lambda]\!] := ((\phi, \lambda), \sigma)$, where $\sigma := \mathsf{Sig}'_{\mathsf{sk}}((\phi, \lambda)))$. Let $\mathsf{KGen}' := \mathsf{KGen}$. To define $\mathsf{Sig}((\phi, \lambda))$ first define $\langle h^1, \ldots, h^{|\phi|} \rangle$ recursively as:

$$h^1 := H(\phi[1]),$$

and for $j := 2, \ldots, |\phi|$:

$$h^j := H(\phi[j], h^{j-1}).$$

Then let $\mathsf{Sig}((\phi, \lambda)) := \langle \sigma^1, \ldots, \sigma^{|\phi|} \rangle$, where for each j we have:

$$\sigma^j := \mathsf{Sig}^{\mathsf{sk}}(h^j, \lambda[j])$$

Verification of this signature is straightforward. It is also easy to see that if $(\mathsf{KGen}, \mathsf{Sig}, \mathsf{Vf})$ is existentially unforgeable under chosen message attack, then so is $(\mathsf{KGen}', \mathsf{Sig}', \mathsf{Vf}')$, assuming the signed messages are of a form (ϕ, λ), where ϕ is the payment path[5]. For a message M let $\{M\}_{P_n}$ denote M signed with $(\mathsf{KGen}', \mathsf{Sig}', \mathsf{Vf}')$. It is easy to see that now a fraud proof from Eq. (6) can be compressed to a sequence

$$\left\{ \left(\left\{ h_i^{|\sigma|}, \lambda_i[|\sigma|] \right\}_{P_n}, \pi_i[1], \left\{ h_i^{|\sigma|+1}, \lambda_i[|\sigma|+1] \right\}_{P_n} \right) \right\}_{i=1}^{m}. \tag{7}$$

such that Eq. (6) holds (above "$\pi_i[1]$" is needed to check correctness of $h_i^{|\sigma|+1}$). Since all the signed values are of size linear in the security parameter, and $m \leqslant \alpha$ we get that Eq. (7) is $O(\alpha \cdot \kappa)$. Note that this requires the parties (and, pessimistically, the state channel contract) to verify m signatures. This can be reduced to 1 signature by using signature aggregation techniques, the simplest one being the Merkle trees technique, where we hash all pairs $(h^j, \lambda[j])$ using Merkle hash and sign only the top of the tree. Note that this introduces additional data costs of size $O(\kappa \cdot \log \alpha)$.

Further proof size reduction using "bisection" Finally, let us remark that the proof Eq. (7) can be further compressed by allowing interaction between the party that discovered cheating (denote it P) and P_n. This is similar to the bisection technique [16, 31]. Suppose P realized that Eq. 6 does not hold. She can then divide the set of paths in $\mathcal{Q}$ into two halves For convince suppose m is even and let

$$A := \sum_{i:=1}^{m/2} \lambda_i[|\sigma|+1],$$

and

$$B := \sum_{i:=m/2+1}^{m} \lambda_i[|\sigma|+1].$$

P can now challenge P_n (on the blockchain) to provide her own calculations of the above sums[6]. Let A' and B' be P_n respective answers. Then one of the following has to hold:

- $\max_{i:=1,\ldots,m} \lambda_i[|\sigma|] < A' + B'$ – then P obtains the fraud proof and we are done.
- $A' < A$ or $B' < B$ – then we can apply this procedure recursively.

It is easy to see that in logarithmic number o rounds P obtains a fraud proof. Note that this fraud proof is short, so it can be easily propagated to other parties (who do not need to repeat the above "game" with P_n).

[5] This assumption is needed since payment paths have a clearly marked "ending", namely they have to finish with (P_n, μ_n), for some μ_n Otherwise it would be possible to attack this scheme by taking a prefix of a signed message and a prefix of its signature.

[6] Since elements of $\mathcal{Q}$ can be sorted such a challenge is short.