Non-Atomic Payment Splitting in Channel Networks

Stefan Dziembowski
University of Warsaw, Poland
IDEAS NCBR, Warsaw, Poland

Paweł Kędzior
University of Warsaw, Poland

Abstract

Off-chain channel networks are one of the most promising technologies for dealing with blockchain scalability and delayed finality issues. Parties 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 the help of intermediaries.

In this paper, we introduce a new notion that we call *Non-Atomic Payment Splitting (NAPS) protocols* that allow the intermediaries in the network to split the payments recursively into several subpayments in such a way that the payment can be successful "partially" (i.e. not all the requested amount may be transferred). This contrasts with the existing splitting techniques that are "atomic" in that they did not allow such partial payments (we compare the "atomic" and "non-atomic" approaches in the paper). We define NAPS formally and then present a protocol that we call "EthNA", that satisfies this definition. EthNA is based on very simple and efficient cryptographic tools; in particular, it does not use expensive cryptographic primitives. We implement a simple variant of EthNA in Solidity and provide some benchmarks. We also report on some experiments with routing using EthNA.

2012 ACM Subject Classification Security and privacy → Systems security

Keywords and phrases Blockchain, Payment Channels Networks

Digital Object Identifier 10.4230/LIPIcs.AFT.2023.17

Related Version *Full Version*: `https://eprint.iacr.org/2020/166` [7]

Supplementary Material *Software*: `https://github.com/Sam16450/NAPS-EthNA`
archived at `swh:1:dir:5336d35d6eae6b63827c47b1c3614ef501e5083e`

Funding This result is part of a project that received funding from the European Research Council (ERC) under the European Union's Horizon 2020 innovation program (grant PROCONTRA-885666). It was also party financed by the Foundation for Polish Science under grant TEAM/2016-1/4 founded within the UE 2014–2020 Smart Growth Operational Program and by the *Ethereum Foundation* grant FY18-0023.

1 Introduction

Blockchain technology allows a large group of parties to reach a consensus about the contents of an (immutable) ledger, typically containing a list of transactions. In blockchain's initial applications, these transactions described transfers of *coins* between the parties. One of the very promising extensions of the original Bitcoin ledger is blockchains that allow to register and execute the so-called *smart contracts* (or simply "contracts"), i.e., formal agreements between the parties, written down in a programming language and having financial consequences (for more on this topic see, e.g., [12, 6]). Probably the best-known example of such a system is *Ethereum*. Several blockchain-based systems' main limitations are delayed finality, lack

© Stefan Dziembowski and Paweł Kędzior;
licensed under Creative Commons License CC-BY 4.0

of scalability, and non-trivial transaction fees. *Off-chain channels* [18, 3] are a powerful approach for dealing with these issues. The simplest examples of this technology are the so-called "*payment* channels". Informally, such a channel between Alice and Bob is an object in which both parties have some coins that they can freely transfer without interacting with the blockchain ("off-chain"). We explain this in Sec. 1.1 below. Readers familiar with this topic can go quickly over it, just paying attention to some terminology and notation that we use

1.1 Background

Assume that the maximal blockchain reaction time is Δ. We model amounts of coins as non-negative integers and write "n¢" to denote n coins. A payment channel is *opened* when Alice and Bob deploy a smart contract on the ledger and deposit some number of coins (say: x, and y, respectively) into it. The initial *balance* of this channel is: "x¢ in Alice's account, y¢ in Bob's account" (or $[\text{Alice} \mapsto x, \text{Bob} \mapsto y]$ for short). This balance can be *updated* (to some new balance $[\text{Alice} \mapsto x', \text{Bob} \mapsto y']$, such that $x' + y' = x + y$) by exchanging messages between the parties. The corresponding smart contract guarantees that each party can at any time *close* the channel and get the money corresponding to her latest balance. Only the opening and closing operations require interaction with the blockchain. Since updates do not require blockchain participation, each update is immediate (the network speed determines its time) and at essentially no cost.

Now, suppose we are given a set of parties $P_1, \ldots, P_n$ and channels between some of 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 "$P_i \multimap P_j$" (instead of $\{P_i, P_j\}$). Every $P_i \multimap P_j$ represents a channel between P_i and P_j, and the *cash function* Γ determines the number of coins available for the parties in every channel. More precisely, every $\Gamma(P_i \multimap P_j)$ is a function f of a type $f : \{P_i, P_j\} \to \mathbb{Z}_{\geq 0}$. We will often write $\Gamma^{P_i \multimap P_j}$ to denote this function. The value $\Gamma^{P_i \multimap P_j}(P)$ denotes the amount of coins that P has in her *account* in channel $P_i \multimap P_j$. A *path* (in $\mathcal{G}$) is a sequence $P_{i_1} \rightarrowtriangle \cdots \rightarrowtriangle P_{i_t}$ such that for every j we have $P_{i_j} \multimap P_{i_{j+1}} \in \mathcal{E}$. In the formal part of the paper (see Sec. 3.1), we will also include "nonces" in the paths, but in this informal description, we ignore them. In this paper, for the sake of simplicity, we assume that (a) the channel system is deployed with some initial value of Γ, which evolves over time, (b) once a channel system is established, no new channels are created, and no channels are closed (i.e., $\mathcal{E}$ remains fixed), and (c) no coins are added to the existing channels, i.e., the total amount of coins available in every channel $e = P_i \multimap P_j$ never exceeds the total amount available in it initially.

Channel graphs can serve for secure payment sending. Let us recall how this works in the most popular payment channel networks, such as *Lightning* or *Raiden*. Our description is very high-level (for the details, see, e.g., [18]). Consider the following example: we have three parties: P_1, P_2, and P_3 and two channels: $P_1 \multimap P_2$ and $P_2 \multimap P_3$ between them. Now, suppose the *sender* P_1 wants to send v¢ to the *receiver* P_3 over the path $P_1 \rightarrowtriangle P_2 \rightarrowtriangle P_3$, with P_2 being an *intermediary* that *routes* these coins. This is done as follows. First, party P_1 asks P_2 to forward v¢ in the direction of P_3 (we call such a request *pushing* coins from P_1 to P_2). The receipt from P_3 confirming that she received these coins has to be presented by P_2 within 2Δ (denote this receipt with ρ). If P_2 manages to do it by this deadline, she gets these coins in her account in the channel $P_1 \multimap P_2$. To guarantee this, P_1 initially blocks these coins in the channel $P_1 \multimap P_2$. These coins can be claimed back by P_1 if time 2Δ have passed, and P_2 did not claim them. In a similar way, P_2 pushes these coins to P_3, i.e., she offers P_3 to

claim (by providing proof ρ within Δ time) v¢ in the channel $P_3 \multimap P_4$. Now suppose that party P_3 claims her v¢ in channel $P_2 \multimap P_3$. This can only be done by providing a receipt ρ confirming that she received these coins. We call this process *acknowledging* payment. Party P_2 can now claim her coins in channel $P_1 \multimap P_2$ by submitting an acknowledgment containing the receipt ρ. In the above example, the number of coins that can be pushed via a channel $P_i \multimap P_{i+1}$ is upper-bounded by the number of coins that P_i has in this channel. Therefore the maximal amount of coins that can be pushed over path $P_1 \twoheadrightarrow P_2 \twoheadrightarrow P_3$ is equal to the minimum of these values. We will call this value the *capacity* of a given path.

On the technical level, in the Lightning network, the receipt ρ is constructed using so-called *hash-locked transactions* and "smart contracts" that guarantee that nobody loses money. This is possible thanks to how the $n\Delta$ deadlines in the channels $P_1 \multimap P_2$ and $P_2 \multimap P_3$ are chosen. An exciting feature of this protocol is that receipt ρ 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 transferred some coins to P_4. In other words: P_1 can use ρ to resolve disputes with P_4, either in some smart contract (deployed earlier and using the given PCN for payments) or outside the blockchain. The notion of payment channels can be generalized to "*state* channels". Informally, such channels can serve not only for payments between the parties but also for executing contracts within them. For more on this, see, e.g., [6, 15, 2].

1.2 Our contribution and related work

One of the main problems with the existing PCNs is that sending a payment between two parties requires a path from the sender to the receiver with sufficient capacity. This problem is amplified by the fact that the capacity of potential paths can change dynamically as several payments are executed in parallel. Although usually, the payments are swift, 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. Therefore it is hard to predict a given path's capacity even in the very near future. This is especially a problem if the capacity of a given channel is close to being completely exhausted (i.e. it is close to zero because of several ongoing payments). A natural solution for solving this problem is to split the payments into several subpayments. This was described in several recent papers (see, e.g., [16, 17, 20, 8]). However, up to our knowledge, all these papers considered so-called "*atomic* payment splitting", meaning that either all the subpayments got through or none of them. In this paper, we prove a new, alternative technique that we call "*non*-atomic payment splitting" that does not have this feature and hence is more flexible. (We compare atomic and non-atomic splitting in Sec. 2.1.2.) More concretely, our contribution can be summarized as follows.

NAPS definition. We introduce the concept of *non-atomic payment splitting* by defining formally a notion of *Non-Atomic Payment Splitting (NAPS)* protocols. In our definition, we require that splitting is done ad-hoc by the intermediaries, possibly in reaction to dynamically changing the capacity of the paths or fees. Perhaps the easiest way to describe NAPS is to look at payment networks as tools for outsourcing payment delivery. For example, in the scenario from Section 1.1 party P_1 outsources to P_2 the task of delivering v¢ to P_4, and gives P_2 time 2Δ 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 money being transferred at once. NAPS protocol permits such recursive non-atomic payment splitting into "subpayments" and partial transfers of coins. This splitting can be done in an ad-hoc way. Moreover, the users can try to route the same payment over the same path multiple times (hoping that some more capacity becomes available in the meantime). We present a UC-like definition of NAPS. An additional advantage of our contribution is that our definition can be easily adapted to cover the *atomic* payment splitting protocols [17, 1, 19, 8].

EthNA construction. We construct a protocol that we call EthNA that satisfies the NAPS definition. We call our protocol EthNA, in reference to Etna, one of the highest active volcanos in Europe. This is because the coin transfers in EthNA resemble a lava flood (with large streams recursively bifurcating into small substreams). The letter "h" is added so that the prefix "Eth-" is reminiscent of ETH, the symbol of Ether (the currency used in Ethereum), and "NA" stands for "Non-Atomic".

In EthNA the "subreceipts" for subpayments are aggregated by the intermediaries into one short subreceipt so that their size does not grow with the number of aggregated subreceipts. This is done efficiently, particularly by avoiding advanced and expensive techniques such as noninteractive zero knowledge or homomorphic signature schemes and hash functions. 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 [21, 6]). We stress that the amount of data that is passed between two consecutive parties on the path does *not* depend on the number of subpayments in which the payment is later divided. The same applies to the data these two parties send to the blockchain if they conflict. We summarize the complexity of EthNA in Sec. 3.3.

Security analysis and implementation. We provide a formal security analysis of EthNA. More precisely, we prove that EthNA satisfies the NAPS definition. We also analyze EthNA's complexity. We also implement EthNA contracts in Solidity (the standard language for programming the smart contracts in Ethereum), and we provide some routing experiments. We describe this implementation and provide some benchmarks. We stress, however, that routing algorithms are *not* the main focus of this work and further research on designing algorithms that exploit the non-atomicity of payment splitting.

Possible applications of NAPS. As mentioned above, one obvious application of NAPS is to help efficiently send one big payment by dividing it into several ad-hoc installments: if it is impossible to route the total amount u, then the client can 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 installment). The same applies to other situations, e.g., when the user wants to exchange coins for another currency. Ideally, he would like to exchange the entire amount u, but exchanging $v < u$ is better than nothing. A related scenario is making a partial "bank deposit" when the user wants to deposit as much money as possible but no more than u.

Moreover, in many cases, the goods the seller delivers in exchange for the payment can be divided into tiny units and sent to the buyer depending on how many coins have been transferred. One example is battery charging, where charging, say 1/2 of the battery is much better than having the battery dead. This applies both to mobile phones and to IoT devices that can trade energy with each other. Let us also mention applications like file sharing,

where the client typically connects to several servers and tries to download as much data as possible from each. NAPS can be an attractive way to perform payments in this scenario. Note also that NAPS can be combined with other means of payment. If a user manages to send only $u < v$ coins via NAPS, then she can decide to send the rest $(v - u)$ in some more expensive way (this makes sense, especially in systems where the fee depends on the amount being transferred, e.g., in the credit card payments).

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 recent exciting developments [9, 15, 13, 6, 4, 5, 11, 14, 2] in this area. The reader can also consult SoK papers on off-chain techniques (e.g. [10]). Partial coin transfers were considered in [17], but with no aggregation techniques and ad-hoc splitting. Atomic payment splitting has been considered in [17, 1, 19, 8]. All of these papers focus on routing techniques, which is not the main topic of this paper.

Organization and notation. Sec. 2 contains an informal description of our ideas. Then, in Sec. 3, we provide the formal NAPS definition and the detailed description of EthNA and its security properties. An overview of our implementation and simulations is presented in Sec. 3.3. When we say that a message is "signed by some party", we mean that it is signed using some fixed signature scheme that is existentially unforgeable under a chosen-message attack. Natural numbers are denoted with $\mathbb{N}$. We will also use the notion of *nonces*. Their set is denoted with $\mathcal{N}$. We assume that $\mathcal{N} = \mathbb{N}$. We use some standard notations for functions, string operations, and trees. 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 A be some finite alphabet. Strings $\delta \in A^*$ are frequently denoted using angle brackets: $\delta = \langle \delta_1, \ldots, \delta_m \rangle$. Let δ be a string $\langle \delta_1, \ldots, \delta_n \rangle$. For $i = 1, \ldots, n$ let $\delta[i]$ denote δ_i. Let ε denote an empty string, and "$||$" denote the concatenation of strings. We overload this symbol, and write $\delta||a$ and $a||\delta$ to denote $\delta||\langle a \rangle$ and $\langle a \rangle||\delta$, respectively (for $\delta \in A^*$ and $a \in A$). For $k \leq n$ let $\delta|_k$ denote δ's prefix of length k. A set of prefixes of δ is denoted $\mathsf{prefix}(\delta)$ (note that it includes ε).

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 $\delta \in T$ we have that any prefix of δ is also in T. Any element of T is called a *node* of this tree. For two nodes $\delta, \beta \in T$ such that $\beta = \delta||a$ (for some a) we say that δ is the *parent* of β, and β is a *child* of δ. 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 $\delta \in T$ we say that $\mathcal{L}(\delta)$ is the *label of* δ.

2 Informal description

Below, in Sec. 2.1 we provide an overview of NAPS definition, and in Sec. 2.2 we informally describe EthNA.

2.1 Overview of the NAPS definition

Let us now explain the NAPS protocol features informally (for a formal definition, see Sec. 3.1). Throughout this paper, we use the following convention: our protocols are run by a set of *parties* denoted $\mathcal{P} = \{P_1, \ldots, P_n\}$, where P_1 be the *sender*, $P_2, \ldots, P_{n-1}$ be the *intermediaries*, and P_n is the *receiver*. A message m signed by a party P_i will be denoted $\wr m \int_{P_i}$. Let v be the number of coins that P_1 wants to send to P_n, and let t be the maximal time until the transfer of coins should be completed. Since, in general, P_1 can perform

multiple payments to P_n, we assume that each payment comes with a nonce $\mu \in \mathcal{N}$ that can be later used to identify this payment. Sometimes we will simply call it "payment μ". For simplicity, we start with an informal description of how NAPS protocols operate when all parties are honest. The security properties (taking into account the malicious behavior of the parties) are described informally in Sec. 2.1.1, and formally defined in Sec. 3.1. Before proceeding with the description of EthNA the reader may look at the example in Fig. 1.

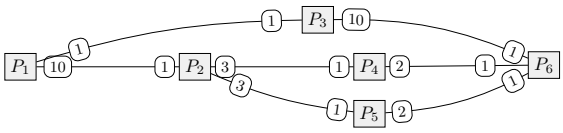
To describe the protocol more generally, let us start by presenting it from the point of view of the sender P_1. Let $P_{i_1}, \ldots, P_{i_t}$ be the neighbors of P_1, i.e., parties with which P_1 has channels. Suppose the balance of each channel $P_1 \multimap P_{i_j}$ is $[P_1 \mapsto x_i, P_{i_j} \mapsto y_j]$ (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 to P_n via some P_{i_j}, and set up a deadline t_j for this (we will also call v_j a *subpayment* of payment μ). This results in: (a) 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 number of coins that P_1 still wants to transfer to P_n is decreased as follows: $v := v - v_j$, and (c) P_{i_j} holding "v_j coins 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 (1) changing the balance of the channel $P_1 \multimap P_{i_j}$ by crediting v'_j coins to P_{i_j}'s account in it, and (2) r coins to P_1's account. Moreover, (3) P_1 adds back the non-transferred amount r to v, by letting $v := v + r$. Above (1) corresponds to the fact that P_{i_j} has to be given the coins that she transferred, and (2) comes from the fact that not all the coins were transferred (if P_{i_j} managed to transfer all the coins, then, of course, $r = 0$). Finally, (3) is used for P_1's "internal bookkeeping" purposes, i.e., P_1 simply writes down that r coins "were returned" and still need to be transferred. While the 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 the same way as transferring coins via P_{i_j}.

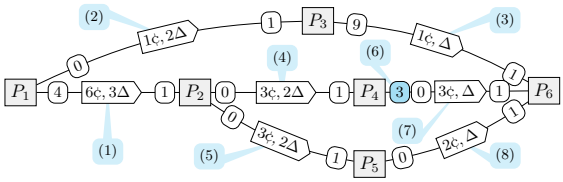
The intermediaries can repeat this process. Let P be a party that holds some coins that were "pushed" to her by some P' (which originate from P_1 and must be delivered to P_n). Now, P can split them further, and moreover, she can decide on her own how this splitting is done depending, e.g., on the current capacity of the possible paths leading to P_n. The payment splitting can be done arbitrarily, except for the two following restrictions. First, we do not allow "loops" (i.e. paths containing 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, we assume that the number of times a given payment subpayment is split by a single party P is bounded by a parameter $\delta \in \mathbb{N}$, called *arity* (for example arity on Figs. 1 is at most 2). In the extended version of this paper [7] we present an improved protocol where δ is unbounded (at the cost of a mild increase of the pessimistic number of rounds of interaction). As mentioned, the essential feature of NAPS is the *non-atomicity of payments*. We discuss it further below.

2.1.1 NAPS security properties

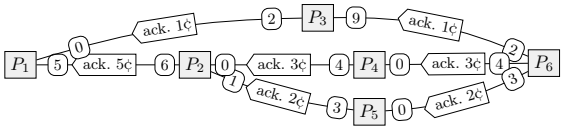
In the description in Sec. 2.1 we assumed that all parties behaved honestly. Like all other PCNs, we require that NAPS protocols work if the parties are malicious. In particular, no honest party P can lose money, even if all the other parties are not following the protocol and are working against P. The corrupt parties can act in a coalition modeled by an adversary $\mathcal{A}$. Formal security definition appears in Sec. 3.1. Let us now informally list the security



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



(b) The sender P_1 wants to send 7¢ to the receiver P_6. She splits these coins into two amounts: 6¢ pushed to P_2 and 1¢ pushed to P_3. This is indicated with labels (1) and (2), respectively. Then (3) party P_3 simply pushes 1¢ further to P_6. Party P_2 splits 6¢ into 3¢ + 3¢, and pushes 3¢ to both P_4 (4) and P_5 (5). Path $P_4 \rightarrowtriangle P_6$ initially had capacity 2 only (see Fig. (a) above), but luckily in the meanwhile 1¢ got unlocked (6) for P_4 in channel $P_4 \multimap P_6$, and hence (7) party P_4 pushes all 3¢ to P_6. Party P_5 pushes only 2¢ to P_6 (8). The channel balances correspond to the situation *after* the coins are pushed (except of channel $P_4 \multimap P_6$ where we also indicated the fact that 1¢ got unlocked (6)). Each party P can also decide on her own about the timeout t of each subpayment she pushes (this timeout is indicated with "$x\Delta$"). The only restriction is that t has to come at least Δ before the time she has to acknowledge that subpayment back. This is because P needs this "safety margin" of Δ in case P' is malicious, and the acknowledgment has to be done "via the blockchain".



(c) Party P_6 acknowledges subpayment of 1¢ to P_3, which, in turn acknowledges it to P_1. Party P_6 also acknowledges subpayment of 3¢ to P_4 and 2¢ 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¢) and sends it to P_1. As a result 5¢ + 1¢ = 6¢ are transferred from P_1 to P_6. The channel balances correspond to the situation *after* the coins were acknowledged.

Figure 1 An example of a NAPS protocol execution. An edge "P_i — x —— y — P_j" denotes the fact that there exists a channel between P_i and P_j, and the parties have x and y¢ in it, respectively.

requirements, which are pretty standard and hold for most PCNs (including Lightning). Below, let u denote the total amount of coins P_1 wants to transfer to P_n within some payment μ.

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 channels with other parties typically changes. Let $net_\mu(P)$ denote the number of coins that P gained in all 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 $\mathsf{Receipt}(\mu, v)$:="*an amount v of coins has been transferred from P_1 to P_n as a result of payment μ*". 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 $v \geq -net_\mu(P_1)$, in which case P_1 holds a receipt for transferring *more* coins than she actually lost in the channels. A receipt can be later used in another smart contract (e.g., a contract that delivers some digital goods whose amount depends on v). *Fairness for the receiver* is defined analogously, i.e.: if P_1 holds a receipt $\mathsf{Receipt}(\mu, v)$ then typically $v = net(P_n)$, and if some parties (other than P_n) are dishonest, then they can make $v \leq net_\mu(P_n)$. In other words, P_1 cannot get a receipt for an amount 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 $net_\mu(P) \geq 0$. Again: if everybody else is also honest, then we have equality instead of inequality.

2.1.2 Atomic vs. non-atomic payment splitting

As already highlighted in Sec. 1.2, the previous protocols on payment splitting always required payments to be atomic, meaning that for a payment to succeed, all the subpayments had to reach the receiver. Technically, this means that to issue a receipt for *any* of the subpayments (this receipt is typically a preimage of a hash function, see, e.g., [8]) all of them need to reach the receiver. This has several disadvantages: (1) the coins remain blocked in every path at least until the last subpayment arrives to the receiver, (2) the success of a given subpayment depends not only on the subsequent intermediaries but also on the other "sibling" paths (this problem was observed in [8] where it is argued that this risk may lead to intermediaries rejecting subpayments that were split before, see Sec. 3.1 of [8]). Finally, atomic payments may result in "deadlock" situations in the network where two competing payments can prevent each other from being executed. We describe an example of such a situation in [7].

Let us also remark that "atomicity" and even "fine-grained atomicity" can also be obtained in EthNA by a small protocol modification. We write more about it in the extended version of this paper [7]. Let us also remark that atomic payment splitting, in general, seems to be easier to achieve, which is probably the reason why there has been more focus on them in the literature (with papers focusing more on other aspects of this problem, such as routing algorithms, e.g. [8]). Finally, let us stress that we do not claim that non-atomicity is superior to atomicity. We think both solutions have advantages and disadvantages, and there exist applications where each is better than the other.

2.2 Overview of the EthNA protocol

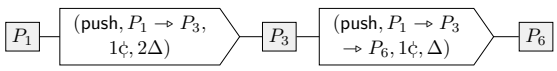
After presenting the NAPS definition, let us now explain the main ideas behind the EthNA protocol that realizes it. An essential feature of EthNA is that it permits "subreceipt aggregation", by which we mean the following. Consider some payment μ. Once P_n receives

some subpayment v that reached it via some path $\pi = P_1 \rightarrowtriangle P_{i_1} \rightarrowtriangle \cdots P_{i_k} \rightarrowtriangle P_n$ she issues a *subreceipt* for this payment and sends it to P_{i_k}. Each intermediary that receives more than one subreceipt can aggregate them into one short subreceipt that she sends further in the direction of 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 Sec. 4. One option would be to let the subreceipt be signed using a homomorphic signature scheme and then exploit this homomorphism to aggregate the subreceipts. This paper uses 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 "subpayment aggregation herself" (this is done when signing a subreceipt and does not require any further interaction with P_n). Then, we 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 proof of this fact (called a "fraud-proof") will count as a receipt that a total amount has been transferred to P_n. From the security point of view, this is ok since an honest P_n will never cheat (hence, no fraud-proof against him will ever be produced). Thanks to this approach, we avoid entirely using any expensive advanced cryptographic techniques (such as homomorphic signatures or noninteractive proofs). Below we explain the main idea of EthNA by considering the example from Fig. 1. Again, we start by describing how the protocol works when everybody is honest, and then (in Sec. 2.2) we show how the malicious behavior is prevented.

Invoice sending. The protocol starts with the receiver P_n sending to P_1 an "invoice" that specifies (among other things) the identifier μ of the payment, and the maximal amount u of coins that P_n is willing to accept. As we explain below, this invoice may be later used together with fraud-proofs to produce proof that all u coins were transferred to P_n (if she turns out to be malicious).

Pushing subpayments. Pushing subpayments is done by sending messages containing information about the path that the subpayment "traveled" so far (together with the number of coins to be pushed and timeout information) and simultaneously blocking coins in the underlying channels. The messages sent between P_1, P_3 and P_6 in Fig. 1 (a)) are presented in the picture below.

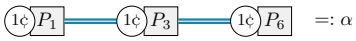


Whenever a message $(\mathsf{push}, \pi, v, t)$ is sent from P to P', the party P blocks v coins in channel $P \multimap P'$ for time t. These coins are claimed by P' if she provides a corresponding subreceipt within time t. Otherwise, they are claimed back by P.

Acknowledging subpayments by the receiver. The receiver P_n acknowledges the subpayments by replying with a signed subreceipt and claiming the coins blocked in the corresponding channels. At the same time, the receiver P_n constructs a labeled graph called the "payment tree" that is stored locally by P_n and grows with each acknowledged subpayment.

Let us now explain how the payment tree is constructed. Consider again Fig. 1 (c). As explained before, the order of message acknowledgment can be arbitrary. In what follows, we assume that the receiver P_6 first acknowledges the subpayment that came along the path $P_1 \rightarrowtriangle P_3 \rightarrowtriangle P_6$. This means that P_6 "accepts" that 1¢ will be transferred to her from P_1 via

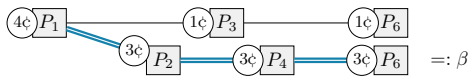
path $P_1 \rightarrowtriangle P_3 \rightarrowtriangle P_6$, or, in other words: 1¢ will be "passed" through each of P_1, P_3, and P_6 (note that we included here the sender P_1 and the receiver P_6). This can be depicted as the following graph that consists of a single path that we denote α:



(1)

To acknowledge the subpayment that was pushed along the path $P_1 \rightarrowtriangle P_3 \rightarrowtriangle P_6$ party P_6 signs α and sends it to P_3. Such signed information will be called a "subreceipt" and denoted $\wr\alpha\int_{P_n}$. By providing this subreceipt, party P_6 also gets 1¢ in the $P_3 \multimap P_6$ (these coins were blocked by P_3 in this channel when the "push" message was sent). The graph from Eq. (1) is the first version of the payment tree that, as mentioned above, the receiver P_6 stores locally.

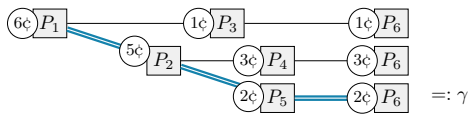
Now, suppose the next subpayment that P_6 wants to acknowledge is the one that came along the path $P_1 \rightarrowtriangle P_2 \rightarrowtriangle P_4 \rightarrowtriangle P_6$, i.e., P_6 accepts that 3¢ will be transferred to her from P_1 via path $P_1 \rightarrowtriangle P_2 \rightarrowtriangle P_4 \rightarrowtriangle P_6$. The receiver P_6 now modifies the payment tree as follows:



(2)

Analogously to what we saw before, this tree represents the total amount of coins that will be "passed" through different parties from P_1 to P_6 after acknowledging this subpayment is completed. In Eq. (2) the thick line (denoted β) corresponds to the "new" path, and the thin one is taken from Eq. (1), except that P_1 is labeled with "4¢". This is because the total amount of coins that will be passed through P_1 is equal to the sum of the coins passed before (1¢) and now (3¢). Party P_6 now signs path β to create a subreceipt that she sends to P_4 to claim 3¢ in the channel $P_4 \multimap P_6$.

Finally, P_6 acknowledges the subpayment along the path $P_1 \rightarrowtriangle P_2 \rightarrowtriangle P_5 \rightarrowtriangle P_6$. This is done similarly to what we did before. The resulting tree is now as follows.



(3)

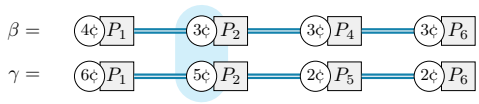
Note that we performed "summing" in two places on Eq. (3): at the node P_1 (where we computed 6¢ as 4¢ + 2¢) and an P_2 (where 5¢ = 2¢ + 3¢). Labeled path γ is now signed by P_6 and sent to P_5 as subreceipt in order to claim 2¢.

The payment trees whose examples we saw in Eqs. (1)–(3) are defined formally (in a slightly more general version) in Sec. 3.2. Their main feature is that the value of coins in the label of each node P is equal to the sum of the labels of the children of P. A recursive application of this observation implies that the coin value of a label of P is equal to the sum of labels in the leaves of the subtree rooted in P. In particular: the label on the root of the entire tree equals the sum of the values in the leaves.

Acknowledging subpayments by the intermediaries. We now show how the intermediaries $P_2, \ldots, P_{n-1}$ acknowledge the subpayments. On a high level, this is done by propagating the subreceipts (issued by P_n) from right to left. Each party may receive several such subreceipts (if she decides to split a given subpayment). Let $\mathcal{W}$ be the set of such subreceipts . When a party P wants to acknowledge the subpayment, she chooses (in a way that we explain below) one of the subreceipts ζ from her set $\mathcal{W}$. She then forwards it back in the left direction to the party P' that pushed the given subpayment to her. As a result P gets v¢ in the channel $P' \multimap P$. To determine the value of v¢ the following rule is used: it is defined as the label

of P on the path ζ. Given this, the rule for choosing $\zeta \in \mathcal{W}$ is pretty natural: P simply chooses such the ζ that maximizes v. Such ζ will be called a "leader" of $\mathcal{W}$ (at node P). To illustrate it, let us look again at our example from Fig. 1.

First, observe that P_3 holds only one subreceipt (i.e., the signed path α). She simply forwards it to P_1 and receives 1¢ in the channel $P_1 \multimap P_3$. Note that this is exactly equal to the value that she "lost" in the channel $P_3 \multimap P_6$, and hence the balance neutrality property holds. The situation is a bit more complicated for P_2 since she holds two paths signed by the receiver: β (defined on Eq. (2)) and γ (from Eq. (3)). By applying the rule described above, P_2 chooses the leader ζ at P_2 to be equal to γ (since 5¢ > 3¢). This is depicted below (the shaded area indicates the labels that are compared).



(4)

What remains is to argue about balance neutrality for P_2, i.e., that number of coins received by P_2 in the channel $P_1 \multimap P_2$ is equal to the sum of coins that she "lost on the right-hand side". In this particular example, it can be easily verified just by looking at Eq. (4) (5¢ are "gained", and 2¢ + 3¢ are "lost"). In the general case, the formal proof is based on the property that the value of coins in the label of each node P in a payment tree is equal to the sum of the labels of the children of P.

Final receipt produced by P_1. Once all subpayments are completed, P_1 decides to conclude the procedure and obtain the final receipt for the entire payment (see Sec. 2.1.1). Again, P_1 holds a "payment report" $\mathcal{W}$, i.e. a set of paths signed by P_6. In the case of our example, these paths are α (sent to P_1 by P_3) and γ (sent by P_2). Party P_1 chooses her "receipt" similarly as the intermediaries choose which subreceipt to forward. More precisely, let ζ be the path that is the leader of $\mathcal{W}$ at node P_1. This path becomes the final receipt. The amount of transferred coins equals the label of P_1 in ζ. In our case, the leader ζ is clearly γ (since its label at P is "6¢", while the label of γ at P is "1¢", cf. Eqs (1) and (3)). Hence, γ becomes the final receipt for the payment of 6 coins.

"Fairness for the sender" follows the same argument as "balance neutrality for the intermediaries". For "fairness for the receiver," observe that ζ is signed by the receiver and is taken from the payment tree (created and maintained by the receiver). To finish the argument, recall that: (a) as observed before, the label in the root of such a tree is always equal to the sum of the labels in its leaves, and (b) this sum is exactly equal to the total amount of coins that the receiver received from its neighbors during this payment procedure.

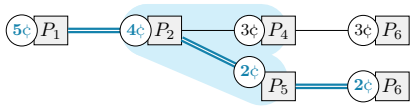
Dealing with malicious behavior. The primary type of malicious behavior that we have to deal with is cheating by the receiver P_n, whose goal could be to get more coins than appears on the final receipt held by the sender P_1. This could potentially be done at the cost of P_1 or some of the intermediaries. So far, we have not described how to guarantee that P_n produces the subreceipts correctly. As already highlighted, our trick is to let a malicious P_n arbitrarily produce the subreceipts and later let other parties verify P_n's operation. This is based on the idea of fraud-proofs: if an intermediary P finds proof that P_n is cheating, she can automatically claim all coins that were pushed to her by forwarding this proof "to the left". In this way, the cheating proof reaches the sender P_1, who can now use it as the

receipt for transferring the total amount that was requested (recall that P_1 holds an "invoice" from P_n). Suppose, e.g., that in our scenario P_6 cheats by sending to P_5, instead of γ (see Eq. (3)), the following subreceipt:

$\widehat{\gamma} :=$

(5)

The receiver does it to make P_1 hold a receipt for 5¢, while in fact receiving 6¢. Party P_5 has no way to discover this fraud attempt (since from her local perspective everything looks ok), so 2¢ get transferred to P_6 in the channel $P_5 \multimap P_6$. Party P_5 forwards $\widehat{\gamma}$ to P_2 and gets 2¢ in the channel $P_2 \multimap P_5$ (hence the "balance neutrality" property for her holds). Now look at this situation from the point of view of P_2. In addition to $\widehat{\gamma}$ she got one more subreceipt, namely β (see, e.g., Eq. (4)). Party P_2 preforms a "consistency check" by combining $\widehat{\gamma}$ and β. This is done by trying to locally reconstruct the part of the payment tree that concerns P_2. This is done as follows. First observe that the value on the label of P_1 in β is 4¢, which is smaller than the label of P_1 in $\widehat{\gamma}$ (which is equal to 5¢). This means that β had to be signed by P_6 *before* she signed $\widehat{\gamma}$. Hence P_2 first writes down β, and then on top of it she writes $\widehat{\gamma}$ (possibly overwriting some values). Normally (i.e. when P_6 is honest), this should result in a subtree of the tree from Eq. (3). However, since P_6 was cheating the resulting graph is different. Namely, P_2 reconstructs the following:



(6)

It is now obvious that P_6 is cheating since the labels on the children of P_2 sum up to 5¢, which is larger than 4¢ (the label of P_2). This "inconsistency" is marked as a shaded region on Eq. (6). Hence the set $\{\beta, \widehat{\gamma}\}$ is a fraud-proof against P_6. As described above, once we get such proof, we are "done": each intermediary can claim all money that was blocked for her, and the receiver can use it as a receipt that *all* the coins were transferred. Let us stress that, of course, none of the parties assumes a priori that P_6 is honest, and hence the "consistency check" is always performed.

3 Technical details

We now proceed to the formal exposition of the ideas already presented informally in Sec. 2. We start with defining a generalization of the term "paths" that were informally introduced before. As already explained, to be as general as possible, the NAPS definition permits that several subpayment of the same payment μ are routed via the same party independently. Consider, e.g., the following scenario: 2¢ is sent from P_1 to P_4 via a path $P_1 \rightarrowtriangle P_2 \rightarrowtriangle P_3 \rightarrowtriangle P_4$. This amount is first split by P_2 as: 1¢ + 1¢, and each 1¢ coin is pushed to P_3, who, in turn, pushes each of them further to P_4. Obviously, both 1¢ coins traveled along $P_1 \rightarrowtriangle P_2 \rightarrowtriangle P_3 \rightarrowtriangle P_4$, but nevertheless, they have to be considered as separate subpayments. In order to uniquely identify each of them, we augment the definition of "path" to include also "nonces" that will make them unique (in the abovementioned situations). To distinguish such paths from those we used in the informal part we denoted them as strings of pairs (party,nonce). A nonce is added in every hop. For example, in the above scenario: the (augmented) paths are as follows $\langle (P_1, \mu_1), (P_2, \mu_2), (P_3, \mu_3), (P_4, \mu_4) \rangle$ and $\langle (P_1, \mu_1), (P_2, \mu_2), (P_3, \mu'_3), (P_4, \mu'_4) \rangle$ (where for both $i = 3, 4$ we have that μ_i and μ'_i are distinct). Moreover, we assume that μ_1 ("contributed" by the sender P_1) is equal to the nonce that identifies the entire payment.

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 *path over* $\mathcal{G}$ *(for payment* μ*)* if each $\mu_i \in \mathcal{N}$ is a nonce, each $P_{i_j} \multimap P_{i_{j+1}}$ is an edge in $\mathcal{G}$, and $P_{i_1} = P_1$. We also assume that a path corresponding to a payment μ always starts with (P_1, μ). We say that P *appears on* π *(at position* j*)* if we have that $P = P_{i_j}$. We assume that every P appears at most once on π, or, in other words: the paths have no loops. In the sequel, every *party* or *functionality* is modeled as poly-time interactive Turing machine. Throughout this section, P denotes a party, u, v and w are non-negative integers denoting the amounts of coins, μ is a nonce, π is a path over $\mathcal{G}$, and t is time.

3.1 NAPS formal security definition

The protocol is parameterized with a security parameter 1^κ known to all machines. The protocol is executed by parties $P_1, \ldots, P_n$, who know each other's public keys (this is easy to achieve in real life using existing underlying blockchain infrastructure). The protocol also comes with an incorruptible party *RVM* called *receipt verification machine*. The role of this machine is to verify a receipt issued by P_n for payment μ. If this machine outputs $(\mathsf{i\text{-}sent}, \mu, w)$ to $\mathcal{Z}$ then we consider payment μ to be completed with the total amount of w coins transferred from P_1 to P_n. It models 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. 2.1.1. Following the tradition in formal cryptography, we first describe how network communication is organized. Then we introduce the notions of "adversary" and "environment". Afterward, we specify the security requirements of the protocol by describing the "ideal" and "real" models. Finally, we define security by comparing these two models. Both the ideal and the real model come with a functionality $\mathsf{Accounts}_{\mathcal{G}}$. This functionality (for the lack of space, presented in detail in the extended version of this paper [7]) is used to model the amounts of coins that the parties have in the channels. It is initialized with $\mathcal{G}$ and accepts messages $(\mathsf{trans}, P_i, P_j, v)$ that are used to transfer v coins from P_i to P_j in channel $P_i \multimap P_j$.

The network model. We assume a synchronous communication network, i.e., the execution of the protocol happens in rounds. The notion of rounds is just an abstraction that simplifies our model and has been used frequently in this area in the past (see, e.g., [5, 6]). Whenever we say that some operation (e.g. sending a message or simply staying in idle state) *takes between* τ *and* τ' *rounds*, we mean that it is up to the adversary to decide how long this operation takes (as long as it takes between τ and τ' rounds). The same convention applies to statements like "it takes at most/at least τ rounds". We assume that every machine is activated in each round. The communication between every two parties P and P' and between a party and an ideal functionality takes 1 round. The adversary can delay messages sent between other machines by at most Δ rounds. This will always be stated explicitly. The links between all the entities in the system are secure (encrypted and authenticated). To avoid replay attacks, we assume that every party (both in the ideal and real scenario) rejects a message m if she already received m before. Messages are tuples starting with keywords written in sans-serif. We also use the following convention. When we say that a party waits to receive a "message m of a form F", we mean that all messages of a different form are ignored. For example, if form F is $(\mathsf{i\text{-}push}, (\pi || \langle (P, \mu), (P', \mu') \rangle, v, t)$ this means that m has to start with an "i-push" keyword, followed by a parameter denoting a path that ends with two elements (denoted (P, μ) and (P', μ') for future reference), parameter v denoting a number of coins, and t denoting time.

The adversary and the environment. The protocol is attacked by a poly-time rushing adversary $\mathcal{A}$ who can *corrupt* some parties (when a party is corrupt $\mathcal{A}$ learns all its secrets and takes complete control over it). A party that has not been corrupt is called *honest*. To model that honest parties can make internal decisions about the protocol actions, we use the concept of an *environment*. This notion is taken from the UC framework; however, to keep things simple, we do not provide a complete UC-composable analysis of our protocol (in particular: since we do not aim at proving composability, we do not have the "session ids," and we use a simplified modeling of time). The environment and the adversary take as input $\mathcal{G}$ and the security parameter. The environment and the adversary can freely read the state of the $\mathsf{Accounts}_{\mathcal{G}}$ functionality. Additionally, we allow the ideal-model adversary to transfer coins from a dishonest party to an honest one. This corresponds to the fact that we allow corrupt parties to behave irrationally and lose coins. $\mathcal{A}$ and $\mathcal{Z}$ can communicate. At the end of its execution, $\mathcal{Z}$ produces an output.

The ideal model. Following the conventions of the UC framework, we assume that in the ideal model, the parties simply forward to the ideal functionality the messages that they receive from $\mathcal{Z}$. For a NAPS protocol with arity δ executed over graph $\mathcal{G}$ the corresponding ideal functionality is denoted $\mathsf{NAPS}^{\delta}_{\mathcal{G}}$ and presented on Fig. 2.

The messages exchanged in the ideal model are indicated with a prefix "i-". Let us now discuss the messages exchanged between the parties and the ideal functionality parties . Note that this functionality does not explicitly send any messages to the simulator. The simulator interacts with the ideal functionality via the corrupt parties she controls. To initiate a new payment μ parties P_1 and P_n send respectively a message $\mathsf{i\text{-}send}(\mu, v, t)$ to P_1 and $\mathsf{i\text{-}receive}(\mu, v, t)$ to P_n. We require that these messages have to be sent simultaneously by P_1 and P_n. This corresponds to an assumption that the parties P_1 and P_n agreed on transferring the coins beforehand. Once the transfer is finished, party *RVM* receives a message $\mathsf{i\text{-}acknowledged}(\langle(P, \mu)\rangle, s)$ from the ideal functionality. The functionality $\mathsf{NAPS}^{\delta}_{\mathcal{G}}$ maintains a set Ψ that contains all the push requests that have not yet been acknowledged. By push *requests* we mean tuples (π, v, t) such that some party sent $(\mathsf{i\text{-}push}, \pi, v, t)$ to the functionality. If such a push request is in Ψ then we say it is *open*. This indicates that the functionality is currently working on pushing v coins that already "traveled" along the path π, and the deadline for this is t. The amount of coins still waiting to be delivered is maintained using the function *remaining*. The push requests are created recursively. Suppose there is an open push request $(\pi||\langle(P, \mu)\rangle, v, t)$. To push it to a party P' party P sends a message $(\mathsf{i\text{-}push}, (\pi||\langle(P, \mu), (P', \mu')\rangle), v', t')$ to $\mathsf{NAPS}^{\delta}_{\mathcal{G}}$. Once the transfer is finished, party P is informed about how many coins were transferred within this push request. This is done via a message $\mathsf{i\text{-}acknowledged}(\langle(P, \mu), (P', \mu')\rangle), v'')$, where v'' specifies the amount of coins that were transferred. If there are no open push requests of a form $(\pi||\langle(P, \mu), (P', \mu')\rangle, v, t)$ then a party P can decide to close a given push request by sending a message $\mathsf{i\text{-}acknowledge}((\pi||\langle(P, \mu), (P', \mu')\rangle, v, t)$ to $\mathsf{NAPS}^{\delta}_{\mathcal{G}}$. The function *remaining* and the accounts in the $P \multimap P'$ channels are updated accordingly (by sending messages to the $\mathsf{Accounts}_{\mathcal{G}}$ functionality). If P_1 wants to finish processing given payment μ (this is possible only if there are no open push requests corresponding to μ other than the request $(\langle(P, \mu)\rangle, v, t)$) then she sends an $\mathsf{acknowledge}$ message to $\mathsf{NAPS}^{\delta}_{\mathcal{G}}$. The "ideal model" adversary will also be called the *simulator* and denoted $\mathcal{S}$. We assume that $\mathcal{S}$ has access to the ideal functionality. The *output of the ideal execution of* $\mathsf{NAPS}^{\delta}_{\mathcal{G}}$ against $\mathcal{S}$ and $\mathcal{Z}$ with security parameter 1^{κ} is a random variable $\mathsf{Ideal}(\mathsf{NAPS}^{\delta}_{\mathcal{G}}, \mathcal{S}, \mathcal{Z}, 1^{\kappa})$ denoting the output of $\mathcal{Z}$ In the extended version of this paper [7] we argue why the informal properties from Sec. 2.1.1 are implied by this ideal functionality.

The ideal functionality $\mathsf{NAPS}^{\delta}_{\mathcal{G}}$ is parametrized by a channel graph $\mathcal{G} = (\mathcal{P}, \mathcal{E}, \Gamma)$ and an arity parameter δ. It maintains a cash function $\widehat{\Gamma}$ initially equal to Γ and a set Ψ initially equal to $\emptyset$. Function $\widehat{\Gamma}$ is used to denote the current amount of coins available in the channels and set Ψ containing all *open push requests*. Moreover, the ideal functionality maintains a function *remaining* $: \Psi \to \mathbb{Z}_{\geq 0}$. It proceeds as follows.

Upon receiving a message of a form $(\mathsf{i\text{-}send}, \mu, u, t)$ from P_1 and $(\mathsf{i\text{-}receive}, \mu, u, t)$ from P_n (in the same round) – check if the following holds:

> *Correctness condition:* (a) you have not received an "i-send" or an "i-receive" message with this μ before and (b) the current time is greater than $t - \Delta$.

If it does not hold, then ignore this message. Otherwise (a) add $(\langle (P, \mu) \rangle, u, t)$ to Ψ and (b) let $\mathit{remaining}(\langle (P, \mu) \rangle, u, t) := v$.

Upon receiving a message of a form $(\mathsf{i\text{-}push}, (\pi || \langle (P, \mu), (P', \mu') \rangle), v, t)$ from P – check if the following holds:

> *Correctness condition:* (a) you have not received an "i-pushed" message with this $\langle (P, \mu), (P', \mu') \rangle$ before, (b) $P' \multimap P \in \mathcal{E}$, (c) $v \leq \widehat{\Gamma}^{P \multimap P'}(P)$, (d) the number of elements $(\pi || \langle (P, \mu), (P', \mu'), (P'', \mu'') \rangle), v', t')$ in Ψ (for any P'', μ'', v', and t) is less than δ, (e) the current time is greater than $t - \Delta$, and (f) if P is honest then $(\pi || \langle (P, \mu) \rangle) \in \Psi$ and $\mathit{remaining}(\pi || \langle (P, \mu) \rangle) \geq v$.

If it does not hold, then ignore this message. Otherwise: (a) add $(\pi || \langle (P, \mu), (P', \mu') \rangle), v, t)$ to Ψ, (b) decrement $\mathit{remaining}(\pi || \langle (P, \mu) \rangle), v, t)$ by v, (c) let $\mathit{remaining}(\pi || \langle (P, \mu), (P', \mu') \rangle), v, t) := v$, (d) decrement $\widehat{\Gamma}^{P \multimap P'}(P)$ by v, and (e) in the next round send a message $(\mathsf{i\text{-}pushed}, (\pi || \langle (P, \mu), (P', \mu') \rangle), v, t))$ to P'.
If time $t + \Delta$ comes and $((\pi || \langle (P, \mu), (P', \mu') \rangle), v, t))$ is still in Ψ then behave as if you received a message $(\mathsf{i\text{-}acknowledge}, (\pi || \langle (P, \mu), (P', \mu) \rangle))$ from P' (see below).

Upon receiving a message of a form $(\mathsf{i\text{-}acknowledge}, (\pi || \langle (P, \mu) \rangle))$ from P – check if the following holds:

> *Correctness condition:* (a) $(\pi || \langle (P, \mu) \rangle, t, v) \in \Psi$ (for some v and t), and (b) there does not exist a push request $((\pi || \langle (P, \mu) \rangle || \langle (P', \mu') \rangle), v', t')$ in Ψ (for any P', μ', v', t').

If it does not hold, then ignore this message. Otherwise let v be the value from the "Correctness condition" and let s be the sum of the v' values in all the messages $\mathsf{i\text{-}acknowledged}((\pi || \langle (P, \mu) \rangle || \langle (P', \mu') \rangle)), v')$ (for any (P', μ')) that were ever sent to P. If P_n is corrupt, allow the simulator to increase the value of s to any amount at most v. Consider the following cases:

- $P = P_1$ (note that in this case π is empty) – then in the next round send $\mathsf{i\text{-}sent}(\langle (P, \mu) \rangle, s)$ to *RVM*.
- $P \in \{P_2, \ldots, P_{n-1}\}$ – then let (P_k, μ_k) be the last element of π and then within time Δ (a) send a message $(\mathsf{trans}, P_k, P, s)$ to $\mathsf{Accounts}_{\mathcal{G}}$, (b) increment $\widehat{\Gamma}^{P_k \multimap P}(P_k)$ by $v - s$, (c) increment $\widehat{\Gamma}^{P_k \multimap P}(P)$ by s, (d) increment $\mathit{remaining}(\pi, v, t)$ by $v - s$, (e) remove $(\pi || \langle (P, \mu) \rangle)$ from Ψ, and (f) send $\mathsf{i\text{-}acknowledged}((\pi || \langle (P, \mu) \rangle), s)$ to P_k.
- $P = P_n$ – then let (P_k, μ_k) be the last element of π and then within time Δ (a) send a message $(\mathsf{trans}, P_k, P, v)$ to $\mathsf{Accounts}_{\mathcal{G}}$, (b) increment $\widehat{\Gamma}^{P_k \multimap P}(P_k)$ by v, (c) remove $(\pi || \langle (P, \mu) \rangle)$ from Ψ, and (d) send $\mathsf{i\text{-}acknowledged}((\pi || \langle (P, \mu) \rangle), s)$ to P_k.

Figure 2 The ideal functionality $\mathsf{NAPS}^{\delta}_{\mathcal{G}}$.

The real model. In the real model, the parties communicate with the environment and interact with each other directly. Before the protocol starts, we generate a (public key, secret key) pair for each P_i and give to P_i its secret key as input. Moreover, all parties (including RVM and $\mathcal{A}$) get the public keys of the other parties. For each pair $\{P_i, P_j\}$ such that $P_i \multimap P_j \in \mathcal{E}$ the parties P_i and P_j also have access to an uncorruptible *state channel machine* $C^{P_i \multimap P_j}$, which in turn, has access to $\mathsf{Accounts}_{\mathcal{G}}$ (the parties do not have a direct access to $\mathsf{Accounts}_{\mathcal{G}}$). Sending messages to $\mathsf{Accounts}_{\mathcal{G}}$ takes time at most Δ. The state channel machines and the parties know the public keys of all the parties. Altogether, a *NAPS protocol for a channel graph $\mathcal{G}$ with arity δ* is a tuple of machines $\Pi^{\delta}_{\mathcal{G}} := (RVM, P_1, \ldots, P_n, \{C^{P_i \multimap P_j}\}_{P_i \multimap P_j \in \mathcal{E}})$. The *output of the real execution of* Π with security parameter 1^{κ} is a random variable $\mathsf{Real}(\Pi^{\delta}_{\mathcal{G}}, \mathcal{A}, \mathcal{Z}, 1^{\kappa})$ denoting the output of $\mathcal{Z}$. We define security by requiring that no environment can distinguish between the ideal and the real model. In the definition, we use the concept of computational indistinguishably. From the construction of the ideal functionality, it is easy to see that all the informal security properties (fairness to the sender and the receiver and the balance neutrality) hold for EthNA.

▶ **Definition 1.** *A tuple $\Pi^{\delta}_{\mathcal{G}}$ is a* secure Non-Atomic Payment Splitting (NAPS) protocol for $\mathcal{G}$ and δ *if for every adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that and every $\mathcal{Z}$ the families of random variables $\{\mathsf{Ideal}(\mathsf{NAPS}^{\delta}_{\mathcal{G}}, \mathcal{S}, \mathcal{Z}, 1^{\kappa})\}_{\kappa}$ and $\{\mathsf{Real}(\Pi^{\delta}_{\mathcal{G}}, \mathcal{A}, \mathcal{Z}, 1^{\kappa})\}_{\kappa}$ are computationally indistinguishable*

3.2 Formal description of EthNA

Let us start by providing formal definitions of some of the terms already informally introduced in Sec. 2.2. For a graph $\mathcal{G}$ and a nonce μ, a *subreceipt (over $\mathcal{G}$, for payment μ)* is a pair $\wr \pi, \lambda \int_{P_n}$ signed by P_n such that π is a path over $\mathcal{G}$ (for payment μ) with P_n appearing on the last position of π, and λ is a non-increasing sequence of positive integers, such that $|\lambda| = |\pi|$. A *payment report for μ* is a set $\mathcal{W}$ of subreceipts for μ such that π identifies a member of $\mathcal{W}$ uniquely, i.e.: $(\wr \pi, \lambda \int_{P_n} \in \mathcal{W}$ and $\wr \pi, \lambda' \int_{P_n} \in \mathcal{W})$ implies $\lambda = \lambda'$. For example, α, β, and γ in Sec. 2.2 are subreceipts, and the set $\{\beta, \gamma\}$ (see Eq. (4)) is a payment report (except that in that informal description, we omitted the nonces). For a payment report $\mathcal{W}$ a subreceipt $\wr(\pi, \lambda \int_{P_n}$ is a *leader of $\mathcal{W}$ at node P* if P appears on π at some position i, and for every $\wr \pi', \lambda' \int_{P_n} \in \mathcal{W}$ we have that $\lambda[i] \geq \lambda'[i]$. This notion was already discussed in Sec. 2.2, where in particular, we said that the leader of a payment report $\{\alpha', \gamma\}$ (on Eq. (4)) is γ. In normal cases (i.e. if P_n is honest), the leader is always unique and is equal to the *last* subreceipt of a from $\wr(\pi || \sigma'), \lambda' \int_{P_n}$ signed by P_n, however in general this does not need to be the case. When we talk about *the* leader of $\mathcal{W}$ at P we mean the leader that is the smallest according to some fixed linear ordering.

As mentioned in Sec. 1.2, EthNA is constructed using fraud-proofs. Formally, a *fraud-proof (for μ)* is a payment report $\mathcal{Q}$ for μ of a form $\mathcal{Q} = \{\wr(\sigma || \pi_i), \lambda_i \int\}_{i=1}^{m}$, where all the $\pi_i[1]$'s are pairwise distinct , such that the following condition holds: $\max_{i:=1,\ldots,m} \lambda_i[|\sigma|] < \sum_{i:=1}^{m} \lambda_i[|\sigma| + 1]$. For an example of a fraud-proof (with nonce missing from the picture) see Eq. (6). If EthNA has arity at most δ (see Sec. 2.1), then we require that $m \leq \delta$. Informally speaking, these conditions mean simply that in $\mathcal{Q}$ the largest label of σ is smaller than the sum of all labels of σ's children. If none of the subsets of a payment report $\mathcal{W}$ is a fraud-proof, then we say that $\mathcal{W}$ *is consistent*. As we show later, if P_n is honest, then $\mathcal{W}$ is always consistent. Note that the description of set $\mathcal{Q}$ as defined above can be quite large (it is of size $O(\delta \cdot (\ell + \kappa))$, where δ is EthNA's arity, ℓ is the maximal length of paths, and κ is the security parameter (we need this to account for the signature size). Luckily, there is a

simple way to "compress" it to $O(\delta \cdot \kappa)$ (where κ is the security parameter) by exploiting the fact that the only values that are needed to prove cheating are the positions on the indices $|\sigma|$ and $|\sigma|+1$ of the λ's. We describe the compression ideas in the extended version of this paper [7] .

The formal description of EthNA appears on Figs. 3, 4, and 5. It uses a subroutine algorithm Add_Φ that we describe in a moment. We outsource some of the protocol to a procedure handle-path (depicted in the same figure) to avoid repeating the same instructions. The receipt verification machine *RVM* is presented in Fig. 6

Party P_1

Upon receiving an admissible message of a form $(\mathsf{i\text{-}send}, \mu, u, t)$ from the environment $\mathcal{Z}$ and in the next round a message $(\mathsf{invoice}, \wr\mu, u, t\int_{P_n})$ from P_n – store this message, and execute the $\mathsf{handle\text{-}path}(P_1, \langle(P_1, \mu)\rangle, v, t)$ procedure presented on Fig. 4. Let (R, v) be the output of this procedure. Send $(\mathsf{acknowledged}, \mu, (\wr u, \mu, t\int_{P_n}, R))$ to *RVM*.

Party P_i for $i = 2, \ldots, n-1$

Upon receiving a message of a form $(\mathsf{push}, (\wr(\pi||\langle(P, \mu), (P', \mu')\rangle), v, t\int_P)$ from some party P – ignore this message if at least one of the following happens: (a) $P' \neq P_i$ or (b) $t > \tau + \Delta$ (where τ is the current time). Otherwise run the *path handling procedure* $\mathsf{handle\text{-}path}(P_i, (\pi||\langle(P', \mu'), (P, \mu)\rangle), v, t)$ presented on Fig. 4. Let (R, v') be the output of this procedure and send $(\mathsf{acknowledge}, \wr(\pi||\langle(P', \mu'), (P, \mu)\rangle), v, t\int_{P_j}, R)$ to $C^{P \multimap P_i}$.

Party P_n

Wait to receive admissible messages of a form $(\mathsf{i\text{-}receive}, \mu, u, t)$ from the environment $\mathcal{Z}$. Handle each of them as follows.

Otherwise let β^μ be an integer variable initially equal to u and send a message $(\mathsf{invoice}, \wr\mu, u, t\int_{P_n})$ to P_1. Let Φ^μ be a variable containing a payment report that is initially empty. Then wait (until time t comes) to receive messages of the following form:

Message $(\mathsf{push}, \wr(\pi||\langle(P, \mu), (P_n, \mu')\rangle), v, t'\int_P)$ from some party P (with $t' \leq t$) – send a message $(\mathsf{pushed}, (\pi||\langle(P, \mu), (P_n, \mu')\rangle), v, t')$ to $\mathcal{Z}$.
If within time t you receive a message $(\mathsf{i\text{-}acknowledge}(\pi||\langle(P, \mu), (P_n, \mu')\rangle)$ from $\mathcal{Z}$ and $v > \beta^\mu$ then execute $\mathsf{Add}_{\Phi^\mu}((\pi||\langle(P, \mu), (P_n, \mu')\rangle), v)$. Let λ be the output of this procedure. Send a message $(\mathsf{acknowledge}, \wr(\pi||\langle(P, \mu), (P_n, \mu')\rangle), v, t'\int_P, \wr(\pi||\langle(P, \mu), (P_n, \mu')\rangle), \lambda\int_{P_n})$ to *RVM*.

Figure 3 The EthNA protocol for the parties.

The parties receive the "ideal model" messages (starting with a prefix "i-") from $\mathcal{Z}$. By saying that a message (received from $\mathcal{Z}$) is *admissible*, we mean that it satisfies the "correctness conditions" from Fig. 2. The push requests are executed by direct communication between the parties, and the payment acknowledgment is done via the state channel machines. Let us comment on the types of messages that are sent within the protocol (see also the cheat sheet in [7]). The messages that are used are: "push" to push a subpayment (the corresponding message sent by the channel to the other party is "pushed"), "acknowledge" to acknowledge a subpayment (the corresponding message is "acknowledged"). The value R contains either a subreceipt (this is the most common case), or a fraud-proof, or a message "'empty" denoting the fact that no subpayments have been acknowledged by P_n.

Let $\mathcal{W}^\pi$ be a variable containing a set of subreceipts that initially is empty and let $\omega^\pi := \delta$. Send $(\mathsf{i\text{-}pushed}, \pi, v, t)$ to $\mathcal{Z}$ and wait for the following messages forms from $\mathcal{Z}$:

Message $(\mathsf{i\text{-}push}, (\pi || \langle (P, \mu), (P', \mu') \rangle, v', t')$ (for some $v' \leq \alpha^\pi$ and μ and μ' and P' such that $P \multimap P' \in \mathcal{E}$) – handle each such a message as follows. If $\omega^\pi = 0$ then ignore this message. Otherwise let $\alpha^\pi := \alpha^\pi - v'$ and decrease ω^π by 1. Then send a message $(\mathsf{push}, \wr(\pi || \langle (P', \mu') \rangle), v', t' \int_P)$ to P' and wait until round t to receive a message od one of the following forms:

- $(\mathsf{acknowledged}, (\pi || \langle (P', \mu') \rangle), \mathsf{empty})$ from $C^{P_i \multimap P'}$ – then let $\alpha^\pi := \alpha^\pi + v'$ and send a message $(\mathsf{i\text{-}acknowledged}, (\pi || \langle (P', \mu') \rangle), 0)$ to $\mathcal{Z}$,
- $(\mathsf{acknowledged}, (\pi || \langle (P', \mu') \rangle), \wr \psi, \lambda \int_{P_n})$, where ψ is such that $(\pi || \langle (P', \mu') \rangle)$ is a prefix of ψ – then store $\wr \psi, \lambda \int_{P_n}$ in $\mathcal{W}^\pi$ by letting $\mathcal{W}^\pi := \mathcal{W}^\pi \cup \{\wr \psi, \lambda \int_{P_n}\}$.

Let $\widehat{v} := \lambda[|\pi| + 1]$. Let $\alpha^\pi := \alpha^\pi + v' - \widehat{v}$ and send $(\mathsf{i\text{-}acknowledged}, (\pi || \langle (P', \mu') \rangle), \widehat{v})$ to $\mathcal{Z}$, or

- $(\mathsf{acknowledged}, (\pi || \langle (P', \mu') \rangle), (\mathsf{fraud\text{-}proof}, w))$ – then store $(\mathsf{fraud\text{-}proof}, w)$ and send a message $(\mathsf{i\text{-}acknowledged}, (\pi || \langle (P', \mu') \rangle), v')$ to $\mathcal{Z}$.

Message $(\mathsf{i\text{-}acknowledge}, \pi)$ (or time t comes) – if you are still waiting in the procedure of handling some "i-push" message (see above), then ignore this message. Otherwise, do the following

- If you stored $(\mathsf{fraud\text{-}proof}, w)$ (for some (P', μ')) or if $\mathcal{W}^\pi$ is inconsistent and w is the fraud-proof – then output $((\mathsf{fraud\text{-}proof}, w), v)$.
- Otherwise: if $\mathcal{W}^\pi$ is empty then output empty.
- Otherwise let $\wr \psi, \lambda \int_{P_n}$ be the leader of $\mathcal{W}^\pi$ at $\widetilde{P}$, where $\widetilde{P}$ is the last party on π. Output $(\wr \psi, \lambda \int_{P_n}, \lambda(|\pi|))$.

Figure 4 Path handling procedure $\mathsf{handle\text{-}path}(P, \pi, v, t)$.

Recall that the values of registers $\Gamma^{P_i \multimap P_j}(P_i)$ and $\Gamma^{P_i \multimap P_j}(P_j)$ are pre-loaded before the execution started. Wait for messages from P_i and P_j.

Upon receiving a message of a form $(\mathsf{acknowledge}, \wr \pi, v, t \int_{P_k}, \mathsf{empty})$ from a party P (such that $\{P_k, P\} = \{P_i, P_j\}$) – send $(\mathsf{acknowledged}, \pi, \mathsf{empty})$ to P_k.

Upon receiving a message of a form $(\mathsf{acknowledge}, \wr \pi, v, t \int_{P_k}, \wr \psi, \lambda \int_{P_n})$ from a party P where (a) current time is at most t, (b) π is a path with a suffix $\langle (P_k, \mu), (P, \mu') \rangle$ (for some μ and μ'), (c) π is a prefix of ψ, (d) $\lambda[|\pi|] \leq \Gamma^{P_i \multimap P_j}(P_k)$, and (e) $\{P_k, P\} = \{P_i, P_j\}$ – then send a message of a form $(\mathsf{trans}, P_k, P, \lambda[|\pi|])$ to $\mathsf{Accounts}_\mathcal{G}$ and a message $(\mathsf{acknowledged}, \pi, \wr \psi, \lambda \int_{P_n})$ to P_k.

Upon receiving a message of a form $(\mathsf{acknowledge}, \wr \pi, v, t \int_{P_k}, (\mathsf{fraud\text{-}proof}, w))$ from a party P where (a) current time is at most t, (b) π is a path with a suffix $\langle (P_k, \mu), (P, \mu') \rangle$ (for some μ, μ' and P_k), (c) $v \leq \Gamma^{P_i \multimap P_j}(P_k)$, and (d) w is a fraud-proof – then send a message $(\mathsf{trans}, P_k, P, v)$ to $\mathsf{Accounts}_\mathcal{G}$ and send a message $(\mathsf{acknowledged}, \pi, (\mathsf{fraud\text{-}proof}, w))$ to P_k.

Figure 5 The EthNA state channel machine $C^{P_i \multimap P_j}$.

Upon receiving a message $(\mathsf{acknowledged}, \mu, (\wr u, \mu', t \int_{P_n}, R))$ from P_1 (such that $\mu = \mu'$ and you have not received an "acknowledged" message with this μ from P_1 before) – let

$$w := \begin{cases} u & \text{if } R = (\mathsf{fraud\text{-}proof}, w), \\ 0 & \text{if } R = \mathsf{empty} \\ \lambda[1] & \text{if } R = \wr \mathsf{acknowledge}, \psi, \lambda \int_{P_n}, \end{cases}$$

where w is a fraud-proof. Send $(\mathsf{i\text{-}sent}, \mu, w)$ to $\mathcal{Z}$

Figure 6 Receipt Verification Machine RVM.

As described above, the main tasks of each party P_i (for $i = 2, \ldots, n-1$) are: (a) receive push requests from some P, (b) forward corresponding push request in the direction of P_n, (c) receive information about how many coins were transferred, and (d) once you are done with handling all push requests: check if you received or you can find a fraud-proof – if yes, then forward this information back to P (via the state channel), and if not, then choose the leader of the set of receipts and forward it back to P (via the state channel). The procedure for P_1 is similar, except that P_1 is activated by a "send" message from $\mathcal{Z}$, and waits of the invoice from P_n. It then communicates with the receipt verification machine defined on Fig. 6 (see p. 19 in the appendix). Probably the most interesting part is the instructions for P_n. First, P_n (upon receiving an $\mathsf{i\text{-}receive}(\mu, u, t)$ message from $\mathcal{Z}$) sends an invoice to P_1. For every payment, μ party P_n maintains a payment tree Φ^μ that is initially empty. Payment trees were already discussed in Sec. 2.2. For a formal definition, consider some fixed μ and $\mathcal{G}$. During the execution of EthNA for $\mathcal{G}$ and μ, several subpayments are delivered to P_n. Let $\pi^1, \ldots, \pi^t$ denote the consecutive paths over which these subpayments go (of course, they need to be distinct), and let $v^i \in \mathbb{Z}_{>0}$ be the number of coins transmitted with each π^i. Let $\mathcal{W} := \{(\pi^i, v^i)\}_{i=1}^t$. Formally, a *payment tree* $\mathsf{tree}(\mathcal{W})$ is a labeled tree $(T, \mathcal{L})$, where T is the set of all prefixes of the π^i's, i.e., $T := \bigcup_i \mathsf{prefix}(\pi^i)$. . If EthNA has arity δ then the arity of T in every node $(\pi || \langle (P, \mu) \rangle)$ is at most δ. 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 path π gets labeled by the arithmetic sum of the value of the payments that were "passed through it". Clearly, the label $\mathcal{L}(\varepsilon)$ of the root node of $\mathsf{tree}(\mathcal{W})$ is equal to the sum of all v^i's, and hence it is equal to the total number of coins transferred by the subpayments in $\mathcal{W}$. We also have that for every path σ $\mathcal{L}(\sigma) = \sum_{\pi \text{ is a child of } \sigma} \mathcal{L}(\pi)$. It is also easy to see that $\mathsf{tree}(\mathcal{W})$ can be constructed "dynamically" by processing elements of $\mathcal{W}$ 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 $(\pi^1, v^1), (\pi^2, v^2), \ldots$. From the construction of the algorithm, it follows

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

This algorithm operates on a global state $\Phi = (T, \mathcal{L})$. Its side effect is a change of the global state. We assume that $v \in \mathbb{Z}_{>0}$ and $\pi \notin T$.

```
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)
```

immediately that if P_n starts with Φ being an empty tree, and then iteratively applies Add_Φ to (π^i, v^i)'s for $i = 1, \ldots, t$, then the final state of Φ is equal to $\mathsf{tree}(\mathcal{W})$. For example, if P_n applies this procedure to the situation in Fig. 1 she obtains the trees depicted on Eqs. (1)–(3). It is easy to see that if P_n applies the Add_Φ algorithm correctly, then the resulting sets $\mathcal{W}$ are never inconsistent (and hence no fraud-proof will ever be produced against an honest P_n).

The formal security analysis of this protocol is given in the following lemma, whose proof appears in [7].

▶ **Lemma 2.** *Assuming that the underlying signature scheme is existentially unforgeable under a chosen-message attack, EthNA is a secure NAPS protocol for every $\mathcal{G}$ and δ.*

3.3 Efficiency analysis

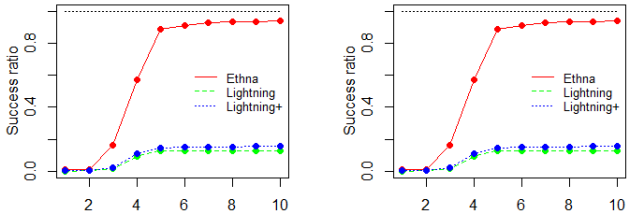
We consider separately the *optimistic* scenario (when the parties are cooperating) and the *pessimistic* one when the malicious parties slow down the execution. In the optimistic case, the payments are almost immediate. It takes 1 round for a payment to be pushed and 2 rounds to be acknowledged (due to the communication with the state channel machine). 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). During the acknowledgment, every malicious party can delay the process by time at most Δ. Hence, the maximal pessimistic time is $(1 + \Delta) \cdot \ell$. The second important measure is the blockchain costs, i.e., the fees that the parties need to pay. Below we provide a "theoretical" analysis of such costs. For the results of concrete experiments, see . Note that in the optimistic case, the only costs are channel opening and closing, and hence they are independent of the tree depth and of its arity. In the pessimistic case, all messages in state channels must be sent "via the blockchain". Let us consider two cases. In the first case, there is no fraud-proof. Then, the only message that is sent via the blockchain is $\mathsf{acknowledge}(\wr\phi, \lambda\int_{P_n})$, 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 if a fraud-proof appears. As remarked in Sec. 3.2 the size of a fraud-proof is $O(\delta \cdot (\ell + \kappa))$, where δ is EthNA's arity, ℓ is the maximal length of paths, 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 subpayments into too many subpayments), she might be forced to pay fees that depend on some (potentially larger) arity. This could result in griefing attacks, which is why we introduced a global limit on the arity. There are many ways around this. First, we could modify the protocol so that the fraud-proofs by P_n are posted directly in a smart contract on a blockchain so that all other parties do not need to re-post and can just refer to it. Moreover, the proof size can be significantly reduced (see the extended version of this paper [7]).

4 Practical aspects

Let us now we provide information about practical experiments of EthNA implementation. We implemented a simple version of EthNA in Solidity. The source code is available at github.com/Sam16450/NAPS-EthNA. The table in Fig. 7 (a) summarizes the execution costs *in terms of thousands of gas*, and depending on the arity δ and the maximal path length. The `constructor` denotes the 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 subreceipt when no cheating was discovered, and `closeDisagreement` – to finally close a channel after disagreement.

δ	path length	constructor	close	addState	addChea-tingProof	addComple-tedTransa-ction	close-Disagree-ment
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

(a)



(b) (c)

Figure 7 Experimental results.

Although routing algorithms are not the main topic of this work, we also performed some experiments with a routing algorithm built on top of EthNA. We took the network graph in our experiments from the Lightning network (from the website gitlab.tu-berlin.de/rohrer/discharged-pc-data) with approx. 6K nodes and 30K channels. Channel's capacities are chosen according to the normal distribution $\mathcal{N}(200, 50)$. Each transaction was split by applying the following rules. The sender and the intermediaries look at the channel graph and search for the set $\mathcal{X}$ of shortest paths that lead to the receiver (and have different first elements). Then they split the payment into values proportional to the capacity of the first channel in the path. In our simulations, we performed 100K transaction. The results appear in Fig. 7. The "success ratio" denotes the probability of complete success of an average payment. Each transaction had to be completed in a maximum of 50 rounds. "Lightning" refers to standard Lightning routing, and "Lightning+" refers to the Lightning algorithm that attempts to push payments multiple times. Transaction values are chosen uniformly from set (x_0, x_1), while in (b) we have $(x_0, x_1) = (10, 500)$ and in (c) we have $x_0 := 150, 200, 300, 400$ and $x_1 := 500$. Our experiments show that even this simple routing algorithm for EthNA works much better than Lightning.

5 Conclusions and future work

We have introduced a Non-Atomic Payment Splitting (NAPS) technique for the payment networks, constructed the EthNA protocol that uses it, and proven its security. Due to the limited space, we focused only on introducing the payment-splitting technique. This paper opens several exciting questions for future research. First, it would be interesting to develop routing algorithms that use this feature. Secondly, we did not address privacy and anonymity in this setting, and it would be interesting to explore this topic. Our solution strongly relies on the Turing completeness of the underlying blockchain platform. It would be interesting to examine if NAPS schemes can be implemented Non-Atomic also over legacy blockchains such as Bitcoin.

References

1. Vivek Kumar Bagaria, Joachim Neu, and David Tse. Boomerang: Redundancy improves latency and throughput in payment-channel networks. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*, volume 12059 of *Lecture Notes in Computer Science*, pages 304–324. Springer, 2020. doi:10.1007/978-3-030-51280-4_17.
2. Manuel M. T. Chakravarty, Sandro Coretti, Matthias Fitzi, Peter Gazi, Philipp Kant, Aggelos Kiayias, and Alexander Russell. Fast isomorphic state channels. In Nikita Borisov and Claudia Díaz, editors, *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part II*, volume 12675 of *Lecture Notes in Computer Science*, pages 339–358. Springer, 2021. doi:10.1007/978-3-662-64331-0_18.
3. Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In Andrzej Pelc and Alexander A. Schwarzmann, editors, *Stabilization, Safety, and Security of Distributed Systems - 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings*, volume 9212 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2015. doi:10.1007/978-3-319-21741-3_1.
4. Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. Multi-party virtual state channels. In Yuval Ishai and Vincent Rijmen, editors, *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*, volume 11476 of *Lecture Notes in Computer Science*, pages 625–656. Springer, 2019. doi:10.1007/978-3-030-17653-2_21.
5. Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel 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*, pages 106–123. IEEE, 2019. doi:10.1109/SP.2019.00020.
6. Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 949–966. ACM, 2018. doi:10.1145/3243734.3243856.
7. Stefan Dziembowski and Pawel Kędzior. Non-atomic payment splitting in channel networks. *IACR Cryptol. ePrint Arch.*, 2020. URL: https://eprint.iacr.org/2020/166.
8. Lisa Eckey, Sebastian Faust, Kristina Hostáková, and Stefanie Roos. Splitting payments locally while routing interdimensionally. *IACR Cryptol. ePrint Arch.*, 2020. URL: https://eprint.iacr.org/2020/555.
9. Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 473–489. ACM, 2017. doi:10.1145/3133956.3134093.
10. Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. Sok: Layer-two blockchain protocols. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*, volume 12059 of *Lecture Notes in Computer Science*, pages 201–226. Springer, 2020. doi:10.1007/978-3-030-51280-4_12.
11. Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. A composable security treatment of the lightning network. In *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*, pages 334–349. IEEE, 2020. doi:10.1109/CSF49147.2020.00031.

12 Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos 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*, pages 839–858. IEEE Computer Society, 2016. doi:10.1109/SP.2016.55.

13 Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 455–471. ACM, 2017. doi:10.1145/3133956.3134096.

14 Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo 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/.

15 Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. Sprites and state channels: Payment networks that go faster than lightning. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, volume 11598 of *Lecture Notes in Computer Science*, pages 508–526. Springer, 2019. doi:10.1007/978-3-030-32101-7_30.

16 Olaoluwa Osuntokun. [lightning-dev] amp: Atomic multi-path payments over lightning. https://tinyurl.com/29m2d7wr, 2018.

17 Dmytro Piatkivskyi and Mariusz Nowostawski. Split payments in payment networks. In Joaquín García-Alfaro, Jordi Herrera-Joancomartí, Giovanni Livraga, and Ruben Rios, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2018 International Workshops, DPM 2018 and CBT 2018, Barcelona, Spain, September 6-7, 2018, Proceedings*, volume 11025 of *Lecture Notes in Computer Science*, pages 67–75. Springer, 2018. doi:10.1007/978-3-030-00305-0_5.

18 Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments, 2016.

19 Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Kathleen Ruan, Parimarjan Negi, Lei Yang, Radhika Mittal, Giulia Fanti, and Mohammad Alizadeh. High throughput cryptocurrency routing in payment channel networks. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 777–796. USENIX Association, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/sivaraman.

20 Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. A^2l: Anonymous atomic locks for scalability in payment channel hubs. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1834–1851. IEEE, 2021. doi:10.1109/SP40001.2021.00111.

21 Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. *CoRR*, abs/1908.04756, 2019. arXiv:1908.04756.