Non-Atomic Payment Splitting in Channel Networks

Stefan Dziembowski and Paweł Kędzior

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

Related Version This is an extended version of a paper that appeared at the 5th Conference on Advances in Financial Technologies (AFT 2023).

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 partly 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., [13, 7]). Probably the best-known example of such a system is Ethereum. Several blockchain-based systems’ main limitations are delayed finality, lack of scalability, and non-trivial transaction fees. Off-chain channels [19, 4] 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 $\Delta$. We model amounts of coins as non-negative integers and write “$n\mathcal{C}$” 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: “$x\mathcal{C}$ in Alice’s account, $y\mathcal{C}$ in Bob’s account” (or [Alice $\mapsto x$, Bob $\mapsto y$] for short). This balance can be updated (to some new balance [Alice $\mapsto x'$, 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 $\textit{close}$ 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 $P_i \leftrightarrow P_j$ (instead of $\{P_i, P_j\}$). Every $P_i \leftrightarrow P_j$ represents a channel between $P_i$ and $P_j$, and the cash function $\Gamma$ determines the number of coins available for the parties in every channel. More precisely, every $\Gamma(P_i \leftrightarrow P_j)$ is a function $f$ of a type $f : \{P_i, P_j\} \rightarrow \mathbb{Z}_{\geq 0}$. We will often write $\Gamma(P_i \leftrightarrow P_j)$ to denote this function. The value $\Gamma(P_i \leftrightarrow P_j)(P)$ denotes the amount of coins that $P$ has in her account in channel $P_i \leftrightarrow P_j$. A path (in $\mathcal{G}$) is a sequence $P_{i_1} \rightarrow \cdots \rightarrow P_{i_t}$ such that for every $j$ we have $P_{i_j} \leftrightarrow P_{i_{j+1}} \in \mathcal{E}$. 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 $\Gamma$, 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 $e = P_i \leftrightarrow P_j$ 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., [19]). Consider the following example: we have three parties: $P_1, P_2$, and $P_3$ and two channels: $P_1 \leftrightarrow P_2$ and $P_2 \leftrightarrow P_3$ between them. Now, suppose the sender $P_1$ wants to send $v\mathcal{C}$ to the receiver $P_3$ over the path $P_1 \rightarrow P_2 \rightarrow P_3$, with $P_2$ being an intermediary that routes these coins. This is done as follows. First, party $P_1$ asks $P_2$ to forward $v\mathcal{C}$ 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 $2\Delta$ (denote this receipt with $\rho$). If $P_2$ manages to do it by this deadline, she gets these coins in her account in the channel $P_1 \leftrightarrow P_2$. To guarantee this, $P_1$ initially blocks these coins in the channel $P_1 \leftrightarrow P_2$. These coins can be claimed back by $P_1$ if time $2\Delta$ 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 $\rho$ within $\Delta$ time) $v\mathcal{C}$ in the channel $P_3 \leftrightarrow P_1$. Now suppose that party $P_3$ claims her $v\mathcal{C}$ in channel $P_2 \leftrightarrow P_3$. This can only be done by providing a receipt $\rho$ confirming that she received these coins. We call this process acknowledging payment. Party $P_2$ can now claim her coins in channel $P_1 \leftrightarrow P_2$ by submitting an acknowledgment.
containing the receipt \( \rho \). In the above example, the number of coins that can be pushed via a channel \( P_i \leftrightarrow P_{i+1} \) 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 \( P_1 \rightarrow P_2 \rightarrow P_3 \) 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 \( \rho \) 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 \( P_1 \leftrightarrow P_2 \) and \( P_2 \leftrightarrow P_3 \) are chosen. An exciting feature of this protocol is that receipt \( \rho \) 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 \( \rho \) 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., [7, 16, 3].

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., [17, 18, 21, 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 \( v \subset P_3 \) to \( P_4 \), and gives \( P_2 \) time \( 2 \Delta \) 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 she 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 [18, 1, 20, 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 [22, 7]). 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, 16, 14, 7, 5, 6, 11, 15, 3] in this area. The reader can also consult SoK papers on off-chain techniques (e.g. [10]). Partial coin transfers were considered in [18], but with no aggregation techniques and ad-hoc splitting. Atomic payment splitting has been considered in [18, 1, 20, 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. 2.1, we provide an overview of NAPS definition, and in Sec. 2.2 we informally describe EthNA. Below, in Sec. 3.3, we provide the formal NAPS definition and the detailed description of EthNA. An overview of our implementation and simulations is presented in Sec. 3. Some of the related work was mentioned already before. Off-chain channels being transferred, e.g., in the credit card payments).

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

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 \( P = \{ P_1, \ldots, P_n \} \), 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 \( [m]_{P_i} \). 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 \( \mu \)”. 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 \( P_{i_1}, \ldots, P_{i_n} \) be the neighbors of \( P_1 \), i.e., parties with which \( P_1 \) has channels. Suppose the balance of each channel \( P_1 \leftrightarrow P_{j} \) is \([P_1 \mapsto x_i, P_{i_j} \mapsto y_j]\) (meaning that \( P_1 \) and \( P_{j} \) 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 \( \mu \)). 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 \( P_{i_j} \) 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 \( P_1 \leftrightarrow P_{i_j} \) 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 Sec. 5.2 we present an improved protocol where \( \delta \) 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 requirements, which are pretty standard and hold for most PCNs (including Lightning).

Below, let \( v \) denote the total amount of coins \( P_1 \) wants to transfer to \( P_n \) within some payment \( \mu \).

The first property is called fairness for the sender. To define it, note that as a result of payment \( \mu \) (with timeout \( t \)), the total amount of coins that each party \( P \) has in the channels
(a) The channel graph with the initial coin distribution.

(b) The sender $P_1$ wants to send $7\zeta$ to the receiver $P_6$. She splits these coins into two amounts: $6\zeta$ pushed to $P_2$ and $1\zeta$ pushed to $P_3$. This is indicated with labels (1) and (2), respectively. Then (3) party $P_3$ simply pushes $1\zeta$ further to $P_6$. Party $P_2$ splits $6\zeta$ into $3\zeta + 3\zeta$, and pushes $3\zeta$ 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 $1\zeta$ got unlocked (6) for $P_4$ in channel $P_4 \leftrightarrow P_6$, and hence (7) party $P_4$ pushes all $3\zeta$ to $P_6$. Party $P_5$ pushes only $2\zeta$ to $P_6$ (8). The channel balances correspond to the situation after the coins are pushed (except of channel $P_4 \leftrightarrow P_6$ where we also indicated the fact that $1\zeta$ 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 $\Delta$ before the time she has to acknowledge that subpayment back. This is because $P$ needs this “safety margin” of $\Delta$ in case $P'$ is malicious, and the acknowledgment has to be done “via the blockchain”.

(c) Party $P_6$ acknowledges subpayment of $1\zeta$ to $P_3$, which, in turn acknowledges it to $P_1$. Party $P_6$ also acknowledges subpayment of $3\zeta$ to $P_4$ and $2\zeta$ to $P_5$, who later acknowledge them to $P_2$. Once $P_2$ receives both acknowledgments, she “aggregates” them into a single acknowledgment (for $5\zeta$) and sends it to $P_1$. As a result $5\zeta + 1\zeta = 6\zeta$ 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 “$\overrightarrow{P_i \rightarrow P_j}$” denotes the fact that there exists a channel between $P_i$ and $P_j$, and the parties have $x$ and $y\zeta$ in it, respectively.
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 $P_1$ holds a receipt of a form $Receipt(\mu, v) := \text{"an amount } v \text{ of coins has been transferred from } P_1 \text{ to } P_n \text{ as a result of payment } \mu\text{"}$. 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 $Receipt(\mu, v)$ 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.

More precisely, consider a channel graph as below (for simplicity, we do not specify the coin amounts on the right-hand sides of the channels, as they are irrelevant to this example).

![Channel Graph](image-url)

Now suppose that $P_1$ and $P_2$ decide to send 2¢ each to $P_5$ via $P_3$ and $P_4$. If now $P_1$ pushes 1¢ to $P_3$ and at the same time $P_2$ pushes 1¢ to $P_4$, then none of the payments can be completed (since the channels $P_3 \leftarrow P_5$ and $P_5 \leftrightarrow P_5$ do not have sufficient capacity). On the other hand: if we allow non-atomic payments, then each payment will partially succeed (i.e. each sender will send 1¢ to the receiver $P_5$). They may then try to send the remaining amounts after some time when new capacity in these channels is available. This can be generalized to larger graphs and more complicated “deadlocks”.

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 Sec. 5.1. 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 $\mu$. Once $P_n$ receives some subpayment $v$ that reached it via some path $\pi = P_1 \rightarrow P_4 \rightarrow \cdots P_k \rightarrow P_n$ she issues a subreceipt for this payment and sends it to $P_k$. 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 $\mu$ 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 $(push, \pi, v,t)$ is sent from $P$ to $P'$, the party $P$ blocks $v$ coins in channel $P \rightarrow P'$ 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 $P_n$ first acknowledges the subpayment that came along the path $P_1 \rightarrow P_3 \rightarrow P_6$. This means that $P_6$ “accepts” that $1c$ will be transferred to her from $P_1$ via path $P_1 \rightarrow P_3 \rightarrow P_6$, or, in other words: $1c$ 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 $\alpha$:

$$
\alpha \rightarrow P_1 \rightarrow P_3 \rightarrow P_6
$$

(1)

To acknowledge the subpayment that was pushed along the path $P_1 \rightarrow P_3 \rightarrow P_6$ party $P_6$ signs $\alpha$ and sends it to $P_3$. Such signed information will be called a “subreceipt” and denoted $[\alpha]_{P_6}$. By providing this subreceipt, party $P_6$ also gets $1c$ in the $P_3 \rightarrow P_6$ (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 $P_1 \rightarrow P_2 \rightarrow P_4 \rightarrow P_6$, i.e., $P_6$ accepts that $3c$ will be transferred to her from $P_1$ via path $P_1 \rightarrow P_2 \rightarrow P_4 \rightarrow P_6$. The receiver $P_6$ now modifies the payment tree as follows:

$$
\beta \rightarrow P_1 \rightarrow P_2 \rightarrow P_4 \rightarrow P_6
$$

(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 $\beta$) corresponds to the “new” path, and the thin one is taken from Eq. (1), except that $P_1$ is labeled with “$4c$”. 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 (1c) and now (3c). Party $P_6$ now signs path $\beta$ to create a subreceipt that she sends to $P_4$ to claim $3c$ in the channel $P_1 \rightarrow P_6$.

Finally, $P_6$ acknowledges the subpayment along the path $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_6$. This is done similarly to what we did before. The resulting tree is now as follows:

$$
\gamma \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_6
$$

(3)

Note that we performed “summing” in two places on Eq. (3): at the node $P_1$ (where we computed $6c$ as $4c + 2c$) and an $P_2$ (where $5c = 2c + 3c$). Labeled path $\gamma$ is now signed by $P_6$ and sent to $P_3$ as subreceipt in order to claim $2c$.

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 $W$ be the set of such subreceipts (such sets will be called “payment reports”, see Sec. 3.2 for their formal definition). When a party $P$ wants to acknowledge the subpayment, she chooses (in a way that we explain below) one of the subreceipts $\zeta$ from her set $W$. 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 $v\zeta$ in the channel $P' \rightsquigarrow P$.

To determine the value of $v\zeta$ the following rule is used: it is defined as the label of $P$ on the path $\zeta$. Given this, the rule for choosing $\zeta \in W$ is pretty natural: $P$ simply chooses such the $\zeta$ that maximizes $v$. Such $\zeta$ will be called a “leader” of $W$ (at node $P$). See Sec. 3.2 for the formal definition of this notion. 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 $\alpha$). She simply forwards it to $P_1$ and receives $1\zeta$ in the channel $P_1 \rightsquigarrow P_3$. Note that this is exactly equal to the value that she “lost” in the channel $P_3 \rightsquigarrow P_6$, 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: $\beta$ (defined on Eq. (2)) and $\gamma$ (from Eq. (3)). By applying the rule described above, $P_2$ chooses the leader $\zeta$ at $P_2$ to be equal to $\gamma$ (since $5\zeta > 3\zeta$). This is depicted below (the shaded area indicates the labels that are compared).

\[
\beta = \begin{array}{c}
\alpha
\end{array}, \begin{array}{c}
\beta
\end{array}, \begin{array}{c}
\beta
\end{array}, \begin{array}{c}
\beta
\end{array}, \begin{array}{c}
\beta
\end{array}, \begin{array}{c}
\beta
\end{array}, \begin{array}{c}
\beta
\end{array}
\gamma = \begin{array}{c}
\alpha
\end{array}, \begin{array}{c}
\beta
\end{array}, \begin{array}{c}
\beta
\end{array}, \begin{array}{c}
\beta
\end{array}, \begin{array}{c}
\beta
\end{array}, \begin{array}{c}
\beta
\end{array}, \begin{array}{c}
\beta
\end{array}
\]

What remains is to argue about balance neutrality for $P_2$, i.e., that number of coins received by $P_2$ in the channel $P_1 \rightsquigarrow P_2$ 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) ($5\zeta$ are “gained”, and $2\zeta + 3\zeta$ 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 $P$. See Sec. 3.2 for the details.

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” $W$, i.e. a set of paths signed by $P_6$. In the case of our example, these paths are $\alpha$ (sent to $P_1$ by $P_3$) and $\gamma$ (sent by $P_2$). Party $P_1$ chooses her “receipt” similarly as the intermediaries choose which subreceipt to forward. More precisely, let $\zeta$ be the path that is the leader of $W$ at node $P_1$. This path becomes the final receipt. The amount of transferred coins equals the label of $P_1$ in $\zeta$. In our case, the leader $\zeta$ is clearly $\gamma$ (since its label at $P$ is “6\zeta”, while the label of $\gamma$ at $P$ is “1\zeta”, cf. Eqs (1) and (3)). Hence, $\gamma$ 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 $\zeta$ 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. For the details see Lemma 2.

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 $y$ (see Eq. (3)), the following subreceipt:

$$\gamma := \gamma_1 \gamma_2 \gamma_3 \gamma_4 \gamma_5 \gamma_6$$

(5)

The receiver does it to make $P_1$ hold a receipt for $5\epsilon$, while in fact receiving $6\epsilon$. Party $P_3$ has no way to discover this fraud attempt (since from her local perspective everything looks ok), so $2\epsilon$ get transferred to $P_5$ in the channel $P_5 \rightsquigarrow P_6$. Party $P_3$ forwards $\gamma$ to $P_2$ and gets $2\epsilon$ in the channel $P_2 \rightsquigarrow P_3$ (hence the “balance neutrality” property for her holds). Now look at this situation from the point of view of $P_2$. In addition to $\gamma$ she got one more subreceipt, namely $\beta$ (see, e.g., Eq. (4)). Party $P_2$ preforms a “consistency check” by combining $\gamma$ and $\beta$. 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 $\beta$ is $4\epsilon$, which is smaller than the label of $P_1$ in $\gamma$ (which is equal to $5\epsilon$). This means that $\beta$ had to be signed by $P_6$ before she signed $\gamma$. Hence $P_2$ first writes down $\beta$, and then on top of it she writes $\gamma$ (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:

$$\gamma := \gamma_1 \gamma_2 \gamma_3 \gamma_4 \gamma_5 \gamma_6$$

(6)

It is now obvious that $P_6$ is cheating since the labels on the children of $P_2$ sum up to $5\epsilon$, which is larger than $4\epsilon$ (the label of $P_2$). This “inconsistency” is marked as a shaded region on Eq. (6). Hence the set $\{\beta, \gamma\}$ 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 $\mu$ are routed via the same party independently. Consider, e.g., the following scenario: $2\epsilon$ is sent from $P_1$ to $P_4$ via a path $P_1 \rightsquigarrow P_2 \rightsquigarrow P_3 \rightsquigarrow P_4$. This amount is first split by $P_2$ as: $1\epsilon + 1\epsilon$, and each $1\epsilon$ coin is pushed to $P_3$, who, in turn, pushes each of them further to $P_4$. Obviously, both $1\epsilon$ coins traveled along $P_1 \rightsquigarrow P_2 \rightsquigarrow P_3 \rightsquigarrow P_4$, 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 $\{(P_1, \mu_1), (P_2, \mu_2), (P_3, \mu_3), (P_4, \mu_4)\}$ and $\{(P_1, \mu_1), (P_2, \mu_2), (P_3, \mu'_3), (P_4, \mu'_4)\}$ (where for both $i = 3, 4$ we have that $\mu_i$ and $\mu'_i$ are distinct). Moreover, we assume that $\mu_1$ (“contributed” by the sender $P_1$) is equal to the nonce that identifies the entire payment.

Formally, for a channel graph $\mathcal{G} = (P, E, \Gamma)$ a string $\pi = \{(P_{i_1}, \mu_1), \ldots, (P_{i_n}, \mu_n)\}$ is a path over $\mathcal{G}$ (for payment $\mu$) if each $\mu_i \in \mathcal{N}$ is a nonce, each $P_i \xrightarrow{\text{trans}} P_{i+1}$ is an edge in $\mathcal{G}$, and $P_1 = P_1$. We also assume that a path corresponding to a payment $\mu$ always starts with $(P_1, \mu)$. We say that $P$ appears on $\pi$ (at position $j$) if we have that $P = P_j$. We assume that every $P$ appears at most once on $\pi$, 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, $\mu$ is a nonce, $\pi$ is a path over $\mathcal{G}$, and $t$ is time. For reference, the notation used in this section is summarized in Fig. 11 in the appendix.

### 3.1 NAPS formal security definition

The protocol is parameterized with a security parameter $1^\kappa$ 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 $\mu$. If this machine outputs (i-sent, $\mu$, $w$) to $Z$ then we consider payment $\mu$ 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 $\text{Accounts}_G$. This functionality (depicted on Fig. 2) is used to model the amounts of coins that the parties have in the channels. It is initialized with $G$ and accepts messages $(\text{trans}, P_i, P_j, v)$ that are used to transfer $v$ coins from $P_i$ to $P_j$ in channel $P_i \xrightarrow{\text{trans}} P_j$.

**Fig. 2: The functionality $\text{Accounts}_G$**

The functionality $\text{Accounts}_G$ is initialized with a channel graph $\mathcal{G} = (P, E, \Gamma)$.

---

Upon receiving a message of a form $(\text{trans}, P_i, P_j, v)$ (with $v \leq \Gamma^{P_i \rightarrow P_j}(P_i)$) from an ideal functionality $\text{NAPS}_G$ or from a state channel machine $\Gamma^{P_i \rightarrow P_j} - \text{decrease } \Gamma^{P_i \rightarrow P_j}(P_i)$ by $v$ and increase $\Gamma^{P_i \rightarrow P_j}(P_j)$ by $v$.

---

We assume that if $P$ is corrupt then for every channel $P \xrightarrow{\text{trans}} P'$ and every $v \geq \Gamma^{P \rightarrow P'}(P)$ the simulator can at any moment decrease $\Gamma^{P \rightarrow P'}(P)$ by $v$ and increase some other $\Gamma^{P' \rightarrow P''}(P')$ by $v$. The state of $\Gamma$ is visible to $Z$. 

---
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., [6, 7]). Whenever we say that some operation (e.g. sending a message or simply staying in idle state) takes between $\tau$ and $\tau'$ rounds, we mean that it is up to the adversary to decide how long this operation takes (as long as it takes between $\tau$ and $\tau'$ rounds). The same convention applies to statements like “it takes at most/at least $\tau$ 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 $\Delta$ 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 \textsf{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 $(i\text{-}push,(\pi||\langle (P,\mu),(P',\mu')\rangle),v,t)$ this means that $m$ has to start with an “$i\text{-}push$” keyword, followed by a parameter denoting a path that ends with two elements (denoted $(P,\mu)$ and $(P',\mu')$ 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 $A$ who can corrupt some parties (when a party is corrupt $A$ 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 \textit{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 $\textsc{Accounts}_n$ 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. $A$ and $Z$ can communicate. At the end of its execution, $Z$ 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 $Z$. For a NAPS protocol with arity $\delta$ executed over graph $\mathcal{G}$ the corresponding ideal functionality is denoted $\textsc{NAPS}_\delta^\mathcal{G}$ and presented on Fig. 3.

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 (for reference, these messages and their syntax are summarized on a cheat sheet on Fig. 9, in the appendix, see p. 29). 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 $\mu$ parties $P_1$ and $P_n$ send respectively a message $i\text{-}send(\mu,v,t)$ to $P_1$ and $i\text{-}receive(\mu,v,t)$ 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 $i\text{-}acknowledged(((P,\mu)),s)$ from the ideal functionality. The
Fig. 3: The ideal functionality $\text{NAPS}^4_G$

The ideal functionality $\text{NAPS}^4_G$ is parametrized by a channel graph $G = (P, E, \Gamma)$ and an arity parameter $\delta$. It maintains a cash function $\Gamma$ initially equal to $\Gamma$ and a set $\Psi$ initially equal to $\emptyset$. Function $\hat{\Gamma}$ is used to denote the current amount of coins available in the channels and set $\Psi$ containing all open push requests. Moreover, the ideal functionality maintains a function $\text{remaining} : \Psi \rightarrow \mathbb{Z}_{\geq 0}$. It proceeds as follows.

Upon receiving a message of a form $(\text{i-send}, \mu, u, t)$ from $P_1$ and $(\text{i-receive}, \mu, u, t)$ 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 $\mu$ before and (b) the current time is greater than $t - \Delta$. |

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

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

| Correctness condition: | (a) you have not received an “i-pushed” message with this $\pi$ before (for any $P', \mu', v'$ and $t$) is less than $\delta$, (b) the current time is greater than $t - \Delta$, and (f) if $P$ is honest then $(\pi||((P, \mu))) \in \Psi$ and $\text{remaining}(\pi||((P, \mu))) \geq v$. |

If it does not hold, then ignore this message. Otherwise (a) add $(\pi||((P, \mu), (P', \mu')), v, t)$ to $\Psi$, (b) decrement $\text{remaining}(\pi||((P, \mu)), v, t)$ by $v$, (c) let $\text{remaining}(\pi||((P, \mu), (P', \mu')), v, t) := v$, (d) decrement $\hat{\Gamma}^{P \rightarrow P'}(P)$ by $v$, and (e) in the next round send a message $(\text{i-pushed}, (\pi||((P, \mu), (P', \mu')), v, t))$ to $P'$.

If time $t + \Delta$ comes and $(\pi||((P, \mu), (P', \mu')), v, t))$ is still in $\Psi$ then behave as if you received a message $(\text{i-acknowledge}, (\pi||((P, \mu), (P', \mu'))))$ from $P'$ (see below).

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

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

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 i-acknowledged$(\pi||((P, \mu)||((P', \mu')))), v)$ (for any $(P', \mu'))$) 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 $\pi$ is empty) — then in the next round send i-sent$((\pi||((P, \mu)), s)$ to $\text{RVM}$.  
- $P \in \{P_2, \ldots, P_{n-1}\}$ — then let $(P_\ell, \mu_\ell)$ be the last element of $\pi$ and then within time $\Delta$ (a) send a message $(\text{trans}, P_\ell, P, s)$ to $\text{Accounts}_G$, (b) increment $\hat{\Gamma}^{P_\ell \rightarrow P}(P)$ by $v - s$, (c) increment $\hat{\Gamma}^{P_\ell \rightarrow P}(P)$ by $s$, (d) increment $\text{remaining}(\pi, v, t)$ by $v - s$, (e) remove $(\pi||((P, \mu)))$ from $\Psi$, and (f) send i-acknowledged$(\pi||((P, \mu)), s)$ to $P_\ell$.
- $P = P_n$ — then let $(P_\ell, \mu_\ell)$ be the last element of $\pi$ and then within time $\Delta$ (a) send a message $(\text{trans}, P_\ell, P, v)$ to $\text{Accounts}_G$, (b) increment $\hat{\Gamma}^{P_\ell \rightarrow P}(P)$ by $v$, (c) remove $(\pi||((P, \mu)))$ from $\Psi$, and (d) send i-acknowledged$(\pi||((P, \mu)), s)$ to $P_\ell$.  

functionality $\text{NAPS}_G^δ$ maintains a set $Ψ$ that contains all the push requests that have not yet been acknowledged. By push requests we mean tuples $(\pi, v, t)$ such that some party sent (i-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 $\pi$, 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 $(\pi||((P, μ), v, t))$. To push it to a party $P'$ party $P$ sends a message (i-push, $(\pi||((P, μ), (P', μ'))), v', t'$) to $\text{NAPS}_G^δ$. 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 i-acknowledged$(((P, μ), (P', μ')), v''$, where $v''$ specifies the amount of coins that were transferred. If there are no open push requests of a form $(\pi||((P, μ), (P', μ')), v, t)$ then a party $P$ can decide to close a given push request by sending a message i-acknowledged$(\pi||((P, μ), (P', μ')), v, t)$ to $\text{NAPS}_G^δ$. The function remaining and the accounts in the $P \rightsquigarrow P'$ channels are updated accordingly (by sending messages to the Accounts$_G$ 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 $(((P, μ), v, t)$ then she sends an acknowledge message to $\text{NAPS}_G^δ$. The “ideal model” adversary will also be called the simulator and denoted $S$. We assume that $S$ has access to the ideal functionality. The output of the ideal execution of $\text{NAPS}_G^δ$ against $S$ and $Z$ with security parameter $1^κ$ is a random variable Ideal$(\text{NAPS}_G^δ, S, Z, 1^κ)$ denoting the output of $Z$.

It is easy to see that the informal properties from Sec. 2.1.1 are implied by this ideal functionality. To see why, look at the “Upon receiving a message of a form (i-acknowledge)...” part of Fig. 3. Recall that $s$ denotes the sum of all the coins a given party “lost” in the channels. From the construction of the ideal functionality, $P_1$ sends to RVM a receipt for exactly $s$ coins (hence the “fairness for the sender” holds). Moreover, every intermediary $P_j$ gets back exactly $s$ coins in the channel that she has with $P_k$ (this implies “balance neutrality for the intermediaries”). Finally, to see why “fairness for the receiver” holds, observe, by looking recursively at the flow of the financial transfers, that $P_1$ will never get a receipt for a value higher than the sum of the amounts of coins that the receiver “gained” in her channels.

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 $A$) get the public keys of the other parties. For each pair $\{P_i, P_j\}$ such that $P_i \rightsquigarrow P_j \in \mathcal{E}$ the parties $P_i$ and $P_j$ also have access to an uncorruptible state channel machine $C^{P_i \rightsquigarrow P_j}$, which in turn, has access to Accounts$_G$ (the parties do not have a direct access to Accounts$_G$). Sending messages to Accounts$_G$ 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 $G$ with arity $δ$ is a tuple of machines $\Pi_G^δ := \{\text{RVM}, P_1, \ldots, P_n, \{C^{P_i \rightsquigarrow P_j}\}_{P_i \rightsquigarrow P_j \in \mathcal{E}}\} \in \mathcal{E}$. The output of the real execution of $\Pi$ with security parameter $1^κ$ is a random variable $\text{Real}(\Pi_G^δ, A, Z, 1^κ)$ denoting the output of $Z$. 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 $\Pi_G^δ$ is a secure Non-Atomic Payment Splitting (NAPS) protocol for $G$ and $δ$ if for every adversary $A$ there exists a simulator $S$ such that and every $Z$
the families of random variables \( \{ \text{Ideal}(\text{NAPS}_Q^A, S, Z, 1^n) \}_\kappa \) and \( \{ \text{Real}(\Pi_Q^A, A, Z, 1^n) \}_\kappa \) 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 \( \mu \), a subreceipt (over \( \mathcal{G} \), for payment \( \mu \)) is a pair \( \{ \pi, \lambda \}_P \) signed by \( P_n \) such that \( \pi \) is a path over \( \mathcal{G} \) (for payment \( \mu \)) with \( P_n \) appearing on the last position of \( \pi \), and \( \lambda \) is a non-increasing sequence of positive integers, such that \( |\lambda| = |\pi| \). A payment report for \( \mu \) is a set \( \mathcal{W} \) of subreceipts for \( \mu \) such that \( \pi \) identifies a member of \( \mathcal{W} \) uniquely, \( i.e.\): \( \{ \{ \pi, \lambda \}_P \in \mathcal{W} \text{ and } \{ \pi, \lambda' \}_P \in \mathcal{W} \} \implies \lambda = \lambda' \). For example, \( \alpha, \beta, \text{ and } \gamma \) 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 \( \{ \{ \pi, \lambda \}_P \) is a leader of \( \mathcal{W} \) at node \( P \) if \( P \) appears on \( \pi \) at some position \( i \), and for every \( \{ \pi', \lambda' \}_P \in \mathcal{W} \) 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 \( \gamma \). 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 \( \{ (\pi)[\sigma] \}, \lambda' \}_P \) 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 \( \mu \)) is a payment report \( \mathcal{Q} \) for \( \mu \) of a form \( \mathcal{Q} = \{ \{ (\sigma)[\pi_i], \lambda_i \} \}_{i=1}^m \), where all the \( \pi_i[1]'s \) are pairwise distinct, such that the following condition holds: \( \max_{i=1,...,m} \lambda_i[|\sigma|] < \sum_{i=1}^m \lambda_i[|\sigma|] + 1 \). For an example of a fraud-proof (with nonce missing from the picture) see Eq. (6). If EthNA has arity at most \( \delta \) (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 \( \sigma \) is smaller than the sum of all labels of \( \sigma \)'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 \( \delta \) is EthNA’s arity, \( \ell \) is the maximal length of paths, and \( \kappa \) 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 \( \kappa \) 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 \( \lambda \)'s. We describe the compression ideas in Sec. 5.2.

The formal description of EthNA appears on Figs. 4, 5, and 6. It uses a subroutine algorithm \( \text{Add}_5 \) that we describe in a moment. We outsource some of the protocol to a procedure \( \text{handle-path} \) (depicted in the same figure) to avoid repeating the same instructions. The receipt verification machine \( \text{RVM} \) is presented in Fig. 7.

The parties receive the “ideal model” messages (starting with a prefix “i-“) from \( Z \). By saying that a message (received from \( Z \)) is admissible, we mean that it satisfies the “correctness conditions” from Fig. 3. 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 on Fig. 10 on p. 30 in the appendix). 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 \( P_n \).
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). The procedure for $P_1$ is similar, except that $P_1$ is activated by a “send” message from $Z$, and waits of the invoice from $P_n$. It then communicates with the receipt verification machine defined on Fig. 7 (see p. 20 in the appendix). Probably the most interesting part is the instructions for $P_n$. First, $P_n$ (upon receiving an i-receive($\mu, u, t$) message from $Z$) sends an invoice to $P_1$. For every payment, $\mu$ party $P_n$ maintains a payment tree $\Phi^\mu$ that is initially empty. Payment trees were already discussed in Sec. 2.2. For a formal definition, consider some fixed $\mu$ and $\mathcal{G}$. During the execution of EthNA for $\mathcal{G}$ and $\mu$, 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 $v^i \in \mathbb{Z}_{\geq 0}$ be the number of coins transmitted with each $\pi^i$. Let $\mathcal{W} := \{v^i, v^i\}_{i=1}^t$. Formally, a payment tree tree($\mathcal{W}$) is a labeled tree $(T, \mathcal{L})$, where $T$ is the set of all prefixes of the $\pi^i$s, i.e., $T := \bigcup_i \text{prefix}(\pi^i)$. If EthNA has arity
Fig. 5: Path handling procedure handle-path($P, \pi, v, t$)

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

Message (i-push, $\pi || ((P', \mu'), (P', \mu'))$, $v', t'$) (for some $v' \leq \alpha^\pi$ and $\mu$ and $\mu'$ and $P'$ such that $P \rightarrow P' \in E$) — handle each such a message as follows. If $\omega^\pi = 0$ then ignore this message. Otherwise let $\alpha^\pi := \alpha^\pi - v'$ and decrease $\omega^\pi$ by 1. Then send a message (push, $((\pi || ((P', \mu'))), v', t'$) to $P'$ and wait until round $t$ to receive a message od one of the following forms:

- (acknowledged, $\pi || ((P', \mu'))$, empty) from $C^{P\rightarrow P'}$ — then let $\alpha^\pi := \alpha^\pi + v'$ and send a message (i-acknowledged, $\pi || ((P', \mu'))$, 0) to $Z$,
- (acknowledged, $\pi || ((P', \mu'))$, $\psi, \lambda, p$), where $\psi$ is such that $\pi || ((P', \mu'))$ is a prefix of $\psi$ — then store $\psi, \lambda, p$ in $\mathcal{W}$ by letting $\mathcal{W} := \mathcal{W} \cup \{\psi, \lambda, p\}$.

Let $\bar{\nu} := \lambda[\pi] + 1$. Let $\alpha^\pi := \alpha^\pi + v' - \bar{\nu}$ and send (i-acknowledged, $\pi || ((P', \mu'))$, $\bar{\nu}$) to $\mathcal{Z}$, or
- (acknowledged, $\pi || ((P', \mu'))$, (fraud-proof, $w$)) — then store (fraud-proof, $w$) and send a message (i-acknowledged, $\pi || ((P', \mu'))$, $v'$) to $\mathcal{Z}$.

Message (i-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 (fraud-proof, $w$) (for some $P', \mu'$) or if $\mathcal{W}$ is inconsistent and $w$ is the fraud-proof — then output ((fraud-proof, $w$), $v$).
- Otherwise: if $\mathcal{W}$ is empty then output empty.
- Otherwise let $\psi, \lambda, p$ be the leader of $\mathcal{W}$ at $P$, where $P$ is the last party on $\pi$. Output $\langle \psi, \lambda, p, \lambda[\pi] \rangle$.

Fig. 6: The EthNA state channel machine $C^{P_k \rightarrow P}$

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

Upon receiving a message of a form (acknowledged, $\pi, v, t, p_k$, empty) from a party $P$ (such that $\{P_k, P\} = \{P, P\}$) — send (acknowledged, $\pi, v, t, p_k$, empty) to $P_k$.

Upon receiving a message of a form (acknowledged, $\pi, v, t, p_k$, $\psi, \lambda, p$) from a party $P$ where (a) current time is at most $t$, (b) $\pi$ is a path with a suffix $((P_k, \mu), (P, \mu'))$ (for some $\mu$ and $\mu'$), (c) $\pi$ is a prefix of $\psi$, (d) $\lambda[\pi] \leq \Gamma^{P_k \rightarrow P}$ ($P_k$), and (e) $\{P_k, P\} = \{P, P\}$ — then send a message of a form (trans, $P_k, P, \lambda[\pi]$) to Accounts and a message (acknowledged, $\pi, \psi, \lambda, p$) to $P_k$.

Upon receiving a message of a form (acknowledged, $\pi, v, t, p_k$, (fraud-proof, $w$)) from a party $P$ where (a) current time is at most $t$, (b) $\pi$ is a path with a suffix $((P_k, \mu), (P, \mu'))$ (for some $\mu, \mu'$ and $P_k$), (c) $v \leq \Gamma^{P_k \rightarrow P}$ ($P_k$), and (d) $w$ is a fraud-proof — then send a message (trans, $P_k, P, v$) to Accounts and send a message (acknowledged, $\pi$, (fraud-proof, $w$)) to $P_k$. 


δ then the arity of T in every node \((\pi||((P,\mu)))\) is at most δ. Then for every \(\pi \in T\) we let \(L(\pi) := \sum_{v \in \text{prefix}(\pi)} v^\epsilon\). In other words: every path \(\pi\) gets labeled by the arithmetic sum of the value of the payments that were “passed through it”. Clearly, the label \(L(\epsilon)\) of the root node of \(\text{tree}(W)\) is equal to the sum of all \(v^\epsilon\)s, and hence it is equal to the total number of coins transferred by the subpayments in \(W\). We also have that for every path \(\sigma\) \(L(\sigma) = \sum_{\pi \in \text{path}(\sigma)} L(\pi)\). It is also easy to see that \(\text{tree}(W)\) can be constructed “dynamically” by processing elements of \(W\) one after another. More precisely, this is done as follows. We start with an empty tree \(\Phi\), and then iteratively apply the algorithm \(\text{Add}_\delta\) (see Alg. 1) for \((\pi^1, v^1), (\pi^2, v^2), \ldots\). From the construction of the algorithm, it follows immediately that if \(P_n\) starts with \(\Phi\) being an empty tree, and then iteratively applies \(\text{Add}_\delta\) to \((\pi^i, v^i)\)’s for \(i = 1, \ldots, t\), then the final state of \(\Phi\) is equal to \(\text{tree}(W)\). 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 \(\text{Add}_\delta\) algorithm correctly, then the resulting sets \(\mathcal{W}\) are never inconsistent (and hence no fraud-proof will ever be produced against an honest \(P_n\)). Formally, this is proven in Fact 1.

The formal security analysis of this protocol is given in the proof of the following lemma.

\begin{transaction}
\textbf{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 \(\delta\).
\end{transaction}

\begin{proof}
To prove that EthNA is a secure NAPS scheme, fix a channel graph \(\mathcal{G} = (\{P_1, \ldots, P_n\}, \mathcal{E}, \Gamma)\) and an adversary \(A\). We need to construct a simulator \(S\) such that the real and ideal executions are indistinguishable for every environment \(Z\).

\end{proof}

\begin{construction}
The simulator gets as input the security parameter 1\(^x\) and starts simulating the adversary \(A\) with this input. She also generates (public key, secret key) pairs for all \(P_i\)’s and sends the public keys to the adversary. The simulator \(S\) checks which
parties are corrupt by \( A \) and corrupt the same parties in the ideal model. She also passes to \( A \) all the secret keys of the corrupt parties. Recall that the honest parties are executed according to the ETHNA protocol. The simulator maintains a “simulated copy” of each \( P_i \), denoted \( \hat{P}_i \). She passes \( \hat{P}_i \) to \( A \) and maintains a “simulated copy” of each state channel machine \( C^{\hat{P}_i \rightarrow P_j} \) (denoted \( \hat{C}^{\hat{P}_i \rightarrow P_j} \)) and a simulated copy of the receipt verification machine (denoted \( \overline{RVM} \)). Then the simulator performs the execution of simulated \( A \) against the honest parties. This simulation proceeds in rounds.

Whenever the adversary \( A \) sends (in the name of some corrupt \( P_i \)) a message \( m \) to an honest \( P_j \), the simulator forwards this message to \( \hat{P}_j \). If \( \hat{P}_j \) ignores this message, then the simulator does nothing. Otherwise, consider the following cases:

- \( m = (\text{invoice}, [u, \mu, t]_{P_i}) \) (this happens only if \( P_i = P_n \) and \( P_j = P_1 \)) and in the same round \( P_i \) sent a message \( \text{i-send}(\mu, u, t) \) to the ideal functionality — then send \( i \)-receive(\( \mu, u, t \)) to the ideal functionality (in the name of \( P_n \)).
- \( m = (\text{push}, [\pi||((P_i, \mu), (P_j, \mu'))]_{P_j}, τ, t'_P \) — then send a message \( (\text{i-push}, (\pi||((P_i, \mu), (P_j, \mu'))), v, t) \) to the ideal functionality (in the name of \( P_1 \)).

Now consider the state channel machine \( C^{\hat{P}_i \rightarrow P_j} \) (recall that we assumed that it is incorruptible) and suppose the adversary \( A \) sends to it (in the name of a corrupt \( P_j \)) a message \( \text{acknowledge}, [\pi, v, t'_P, R] \). Then forward this message to \( C^{\hat{P}_i \rightarrow P_j} \) and observe its reaction. It is easy to see that the only interesting case is when the other user of this channel (\( P_j \)) is honest. Recall that the execution of \( C^{\hat{P}_i \rightarrow P_j} \) can result both in the change the \text{Accounts} functionality (via the trans messages) and in sending \text{acknowledged} messages to \( P_i \). Handle this as follows. The \text{trans} messages are simply forwarded to \( C^{\hat{P}_i \rightarrow P_j} \). For each \( \text{acknowledged}, [\pi, v, t'_P, R] \) message first send a message \( (\text{i-acknowledge}, \pi) \) to the ideal functionality (in the name of \( P_j \)), and then consider the following cases: (a) if \( R = \text{empty} \) then send a message \( \text{acknowledged}, \pi, 0 \) to \( P_j \), (b) if \( R = \text{acknowledged, } \pi, [\psi, \lambda]_{P_n} \) then send \( \text{acknowledged, } \pi, \lambda[1] \) to \( P_j \), and (c) if \( R = \text{acknowledged, } \pi, ([\text{fraud-proof}, w]) \) then send a message \( \text{acknowledged, } \pi, v \) to \( P_j \) (where \( v \) is taken from the signed tuple \( [\pi, v, t'_P] \) in the code of \( C^{\hat{P}_i \rightarrow P_j} \)). Recall that all messages sent by \( C^{\hat{P}_i \rightarrow P_j} \) can be delayed by some time \( \Delta' \leq \Delta \). The simulator delays the messages she sends by the same time. Finally, consider the \( RVM \) machine (again, we assumed it is incorruptible). In this case, the simulator simply forwards to \( \overline{RVM} \) every message \( \text{acknowledge}, [\pi, v, t'_P, R] \) that it receives from the adversary \( A \) (in the name of some corrupt party).

**Analysis of the simulator – the honest \( P_n \) case.** We now proceed to the analysis of the simulator \( S \) constructed above. We start with the case when \( P_n \) is honest. We first show a fact that essentially states that if \( P_n \) is honest, no fraud-proof will ever be produced.

> **Fact 1.** Suppose a party \( P_n \) executes \( \text{Add}_\phi \) multiple times (for some payment \( \mu \), starting from \( \Phi = \emptyset \)) and signs every output. Let \( W \) be the set of subreceipts signed by party \( P_n \) during the execution of the \( \text{Add}_\phi \) algorithm. Then \( W \) is consistent.

**Proof.** Take an arbitrary path \( \sigma \) and an arbitrary set \( Q \subseteq W \) that has a form \( Q = \{ [\sigma||p_i], \lambda_i]_{P_n} \}_{i=1}^m \). Without loss of generality, assume paths in \( Q \) are sorted according to the time the paths in this set were signed (starting from the first). From the fact that in the \( \text{Add} \) algorithm, the values in the labels can only increase, we get that \( \max_{i=1,...,m} \lambda_i||\sigma|| = \lambda_m||\sigma|| \). From the fact that \( L(\sigma) = \sum p_i \) is a child of \( \sigma \) \( L(\pi) \) (see Sec. 3.2) we know that the time when path \( [\sigma||p_m], \lambda_m]_{P_n} \) was signed all the children on \( \sigma \) in the tree \( T \) were labeled by values that sum up to \( \lambda_m||\sigma|| \). The sum \( \sum_{i=1}^m \lambda_i||\sigma|| + 1 \) is at most equal to this value. This is because (a) it is a subset of the set of all children of \( \sigma \), and (b) these paths were signed earlier than when
22 Non-Atomic Payment Splitting in Channel Networks

\[(\sigma | \pi_m), \lambda_m | P_n \] 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 \(\max_{1 \leq i \leq m} \lambda_i[|\sigma|] \geq \sum_{\ell=1}^m \lambda_i[|\sigma| + 1]\), and hence \(Q\) cannot be a fraud-proof (see Sec. 3.2 for the definition of fraud-Proofs). Therefore \(W\) does not have fraud-Proofs, so it is consistent.

Hence, no valid \((\text{acknowledge}, [\pi, v, t] | P_n, \text{fraud-proof}, w))\) message will be very sent to any state channel machine. The only things that a corrupt party \(P\) can do are: (a) send a message \((\text{acknowledge}, [\pi, v, t], \text{empty})\) to the state channel machine while in fact your set \(W^a\) was not empty, or (b) send a message \((\text{acknowledge}, [\pi, v, t], [\psi, \lambda] | P_n)\), where \([\psi, \lambda] | P_n\) is chosen in some other way than described in the protocol. It is easy to see that in both cases, \(P\) acts “against her own financial interest”. First, in the case of (a), party \(P\) claims 0 coins in the corresponding channel, while she might have lost some coins (corresponding to the same payment \(\mu\)) in other channels. Second, in case of (b) the only thing that \(P\) can do is to send some other \([\psi', \lambda'] | P_n \in W^a\). However, since we assumed that \(P_n\) is honest and that an honest \(P\) chooses \([\psi, \lambda] | P_n\) that maximizes her gain, this can only lead to losing coins by \(P\). Observe that this may lead to the situation in which the honest parties gain more coins in the real model than in the ideal model, meaning that \(Z\) can distinguish between both models. To remedy this, we use the feature of the Accounts\(_G\) functionality that the simulator (in the ideal model) can always transfer some coins from a corrupt party to an honest one.

Thanks to this, we can “correct” balances in Accounts\(_G\) (in the ideal model) so that they are the same in both models.

Analysis of the simulator – the corrupt \(P_n\) case. Now consider the case when \(P_n\) is corrupt. Let \(\Phi^\mu\) be the set of all signed tuples \([\psi, \lambda] | P_n\) that \(P_n\) ever sent to other parties (for payment \(\mu\)). Clearly, the only interesting case is when \(\Phi^\mu\) is inconsistent (otherwise, we can use the same reasoning as in the “honest \(P_n\) case”). Suppose some honest intermediary \(P\) finds inconsistency proof. Then she can claim the full amount \(v\) of coins that she was supposed to push. Since she never pushes further a total amount higher than \(v\) she can only gain coins from this. Again, we handle this in the ideal model by transferring coins from corrupt \(P_n\) to \(P\). If \(P\) receives an inconsistency proof due to one of her push requests, then by a similar argument, \(P\) can only “gain” coins (which we can handle by transferring coins from corrupt \(P_n\) to \(P\)). It is also easy to see that honest \(P_1\) cannot lose coins if she finds or receives fraud-proof for similar reasons. The complete security proof will be provided in the full version of this paper.

\subsection{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 \(\ell\) is the depth of the payment tree). During the acknowledgment, every malicious party can delay the process by time at most \(\Delta\). 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 Sec. 4. 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 $\text{acknowledge}(\phi, \lambda_{\mathcal{P}_n})$, which has size linear $O(\ell + \kappa)$ (where $\ell$ is as above, and $\kappa$ 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 $\delta$ is EthNA’s arity, $\ell$ is the maximal length of paths, and $\kappa$ 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 $\mathcal{P}_n$ 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 Sec. 5.2).

4 Practical aspects

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>path length</th>
<th>constructor</th>
<th>close</th>
<th>addState</th>
<th>addCheatingProof</th>
<th>addCompletedTransaction</th>
<th>closeDisagreement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>2,391</td>
<td>14</td>
<td>93</td>
<td>1,144</td>
<td>150</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2,249</td>
<td>14</td>
<td>94</td>
<td>871</td>
<td>145</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2,088</td>
<td>14</td>
<td>93</td>
<td>779</td>
<td>145</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2,191</td>
<td>14</td>
<td>93</td>
<td>590</td>
<td>140</td>
<td>14</td>
</tr>
</tbody>
</table>

**Figure 8** Experimental results.

Let us now 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. 8 (a) summarizes the execution costs in terms of thousands of gas, and depending on the arity $\delta$ 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.

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 \( N(200, 50) \). Each transaction was split by applying the following rules. The sender and the intermediaries look at the channel graph and search for the set \( X \) 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. 8. 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 Extensions

In this section, we show some extensions of EthNA. The formal proof that such “extended EthNAs” satisfy the NAPS definition will be presented in the full version of this paper.

5.1 Obtaining atomicity and partial atomicity in EthNA

EthNA can be easily converted into a payment system for atomic payments in the following way. Consider some payment \( \mu \) for \( v \). We simply let any subreceipt for a subpayment count as the receipt for the entire payment \( \mu \), and at the same time, we instruct the receiver \( P_n \) to start acknowledging payments, i.e., signing such receipts only if she receives all the subpayments (for the full amount \( v \)). This works since (a) as long as \( P_n \) did not receive the full amount, there is no receipt that she received any coins, and (b) once she does it, it is in her own best interest to acknowledge all subpayments (and claim all coins). This can be naturally generalized further to obtain “partial atomicity” where, e.g., the receiver can either receive \( 0 \cdot v, v/2 \cdot v \), or the full amount of \( v \). This way of obtaining atomicity may be used in the applications like the one described very recently in [8], where, in Sec. 3.1, we describe a way to obtain “unlinkability” in atomic payment splitting. The main idea here is to hide that a given payment has already been split. The “atomic EthNA” satisfies this property while avoiding using homomorphic hash functions (used in [8]). We leave a full comparison of these two approaches as a direction for future work.

5.2 Reducing the size of the fraud-proofs

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

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

(7)

Hence, in the most straightforward implementation, it is of length \( \Omega(\delta \cdot (\ell + \kappa)) \), where \( \delta \) is EthNA’s arity, \( \ell \) is the maximal length of paths, and \( \kappa \) is the security parameter.

We now show how to reduce this to \( O(\delta \cdot \kappa) \). We do it by designing an algorithm that signs the subreceipts \( \phi, \lambda \) differently. Let \( H \) be a collision-resistant hash function, and let \((KGen, Sig, Vf)\) be a signature scheme. Suppose \((sk, pk) \leftarrow KGen(1^\kappa)\) is the key pair of \( P_n \). To sign \((\phi, \lambda)\) we define a new signature scheme \((KGen, Sig, Vf)\) (i.e. we later let
\[
[\phi, \lambda]_{\text{P}_n} := (\langle \phi, \lambda, \sigma \rangle), \text{ where } \sigma := \text{Sig}_{\text{sk}}^\prime((\phi, \lambda))). \text{ Let } \text{KGen}' := \text{KGen}. \text{ To define } \text{Sig}((\phi, \lambda)) \text{ first define } \langle h^1, \ldots, h^{||\phi||} \rangle \text{ recursively as:}
\]
\[
h^1 := H(\phi[1]),
\]
and for \( j := 2, \ldots, ||\phi|| \):
\[
h^j := H(\phi[j], h^{j-1}).
\]
Then let \( \text{Sig}((\phi, \lambda)) := \langle \sigma^1, \ldots, \sigma^{||\phi||} \rangle \), where for each \( j \) we have:
\[
\sigma^j := \text{Sig}_{\text{sk}}(h^j, \lambda[j])
\]
Verification of this signature is straightforward. It is also easy to see that if \((\text{KGen}, \text{Sig}, \text{Vf})\) is existentially unforgeable under chosen message attack, then so is \((\text{KGen}', \text{Sig}', \text{Vf}')\), assuming the signed messages are of a form \((\phi, \lambda)\), where \(\phi\) is the path\(^1\). For a message \(M\) let \(\{M\}_{\text{P}_n}\) denote \(M\) signed with \((\text{KGen}', \text{Sig}', \text{Vf}')\). It is easy to see that now a fraud-proof from Eq. (7) can be compressed to a sequence
\[
\left\{ \left\{ \left\{ h_i^{||\sigma||}, \lambda_i[[\sigma]] \right\}_{\text{P}_n}, \pi_i[1], \right\}_{i=1}^m \right\}_{i=1}^m.
\]
such that Eq. (7) holds (above “\(\pi_i[1]\)” is needed to check correctness of \(h_i^{||\sigma||+1}\)). Since all signed values are of size linear in the security parameter, and \(m \leq \delta\) we get that Eq. (8) is \(O(\delta \cdot \kappa)\). 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 \(O(\kappa \cdot \log \delta)\).

**Further proof size reduction using “bisection”**. Finally, let us remark that the proof Eq. (8) can be further compressed by allowing interaction between the party that discovered cheating (denote it \(P\)) and \(\text{P}_n\). This is similar to the bisection technique \([22]\). Suppose \(P\) realizes that Eq. 7 does not hold. She can then divide the set of paths in \(\mathcal{Q}\) into two halves. 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 \(\text{P}_n\) (on the blockchain) to provide her own calculations of the above sums\(^2\). Let \(A'\) and \(B'\) be \(\text{P}_n\) 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.

\(^1\) This assumption is needed since paths have a clearly marked “ending”, namely they have to finish with \((\text{P}_n, \mu_n)\), for some \(\mu_n\). Otherwise it would be possible to attack this scheme by taking a prefix of a signed message and a prefix of its signature.
\(^2\) Since elements of \(\mathcal{Q}\) can be sorted, such a challenge is short.
26 Non-Atomic Payment Splitting in Channel Networks

It is easy to see that in the logarithmic number of 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 $P_n$).

6 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 possibly using techniques such as “scriptless scripts” (see, e.g., [23]), see also [2], or those of [12]). Another important question is to examine if one can reduce the pessimistic payment acknowledgment time from linear to constant analogously to the “Sprites” method [16].

References


Non-Atomic Payment Splitting in Channel Networks


Types of variables:
- $v, u$ — positive integers denoting amounts of coins,
- $\mu$ — a nonce,
- $\pi$ — path over $G$, and
- $t$ — time.

Messages
The parties send to the $\text{NAPS}_G^\delta$ functionality messages of the following form:
- (i-send, $\mu, u, t$) (such messages are sent only to $P_1$),
- (i-receive, $\mu, u, t$) (such messages are sent only to $P_n$),
- (i-push, $\pi, v, t$), and
- (i-acknowledge, $\pi$).

The $\text{NAPS}_G^\delta$ functionality sends to the parties messages of the following forms:
- (i-sent, $\mu$, receipt) (such messages are sent only by $P_1$, in case of EthNA receipt has a form $(u, \mu, t^*_{P_n}, R)$) (see Fig. 10),
- (i-pushed, $\pi, v, t$), and
- (i-acknowledged, $\pi, v$).

Figure 9 Messages and variables exchanged in the ideal execution $\text{NAPS}_G^\delta$ (assuming the payment path is $P_1 \rightarrow \cdots \rightarrow P_n$).
Types of variables

- \( v, u \) — positive integers denoting amounts of coins,
- \( \mu \) — a nonce,
- \( \pi, \psi \) — path over \( \mathcal{G} \), and
- \( t \) — time.

The variable \( R \) has one of the following forms:

- \( R = \text{empty} \) (where “empty” is a keyword),
- \( R = (\psi, \lambda)_{\pi} \) (where \( (\psi, \lambda) \) is a subreceipt), or
- \( R = (\text{fraud-proof}, w) \), where \( w \) is a fraud-proof.

Messages exchanged between the parties

Each party \( P \) sends to other parties messages of a form \( (\text{push}, \pi, v, t + P) \).

Party \( P_1 \) sends to \( \text{RVM} \) a message of a form \( (\text{acknowledged}, \mu, [(u, \mu, t + P)]) \).

Messages exchanged between the parties and the state channel machines

The parties send to the state channel machines messages of a form \( (\text{acknowledged}, (\psi, \lambda, t + P), R) \).

The state channel machines send the parties messages of a form \( (\text{acknowledged}, \pi, R) \).

Messages sent and received by \( Z \)

The environment \( Z \) sends the parties messages of the following forms:

- \( (i\text{-send}, \mu, u, t) \) (such messages are sent only to \( P_1 \)),
- \( (i\text{-receive}, \mu, u, t) \) (such messages are sent only to \( P_n \)),
- \( (i\text{-push}, \pi, v, t) \), and
- \( (i\text{-acknowledge}, \pi) \).

The parties send to \( Z \) messages of the following forms:

- \( (i\text{-sent}, \mu, \text{receipt}) \) (such messages are sent only by \( P_1 \), in case of \( \text{EthNA} \) receipt has a form \( [(u, \mu, t + P), R] \)),
- \( (i\text{-pushed}, \pi, v, t) \), and
- \( (i\text{-acknowledged}, \pi, v) \).

Figure 10: The messages exchanged in \( \text{EthNA} \) (assuming the payment path is \( P_1 \rightarrow \cdots \rightarrow P_n \)).
A **channel graph** is a tuple $\mathcal{G} = (\mathcal{P}, \mathcal{E}, \Gamma)$ with the set of vertices $\mathcal{P} = \{P_1, \ldots, P_n\}$ and set $\mathcal{E}$ of edges being a family of two-element subsets of $\mathcal{P}$. The elements of $\mathcal{P}$ are denoted as “$P_i \rightsquigarrow P_j$” (instead of $\{P_i, P_j\}$). Every $P_i \rightsquigarrow P_j$ represents a channel between $P_i$ and $P_j$. The **cash function** $\Gamma$ determines the number of coins available for the parties in every channel: every $\Gamma(P_i \rightsquigarrow P_j)$ is a function $f$ of a type $f: \{P_i, P_j\} \rightarrow \mathbb{Z}_{\geq 0}$. We often write $\Gamma^P_i \rightsquigarrow P_j$ to denote this function. The value $\Gamma^P_i \rightsquigarrow P_j (P)$ denotes the amount of coins that $P$ has in her **account** in channel $P_i \rightsquigarrow P_j$.

A **payment report** for $\mu$ is a set $\mathcal{W}$ of subreceipts for $\mu$ such that $\pi$ identifies a member of $\mathcal{W}$ uniquely, i.e.:

$$\forall \pi, \lambda \in \mathcal{W} \text{ s.t. } \pi, \lambda \not\in \mathcal{W} \land \pi \neq \lambda \Rightarrow \exists \lambda' \in \mathcal{W} : \pi, \lambda \not\in \mathcal{W} \land \pi \neq \lambda' \Rightarrow \lambda = \lambda'. $$

For a payment report $\mathcal{W}$ a subreceipt $\pi, \lambda \in \mathcal{W}$ is a **leader of $\mathcal{W}$ at node $P$** if $P$ appears on $\pi$ at some position $i$, and for every $\pi', \lambda' \in \mathcal{W}$ we have that $\lambda[i] \geq \lambda'[i]$. 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.

A **fraud-proof** (for $\mu$) is a payment report $Q$ for $\mu$ of a form $Q = \{[(\sigma, \pi_i), \lambda_i]\}_{i=1}^m$, where all the $\pi_i[1]'s$ are pairwise distinct, such that the following condition holds: $\max_{i=1,...,m} \lambda_i[\sigma_i] < \sum_{i=1}^m \lambda_i[\sigma_i + 1]$.

A **payment tree** $\text{tree}(\mathcal{W})$ is a pair $(T, \mathcal{L})$, where $T$ is the set of all prefixes of the $\pi^i$'s, i.e.,

$$T := \bigcup_i \text{prefix}(\pi^i),$$

and for every $\pi \in T$ we let

$$\mathcal{L}(\pi) := \sum_{i: \pi \in \text{prefix}(\pi^i)} v^i.$$