Astrape: Anonymous Payment Channels with Boring Cryptography[⋆]

Yuhao Dong, Ian Goldberg, Sergey Gorbunov, and Raouf Boutaba

University of Waterloo

Abstract. The increasing use of blockchain-based cryptocurrencies like Bitcoin has run into inherent scalability limitations of blockchains. Payment channel networks, or PCNs, promise to greatly increase scalability by conducting the vast majority of transactions outside the blockchain while leveraging it as a final settlement protocol. Unfortunately, first-generation PCNs have significant privacy flaws. In particular, even though transactions are conducted off-chain, anonymity guarantees are very weak. In this work, we present Astrape, a novel PCN construction that achieves strong security and anonymity guarantees with simple, black-box cryptography, given a blockchain with flexible scripting. Existing anonymous PCN constructions often integrate with specific, often custom-designed, cryptographic constructions. But at a slight cost to asymptotic performance, Astrape can use any generic public-key signature scheme and any secure hash function, modeled as a random oracle, to achieve strong anonymity, by using a unique construction reminiscent of onion routing. This allows Astrape to achieve provable security that is "generic" over the computational hardness assumptions of the underlying primitives. Astrape's simple cryptography also lends itself to more straightforward security proofs compared to existing systems.
Furthermore, we evaluate Astrape's performance, including that of a concrete implementation on the Bitcoin Cash blockchain. We show that despite worse theoretical time complexity compared to state-of-the-art systems that use custom cryptography, Astrape operations on average have a very competitive performance of less than 10 milliseconds of computation and 1 KB of communication on commodity hardware. Astrape explores a new avenue to secure and anonymous PCNs that achieves similar or better performance compared to existing solutions.

1 Introduction

1.1 Payment channel networks

Blockchain cryptocurrencies are gaining in popularity and becoming a significant alternative to traditional government-issued money. For instance, over 300,000

[⋆] This is an extended version of our paper that appeared in the proceedings of the 20th International Conference on Applied Cryptography and Network Security (ACNS 2022) [12]. The accompanying source code is available at https://github.com/nullchinchilla/astrape-paper/.

Bitcoin transactions alone [2] are processed every day. Unfortunately, such high demand inevitably leads to well-known scalability barriers [8]. Bitcoin, for instance, processes less than 10 transactions every second [21], far less than a reasonable global payment system.

Payment channels [11] are a common technique to scale cryptocurrency transactions. In a nutshell, Alice and Bob open a payment channel by submitting a single transaction to the blockchain, locking up a sum of cryptocurrency from both of the parties. They can then pay each other by simply mutually signing a division of the locked money. Additional blockchain transactions are required only when the channel is closed by submitting an up-to-date signed division, unlocking the latest balances of Alice and Bob. This allows most activity to remain off-chain, while retaining the blockchain for final settlement: as long as the blockchain is secure, nobody can steal funds. More importantly, payment channels can be organized into *payment-channel networks* (PCNs) [21], where users without any open channels between them can pay each other through intermediaries.

1.2 Anonymity in PCNs

Unfortunately, "first-generation" PCNs based on the HTLC (hash time-locked contract), such as Lightning Network [11], have a significant problem — poor anonymity [19]. In the worst case, HTLC payments are as transparently linkable as blockchain payments [19], threatening the improved privacy that is often cited [25, 26] as a benefit of PCNs. Furthermore, naive implementations fall victim to subtle fee-stealing attacks, like the "wormhole attack" [20], that threaten economic viability.

A sizable body of existing work on fixing PCN security and privacy exists. On one hand, specialized constructions achieve strong anonymity in specific settings, such as Bolt [15] for hub-based PCNs on the Zcash blockchain, providing for indistinguishability of two concurrent transactions even when all intermediaries are malicious. On the other hand, general solutions for all PCN topologies, like Fulgor [19] and the AMHL (Anonymous Multi-hop Locks) family [20], achieve a somewhat weaker, topology-dependent notion of anonymity: *relationship anonymity* [6, 18]. This property, common to onion-routing and other anonymous communication protocols, means that two concurrent transactions cannot be distinguished as long as they share at least one honest intermediary.

1.3 Why boring cryptography?

Unfortunately, there remains a shortcoming common to all existing anonymous PCN constructions — custom, often number-theoretic and sometimes complex cryptographic primitives. No existing anonymous PCN construction limits itself to the bare-bones cryptographic primitives used in HTLC — black-box access to a generic signature scheme and hash function. For example, AMHL uses either homomorphic one-way functions or special constructions that exploit the mathematical structure of ECDSA or Schnorr signatures and Tumblebit uses a custom cryptosystem based on the RSA assumption. Blitz [5], though relying on an ostensibly black-box signature scheme, requires it to have a property[1] that rules out many post-quantum signature schemes.

The key objective of these constructions is reconciling anonymity with *balance security* — ensuring that the entire multi-hop transaction either completes correctly or reverts all money to the sender. The former requires hiding important information, while the second requires verifying it, making a zero-knowledge protocol a seemingly natural fit. Indeed, for hub-based anonymous PCNs, a strong zero-knowledge cryptosystem is likely necessary — privacy against an adversary controlling *everyone* other than the counterparties is a difficult goal to achieve.

However, it is unclear that relationship anonymity requires sophisticated techniques. Relationship anonymity appears to be relatively "easy" elsewhere. Well-understood anonymous constructs like onion routing and mix networks exist for communication with no more than standard primitives used in secure communication (symmetric and asymmetric encryption). Of course, communication is probably easier — indeed some go beyond relationship anonymity with only simple cryptography — but it seems plausible that PCNs can use similarly elementary primitives to achieve anonymity.

Furthermore, "boring" cryptography has practical advantages. For one, non-standard cryptography poses significant barriers to adoption. Reliable and performant implementations of novel cryptographic functions are difficult to obtain, and tight coupling between a PCN protocol and a particular cryptographic construction makes swapping out primitives impossible. With use of black-box cryptography, a system is *generic over cryptographic hardness assumptions* — instead of assuming that, say, the RSA or discrete-log problems are hard, we only need to assume that there exists, for example, *some* secure signature scheme and *some* secure hash function.

Thus, we believe that efficient yet privacy-preserving PCNs that only use well-understood and easily replaced black-box cryptographic primitives are crucial to usable PCNs. In fact, AMHL's authors already proposed that "an interesting question related to [anonymous PCN constructions] is under which class of hard problems such a primitive exists" [20] that they conjecturally answered with linear homomorphic one-way functions.

1.4 Our contributions

In this paper, we present Astrape,[2] a PCN protocol that limits itself to "boring", generic cryptography already used in HTLC, yet achieves strong relationship anonymity. Despite achieving comparable security, privacy, and performance to

[1] In particular, the ability for any party, given any public key, to generate new public keys that correspond to the same private key yet are unlinkable to the previous public key. This is crucial to the "stealth addresses" that Blitz's pseudonymous privacy rests upon.

[2] Greek for "lightning", pronounced "As-trah-pee".

other anonymous PCN constructions, Astrape does not introduce any cryptographic constructs other than those used in HTLC. This is accomplished using a novel construct reminiscent of onion routing that avoids the use of any form of zero-knowledge verification.

– Section 2 discusses existing payment channel networks, focusing on their security and privacy properties. We show that first-generation PCNs have significant privacy and security problems, and discuss existing work on improving them.
– Section 3 introduces a formal model of PCN constructions, "generalized multi-hop locks" (GMHL), by generalizing "anonymous multi-hop locks" [20]. We model the security and privacy properties we desire within this model.
– Within the GMHL framework, we introduce Astrape, a new PCN construction that solves the security and privacy issues of HTLC-based constructs with only black-box cryptographic primitives in Section 4. Strong privacy without new cryptographic primitives solves an open problem in the field. Astrape builds upon existing work [19] on private PCNs but uses a new technique that, at a modest cost in lock and unlock size as well as script complexity, avoids the need for verifying hidden information.
– Section 5 presents a security and privacy analysis, showing that Astrape achieves the same security and privacy goals as existing work like Fulgor [19] and AMHL [20]. Section 6 discusses some potential difficulties in practical deployment of Astrape — in particular, difficulty in deployment to blockchains without smart contracts — and presents solutions.
– Finally, in Section 7 we implement Astrape and show that on average, Astrape requires less computation or communication than state-of-the-art private PCN constructions. We show that Astrape can easily be ported to different blockchains.

2 Background and related work

2.1 Payment channels

The fundamental building block of payment channel networks is the bilateral payment channel [21]. A payment channel is a construct where two parties Alice and Bob deposit money into a blockchain-enforced "vault" that can be only opened with signatures from both Alice and Bob. Alice and Bob can then send each other money by privately agreeing on the distribution of funds between them by signing dated statements. When one of the parties needs to access the funds, they can simply open the vault using any statement that was produced in the course of using the payment channel, subject to a short period of time where the other party may override it with a later-dated statement. This mechanism ensures security of funds even if one of the parties is malicious.

2.2 First-generation PCNs with HTLC

An extremely useful property of payment channels is that they can be used to construct *payment channel networks* (PCNs) [21, 10, 8], allowing users without channels directly between each other to pay each other via intermediaries. At the heart of any PCN is a *secure multi-hop transaction* mechanism — some way of Alice paying Bob to pay Carol without any trust in Bob. Most PCNs implement this using a smart contract known as the *Hash Time-Lock Contract* (HTLC). An HTLC is parameterized over a *sender* Alice, the *recipient* Bob, a deadline t, and a *puzzle* s. It locks up a certain amount of money, unlocking it according to the following rules:

– The money goes to Bob if he produces π where $H(\pi) = s$ before time t, where H is a secure hash function.
– Otherwise, the money goes to Alice.

We can use HTLC to construct secure multi-hop transactions. Consider a sender U_0 wishing to send money to a recipient U_n through untrusted intermediaries $U_1, \ldots, U_{n-1}$. At first, U_0 will generate a random π and $s = H(\pi)$, while sending the pair (π, s) to U_n over a secure channel. U_0 can then lock money in a HTLC parameterized over U_0, U_1, s, t_1, notifying U_1. U_1 would send an HTLC over U_1, U_2, s, t_2, notifying U_2, and so on. The deadline must become earlier at each step — $t_1 > t_2 > \cdots > t_n$ — this ensures that in case of an uncooperative or malicious intermediary, funds always revert to the sender.

The payment eventually will be routed to U_n, who will receive an HTLC over U_{n-1}, U_n, s, t_n. The recipient will claim the money by providing π; this allows U_{n-1} to claim money from U_{n-2} using the same π, and so on, until all outstanding HTLC contracts are fulfilled. U_0 has successfully sent money to U_n, while the preimage resistance of H prevents any intermediary from stealing the funds.

2.3 Hub-based anonymous payment channels

Unfortunately, HTLC has an inherent privacy problem — a common identifier $s = H(\pi)$ visible to all nodes in the payment path [15, 19, 20]. This motivates *anonymous* PCN design. *Hub-based* approaches form the earliest kind of anonymous PCN design. Here, the shape of the network is limited to a star topology with users communicating with a centralized hub. Some solutions are highly specialized, such as Green and Miers' Bolt [15], which relies on the Zcash blockchain's zero-knowledge cryptography. Other solutions, such as Tumblebit [16] and the more recent A2L [24], provide more general solutions that work on a wide variety of blockchains.

Hub-based PCN constructions tackle the difficult problem of providing unlinkability between transactions despite the existence of only a single untrusted intermediary. It is therefore unsurprising that specialized cryptography is needed to protect anonymity. On the other hand, observations of real-world PCNs like the Lightning Network, as well as economic analysis [13], show that actual PCNs often have intricate topologies without dominating hubs. General, topology-agnostic solutions are thus more important to deploying private PCNs in practice.

2.4 Relationship-anonymous payment channels

Unlike hub-based approaches, where no intermediaries are trusted, general private PCN constructions target *relationship anonymity*. This concept, shared with onion routing and other anonymous communication protocols, assumes at least one honest intermediary. Thus intermediaries are in fact crucial to relationship-anonymous PCNs' privacy properties. Like most hub-based approaches, relationship-anonymous payment channels do not by themselves deal with information leaked by side channels such as timing and value. We return to this subject in Section 6.2.

The earliest solution to PCN privacy in this family was probably Fulgor and Rayo [19], a closely related pair of constructions that can be ported to almost all HTLC-based PCNs. Fulgor/Rayo combines a "multi-hop HTLC" contract with out-of-band ZKPs to remove the common identifier across payment hops.

In a later work, Malavolta et al. [20] introduced *anonymous multi-hop locks* (AMHL), a rigorous theoretical framework for analyzing private PCN contracts. The AMHL paper provided a concrete instantiation using linear homomorphic one-way functions (hOWFs), as well as a conjecture that hOWFs are necessary for implementing anonymous PCNs. They also presented a variant that uses a clever encoding of homomorphic encryption in ECDSA to be used in ECDSA-based cryptocurrencies like Bitcoin. The latter "scriptless" variant was generalized in later work to a notion of adaptor signatures [4], where a signature scheme like ECDSA is "mangled" in such a way that a correct signature reveals a secret based on a cryptographic condition. The authors of AMHL also discovered "wormhole attacks" on HTLC-based PCNs. These attacks exploit a fundamental flaw in the HTLC construction to allow malicious intermediaries to steal transaction fees from honest ones, a problem that AMHL's anonymity techniques also solve.

More recently, Blitz [5] introduced *one-phase* payment channels that support multi-hop payments without a two-phase separation of coin creation and spending, improving performance and reliability. Blitz also achieves stronger anonymity than HTLC, but its notion of anonymity is strictly weaker than the relationship anonymity of AMHL and Fulgor/Rayo. Other relationship-anonymous systems consider powerful adversaries that control most nodes and achieve indistinguishability of concurrent transactions, but Blitz considers local adversaries controlling a single intermediary and limits itself to hiding the rest of the path from this intermediary.

3 Our approach

As we argued in Section 1.3, all of these existing solutions share an undesirable reliance on either custom cryptographic constructions or primitives with special properties, like Blitz's stealth-address signature schemes. This causes inflexibility, difficult implementation, and an inability to respond to cryptanalytic breakthroughs like practical quantum computing.

Astrape is our solution to this problem. We show with a novel design that avoids the zero-knowledge verification paradigm, anonymous and atomic multi-hop transactions can be constructed with nothing but the two building blocks of HTLCs — hashing and signatures. Unlike existing work, no specific assumptions about the structure of the hash function or signature scheme are made, allowing Astrape to be easily ported to different concrete cryptographic primitives and its security properties to "fall out" from those of the primitives. This also allows Astrape to achieve high performance on commodity hardware using standard cryptographic libraries.

3.1 Generalized multi-hop locks

In our discussion of Astrape, we avoid describing the concrete details of a specific payment channel network and cryptocurrency. Instead, we introduce an abstract model — generalized multi-hop locks. This model readily generalizes to different families of payment channel networks.

We model a *sender*, U_0, sending money to a *receiver*, U_n, through intermediaries $U_1, \ldots, U_{n-1}$. We assume a "source routing" model, where the graph of all valid payment paths in the network is publicly known and the sender can choose any valid path to the recipient. After an *initialization* phase where the sender may securely communicate parameters to each hop, each user U_i where $i < n$ *creates* a *coin* and notifies U_{i+1}. This coin is simply a contract ℓ_{i+1} known as a *lock script*, that essentially releases money to U_{i+1} given a certain key k_{i+1}. We call this lock the *right lock* of U_i and the *left lock* of U_{i+1}.

Finally, the payment completes once all coins created in the protocol have been unlocked and spent by fulfilling their lock scripts. Typically, this happens through a chain reaction where the recipient's left lock ℓ_n is unlocked, allowing U_{n-1} to unlock its left lock, etc.

Formally, we model a GMHL over a set of participants U_i as a tuple of four PPT algorithms $\mathbb{L} = (\mathsf{Init}, \mathsf{Create}, \mathsf{Unlock}, \mathsf{Vf})$, defined as follows:

Definition 1. *A GMHL* $\mathbb{L} = (\mathsf{Init}, \mathsf{Create}, \mathsf{Unlock}, \mathsf{Vf})$ *consists of the following polynomial-time protocols:*

1. $\langle s_0^I, \ldots, (s_n^I, k_n))\rangle \Leftarrow \langle \mathsf{Init}_{U_0}(1^\lambda, U_1, \ldots, U_n), \mathsf{Init}_{U_1}, \ldots, \mathsf{Init}_{U_n}\rangle$*: the initialization protocol, started by the sender* U_0*, that takes in a security parameter* 1^λ *and the identities of all hops* U_i *and returns an initial state* s_i^I *to all users* U_i*. Additionally, the recipient receives a key* k_n*.*
2. $\langle(\ell_i, s_{i-1}^R), (\ell_i, s_i^L)\rangle \Leftarrow \langle \mathsf{Create}_{U_{i-1}}(s_{i-1}^I), \mathsf{Create}_{U_i}(s_i^I)\rangle$*: the coin-creating protocol run between two adjacent hops* U_{i-1} *and* U_i*, creating the "coin sent from* U_{i-1} *to* U_i*". This includes a lock representation* ℓ_i *as well as additional state on both ends — unlocking the lock represented by* ℓ_i *releases the money.*
3. $k_i \Leftarrow \mathsf{Unlock}_{U_i}(\ell_{i+1}, (s_i^I, s_i^L, s_i^R), k_{i+1})$*: the coin-spending protocol, run by each intermediary* U_i *where* $i < n$*, obtains a valid unlocking key* k_i *for the*

"left lock" ℓ_i given its "right lock" ℓ_{i+1}, its unlocking key k_{i+1} (already verified by Vf *below), and U_i's internal state.*

4. $\{0, 1\} \Leftarrow \mathsf{Vf}(\ell, k)$*: given a lock representation ℓ and an unlocking key k, return* 1 *iff the k is a valid solution to the lock ℓ*

As an example, a formalization of HTLC in the GMHL model can be found in Appendix A.

Generalizability to non-PCN systems. We note here that GMHL makes no mention of typical PCN components such as channels, the blockchain, etc. This is because GMHL is actually agnostic of *how* exactly the locks are evaluated and enforced. In a typical PCN, these locks will be executed within bilateral payment channels, falling back to a public blockchain for final settlement.

However, other enforcement mechanisms can be used. Notably, all the locks could simply be contracts directly executing on a blockchain. In this way, any anonymous PCN formulated in the GMHL model is equivalent to a specification for a provably anonymous *on-chain, multi-hop coin tumbling service* that can anonymize entirely on-chain payments by routing them through multiple intermediaries.

Comparison to existing work. GMHL is an extension of *anonymous multi-hop locks*, the model used in the eponymous paper by Malavolta et al. [20]. In particular, AMHL defines an anonymous PCN construction in terms of the operations KGen, Setup, Lock, Rel, Vf, four of which correspond to GMHL functions.

Although AMHL's model is useful, we could not use it verbatim. This is largely because AMHL's original definition [20] also included its security and privacy properties, while we wish to be able to use the same framework in a purely *syntactic* fashion to discuss PCNs with other security and anonymity goals.

Nevertheless, GMHL can be considered as AMHL, reworded and used in a more general context. As we will soon see, Astrape's desired security and privacy properties are actually very similar to those of AMHL, though we will consider other systems formulated in the GMHL framework along the way. Astrape can be considered an alternative implementation of the same "anonymous multi-hop locks" [20] construct.

3.2 Security and execution model

Now that we have a model to discuss PCN constructions, we can discuss our security model, as well as a model of the GMHL execution environment in which Astrape will execute.

Active adversary. We use a similar adversary model to that of AMHL [20]. That is, we model an adversary $\mathcal{A}$ with access to a functionality $\mathsf{corrupt}(U_i)$ that takes in the identifier of any user U_i and provides the attacker with the complete internal state of U_i. The adversary will also see all incoming and outgoing communication of U_i. $\mathsf{corrupt}(U_i)$ will also give the adversary active control of U_i, allowing it to impersonate U_i when communicating with other participants.

Anonymous communication. We assume there is a secure and anonymous message transmission functionality $\mathcal{F}_{\text{anon}}$ that allows any participant to send messages to any other participant. Messages sent by an honest (non-corrupted) user with $\mathcal{F}_{\text{anon}}$ hide the identity of the sender and cannot be read by the adversary, although the adversary may arbitrarily delay messages.

There are many ways of implementing $\mathcal{F}_{\text{anon}}$, the exact choice of which is outside the scope of this paper. One solution recommended by existing work [19, 20] is an onion-routing circuit constructed over the same set of users U_i, constructed with a provably private protocol like Sphinx [9]. Public networks such as Tor may also be used to implement $\mathcal{F}_{\text{anon}}$.

Exposed lock activity. In contrast to communication, *lock activity* — the content of all locks being created, as well as the unlocking keys during unlocking — is not secure. This is because in practice, lock activity often happens on public media like blockchains. We pessimistically assume that the adversary can see all lock activity, while a non-adversary only sees lock activity concerning locks that it sends and receives.

Liveness and timeouts. We assume that every coin lock ℓ_i comes with an appropriate timeout that will return money to U_{i-1} (i.e., able to be unlocked by a signature from U_{i-1} after the timeout) if U_i does not take action. We also assume that each left lock ℓ_i's deadline is at least δ later than that of the right lock ℓ_{i+1}, where δ is an upper bound on network latency between honest parties, even under disruption by the adversary. In the most common setting of a PCN consisting of bilateral payment channels backed by a blockchain, this is essentially a blockchain censorship-resistance assumption. With a liveness assumption, we can then omit timeout handling from the description of the protocol, in line with related work (such as AMHL [20]).

Infallible lock execution. We formulate Astrape in the GMHL model, and assume the existence of a mechanism that will guarantee that cryptocurrency locks are always correctly executed in the face of arbitrary adversarial activity. In practice, both bilateral payment channels falling back to a general-purpose public blockchain (like Ethereum) and direct use of this blockchain are good approximations of this mechanism.

Lock functionality. We assume that inside our on-chain contracts we are able to use at least the following operations:

- *Concatenation*, producing a bitstring $x||y$ of length $|n+m|$ from two bitstrings x, y, where x has length n and y has length m.
- *Bitwise XOR*, producing a bitstring $x \oplus y$ from two bitstrings x, y

as well as the cryptographic hash function H defined below. An implication of this assumption is that PCNs on blockchains with highly restricted scripting languages, like Bitcoin, cannot use Astrape.

Cryptographic assumptions. One of Astrape's main goals is to make minimal cryptographic assumptions. We assume only:

– *Generic cryptographic hash function.* We assume a hash function H, modeled as a random oracle for the purpose of security proofs, producing λ bits of output, where 1^λ is the security parameter. We use the random oracle both as a pseudorandom function and as a commitment scheme, which is well known [7] to be secure.
– *Generic signature scheme.* We assume a secure signature scheme that allows for authenticated communication between any two users U_i and U_j.

3.3 Security and privacy goals

Against the adversary we described above, we want to achieve the following security and privacy objectives:

Relationship anonymity. Given two simultaneous payments between different senders $S_{\{0,1\}}$ and receivers $R_{\{0,1\}}$ with payment paths of the same length intersecting at the same position at at least one honest intermediate user, an adversary corrupting all of the other intermediate users cannot determine, with probability non-negligibly better than 1/2 (guessing), whether S_0 paid R_0 and S_1 paid R_1, or S_0 paid R_1 and S_1 paid R_0. This is an established standard for anonymity in payment channels [19, 20] and is analogous to similar definitions for anonymous communication [23, 6]. It is important to note that the adversary is not allowed to corrupt the sender — senders always know who they are sending money to.

Balance security. For an honest user U_i, if its right lock ℓ_i is unlocked, U_i must always be able to unlock its left lock ℓ_{i-1} even if all other users are corrupt. Combined with the timeouts mentioned in our security model, this guarantees that no intermediary node can lose money even if everybody else conspires against it.

Wormhole resistance. We need to be immune to the *wormhole attack* on PCNs, where malicious intermediaries steal fees from other intermediaries. The reason why is rather subtle [20], but for our purposes this means that given an honest sender and an honest intermediary U_{i+1}, ℓ_i cannot be spent by U_i until U_{i+1} spends ℓ_{i+1}. Intuitively, this prevents honest intermediaries from being "left out".

4 Construction

4.1 Core idea: balance security + honest-sender anonymity

Unlike existing systems that utilize the mathematical properties of some cryptographic construction to build a secure and anonymous primitive, Astrape is constructed out of two separate *broken* constructions, both of which use boring cryptography and are straightforward to describe:

– **XorCake**, which has relationship anonymity but lacks balance security if the sender U_0 is malicious
– **HashOnion** which has balance security, but *loses relationship anonymity* in the Unlock phase. That is, an adversary limited only to observing Init and Create cannot break relationship anonymity, but an adversary observing Unlock can.

The key insight here is that if we can combine XorCake and HashOnion in such a way to ensure that HashOnion's Unlock phase *can only reveal information when the sender is malicious*, we obtain a system, Astrape, that has both relationship anonymity and balance security. This is because the definition of relationship anonymity assumes an honest sender: if the sender is compromised, it can always simply tell the adversary the identity of its counterparty, breaking anonymity trivially. It is important to note that such a composition *does not in any way weaken anonymity* compared to existing "up-front anonymity" systems like AMHL, even in the most pessimistic case. If the sender is honest, Astrape reduces to XorCake and has strong anonymity. If the sender is compromised, on the other hand, the adversary already knows who is sending money to whom in all relationship-anonymous PCNs, as the sender must know the recipient to initiate the payment. In both cases, Astrape has anonymity equivalent to that of existing systems like AMHL and Fulgor.[3]

We now describe XorCake and HashOnion, and their composition into Astrape.

4.2 XorCake: anonymous but insecure against malicious senders

Let us first describe XorCake's construction. XorCake is an extremely simple construction borrowed from "multi-hop HTLC", a building block of Fulgor [19]. It has relationship anonymity, but not balance security against malicious senders.

Recall that in GMHL, the sender (U_0) wishes to send a sum of money to the recipient (U_n) through $U_1, \dots, U_{n-1}$. At the beginning of the transaction, the sender samples n independent λ-bit random strings $(r_1, \dots, r_n)$. Then, for all $i \in 1, \dots, n$, she sets n values $s_i = H(r_i \oplus r_{i+1} \oplus \cdots \oplus r_n)$, where H is a secure hash function. That is, s_i is simply the hash of the XOR of all the values r_j for $j \geq i$. U_0 then uses the anonymous channel $\mathcal{F}_{\text{anon}}$ to provide U_n the values (r_n, s_n) and all the other U_i with (r_i, s_i, s_{i+1}).

Then, for each pair of neighboring nodes (U_i, U_{i+1}), U_i sends U_{i+1} a coin encumbered by a regular HTLC ℓ_{i+1} asking for the preimage of s_{i+1}. U_n knows how to unlock ℓ_n, and the solution would let U_{n-1} unlock ℓ_{n-1}, and so on. That is, each lock ℓ_i is simply an HTLC contract asking for the preimage of s_i.

In Appendix A, we give the formal definition of XorCake in the GMHL framework.

[3] In a sense then, Astrape has "pseudo-optimistic" anonymity. Its design superficially suggests an optimistic construction with an anonymous "happy path" and a non-anonymous "unhappy path", but a more careful analysis in Section 5.1 reveals that the latter non-anonymity is illusory — the sender can always prevent the "unhappy" path from deanonymizing the transaction even if all other parties are malicious.

XorCake by itself satisfies relationship anonymity. A full proof is available in the Fulgor paper from which XorCake was borrowed [20], but intuitively this is because r_i will be randomly distributed over the space of possible strings because H behaves like a random oracle. This means that unlike in HTLC, no two nodes U_i and U_j can deduce that they are part of the same payment path unless they are adjacent.

State-mismatch attack. Unfortunately, *XorCake does not have balance security.* Consider a malicious sender who follows the protocol correctly, except for sending an incorrect r_i to U_i. (Note that U_i cannot detect that r_i is incorrect given a secure hash function.) Then, when ℓ_{i+1} is unlocked, ℓ_i cannot be spent! In an actual PCN such as the Lightning Network, all coins "left" of U_i will time out, letting the money go back to U_0. U_0 paid U_n with U_i's money instead of her own. We call this the "state-mismatch attack", and because of it, XorCake is not a viable PCN construction on its own. In Fulgor, XorCake was combined with out-of-band zero-knowledge proofs of the correctness of r_i, but as we will see shortly, Astrape can dispense with them.

4.3 HashOnion: secure but eventually non-anonymous

We now present HashOnion, a PCN construction that has balance security but not relationship anonymity. Note that unlike HTLC, HashOnion's non-anonymity stems entirely from information leaked in the Unlock phase, a property we will leverage to build a fully anonymous construction combining HashOnion and XorCake.

At the beginning of the transaction, U_0 generates random values s_i for $i \in \{1, \ldots, n\}$, then "onion-like" values x_i, recursively defined as $x_i = H(s_i || x_{i+1})$, $x_n = H(s_n || 0^\lambda)$.

Essentially, x_i is a value that commits to all s_j where $j \geq i$. An onion-like commitment is used rather than a "flat" commitment (say, a hash of all s_j where $j \geq i$) as it is crucial for balance security, as we will soon see.

For all intermediate nodes $0 < i < n$, the sender sends (x_{i+1}, s_i) to U_i, while for the destination, the sender sends s_n. Then, each intermediary U_{i-1} sends to its successor U_i a lock ℓ_i, which can be only be unlocked by some $k_i = (s_i, \ldots, s_n)$ where $H(s_i || H(s_{i+1} || H(\ldots H(s_n || 0^\lambda)))) = x_i$. U_{i-1} constructs this lock from the x_i it received from the sender. Finally, during the unlock phase, the recipient U_n solves ℓ_n with $k_n = (s_n)$. This allows each U_i to spend ℓ_i, completing the transaction.

For balance security, we need to show that with a solution $k_{i+1} = (s_{i+1}, \ldots, s_n)$ to ℓ_{i+1}, and s_i, we can always construct a solution to ℓ_i . This is obvious: we just add s_i to the solution: $k_i = (s_i, s_{i+1}, \ldots, s_n)$.

One subtle problem is that U_i needs to make sure that its left lock is actually the correct ℓ_i and not some bad ℓ'_i parameterized over some $x'_i \neq H(s_i || x_{i+1})$. Otherwise, its right lock might get unlocked with a solution that does not let it unlock its left lock. Fortunately, this is easy: given s_i, x_{i+1} from the sender, U_i can just check that its left lock, parameterized over some x_i, matches $x_i = H(s_i || x_{i+1})$

before sending out ℓ_{i+1} (parameterized with x_{i+1}) to the next hop. Thus, every user can make sure that if its right lock is unlocked, so can its left lock, so balance security holds.

We also see that although the unlocking procedure breaks relationship anonymity by revealing all the s_i, before the unlock happens, HashOnion does have relationship anonymity. This is because the adversary cannot connect the different x_i as long as one s_i remains secret — that of the one honest intermediary.

4.4 Securing XorCake+HashOnion

We now move on to composing XorCake and HashOnion. We do so by creating a variant of HashOnion that embeds XorCake and recognizes an *inconsistency witness*. That is, this variant of HashOnion will unlock only when given a combination of values that proves an attempt by the sender to execute a state-mismatch attack for XorCake.

To construct such a lock, after generating the XorCake parameters, U_0 creates n λ-bit values x_i recursively:

$$x_n = o_n, \qquad x_i = H(\overbrace{r_i || s_i || s_{i+1}}^{\text{XorCake parameters}} || o_i || x_{i+1})$$

where o_i is a random nonce sampled uniformly from all possible λ-bit values.[4] The intuition here is that x_i *commits* to all the information U_0 would give to all hops U_j where $j \geq i$.

Afterwards, the sender then uses $\mathcal{F}_{\text{anon}}$ to send (o_i, x_i, x_{i+1}) , in addition to the XorCake parameters (r_i, s_i, s_{i+1}), to every hop i. Every hop U_i checks that all the parameters are consistent with each other.

We next consider what will happen if the sender attempts to fool an intermediate hop U_i with a state-mismatch attack. U_{i+1} would unlock its left lock ℓ_{i+1} by giving k_{i+1} where $H(k_{i+1}) = s_{i+1}$ but $H(r_i \oplus k_{i+1}) \neq s_i$. This then causes U_i to fail to unlock its left lock.

But this attempt allows U_i to generate a cryptographic witness verifiable to anybody knowing x_i: λ-bit values k_{i+1}, r_i, s_i, s_{i+1}, o_i, x_{i+1} where:

$$H(k_{i+1}) = s_{i+1}, H(r_i \oplus k_{i+1}) \neq s_i, H(r_i || s_i || s_{i+1} || o_i || x_{i+1}) = x_i$$

This inconsistency witness proves that the preimage of s_{i+1} XOR-ed with r_i does not equal the preimage of s_i, demonstrating that the values given to U_i are inconsistent and that U_0 is corrupt. Since U_{i-1} knows x_i, U_i can therefore prove that it was a victim of a state-mismatch attack to U_{i-1}.

Since x_i commits to *all* XorCake initialization states "rightwards" of U_i, U_i, in cooperation with U_{i-1}, can also produce a witness that U_{i-2} can verify using x_{i-1}. This is simply a set of λ-bit values k_{i+1}, r_{i-1}, s_{i-1}, r_i, s_i,s_{i+1}, x_i, o_i, o_{i-1}, x_{i+1} where:

$$H(k_{i+1}) = s_{i+1}, H(r_i \oplus k_{i+1}) \neq s_i,$$

[4] || denotes concatenation. In our case, it is possible to unambiguously separate concatenated values, since we only ever concatenate λ-bit values.

```
function $\mathsf{Init}^{\mathrm{AS}}_{U_0}(1^\lambda, U_1, \dots, U_n)$
Upon invocation by $U_0$:
    generate $\lambda$-bit random numbers $\{r_1, \dots, r_n\}$
    $x_n \leftarrow$ random $\lambda$-bit number
    for $i$ in $n-1, \dots, 1$ do
        $s_i \leftarrow H(r_i \oplus r_{i+1} \oplus \cdots \oplus r_n)$
        $o_i \leftarrow$ random $\lambda$-bit number
        if $i < n$ then
            $x_i \leftarrow H(r_i || s_i || s_{i+1} || o_i || x_{i+1})$
    for $i$ in $1, \dots, n$ do
        if $i = n$ then
            send $s^I_n = (k_n = \mathsf{HSoln}[r_n], s_n)$ to $U_n$
        else
            send $s^I_i = (r_i, s_i, s_{i+1}, x_i, x_{i+1}, o_i)$ to $U_i$

function $\mathsf{Create}^{\mathrm{AS}}_{U_i}(s^I_i = (r_i, s_i, s_{i+1}, x_i, x_{i+1}, o_i))$
Upon invocation by $U_i$, where $i < n$:
    if $x_i \neq H(r_i || s_i || s_{i+1} || o_i || x_{i+1})$ then
        abort bad initial state
    if $i > 0$ then
        wait for $\ell_i = \mathsf{Astrape}[\hat{x}_i, \hat{s}_i]$ to be created
        if $\hat{x}_i \neq x_i$ or $\hat{s}_i \neq s_i$ then
            abort invalid left lock
    return $\ell_{i+1} = \mathsf{Astrape}[x_{i+1}, s_{i+1}]$

function $\mathsf{Unlock}^{\mathrm{AS}}_{U_i}(\ell_{i+1}, s^I_i, k_{i+1})$
Upon invocation by $U_i$, where $i < n$:
    $\mathit{\Gamma}_i \leftarrow r_i || s_i || s_{i+1} || o_i$
    parse $s^I_i = (r_i, s_i, s_{i+1}, x_i, x_{i+1}, o_i)$
    if parse $k_{i+1} = \mathsf{HSoln}[\kappa_{i+1}]$ then
        if $H(r_i \oplus \kappa_{i+i}) = s_i$ then
            return $k_i = \mathsf{HSoln}[r_i \oplus \kappa_{i+1}]$
        else
            return $k_i = \mathsf{WSoln}[\kappa_{i+1}, x_{i+1}, \{\mathit{\Gamma}_i\}]$
    else
        parse $k_{i+1} = \mathsf{WSoln}[\kappa_j, x_j, \{\mathit{\Gamma}_{i+1}, \dots, \mathit{\Gamma}_j\}]$
        return $k_i = \mathsf{WSoln}[\kappa_j, x_j, \{\mathit{\Gamma}_i, \mathit{\Gamma}_{i+1}, \dots, \mathit{\Gamma}_j\}]$

function $\mathsf{Vf}^{\mathrm{AS}}(\ell, k)$
    parse $\ell = \mathsf{Astrape}[x, s]$
    if parse $k = \mathsf{HSoln}[\kappa]$ then
        return 1 iff $H(\kappa) = s$        ▷ "normal" case
    else if parse $k = \mathsf{WSoln}[\kappa, \chi, \{\mathit{\Gamma}_i, \dots, \mathit{\Gamma}_j\}]$ then
        if $\exists i$ s.t. $\mathit{\Gamma}_i.\mathsf{length} \neq 4\lambda$ bits then
            return 0
        parse $\mathit{\Gamma}_j = r_j || s_j || s_{j+1} || o_j$
        if $H(r_j \oplus \kappa) = s_j$ then
            return 0        ▷ state good
        $\hat{x} \leftarrow H(\mathit{\Gamma}_i || H(\mathit{\Gamma}_{i+1} || \dots H(\mathit{\Gamma}_j || \chi)))$
        return 1 iff $\hat{x} = x$        ▷ "inconsistency" case
```

Fig. 1: Astrape as a GMHL protocol

$$H(r_i || s_i || s_{i+1} || o_i || x_{i+1}) = x_i, H(r_{i-1} || s_{i-1} || s_i || o_{i-1} || x_i) = x_{i-1}$$

We can clearly extend this idea all the way back to U_1 — given a witness demonstrating a state-mismatch attack against U_i, U_{i-1} can verify the witness and generate a similar one verifiable by U_{i-2}, and so on. This forms the core construction that Astrape uses to fix XorCake's lack of balance security.

4.5 Complete construction

We now present the complete construction of Astrape, as formalized in Figure 1 within the GMHL framework. Note that we use the notation $\mathsf{Tag}[x_1, \dots, x_n]$ to represent a n-tuple of values with an arbitrary "tag" that identifies the type of value.

Initialization. In the first phase, represented as Init in GMHL, the sender U_0 first establishes communication to the n hops $U_1, \dots, U_n$, the last one of which is the receiver. When talking to intermediaries and the recipient, U_0 uses our abstract functionality $\mathcal{F}_{\text{anon}}$.

The sender then generates random λ-bit strings $(r_1, \ldots, r_n)$ and $(o_1, \ldots, o_n)$, deriving $s_i = H(r_i \oplus r_{i+1} \oplus \cdots \oplus r_n)$ and $x_i = H(r_i || s_i || s_{i+1} || o_i || x_{i+1})$; $x_n = H(o_n)$. She then sends to each intermediate hop U_i the tuple $s_i^I = (r_i, s_i, s_{i+1}, x_i, x_{i+1}, o_i)$ and gives the last hop U_n the initial state $s_n^I = (r_n, s_n)$ and the unlocking key $k_n = \mathsf{HSoln}[r_n]$.

Creating the coins. We now move on to Create, where all the coins are initially locked. U_0 then sends U_1 a coin encumbered with a lock represented as $\ell_1 = \mathsf{Astrape}[x_1, s_1]$. When each hop U_i receives a correctly formatted coin from its previous hop U_{i-1}, it sends the next hop U_{i+1} a coin with a lock $\ell_{i+1} = \mathsf{Astrape}[x_{i+1}, s_{i+1}]$. Note that U_i checks whether its left lock is consistent with the parameters it received from U_0; this ensures that when U_i's right lock unlocks later, U_i can always construct a solution for its left lock. If the checks fail, Create aborts, and all of the locks will eventually time out (see Section 3.2), returning money to the sender.

As specified in Vf, each of these locks ℓ_i can be spent either through solving a XorCake-type puzzle to find the preimage of s_i (the "normal" case) or by presenting an inconsistency witness with a HashOnion-type witness demonstrating x_i's commitment to inconsistent data (the "inconsistency" case). After all the transactions with Astrape-encumbered coins are sent, U_n can finally claim its money, triggering the next phase of the protocol.

Unlocking the coins. The last step is Unlock. After receiving the final coin from U_{n-1}, the recipient unlocks its lock ℓ_n by providing to Vf the preimage of the HTLC puzzle: $k_n = \mathsf{HSoln}[r_n]$ — this is the only way an honest recipient can claim the money in a payment originating from an honest sender. Each intermediate node U_i reacts when its right lock ℓ_{i+1} is unlocked with key k_{i+1}:

- If U_{i+1} solved the HTLC puzzle with $k_{i+1} = \mathsf{HSoln}[\kappa_{i+1}]$, construct $\kappa_i = r_i \oplus \kappa_{i+1}$
  - If $H(\kappa_i) = s_i$, this means that there is no state-mismatch attack happening. We unlock our left lock with $k_i = \mathsf{HSoln}[\kappa_i]$.
  - Otherwise, there must be an attack happening. We construct a witness and create a key that embeds the witness verifiable with x_i. This gives us $k_i = \mathsf{WSoln}[\kappa_{i+1}, x_{i+1}, \{\Gamma_i\}]$, where $\Gamma_i = r_i || s_i || s_{i+1} || o_i$.
- Otherwise, U_{i+1} demonstrated that the sender attempted to defraud some U_j, where $j > i$ unlocked ℓ_{i+1} by presenting an inconsistency witness $k_{i+1} = \mathsf{WSoln}[\kappa_j, x_j, \{\Gamma_{i+1}, \Gamma_{i+2}, \ldots, \Gamma_j\}]$.
  - We can simply construct $k_i = \mathsf{WSoln}[\kappa_j, x_j, \{\Gamma_i, \Gamma_{i+1}, \ldots, \Gamma_j\}]$ where $\Gamma_i = r_i || s_i || s_{i+1} || o_i$. This transforms the witness verifiable with x_{i+1} to a witness verifiable with x_i.

Note that both cases are covered by Vf — it accepts and verifies both "normal" unlocks with HSoln-tagged tuples, and "inconsistency" unlocks with WSoln. Thus, even though Astrape is a composition of XorCake and HashOnion, the final construction fully "inlines" the two into the same flow of initialization, coin creation, and unlocking, with no separate procedure to process inconsistency witnesses. Unlocking continues backwards towards the sender until all the locks created in the previous step are unlocked. We have balance security — node U_i can unlock its left lock ℓ_i if and only if node U_{i+1} has unlocked ℓ_{i+1}, so no intermediaries can lose any money.

5 Security and privacy analysis

To prove security and privacy, we show that the checks done in the Init and Unlock phases fully prevent either violations of relationship anonymity through incorrect application of HashOnion or any violations of balance security, even when adversaries can arbitrary corrupt messages. Importantly, we show that under the definition of relationship anonymity, an adversary cannot forge messages in such a way to "fake" the propagation of an inconsistency witness and break anonymity. We also see that just as in AMHL, committing to the entire path during initialization eliminates the wormhole attack.

5.1 Relationship anonymity

Out of our three desired properties, relationship anonymity is the most important. Unlike plain XorCake, Astrape achieves relationship anonymity in a less trivial fashion; in particular we must show that the HashOnion component cannot lead to deanonymization.

Thus, we present proof that Astrape does achieve relationship anonymity. Note that within this proof, we assume that the sender U_0 and one particular U_i are honest according to the definition of relationship anonymity, while everybody else can be compromised by the adversary $\mathcal{A}$. We show that the only situation when information hurting relationship anonymity leaks is when U_0 is dishonest. Recall that relationship anonymity is only meaningful when U_0 is honest (otherwise U_0 can simply tell the adversary all the information), so this property *does not lose any anonymity* compared to systems like AMHL that unconditionally hide relationship-revealing information.

Recall that in $\mathcal{F}_{\text{anon}}$, messages are encrypted to the recipient, but the sender is hidden. Therefore, $\mathcal{A}$ cannot *read* the Init message sent from U_0 to U_i, but can *corrupt* (replace) it with a message of its own. This can prove problematic, because in the rest of our analysis of relationship anonymity we want to treat messages that U_i obtained through $\mathcal{F}_{\text{anon}}$ as authentic messages from U_0. However, if we prove that the adversary cannot gain any information from corrupting these messages, corrupting these messages cannot lead to a break in relationship anonymity, so we can then assume the messages are not corrupted. We also do not need to consider attacks where the adversary corrupts messages to the recipient or another intermediary, as the adversary is already allowed to directly corrupt those parties.

Lemma 1. *The adversary cannot gain any information by corrupting $\mathcal{F}_{\text{anon}}$ messages sent by U_0 to the honest intermediary U_i.*

Proof. Recall the Init message from U_0 to U_i is of the form $s_i^I = (r_i, s_i, s_{i+1}, x_i, x_{i+1}, o_i)$. Suppose the adversary replaces this message with $s_i^{I'} = (r'_i, s'_i, s'_{i+1}, x'_i, x'_{i+1}, o'_i)$. We first note that due to U_i's self-consistency check of $x'_i = H(r'_i || s'_i || s'_{i+1} || o'_i || x'_{i+1})$, any forgery of one variable requires the adversary either to *know* or *forge* every other variable. Therefore, after a corrupted Init, the only values U_0 wished to send to U_i that the adversary does not know — r_i, o_i — are destroyed. There is thus nothing in $s_i^{I'}$ now that U_i could reveal to the adversary in later stages of the protocol to possibly break relationship anonymity.

Lemma 2. *Given an honest sender and uncorrupted communication between it and the honest intermediary U_i, the left and right locks of U_i, ℓ_i and ℓ_{i+1}, are always unlocked normally and never by inconsistency witness. That is, the adversary cannot "frame" an honest sender to an honest intermediary.*

Proof. There are only two cases where an honest U_i unlocks the left lock ℓ_i by inconsistency witness:

1. The right lock was unlocked normally, but with an XorCake solution κ where $H(\kappa) = s_{i+1}$ yet $H(r_i \oplus \kappa) \neq s_i$.
2. The right lock was unlocked with a witness recognized by HashOnion.

The first case is impossible with an honest sender, since it would imply finding a hash collision in H.

In the second case, under the random oracle model, where we can assume that H acts as a binding commitment scheme, this means that x_{i+1} commits to inconsistent information that violates the protocol. But since we know that U_0 is honest, such an x_{i+1} could not have been sent to U_i, so this is impossible as well.

Thus, with an honest sender, there cannot be any inconsistency witnesses used when unlocking ℓ_i and ℓ_{i+1}.

Note: Lemma 2 *does not* claim that if U_0 is honest, no inconsistency witnesses can be generated at all — it only covers a very particular situation: the locks immediately adjacent to U_i in the case that communication is uncorrupted. In fact, two adjacent adversarial nodes U_j, U_k elsewhere can easily unlock the lock ℓ_k between them with a witness even when U_0 is honest by simply incorrectly executing Unlock (in case of off-chain payment channels) or replacing ℓ_k with a corrupt lock.

But can other inconsistency witnesses getting propagated break anonymity? Fortunately, no. First of all, dishonest intermediaries "acting out" an inconsistency witness with invalid locks between them cannot lead to a valid inconsistency witness that can unlock a *valid* lock, and thus cannot reveal any information that the dishonest intermediaries do not already know. The presence of these inconsistency witness therefore cannot lead to breaking anonymity.

A more interesting case is if the adversary *corrupts the locks around the honest intermediary.* After all, $\mathcal{F}_{\text{anon}}$ is not authenticated. Then, we would indeed be able to get an inconsistency witness to propagate across the honest intermediary even when the sender is honest — this sounds dangerous!

The catch is that in corrupting the locks, the adversary must replace the internal state of the honest intermediary entirely with information the adversary already knows. This destroys any information the adversary might be interested in, and any further behavior in the network by definition cannot leak anonymity-harming information. We showed this in Lemma 1. Intuitively, it is *untampered* secret information from U_0 held by the honest intermediary U_i, which can only be revealed by an inconsistency witness "propagating across" U_i, that must be leaked to violate relationship anonymity.

We now proceed to formalize the notion that without this information, the adversary cannot break relationship anonymity.

Lemma 3. *Given two simultaneous payments with the same value and paths* $\cdots \to U_{i-1} \to U_i \to U_{i+1} \to \cdots$ *and* $\cdots \to U'_{i-1} \to U_i \to U'_{i+1} \to \cdots$, *the attacker cannot distinguish this situation from two payments with paths* $\cdots \to U_{i-1} \to U_i \to U'_{i+1} \to \cdots$ *and* $\cdots \to U'_{i-1} \to U_i \to U_{i+1} \to \cdots$.

Proof. We assume that the attacker $\mathcal{A}$ has corrupted all intermediate nodes before and after U_i. This means that in the Init phase, $\mathcal{A}$ learns all states s_j^I where $j \neq i$ for the first transaction, and also all states $s_j'^I$ where $j \neq i$ for the second transaction, but not the internal state of U_i, which is $s_i^{(')I}$. The only values revealed to $\mathcal{A}$ that contain the internal state of U_i are commitments x_j where $j \leq i$ — but reconstructing U_i's initial state from x_j implies breaking the preimage resistance of H, which is impossible if H is a random oracle.

In the Create phase, we note that U_i obviously does not reveal any internal state except $s_{i+1}^{(')}$ and $x_{i+1}^{(')}$, which the attacker already knows from Init. Furthermore, by Lemma 2 we know that the two locks surrounding U_i can only be unlocked through the HTLC-based "normal" case, which again does not reveal more information about $s_i^{(')}$.

Thus, the attacker's game reduces to guessing the predecessor and successor of U_i for both transactions, given all states $s_j^{(')I}$ for $j \neq i$, with probability non-negligibly more than $1/2$ (as a function of λ). We can further reduce this to $\mathcal{A}$ guessing whether state s_{i-1}^I comes from the same transaction as s_{i+1}^I or $s_{i+1}'^I$; note that all of these states come from corrupt users adjacent to U_i.

$\mathcal{A}$ knows

$$s_{i-1}^I = (r_{i-1}, s_{i-1}, s_i, x_{i-1}, x_i, o_{i-1})$$
$$s_{i+1}^I = (r_{i+1}, s_{i+1}, s_{i+2}, x_{i+1}, x_{i+2}, o_{i+1})$$
$$s_{i+1}'^I = (r'_{i+1}, s'_{i+1}, s'_{i+2}, x'_{i+1}, x'_{i+2}, o'_{i+1})$$

$\mathcal{A}$ must then determine whether U_0 knows r_i, o_i, and κ_{i+1} such that ($x_i = H(r_i||s_i||s_{i+1}||o_i||x_{i+1})$ and $s_{i+1} = H(\kappa_{i+1})$) or else ($x_i = H(r_i||s_i||s'_{i+1}||o_i||x'_{i+1})$ and $s'_{i+1} = H(\kappa_{i+1})$), knowing that $s_i = H(r_i \oplus \kappa_{i+1})$. However, modelling H as a random oracle, these two cases are indistinguishable to $\mathcal{A}$, as required.

Theorem 1. *Astrape has relationship anonymity.*

Proof. This follows immediately from the preceding lemmas.

5.2 Balance security

We now show that Astrape has balance security. An interesting note is that balance security can be proven in a purely *local* way. As long as every intermediary can locally ensure that any right-lock solution can turn into a left-lock solution, no honest intermediary will end the protocol with less money than it started with, and balance security holds. Recall also from Section 4.5 that even the exceptional, inconsistency-witness-handling case fits into the same GMHL model — each intermediary U_i waiting for its right lock to unlock, and using the information revealed to construct a solution for its left lock that satisfies Vf. Thus, unlike the case with relationship anonymity, we do not need to consider the "rest of the world" other than U_i. Other participants and communication can be arbitrarily corrupt without affecting balance security.

Lemma 4. *An honest user U_i always receives enough information to unlock its left lock ℓ_i if its right lock ℓ_{i+1} is unlocked.*

Proof. We have two cases to consider.

In the first case, ℓ_{i+1} is unlocked by solving the XorCake component — with $k_{i+1} = \mathsf{HSoln}[\kappa_{i+1}]$.

Either κ satisfies $H(r_i \oplus \kappa) = s_i$ or it does not. If it does, U_i can construct $\kappa' = H(r_i \oplus \kappa)$ and use $k_i = \mathsf{HSoln}[\kappa]$ to unlock $\ell_i = \mathsf{Astrape}[s_i, x_i]$. If not, we have an inconsistency witness, and we can construct $k_i = \mathsf{WSoln}[\kappa, x_{i+1}, \{\Gamma_i\}]$ where $\Gamma_i = r_i||s_i||s_{i+1}||o_i$, and $H(\Gamma_i||x_{i+1}) = x_i$. By plugging into Vf, we see that this k_i is able to unlock ℓ_i.

In the second case, our right lock ℓ_{i+1} is unlocked by inconsistency using the HashOnion-like component. We can construct our own k_i to unlock ℓ_i by simply extending the witness with an additional Γ_i, which Vf will then accept.

Theorem 2. *Astrape has balance security.*

Proof. For every intermediary U_i, its left lock has funds equal to its right lock (plus fees). Following Lemma 4, this means that any money lost by U_i's right lock being spent can always be recovered by spending its left lock. Thus, intermediaries cannot lose any money.

Therefore, Astrape has balance security according to its definition in Section 3.3.

5.3 Wormhole immunity

The wormhole attack [20] is a generic attack on PCNs where given an honest sender and two corrupted nodes U_i, U_j, $j > i + 1$ separated by at least one honest node, U_i can unlock its left lock once U_j's right lock is unlocked. This "cuts out" all the participants in between, depriving them of fees from the sender.

Astrape is immune to the wormhole attack. This is because when U_{i+1} is honest, ℓ_{i+1} must be unlocked before U_i even has the information to unlock ℓ_i — information from "further down the line" at U_j where $j > i + 1$ is not sufficient. We note that Astrape fits the model given in the AMHL paper [20] for wormhole-immune PCN constructions, since the sender must know the complete path before sending money and there are two rounds of communication.

6 Practical concerns

6.1 Blockchain implementation

Astrape is easy to implement on blockchains with Turing-complete scripting languages, like Ethereum, as well as layer-2 PCNs such as Raiden built on these blockchains, but blockchains without Turing-complete scripts involve two main challenges.

First, these blockchains typically do not allow recursion or loops in lock scripts. This means that we cannot directly implement the Vf function. Instead, we must "unroll" Vf to explicitly check for witnesses to inconsistencies in the parameters given to U_i, U_{i+1}, etc. So for an n-hop payment the size of every lock script grows to $\Theta(n)$. In practice, the mean path length in the Lightning Network is currently around 5 (see our measurements in Section 7.5), and privacy-focused onion routing systems such as Tor or I2P typically use 3 to 5 hops. We believe linear-length script sizes are not a significant concern for Astrape deployment. An Astrape deployment can simply pick an arbitrary maximum for the number of hops supported and achieve reasonable worst-case performance.

The second issue is more serious: some blockchains have so little scripting that Astrape cannot be implemented. Astrape requires an "append-like" operation $||$ that can take in two bytestrings and combine them in a collision-resistant manner. Unfortunately, the biggest blockchain Bitcoin has disabled all string-manipulation opcodes. Whether an implementation based solely on the 32-bit integer arithmetic that Bitcoin uses is possible is an interesting open question.

6.2 Side-channel deanonymization

We follow existing literature on anonymous PCNs [19, 20] in pursuing relationship anonymity as our privacy goal. This means that a powerful adversary cannot distinguish between two payments sharing one honest intermediary that happen at exactly the same time with exactly the same values. However, real-life scenarios are rarely so clean, and violations of the assumptions of relationship anonymity can easily break privacy. For example, timing patterns may deanonymize users — perhaps only AliceCorp pays employees large lump sums on the 3rd and 14th of each month, letting the adversary detect AliceCorp employees.

Why, then, do we use relationship anonymity as our privacy goal? The main reason is that timing and value anonymity are largely *orthogonal* to relationship anonymity. Once we achieve relationship anonymity, we can adapt a large body of existing research in defeating side-channel attacks against anonymity systems like onion routing. Timing attack defenses can be directly lifted from works like Feigenbaum et al. [14], while the information leaked by cryptocurrency values can be vastly diminished simply by using fixed denominations.

6.3 Griefing attacks

An unusual property of Astrape is that its worst-case cost (the HashOnion backup mechanism) diverges from its "average"-case cost (XorCake). Compared to HTLC, Astrape might open the door to *griefing* attacks, where an adversary sends coins to itself to intentionally burden victim intermediaries with large lock sizes and verification costs. These costs can be substantial, as there are limits on the number of concurrent locks a single payment channel can support, imposed by the underlying blockchain.

Fortunately, both griefing based on verification costs and griefing based on lock sizes are easy to mitigate. In the first case, we can charge a penalty P for every hop the HashOnion case is used. For example, for each hop ℓ_i, in addition to the main payment coin encumbered by Astrape's composite XorCake/HashOnion lock, we can add a coin of value $P(n-i)$ which is unlockable by the same parameters that unlock the HashOnion case of the main coin. This way, if HashOnion is activated, every hop burdened by verifying HashOnion proofs can collect an extra P from its predecessor, which ultimately penalizes U_0.

Lock sizes are another possible way of griefing Astrape. For instance, the adversary can pay himself with a long path, but simply never unlock the coins and allow them to time out, burdening the victims with the cost of the locks. In Astrape, such a griefer does impose larger costs than in HTLC, as locks can be larger than in HTLC, but similar griefing problems occur in any system where failed transactions cannot incur fees, including HTLC.

But in case lock-size griefing becomes a problem, a robust defense that would also apply to other PCNs is to simply charge a small nonrefundable fee even in the case of timeout (for example, by refunding a small portion of the payment between U_i and U_{i+1} to U_{i+1} rather than U_i). This does allow an intermediary to pocket this small fee and not forward the payment, weakening balance security somewhat, but setting this fee smaller than the fee for actually forwarding the payment would likely make it unprofitable for intermediaries to do so.

Finally, an important point to note is that because Astrape messages are unauthenticated, *non-local* anti-griefing mitigations that attempt to punish griefers network-wide rather at the lock-script level are unlikely to work. In particular, it may be tempting to impose a network-wide penalty for senders who send inconsistent data that force intermediaries to generate expensive inconsistency witnesses, say through a reputation-based blacklist or destroying some staking bond. But in fact inconsistency witnesses cannot be tied to a specific sender who triggered them — there might not even be a "guilty" sender, since inconsistency might come from corrupted transmission (which as we argued in the security analysis cannot lead to deanonymization), and in any case the identity of the sender is hidden by $\mathcal{F}_{\text{anon}}$.

Fortunately, globally punishing nodes that send inconsistent messages is completely unnecessary for Astrape's security. As we discussed earlier, inconsistency witnesses are already fully handled by the normal Unlock and Vf GMHL functions, and do not require a separate recovery path or punishment mechanism to restore balance security. The additional burden that witnesses pose on network resources can also be fully mitigated through the hop-by-hop punishments that we just outlined.

7 Comparison with existing work

In this section, we compare Astrape with existing PCN constructions. First, we compare Astrape's design choices and features with that of other systems, showing that it explores a novel design space. Then, we evaluate Astrape's concrete performance. We compare Astrape's performance with that of other PCN constructions, both anonymous and non-anonymous. Finally, we explore Astrape's performance on a real-world network graph from the Lightning Network.

Table 1: Comparison of different PCNs

	Topology	Anon[a]	Efficient[b]	Crypto
HTLC	Mesh	No	Yes	Sig. + hash
Tumblebit	Hub	Yes	No	Custom RSA
Bolt	Hub	Yes	Yes	NIZKP
Teechain	Hub	Yes	Yes	Trusted comp.
Fulgor/Rayo	Mesh	Yes	No	ZKP
AMHL_{van}[c]	Mesh	Yes	Yes	Homom. OWF
AMHL_{ecd}	Mesh	Yes	Yes	ECDSA, Homom. enc.
AMHL_{sch}	Mesh	Yes	Yes	Schnorr sigs
Blitz	Mesh	Weak[d]	Yes	SA[e]sig. + hash
Astrape	Mesh	Yes	Yes	Sig. + hash

[a] Relationship anonymity
[b] Roughly comparable performance to HTLC. For example, ZKPs requiring many orders of magnitude more computation time than HTLC are not considered "efficient".
[c] AMHL is a family of three closely related constructions. We denote by *van*, *ecd*, *sch* the "vanilla", ECDSA, and Schnorr implementations respectively.
[d] See discussion in Section 2.4.
[e] A "stealth-address" signature scheme; i.e., a signature scheme where any party knowing a public key can generate unlinkably different public keys that correspond to the same private key.

7.1 Design comparison

In Table 1, we compare Astrape's properties with those of existing payment channel networks. We see that except for HTLC, which does not achieve anonymity, all previous PCN networks use cryptographic constructions specialized for their use case. Furthermore, only more recent constructions achieve efficiency comparable to HTLC. It is thus clear that Astrape is the first and only PCN construction that works on all PCN topologies, achieves strong anonymity, and performs at high efficiency, while using the same simple cryptography as HTLC.

7.2 Implementation and benchmark setup

To demonstrate the feasibility and performance of our construction, we developed a prototype implementation in the Go programming language. We implemented

Table 2: Resource usage of different PCN systems (n hops, c-byte HTLC contract, d-byte Astrape contract); AHML variants van,ecd,sch as in Table 1

	Plain HTLC	Fulgor/ Rayo	AMHL	Astrape (Bitcoin Cash)	Astrape
Comput. time (ms)	< 0.001	$\approx 200n$	$\approx n$ (van) $\approx 3n$ (ecd) $\approx 3n$ (sch)	$\approx 0.7n$	$\approx 0.25n$
Comm. size (bytes)	$32n$	$1650000n$	$32 + 96n$ (van) $416 + 128n$ (ecd) $256 + 128n$ (sch)	$192n$	$192n$
Lock (bytes)	$32 + c$	$32 + c$	$32 + c$	$108 + 39 \cdot n$	$64 + d$
Unlock, normal case (bytes)	32	32	32 (van) / 64	32	32
Unlock, worst case (bytes)	32	32	32 (van) / 64	$64 + 128 \cdot n$	$64 + 128 \cdot n$

all the cryptographic constructions of Astrape inside a simulated GMHL model. We used the libsodium library's implementation of the ed25519 [17] signature scheme and blake2b [3] hash function. In addition, we generated script locks in Bitcoin Cash's scripting language to illustrate script sizes for scripting languages with no loops. The Bitcoin Cash scripts, written in the higher-level CashScript language, can be found in Appendix B.

All tests were done on a machine with a 3.2 GHz Intel Core i7 and 16 GB RAM. Network latency is not simulated, as this is highly application dependent. These conditions are designed to be maximally similar to those under which Fulgor [19] and AMHL [20] were evaluated, allowing us to compare the results directly.

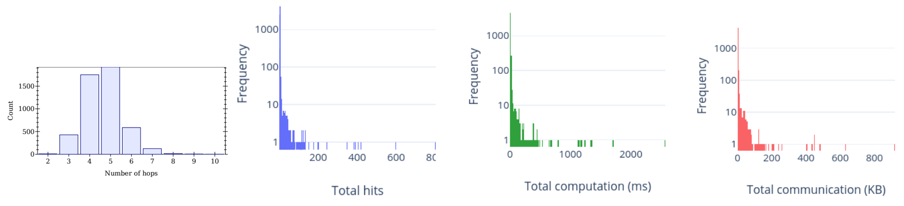
7.3 Resource usage

Our first set of tests compares Astrape's resource usage to that of other PCN constructions. We compare both a simulation of Astrape and a concrete implementation using Bitcoin Cash's scripting language to traditional HTLC, Fulgor/Rayo, and all three variants of AMHL.

We summarize the results in Table 2, where n refers to the number of hops, c to the size of an HTLC contract, and d to the size of an Astrape contract. We copy results for Fulgor/Rayo [19] and AMHL (ECDSA) [20] from their original sources, which use an essentially identical setup.

Computation time. We measure computation time, with communication and other overhead ignored. The time measured is the sum of the CPU time taken by each hop, for all steps of the algorithm. We note that by eschewing non-standard cryptographic primitives, Astrape achieves lower computation times across the board compared to Fulgor/Rayo and AMHL.

Communication overhead. We also measure the communication overhead of each system. This is defined as all the data that needs to be communicated *other than the locks and their opening solutions.* For example, in Astrape, this includes all the setup information sent from U_0, while in AMHL this includes



(a) Shortest paths (b) Hit distribution (c) Computation (d) Communication

Fig. 2: Overhead distribution

everything exchanged during the Setup, Lock, and Rel [20] phases. We see that Astrape has by far the least communication overhead of all the anonymous PCN constructions. Note especially the extreme overhead of the zero-knowledge proofs used in Fulgor/Rayo.

Lock overhead. The last measure is *per-coin* lock overhead — the size of each lock script (the "lock size") and that of the information required to unlock it (the "unlock size"). This is a very important component of a system's resource usage, since lock and unlock sizes directly translate into transaction fees in blockchain cryptocurrencies. Astrape's performance differs in two important ways.

First of all, Astrape's Vf function is expressed in a recursive manner. In blockchains like Bitcoin Cash that support neither recursion nor loops in their scripting language, we must "unroll" the Vf implementation (see Appendix B). This causes lock sizes to be linear in the number of hops. In blockchains with general-purpose scripting languages, though, lock size is generally constant. Second, the worst-case unlock size is larger for Astrape. When the sender is malicious and all coins have to be spent by invoking HashOnion, we need n parameters $(\mathit{\Gamma}_1, \ldots, \mathit{\Gamma}_n)$ to unlock each coin for an n-hop payment. However, despite this asymptotic disadvantage, we believe that Astrape nevertheless offers competitive lock performance. This is because payment routes are quite short in practice, as we will shortly see.

7.4 Privacy vs. overhead

One possible concern with Astrape's performance behavior is that it establishes a tradeoff between privacy and performance. As n increases, we get higher privacy but also higher per-hop overhead, especially for Bitcoin Cash-like blockchains with no support for recursive lock scripts. We next quantify this effect to see whether or not overhead poses a serious barrier to high levels of privacy.

To do so, we calculate the relationship between the fraction of corrupt users and the probability that at least intermediary in n hops would be honest, for varying values of n. This is plotted in Figure 3. We see that by increasing n to 8 we can reduce the chance of compromise to less than 1% even if most of the nodes are corrupt, while still having script sizes less than 10 KB. Although such a script

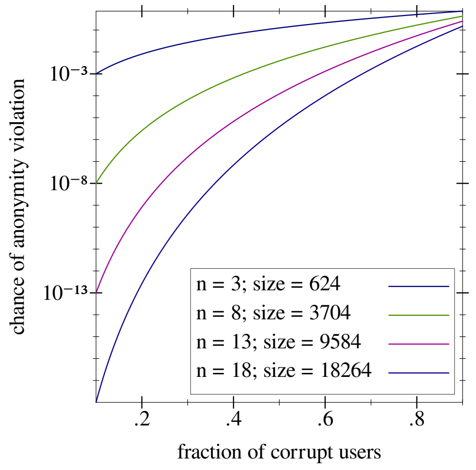


Fig. 3: Privacy at different values for n and Bitcoin Cash lock script sizes

will be hundreds of times the size of an HTLC script, it is still small enough that communicating it across payment channels will not be a large burden.

These script sizes do, however, cause large transaction costs in case transactions must settle on the blockchain. Thus, Astrape is probably not very suitable for on-chain coin-mixing services for blockchains with no recursive lock scripts. Other anonymous multi-hop transaction constructions, like the Fulgor or AMHL, may be more suitable in those situations.

7.5 Statistical simulation

Finally, we simulate the performance of Astrape on the network graph of the Lightning Network (LN).

Setup. We set up a mainnet Lightning Network node using the lnd [1] reference implementation. We then used the lncli describegraph command to capture the network topology of Lightning Network in Feburary 2021. This gives us a graph of 9566 nodes and 72164 edges. Finally, we randomly sample 5,000 pairs of nodes in the network and calculate the shortest paths between them. This gives us a randomly sampled set of real-life payment paths.

Path statistics. As we have previously shown, paths more than 10 hops long still have fairly small overhead even with non-recursive lock scripts, but much longer paths will cause rather large unlock sizes. We examine whether the graph topology will force payments to grow too long; Figure 2a illustrates the distribution of lengths for our 5,000 randomly selected payment paths. On average, a payment path was 5.12 hops long, though the Lightning Network specification allows up to 20 hops. This indicates that shortest payment paths long enough to pose seriously ballooning worst-case overhead are practically nonexistent.

Total scalability. One important attribute we wish to explore with the LN topology is the total scalability of the network — how fast can a PCN process transactions as a whole.

To do so, we keep track of how many times each node appears, or is "hit", in our 5,000 randomly selected payment routes. On average, this is 2.99, but the vast majority of nodes are hit only once, while a few nodes are hit hundreds of times. The distribution of hits is plotted in Figure 2b as a log-linear histogram. We then look at the *distribution of overhead* in the network for both computation and communication. This is by calculating the total computation and communication cost for each node "hit" by the 1,000 random payments, using values from the Bitcoin Cash implementation.

Computation cost is plotted in Figure 2c. We see that the most heavily loaded node in the entire network did around 2,000 ms of computation to process 1,500 transactions. This indicates that the largest hubs in a PCN with the current Lightning Network topology will be able to process around 750 transactions a second per CPU core. Such a throughput is orders of magnitude higher than that of typical blockchains and is within reach of many traditional payment systems. We note that this is only the maximum throughput of a *single CPU core* — in practice hubs likely have multicore machines, and with many hubs the total LN throughput will be many times this number.

Communication cost is plotted in Figure 2d. We pessimistically assume that all payments are settled through HashOnion. Even so, the total network load averages to only about 3.58 KB per node. The largest hubs' total load still do not exceed 1 MB. This illustrates that the bottleneck is actually computation, not communication.

In summary, we see that Astrape's worst-case asymptotic performance poses no barriers to the total throughput of an Astrape-powered payment channel network. PCNs can enjoy the superb scalability associated with them just as easily with Astrape-powered privacy and security.

8 Future work

8.1 Constant-space lock sizes

One interesting problem is whether or not an anonymous multi-hop transaction protocol using only "conventional" cryptography can work with constant lock size. Avoiding worst-case $O(n)$ lock sizes can allow the use of anonymous PCNs in applications where these lock sizes are problematic, such as on-chain coin mixing or bandwidth-constrained environments. However, it seems unlikely that constant space usage is possible with Astrape-type constructions based on multi-hop HLTC and inconsistency witnesses. There does not seem to be a straightforward way of securely committing to rightward (r_i, s_i) parameters with sublinear witnesses.

We wish to investigate as future work whether or not more sophisticated cryptographic commitment mechanisms, such as Merkle trees, can be used to eliminate the linearly growing inconsistency witnesses.

8.2 More robust notions of privacy

Privacy properties not captured by relationship anonymity, such as immunity from timing and other side-channel attacks, are notoriously hard to model.

Unfortunately, anonymity breaks in anonymous communication systems have largely been attacks on these side channels. For instance, the Silk Road 2.0 darknet market was deanonymized largely from timing attacks [22]. In fact, for payment channel networks these side channels probably matter even more. Payment volume is a small fraction of anonymous communication volume, making "innocent" transactions that an adversary can confuse targeted ones for rare.

We urgently need a notion of payment channel privacy that captures the "messy" concepts of time information and distinguishability from other payments across the whole network — not just distinguishability between two payments of the same length, time, etc., intersecting at the same honest intermediary. More research into this area could lead to a much more robustly anonymous payment channel network that can truly replace less scalable alternatives such as physical cash or on-chain payments with zero-knowledge-proof based cryptocurrencies like Zcash.

9 Conclusion

First-generation payment channel networks and other trust-minimizing intermediarized cryptocurrency payment systems lack strong privacy and security guarantees. Existing research, although solving the privacy and security problems, tend to rely on custom cryptographic primitives that cannot be easily swapped with alternatives based on different computational hardness assumptions.

We presented Astrape, a novel PCN construction that breaks this conundrum. Astrape is the first PCN that achieves relationship anonymity and balance security with only black-box access to generic conventional cryptography. It relies on a general idea of using non-anonymous post-hoc inconsistency witnesses to achieve balance security, while avoiding any information leaks when senders are not corrupt. This allows Astrape to avoid dealing with the zero-knowledge verification used to achieve balance security in existing relationship-anonymous PCNs without sacrificing any anonymity or security properties.

Furthermore, we demonstrate that Astrape is practical to deploy in the real world. Performance is superior on average to existing private PCNs, even on blockchains that are unsuitable for free-form smart contracts. We also showed that Astrape achieves high scalability on a real-world payment channel network graph.

Acknowledgements

We thank the reviewers and Sherman Chow for helping to improve this paper. We thank NSERC for Discovery Grant RGPIN-07014 and Create 498002-2017. This research was undertaken, in part, thanks to funding from the Canada Research Chairs program.

References

1. Lightning Network Daemon. https://github.com/lightningnetwork/lnd (2019)
2. Blockchain Charts. https://www.blockchain.com/charts (2021), accessed on 2022-04-01
3. Aumasson, J.P., Neves, S., Wilcox-O'Hearn, Z., Winnerlein, C.: BLAKE2: simpler, smaller, fast as MD5. In: International Conference on Applied Cryptography and Network Security. pp. 119–135. Springer (2013)
4. Aumayr, L., Maffei, M., Ersoy, O., Erwig, A., Faust, S., Riahi, S., Hostáková, K., Moreno-Sanchez, P.: Bitcoin-compatible virtual channels. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 901–918. IEEE (2021)
5. Aumayr, L., Monero-Sanchez, P., Maffei, M.: Blitz: Secure Multi-Hop Payments Without Two-Phase Commits. In: 30th USENIX Security Symposium (2021)
6. Backes, M., Kate, A., Manoharan, P., Meiser, S., Mohammadi, E.: Anoa: A framework for analyzing anonymous communication protocols. In: 2013 IEEE 26th Computer Security Foundations Symposium. pp. 163–178. IEEE (2013)
7. Camenisch, J., Drijvers, M., Gagliardoni, T., Lehmann, A., Neven, G.: The wonderful world of global random oracles. In: IACR EuroCrypt. pp. 280–312. Springer (2018)
8. Croman, K., Decker, C., Eyal, I., Gencer, A.E., Juels, A., Kosba, A., Miller, A., Saxena, P., Shi, E., Sirer, E.G., et al.: On scaling decentralized blockchains. In: International Conference on Financial Cryptography and Data Security. pp. 106–125. Springer (2016)
9. Danezis, G., Goldberg, I.: Sphinx: A compact and provably secure mix format. In: 30th IEEE Symposium on Security and Privacy. pp. 269–282. IEEE (2009)
10. Decker, C., Russell, R., Osuntokun, O.: eltoo: A Simple Layer2 Protocol for Bitcoin. https://blockstream.com/eltoo.pdf (2018)
11. Decker, C., Wattenhofer, R.: A fast and scalable payment network with bitcoin duplex micropayment channels. In: Symposium on Self-Stabilizing Systems. pp. 3–18. Springer (2015)
12. Dong, Y., Goldberg, I., Gorbunov, S., Boutaba, R.: Astrape: Anonymous Payment Channels with Boring Cryptography. In: International Conference on Applied Cryptography and Network Security (ACNS) (2022)
13. Engelmann, F., Kopp, H., Kargl, F., Glaser, F., Weinhardt, C.: Towards an economic analysis of routing in payment channel networks. In: Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers. pp. 1–6 (2017)
14. Feigenbaum, J., Johnson, A., Syverson, P.: Preventing active timing attacks in low-latency anonymous communication. In: International Symposium on Privacy Enhancing Technologies Symposium. pp. 166–183. Springer (2010)
15. Green, M., Miers, I.: Bolt: Anonymous payment channels for decentralized currencies. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 473–489. ACM (2017)
16. Heilman, E., Alshenibr, L., Baldimtsi, F., Scafuro, A., Goldberg, S.: Tumblebit: An untrusted bitcoin-compatible anonymous payment hub. In: Network and Distributed System Security Symposium (2017)
17. Josefsson, S., Liusvaara, I.: Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032 (Jan 2017). https://doi.org/10.17487/RFC8032, https://rfc-editor.org/rfc/rfc8032.txt
18. Lai, R.W.F., Cheung, H.K.F., Chow, S.S.M., So, A.M.C.: Another look at anonymous communication. In: Phan, R.C.W., Yung, M. (eds.) Paradigms in Cryptology – Mycrypt 2016. Malicious and Exploratory Cryptology. pp. 56–82. Springer International Publishing, Cham (2017)

19. Malavolta, G., Moreno-Sanchez, P., Kate, A., Maffei, M., Ravi, S.: Concurrency and privacy with payment-channel networks. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 455–471. ACM (2017)
20. Malavolta, G., Moreno-Sanchez, P., Schneidewind, C., Kate, A., Maffei, M.: Anonymous multi-hop locks for blockchain scalability and interoperability. In: NDSS (2019)
21. McCorry, P., Möser, M., Shahandasti, S.F., Hao, F.: Towards bitcoin payment networks. In: Australasian Conference on Information Security and Privacy. pp. 57–76. Springer (2016)
22. Murdoch, S.J.: How Tor's privacy was (momentarily) broken, and the questions it raises (2015), https://theconversation.com/how-tors-privacy-was-momentarily-broken-and-the-questions-it-raises-52048, accessed on 2022-04-01
23. Pfitzmann, A., Hansen, M.: A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management (2010)
24. Tairi, E., Moreno-Sanchez, P., Maffei, M.: A2L: Anonymous atomic locks for scalability and interoperability in payment channel hubs. In: 42nd IEEE Symposium on Security and Privacy (2021)
25. Van Wirdum, A.: How the Lightning Network Layers Privacy on Top of Bitcoin. https://bitcoinmagazine.com/articles/how-the-lightning-network-layers-privacy-on-top-of-bitcoin-1482183775 (2016), accessed on 2022-04-01
26. Yousaf, H., Kappos, G., Piotrowska, A., Kanjalkar, S., Delgado-Segura, S., Miller, A., Meiklejohn, S.: An empirical analysis of privacy in the lightning network (2021)

A HTLC and XorCake in the GMHL framework

In this appendix, we cast two PCN constructions that do not meet our security goals — HTLC (Figure 4) and XorCake (Figure 5) — into the GMHL framework, to illustrate the framework's generality.

We see that in HTLC, there is a common identifier s shared between all the participants — this is why HTLC lacks meaningful privacy. In XorCake, the "bad state" case causes Unlock to abort and not return a valid key for U_i's left lock, and this is why XorCake lacks balance security.

```
function $\mathsf{Init}_{U_0}^{\mathrm{HTLC}}(1^\lambda, U_1, \ldots, U_n)$
Upon invocation by $U_0$:
    generate a $\lambda$-bit random number $r$
    $s \leftarrow H(r)$
    for $i$ in $1, \ldots, n$ do
        if $i = n$ then
            send $r$ to $U_n$
        else
            send $s$ to $U_i$

function $\mathsf{Create}_{U_i}^{\mathrm{HTLC}}(s_i^I = s)$
Upon invocation by $U_i$, where $i < n$:
    return $\ell_{i+1} = \mathsf{HTLC}[s]$

function $\mathsf{Unlock}_{U_i}^{\mathrm{HTLC}}(\ell_{i+1}, s_i^I = s, k_{i+1})$
Upon invocation by $U_i$, where $i < n$:
    return $k_{i+1}$

function $\mathsf{Vf}^{\mathrm{HTLC}}(\ell, k)$
    parse $\ell$ as $\mathsf{HTLC}[s]$
    if $H(k) = s$ then
        return 1
    else
        return 0
```

Fig. 4: HTLC in the GMHL framework

```
function $\mathsf{Init}^{\mathrm{XC}}_{U_0}(1^\lambda, U_1, \ldots, U_n)$
Upon invocation by $U_0$:
    generate $\lambda$-bit random numbers $\{r_1, \ldots, r_n\}$
    for $i$ in $1, \ldots, n$ do
        $s_i \leftarrow H(r_i \oplus r_{i+1} \oplus \cdots \oplus r_n)$
        if $i = n$ then
            send $(k_n = r_n, s_n)$ to $U_n$
        else
            send $(r_i, s_i, s_{i+1})$ to $U_i$

function $\mathsf{Create}^{\mathrm{XC}}_{U_i}(s^I_i = (r_i, s_i, s_{i+1}))$
Upon invocation by $U_i$, where $i < n$:
    return $\ell_{i+1} = \mathsf{HTLC}[s_{i+1}]$

function $\mathsf{Unlock}^{\mathrm{XC}}_{U_i}(\ell_{i+1}, s^I_i = (r_i, s_i, s_{i+1}), k_{i+1})$
Upon invocation by $U_i$, where $i < n$:
    if $\mathsf{Vf}^{\mathrm{XC}}(\ell_i, r_i \oplus k_{i+1}) = 0$ then
        abort                                              ▷ bad state
    $k_i \leftarrow r_i \oplus k_{i+1}$
    return $k_i$

function $\mathsf{Vf}^{\mathrm{XC}}(\ell, k)$
    parse $\ell$ as $\mathsf{HTLC}[s]$
    if $H(k) = s$ then
        return 1
    else
        return 0
```

Fig. 5: XorCake in the GMHL framework

B CashScript implementation of Astrape

```
pragma cashscript ^0.6.0;

contract Astrape(pubkey sender,
                 pubkey recipient,
                  bytes32 X,
                  bytes32 S,
                  int timeout) {

  // The xorcake case
function unlock_hsoln(sig recipientSig,
                      bytes32 Kappa) {
```

```
  require(checkSig(recipientSig, recipient));
  // HTLC
  require(hash256(Kappa) == S);
}

// The hashonion case
function unlock_WSoln(bytes32 Kappa,
                      bytes32 Xi,
                      bytes Gamma) {
  require(Gamma.length % 128 == 0 && Gamma.length > 0);
  // get the last element of Gamma
  bytes Gamma_j = Gamma.split(Gamma.length - 32)[1];
  bytes r_j = Gamma_j.split(32)[0];
  bytes s_j = Gamma_j.split(32)[1].split(32)[0];
  // check that it's a valid witness!
  require(hash256(r_j + Kappa) != s_j);
  // construct the answer
  int index = Gamma.length - 128;

  bytes32 accum = hash256(Gamma_j + Xi);

  // **** "unrolled loop" ****
  // Unrolled n times for n remaining hops
  // n = 3 in this instance

   if (index >= 128) {
      bytes Gamma_i =
         Gamma.split(index)[1].split(128)[0];
      accum = hash256(Gamma_i + accum);
      index = index - 128;
   }

   if (index >= 128) {
      bytes Gamma_i =
         Gamma.split(index)[1].split(128)[0];
      accum = hash256(Gamma_i + accum);
      index = index - 128;
   }

   if (index >= 128) {
      bytes Gamma_i =
         Gamma.split(index)[1].split(128)[0];
      accum = hash256(Gamma_i + accum);
      index = index - 128;
   }

  // **** end of unrolled loop ****
  require(index == 0 && accum == X);
}

// Require timeout time to be reached
// and sender’s signature to match
function timeout(sig senderSig) {
  require(checkSig(senderSig, sender));
  require(tx.time >= timeout);
}
}
```