Boomerang: Redundancy Improves Latency and Throughput in Payment-Channel Networks

Vivek Bagaria*, Joachim Neu*, and David Tse

Stanford University, Stanford CA 94305, USA

Abstract. In multi-path routing schemes for payment-channel networks, Alice transfers funds to Bob by splitting them into partial payments and routing them along multiple paths. Undisclosed channel balances and mismatched transaction fees cause delays and failures on some payment paths. For atomic transfer schemes, these straggling paths stall the whole transfer. We show that the latency of transfers reduces when redundant payment paths are added. This frees up liquidity in payment channels and hence increases the throughput of the network. We devise *Boomerang*, a generic technique to be used on top of multi-path routing schemes to construct redundant payment paths free of counterparty risk. In our experiments, applying Boomerang to a baseline routing scheme leads to 40 % latency reduction and 2x throughput increase. We build on ideas from publicly verifiable secret sharing, such that Alice learns a secret of Bob iff Bob overdraws funds from the redundant paths. Funds are forwarded using Boomerang contracts, which allow Alice to revert the transfer iff she has learned Bob's secret. We implement the Boomerang contract in Bitcoin Script.

Keywords: Payment-channel networks · Redundancy · Atomic multi-path · Routing · Throughput · Latency · Adaptor signatures

1 Introduction

1.1 Payment Channels and Networks

Blockchains provide a method for maintaining a distributed ledger in a decentralized and trustless fashion [24]. However, these so called layer-1 (L1) consensus mechanisms suffer from low throughput and high confirmation latency. From a scaling perspective it would be desirable if transactions could be processed 'locally' by only the involved participants.

Payment channels (PCs) [9,29,8] provide this. Once two participants have established a PC on-chain, they can transact through the channel off-chain without involving L1 every time. Briefly, the mechanism underlying the different PC implementations is as follows. The participants escrow a pool of funds on-chain which they can spend only jointly. They can perform a transfer through the PC by agreeing on an updated split of the shared funds which reflects the new

* Contributed equally and listed alphabetically.

balances after the transfer and can be enforced on-chain anytime. Special care is taken that participants can only ever execute the most recent agreement on-chain. PCs improve the throughput, latency and privacy of a blockchain system.

Payment networks (PNs) [29,23,11,18,12] can be constructed on top of PCs. PCs can be linked to establish a path between a source and a destination via some intermediaries. Hash- and time-locked contracts (HTLC) are used to perform transfers via intermediaries without counterparty risk, giving rise to so called layer-2 (L2) PNs. To this end, the destination draws a secret (preimage) and reveals a one-way function of the secret (preimage challenge) to the source. The source then initiates a chain of payments to the destination, all conditional on the revelation of a valid preimage. The destination reveals the secret to claim the funds, setting the chain of payments in motion. Net, the destination is paid, the source pays, and the intermediaries are in balance, up to a small service fee earned for forwarding. For recent surveys on PCs and PNs, see [17,15].

1.2 Routing in Payment Networks

Routing algorithms for PNs find paths and forward funds while optimizing objectives such as throughput, latency, or transaction fees. Recently, the routing problem has been studied extensively and various routing algorithms have been proposed [29,30,21,31,16,34,10,35]. Early on, it has been discovered, also through the Lightning Torch experiment [1,2], that single-path routing restricts the maximum transfer size. Multi-path routing [27] allows for a more flexible use of PC liquidity and hence can accommodate larger transfers by splitting transfers into partial payments that are routed along multiple paths. However, while single-path payments are naturally *atomic*, *i.e.*, they either succeed or fail entirely, simply sending multiple partial payments can lead to half-way incomplete transfers. Atomicity is important to keep payment networks manageable from a systems-design perspective, and is achieved by atomic multi-path payments (AMP, [25]), which most state-of-the-art routing algorithms rely on and Lightning developers are actively working towards [3]. Yet, because of its ‘everyone-waits-for-the-last’ philosophy, atomicity comes at a cost for latency and throughput.

1.3 Main Contributions

Due to the stochastic nature of PNs, *i.e.*, random delays and failures of payment paths, AMP routing often idles while waiting for a few straggling paths. This leads to a long time-to-completion (TTC) of transfers. Furthermore, already successful partial payments are kept pending, seizing liquidity from the PN that could have served other transfers. As a result, the throughput of the PN reduces. For example, consider a transfer from Alice to Bob of \$4 via 4 paths, of which 3 succeed after $1\,\mathrm{s}$ and 1 succeeds after $4\,\mathrm{s}$. The transfer has a TTC of $4\,\mathrm{s}$ and consumes liquidity $\$4 \cdot 4\,\mathrm{s} = \$16 \cdot \mathrm{s}$.

Now suppose instead Alice could send 8 partial payments of \$1 each (4 ‘extra’ redundant paths), such that AMP completes once a quorum of 4 out of 8 paths succeeds and Bob cannot steal any extra funds. If 6 paths succeed after $1\,\mathrm{s}$ and

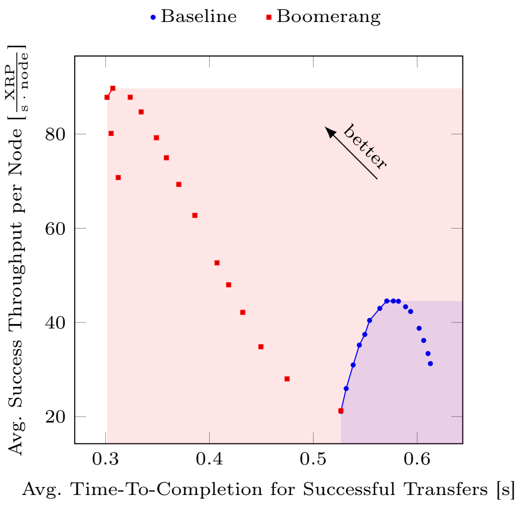


Fig. 1: The blue (resp. red) points mark the tradeoff between latency and throughput of the baseline (resp. Boomerang) AMP routing scheme, obtained by varying an internal parameter. The Pareto fronts (lines) and the achievable regions (shaded) of the tradeoff are shown. Boomerang yields a 2x increase in throughput and a 40 % decrease in latency over the baseline scheme.

2 succeed after 4 s, then the transfer has a TTC of 1 s and consumes liquidity $4 \cdot \$1 \cdot 1\,\mathrm{s} + 2 \cdot \$1 \cdot 1\,\mathrm{s} + 2 \cdot \$1 \cdot 4\,\mathrm{s} = \$14 \cdot \mathrm{s}$. Thus, the use of redundant payments reduces the TTC of AMP transfers. As a result, less liquidity is consumed and the PN achieves higher throughput. Similar observations about straggler mitigation using redundancy have been made in large-scale distributed computing [7,19,4].

We devise *Boomerang*, a technique to be used on top of multi-path routing schemes to construct redundant payment paths free of counterparty risk. Building on ideas from publicly verifiable secret sharing, we use a homomorphic one-way function to intertwine the preimage challenges used for HTLC-type payment forwarding, such that Alice learns a secret of Bob iff Bob overdraws funds from the redundant paths. Funds are forwarded using Boomerang contracts, which allow Alice to revert the transfer iff she has learned Bob's secret. We prove the Boomerang construction to be secure, and present an implementation in Bitcoin Script which applies either adaptor signatures based on Schnorr signatures, or elliptic curve scalar multiplication as a one-way function.

We empirically verify the benefits of Boomerang for throughput and latency of AMP routing. For this purpose, we choose a baseline routing scheme which resembles the scheme currently used in Lightning. We enhance this scheme with Boomerang. As can be seen in Fig. 1, redundancy increases the throughput by 2x and reduces the TTC (latency) by 40 %. Our results suggest that redundancy is a generic tool to boost the performance of AMP routing algorithms.

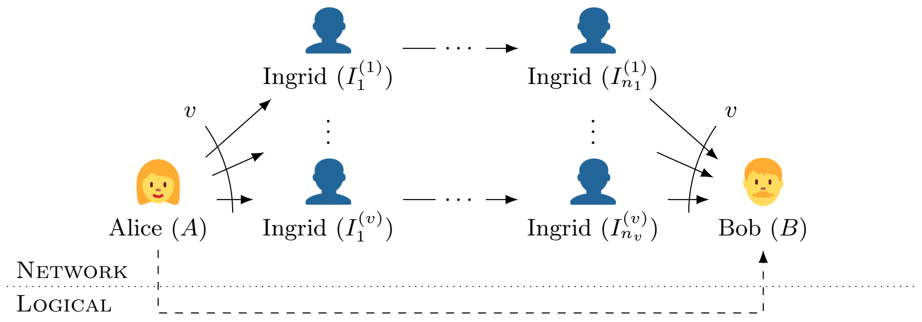


Fig. 2: The TF from A to B (dashed) is implemented by v TXs via intermediaries $(I_1^{(1)}, \dots, I_{n_1}^{(1)})$, ..., $(I_1^{(v)}, \dots, I_{n_v}^{(v)})$ (solid).

1.4 Paper Outline

In Section 2, we recall cryptographic preliminaries, introduce the system model, and summarize causes for delay and failure of transfers in payment networks. We devise the Boomerang construction for redundant transactions in Section 3, and prove it to be secure. An implementation of Boomerang in Bitcoin Script is presented in Section 4. In Section 5, we demonstrate the utility of redundancy for improving routing protocols in experiments.

2 Preliminaries

2.1 System Model & Terminology

The topology of a payment network (PN) is given by an undirected graph whose nodes are agents, and whose edges are payment channels (PCs). Each PC endpoint owns a share of the funds in the PC, which we refer to as its liquidity or balance. For privacy reasons it is undesirable to reveal balances to third party nodes. Following Lightning [29, BOLT #7], we assume that nodes have no information about PC balances when taking routing decisions, but know the PN topology. Finally, we assume that PN nodes communicate in a peer-to-peer (P2P) gossiping fashion only along PCs, *i.e.*, PN topology is P2P network topology.

For multi-path routing, we highlight a strict separation between two layers in our terminology (cf. Fig. 2). The 'logical' layer is concerned with *transfers* (TFs) of an amount v of funds from a *source* (Alice, A) to a *destination* (Bob, B). The 'network' layer implements a TF through multiple *transactions* (TXs) from the *sender* to the *receiver* along different paths through multiple intermediaries (Ingrid, I_i). A preimage p_i is used for HTLC-type forwarding of the i-th TX. The amounts of the TXs add up to the amount of the TF. Routing algorithms schedule a sequence of TXs in an attempt to implement a given TF. Figures of merit are *throughput*, *i.e.*, the average amount of funds transported successfully per time, and *time-to-completion* (TTC), *i.e.*, the average delay between commencement and completion of a TF.

2.2 Delay and Failure of Transactions in Payment Networks

There are various causes for delay and failure of TXs in PNs: • The liquidity of a PC along the desired path is insufficient. • Insufficient fees do not incentivize intermediaries to forward. • Queuing, propagation and processing delay of P2P messages. • PN topology changes, nodes come and go, *e.g.*, due to connectivity or maintenance. • Governments or businesses attempt to censor certain TFs.

2.3 Cryptographic Preliminaries

In this section we briefly recapitulate the cryptographic tools used throughout the paper. Let $\mathbb{G}$ be a cyclic multiplicative group of prime order q with a generator $g \in \mathbb{G}$. We assume that the discrete logarithm problem (DLP, formally introduced in Section A) is hard for g in $\mathbb{G}$, which is commonly assumed to be the case, *e.g.*, in certain elliptic curves (ECs) used in Bitcoin.

Let $H\colon \mathbb{Z}_q \to \mathbb{G}$ with $H(x) \triangleq g^x$, where $\mathbb{Z}_q$ is the finite field of integers modulo q. We require H to be a one-way function, which follows from the DLP hardness assumption. Given a *preimage challenge* $h \triangleq H(x)$, it is difficult to obtain the *preimage* x, but easy to check whether a purported preimage $\hat{x}$ satisfies the challenge (*i.e.*, $H(\hat{x}) = h$).

H has the following homomorphic property which we make extensive use of:

$$\forall n \geq 1\colon \forall c_1, \ldots, c_n\colon \quad H\left(\sum_{i=1}^{n} c_i x_i\right) = \prod_{i=1}^{n} H(x_i)^{c_i} \tag{1}$$

3 The Boomerang Construction

In this section, we show how to add redundant TXs without counterparty risk. To this end, we build up the Boomerang construction in three steps. First, B draws a secret α_0. We use the homomorphic property of H and ideas from publicly verifiable secret sharing [33,5,14,26,32] to construct the preimages p_i and the preimage challenges $H(p_i)$ such that A learns α_0 iff B overdraws funds from the redundant paths. Second, the so called Boomerang contract serves as a building block for contingent transfer of funds, *i.e.*, HTLC-type forwarding with the additional provision that A can revert the TF iff she learns α_0. Finally, the end-to-end procedure of a Boomerang TF is devised from the previously mentioned ingredients. Subsequently, we prove that Boomerang satisfies the relevant notions of security.

3.1 Setup of Preimage Challenges

We assume that A and B have agreed out-of-band to partition their TF into v TXs of a unit of funds (\$1) each, without loss of generality (w.l.o.g.). In addition, they use u redundant TXs to improve their TF, so that the total number of TXs

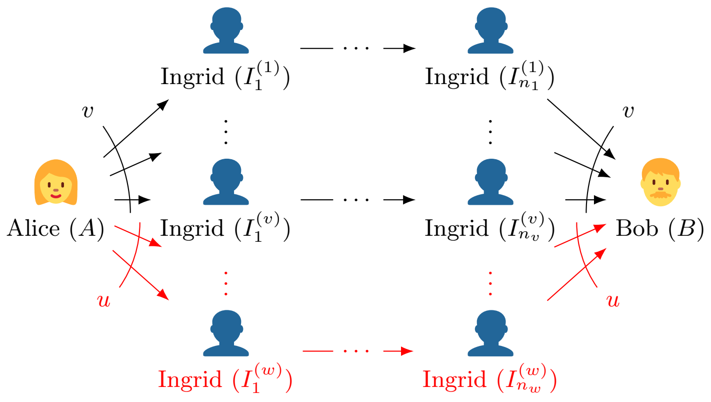


Fig. 3: The TF of amount v from A to B is implemented by the first v out of $w \triangleq v+u$ TXs of unit amount \$1 via $(I_1^{(1)}, \ldots, I_{n_1}^{(1)})$, ..., $(I_1^{(w)}, \ldots, I_{n_w}^{(w)})$.

is $w \triangleq v+u$ (cf. Fig. 3). First, B chooses a polynomial $P(x)$ of degree $\deg(P) = v$ with coefficients $\alpha_0 \overset{\mathrm{R}}{\leftarrow} \mathbb{Z}_q, \ldots, \alpha_v \overset{\mathrm{R}}{\leftarrow} \mathbb{Z}_q$ drawn uniformly at random,

$$P(x) = \sum_{j=0}^{v} \alpha_j x^j. \tag{2}$$

Then, B commits to $P(x)$ by providing A with $H(\alpha_0), \ldots, H(\alpha_v)$. Due to the homomorphic property of H, eq. (1), A can compute

$$\forall i \in \{1, \ldots, v, \ldots, w\}: \quad H(P(i)) = H\left(\sum_{j=0}^{v} \alpha_j i^j\right) = \prod_{j=0}^{v} H(\alpha_j)^{(i^j)}. \tag{3}$$

For the i-th TX, $p_i \triangleq P(i)$ is used as a preimage for HTLC-type forwarding. Hence, A uses $H(p_i) = H(P(i))$ from eq. (3) as preimage challenge, and informs B out-of-band of which i was used.

To redeem the i-th TX, B reveals $p_i = P(i)$. Should B overdraw by revealing more than v evaluations p_i of $P(x)$, then $\alpha_0, \ldots, \alpha_v$ can be recovered using polynomial interpolation due to $\deg(P) = v$. Recall that α_0 serves as a secret which A can use to revert the TF in this case. As long as B reveals no more than v evaluations of $P(x)$, each α_i remains marginally uniformly distributed. In this case, the TF is final.

Note that A and B do not have to agree on u ahead of time. Instead, A can create virtually infinitely many $H(p_i)$ (as long as $w < q$), and hence send a continuous flow of TXs until B redeems v TXs. This property is similar to rateless codes used to implement ‘digital fountains’ [6], and enables B to choose u adaptively during execution of the TF. Furthermore, since there is no risk in ‘losing control’ over a redundant TX, source routing can be abandoned in favor of the PN taking distributed routing decisions. Conceptually, the use of

$$\begin{tikzcd} \text{Party 1 } (P_1) \arrow[r, bend left, dashed, "\$1+\delta : \hat{p}_i \text{ s.t. } H(\hat{p}_i)=H(p_i)"] & \text{Party 2 } (P_2) \arrow[l, bend left, dashed, "\begin{array}{c}\$1 : \hat{p}_i \text{ s.t. } H(\hat{p}_i)=H(p_i)\\ \hat{\alpha}_0 \text{ s.t. } H(\hat{\alpha}_0)=H(\alpha_0)\end{array}"] \end{tikzcd}$$

Fig. 4: Boomerang contract used to forward \$1 plus δ TX fees from P_1 to P_2

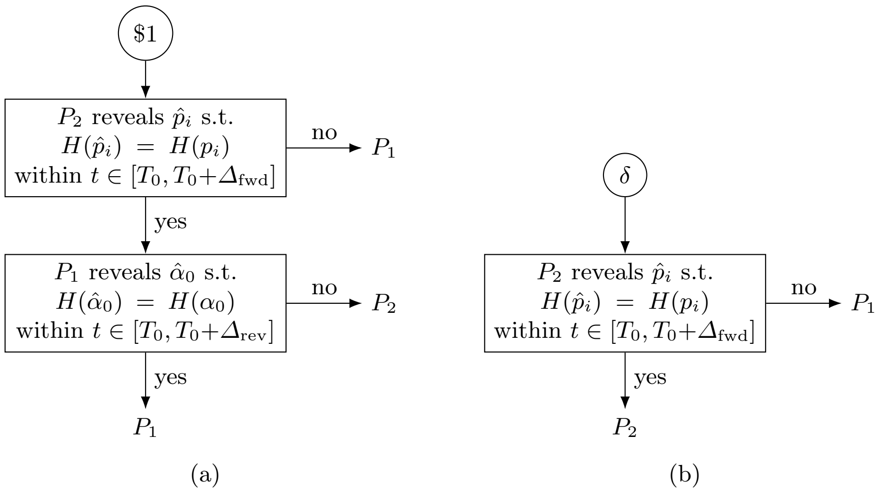
redundant TXs in PNs is analogous to the use of erasure-correcting codes [13] in communication networks consisting of packet erasure channels [20] and for straggler mitigation in large-scale distributed computing and storage [7,19].

3.2 Boomerang Contract

We devise the Boomerang contract, which implements reversible HTLC-type forwarding as required for Boomerang on top of a PC. Still, w.l.o.g. \$1 funds and δ TX fee are to be forwarded from party P_1 to party P_2.

Conceptually, the desired behavior could be accomplished by two conditional forwardings (cf. Fig. 4), the first of which ('forward', top arrow in Fig. 4) transfers $\$1+\delta$ from P_1 to P_2 upon revelation of a preimage $\hat{p}_i$ such that $H(\hat{p}_i) = H(p_i)$, and the second of which ('reverse', bottom arrow in Fig. 4) transfers \$1 back from P_2 to P_1 upon revelation of two preimages $\hat{p}_i$ and $\hat{\alpha}_0$ such that $H(\hat{p}_i) = H(p_i)$ and $H(\hat{\alpha}_0) = H(\alpha_0)$. Note that the two conditions are nested such that if P_1 redeems the second forwarding, then P_2 can redeem the first forwarding. As a result, the Boomerang contract has three possible outcomes. Either, *a)* neither forwarding is redeemed and P_1 retains all funds (*e.g.*, in the case of a timeout or an unused redundant TX), or *b)* only the 'forward' forwarding is redeemed (*i.e.*, B draws funds by revealing $P(i)$ but does not leak α_0), or *c)* both forwardings are redeemed (*i.e.*, B overdraws and reveals both $P(i)$ and α_0). Anyway, P_2 cannot loose funds and thus agrees to deploy the contract on the PC.

Note that Fig. 4 has been simplified for ease of exposition. Indeed, two essential aspects are not captured in Fig. 4 and necessitate the refined final specification of the Boomerang contract in Fig. 5. First, timeouts Δ_{fwd} and Δ_{rev} (relative to current time T_0) need to be chosen such that the source of the TF has time to detect overdraw and reclaim the funds ($\Delta_{\text{fwd}} < \Delta_{\text{rev}}$). Second, having two separate forwardings would requisition twice the liquidity ($\$2+\delta$) while the forwardings are pending. Instead, the Boomerang contract as specified by the flow charts in Fig. 5 allows for consistent timeouts with $\Delta_{\text{fwd}} < \Delta_{\text{rev}}$ and requisitions the TX amount in PC liquidity only once ($\$1+\delta$). We present an implementation of Fig. 5 in Bitcoin Script on top of Eltoo PCs [8] in Section 4. Throughout the paper, we use two circular arrows as in Fig. 4 to visualize a Boomerang contract.



$1
P_2 reveals $\hat{p}_i$ s.t. $H(\hat{p}_i) = H(p_i)$ within $t \in [T_0, T_0+\Delta_{\mathrm{fwd}}]$
no
P_1
yes
P_1 reveals $\hat{\alpha}_0$ s.t. $H(\hat{\alpha}_0) = H(\alpha_0)$ within $t \in [T_0, T_0+\Delta_{\mathrm{rev}}]$
no
P_2
yes
P_1
(a)
δ
P_2 reveals $\hat{p}_i$ s.t. $H(\hat{p}_i) = H(p_i)$ within $t \in [T_0, T_0+\Delta_{\mathrm{fwd}}]$
no
P_1
yes
P_2
(b)

Fig. 5: Flow charts for payout of a Boomerang contract (cf. Fig. 4) concerning (a) \$1 funds and (b) δ TX fees between P_1 and P_2

3.3 End-To-End Procedure for Boomerang Transfers

Given the intertwined preimage challenges of Section 3.1 and the Boomerang contract of Section 3.2, we devise the end-to-end procedure for Boomerang TFs.

To forward a TX along a path from A to B via $I_1, \ldots, I_n$, the timeouts Δ_{fwd} and Δ_{rev} of the Boomerang contracts between A and I_1, I_i and I_{i+1}, and I_n and B need to be chosen in the following way: The forward components' timeouts decrease along the path from A to B. The reverse components' timeouts increase along the path from A to B. Additionally, the earliest reverse component expires later than the latest forward component, to allow A to react to overdraw and activate the reverse components. An example is illustrated in Fig. 6.

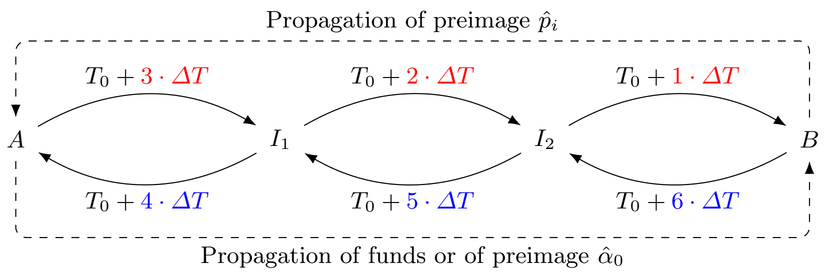


Fig. 6: Staggering of timeouts of Boomerang contracts: The Δ_{fwd} are chosen such that propagation of the preimage $\hat{p}_i$ from B via I_2 and I_1 to A is guaranteed once B reveals $\hat{p}_i$. The Δ_{rev} are chosen such that propagation of the preimage $\hat{\alpha}_0$ from A via I_1 and I_2 to B is guaranteed once A reveals $\hat{\alpha}_0$.

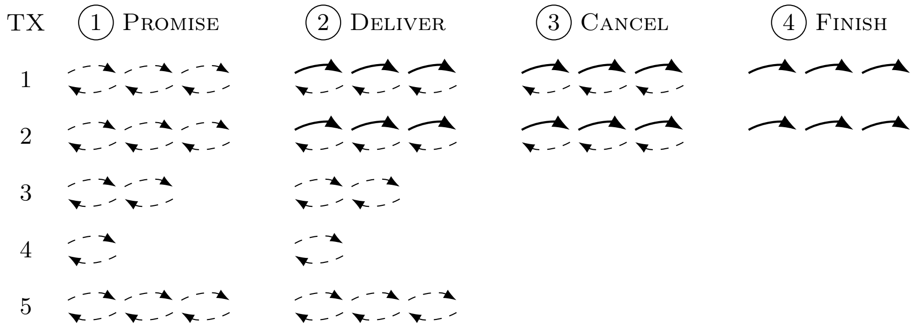


Fig. 7: Stages of a Boomerang TF (here using $w = v + u = 2 + 3$ TXs, each through two I_i): ① A attempts 5 TXs; TXs 1, 2 and 5 reach B, 3 and 4 do not. ② B claims TXs 1 and 2 by revealing $P(1)$ and $P(2)$. ③ The unsuccessful and outstanding TXs 3, 4 and 5 are cancelled upon request from B. ④ A relinquishes the option to retract TXs 1 and 2 as no further funds can be drawn by B.

A Boomerang TF of v units using $w = v + u$ redundant TXs proceeds, after the setup of preimage challenges out-of-band, in four steps (cf. Fig. 7):

① Promise: A attempts $w = v + u$ TXs.
② Deliver: B claims funds from up to v TXs by revealing the preimages $p_i = P(i)$ to the corresponding preimage challenges $H(p_i)$, activating the forward components of the Boomerang contracts along the respective paths.
③ Cancel: Upon a request by B which is passed on to the tip of the path of an unsuccessful or surplus TX, outstanding TXs are cancelled. Note that cancellation is counterparty risk-free if it proceeds along the path from B towards A, and honest as well as rational participants have a self-interest in freeing up liquidity that will foreseeably not earn TX fees.
④ Finish: A renounces the reverse components of the remaining Boomerang contracts to free up liquidity, as there is no more risk of B overdrawing.

3.4 Security Guarantees

We give the following guarantees to the source A and the destination B of a TF, respectively, which formalize security for the outlined Boomerang construction:

Theorem 1 (A-Guarantee). *If more than v of the redundant TXs are drawn from A, then A can recover α_0 and revert all TXs.*

Theorem 2 (B-Guarantee). *As long as B follows the protocol and draws no more than v of the redundant TXs, all TXs are final, except with negligible probability, provided the DLP is hard for $(\mathbb{G}, g)$.*

Corollary 1 (Proofs). *If A knows α_0, this proves that B cheated. If A knows p_i, this proves that B was paid accordingly. A can forge the proofs only with negligible probability, provided the DLP is hard for $(\mathbb{G}, g)$.*

Note that 'drawing' a TX means 'revealing the preimage $p_i = P(i)$ for the challenge $H(p_i)$ of TX i'. Hence, the above guarantees are statements about 'how much preimage information flows' 'into A' and 'out of B', implicitely assuming the worst-case that all intermediary I_i collude with the respective opposing party.

Proof (of Theorem 1: A-Guarantee). Due to the homomorphic property of H, and the fact that A computes the challenges $H(p_i)$ from the commitments $H(\alpha_0), \ldots, H(\alpha_v)$, there exists a unique polynomial of degree v that passes through all $P(i)$. This polynomial can be determined by interpolation, if more than v preimages $p_i = P(i)$ are revealed. Hence, in the case of overdrawing, A can be certain to obtain an α_0, such that A can activate the reverse components of the Boomerang contracts and revert all TXs. □

Proof (of Theorem 2: B-Guarantee). It suffices to show that one cannot recover α_0 from $H(\alpha_0)$, ..., $H(\alpha_v)$, $P(i_1)$, ..., $P(i_v)$, with $i_1, \ldots, i_v$ distinct, except with negligible probability, provided that the DLP is hard for $(\mathbb{G}, g)$.

W.l.o.g., assume that all preimages revealed by B have been forwarded to A. Call the task A faces *Alice's problem* (AP), *i.e.*, given an AP instance $(g^{\alpha_0}, \ldots, g^{\alpha_v}, P(i_1), \ldots, P(i_v))$ for known distinct $i_1, \ldots, i_v$, find α_0. We show that AP is computationally infeasible by reducing the DLP to AP.

The reduction proceeds as follows. Assume $\mathcal{B}$ is a blackbox that solves AP instances efficiently. Using $\mathcal{B}$, we construct a DLP solver $\mathcal{A}$. Present $\mathcal{A}$ with a DLP instance (g, g^a). $\mathcal{A}$ samples $P(i_j) \overset{\mathrm{R}}{\leftarrow} \mathbb{Z}_q$ for $j = 1, \ldots, v$ uniformly at random, such that there exists a unique degree v polynomial interpolating $P(0), P(i_1), \ldots, P(i_v)$, with coefficients $\alpha_0, \ldots, \alpha_v$ chosen uniformly at random. Recall that $P(0) = \alpha_0 = a$ is unknown. By eq. (2),

$$\underbrace{\begin{bmatrix} P(0) \\ P(i_1) \\ \vdots \\ P(i_v) \end{bmatrix}}_{\triangleq \boldsymbol{p}} = \underbrace{\begin{bmatrix} 0^0 & 0^1 & \ldots & 0^v \\ i_1^0 & i_1^1 & \ldots & i_1^v \\ \vdots & \vdots & & \vdots \\ i_v^0 & i_v^1 & \ldots & i_v^v \end{bmatrix}}_{\triangleq \boldsymbol{M}} \underbrace{\begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_v \end{bmatrix}}_{\triangleq \boldsymbol{\alpha}}, \tag{4}$$

where the entries of $\boldsymbol{M}$ are fixed.

As $\boldsymbol{M}$ is a Vandermonde matrix and thus invertible,

$$\boldsymbol{p} = \boldsymbol{M}\boldsymbol{\alpha} \quad \Longleftrightarrow \quad \boldsymbol{\alpha} = \boldsymbol{M}^{-1}\boldsymbol{p}. \tag{5}$$

Let $\tilde{m}_{ij} \triangleq \left\{\boldsymbol{M}^{-1}\right\}_{ij}$ and $p_j \triangleq \{\boldsymbol{p}\}_j$, where $\{X\}_y$ is the y-th entry of X. Then,

$$\alpha_i = \sum_{j=0}^{v} \tilde{m}_{ij} p_j. \tag{6}$$

Hence, using the homomorphic property of H from eq. (1), obtain

$$\forall i \in \{0, \ldots, v\}\colon \quad H(\alpha_i) = H\left(\sum_{j=0}^{v} \tilde{m}_{ij} p_j\right) = \prod_{j=0}^{v} H(p_j)^{\tilde{m}_{ij}}. \tag{7}$$

Now, $\mathcal{A}$ has a tuple $(H(\alpha_0), \ldots, H(\alpha_v), P(i_1), \ldots, P(i_v))$ drawn from the distribution of AP instances implied by the Boomerang protocol. $\mathcal{A}$ invokes the AP oracle $\mathcal{B}$ and obtains $\alpha_0 = a$, which solves the DLP instance.

Thus, an efficient solution to AP implies an efficient solution to DLP. But since DLP is assumed to be hard, AP has to be hard, proving the claim. $\square$

Proof (of Corollary 1: Proofs). Theorems 1 and 2 imply that A knows α_0 iff B cheated (except with negligible probability, provided the DLP is hard for $(\mathbb{G}, g)$).

For p_i to serve as proof of payment, it requires that for any $n \leq v$ and known distinct $i_1, \ldots, i_{n+1}$, A cannot obtain $P(i_{n+1})$ from $H(\alpha_0)$, ..., $H(\alpha_v)$, $P(i_1)$, ..., $P(i_n)$ (except with negligible probability, provided the DLP is hard for $(\mathbb{G}, g)$). For $n = v$, this follows from Theorem 2 by contradiction, as α_0 can be computed from $P(i_1), \ldots, P(i_{v+1})$. For $n < v$, the problem is harder than for $n = v$. $\square$

4 Implementation in Bitcoin Script

We present an implementation of Boomerang in Bitcoin Script and its deployment on an Eltoo PC. For simplicity, we first present an implementation which hinges on a new opcode for $H(x)$. Subsequently, we refine the implementation to apply adaptor signatures based on Schnorr signatures instead.

4.1 Implementation Using an Opcode for $H(x)$

Assume the addition of a new command $\texttt{ECEXP}\langle g \rangle$ to Bitcoin Script for computing $H(x) = g^x \in \mathbb{G}$. The command would pop x off the stack and push g^x back onto the stack. The necessary cryptographic primitives to do this in ECs are already part of Bitcoin and only need to be exposed to the scripting engine. Then, $\texttt{ECEXP}\langle g \rangle$ can be used as a one-way function in similar situations as the `SHA*` or `HASH*` commands, but provides the homomorphic property that can be necessary or useful for applications beyond Boomerang.

We show how the mechanics of the Boomerang contract specified in Fig. 5 can be implemented using Bitcoin Script. To this end, we refer to the settlement transaction of the PC on top of which the Boomerang contract is deployed as $\mathsf{TX}_{\text{settle}}$, and to transactions that P_i would use to pay a certain output to themselves as $\mathsf{TX}_{\text{payout},P_i}$. For the signature scheme employed by Bitcoin, an identity P has a public key pk_P and a private (secret) key sk_P. Using sk_P, a signature $\sigma \triangleq \mathsf{sig}_P(x)$ for string x can be created.

The implementation of Fig. 5(a) is provided in Fig. 8. The first condition of the flow chart in Fig. 5(a), which captures the forward component of the Boomerang contract, is implemented as an output of $\mathsf{TX}_{\text{settle}}$. If this condition is met, the distribution of the funds is decided by an additional transaction $\mathsf{TX}_{\text{retaliate}}$ which implements the reverse component of the Boomerang contract. $\mathsf{TX}_{\text{retaliate}}$ is agreed-upon and signed by both parties using temporary one-time identities $P_{i,\text{tmp}}$ as part of the deployment of a Boomerang contract. The implementation of Fig. 5(b) is provided in Fig. 11 of Section B.

Bitcoin Script implementation of output on $\mathsf{TX}_{\text{settle}}$:

```
1  IF
2      ECEXP⟨$g$⟩  PUSH⟨$H(p_i)$⟩  EQUALVERIFY
3      2  PUSH⟨$\mathsf{pk}_{P_{1,\text{tmp}}}$⟩  PUSH⟨$\mathsf{pk}_{P_{2,\text{tmp}}}$⟩  2  CHECKMULTISIGVERIFY
4  ELSE
5      PUSH⟨$T_0 + \Delta_{\text{fwd}}$⟩  CHECKLOCKTIMEVERIFY  DROP
6      PUSH⟨$\mathsf{pk}_{P_1}$⟩  CHECKSIGVERIFY
7  ENDIF
```

Redemption for P_1 (‘no’ branch):

```
1  $\mathsf{sig}_{P_1}(\mathsf{TX}_{\text{payout},P_1})$
2  FALSE
```

Redemption for P_2 (‘yes’ branch):

```
1  $\mathsf{sig}_{P_{1,\text{tmp}}}(\mathsf{TX}_{\text{retaliate}})$
2  $\mathsf{sig}_{P_{2,\text{tmp}}}(\mathsf{TX}_{\text{retaliate}})$
3  $\hat{p}_i$
4  TRUE
```

Bitcoin Script implementation of output on $\mathsf{TX}_{\text{retaliate}}$:

```
1  IF
2      ECEXP⟨$g$⟩  PUSH⟨$H(\alpha_0)$⟩  EQUALVERIFY
3      PUSH⟨$\mathsf{pk}_{P_1}$⟩  CHECKSIGVERIFY
4  ELSE
5      PUSH⟨$T_0 + \Delta_{\text{rev}}$⟩  CHECKLOCKTIMEVERIFY  DROP
6      PUSH⟨$\mathsf{pk}_{P_2}$⟩  CHECKSIGVERIFY
7  ENDIF
```

Redemption for P_1 (‘yes’ branch):

```
1  $\mathsf{sig}_{P_1}(\mathsf{TX}_{\text{payout},P_1})$
2  $\hat{\alpha}_0$
3  TRUE
```

Redemption for P_2 (‘no’ branch):

```
1  $\mathsf{sig}_{P_2}(\mathsf{TX}_{\text{payout},P_2})$
2  FALSE
```

Fig. 8: Implementation of the \$1 outputs (cf. Fig. 5(a)), and witness stacks for redemption, for the two staggered $\mathsf{TX}_{\text{settle}}$ and $\mathsf{TX}_{\text{retaliate}}$ (cf. Fig. 9). For $\mathsf{TX}_{\text{settle}}$: l. 2 enforces revelation of $\hat{p}_i$ s.t. $H(\hat{p}_i) = H(p_i)$, l. 3 requires signatures of the temporary identities $P_{i,\text{tmp}}$ (created separately for each instance of the contract, and used to sign $\mathsf{TX}_{\text{retaliate}}$ as part of the commitment to a forwarding), l. 5 enforces the timelock of $T_0 + \Delta_{\text{fwd}}$, l. 6 requires a signature of P_1. For $\mathsf{TX}_{\text{retaliate}}$: l. 2 enforces revelation of $\hat{\alpha}_0$ s.t. $H(\hat{\alpha}_0) = H(\alpha_0)$, l. 3 requires a signature of P_1, l. 5 enforces the timelock of $T_0 + \Delta_{\text{rev}}$, l. 6 requires a signature of P_2.

4.2 Implementation Using Adaptor Signatures

The dependency on a new opcode ECEXP⟨g⟩ can be lifted by replacing the hash-lock with adaptor signatures [28] based on Schnorr signatures [22]. A primer on adaptor signatures is given in Section C. Rather than one party revealing a preimage x for a challenge $h \triangleq H(x)$ when publishing a TX to claim funds or advance a contract, P_1 and P_2 exchange an adaptor signature σ' for TX which either party can turn into a proper signature σ once they obtain x. Once TX gets published together with σ, the other party can derive x from σ and σ'.

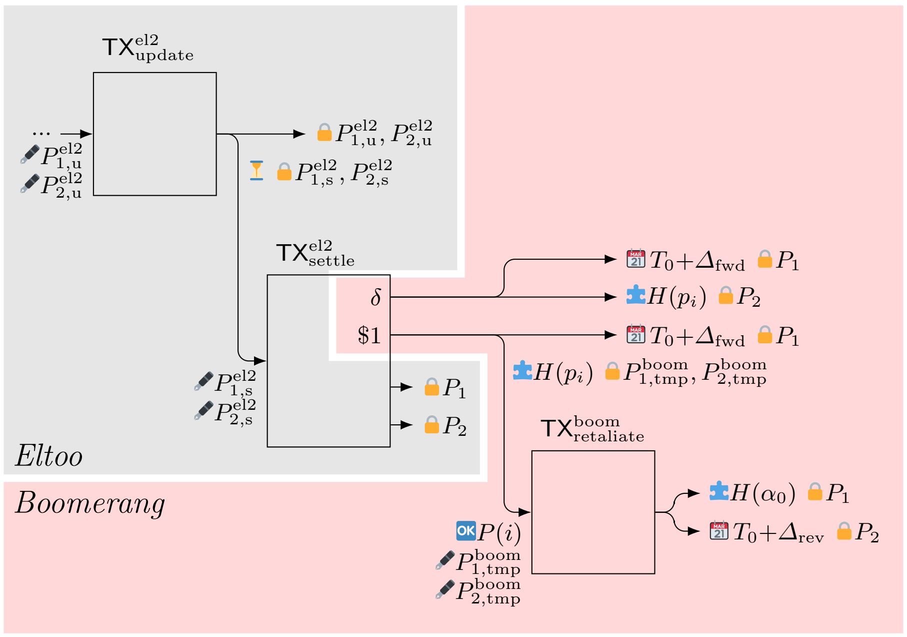

$\mathsf{TX}^{\mathrm{el2}}_{\mathrm{update}}$
...
$P^{\mathrm{el2}}_{1,\mathrm{u}}$
$P^{\mathrm{el2}}_{2,\mathrm{u}}$
$P^{\mathrm{el2}}_{1,\mathrm{u}}, P^{\mathrm{el2}}_{2,\mathrm{u}}$
$P^{\mathrm{el2}}_{1,\mathrm{s}}, P^{\mathrm{el2}}_{2,\mathrm{s}}$
$\mathsf{TX}^{\mathrm{el2}}_{\mathrm{settle}}$
δ
\$1
$P^{\mathrm{el2}}_{1,\mathrm{s}}$
$P^{\mathrm{el2}}_{2,\mathrm{s}}$
P_1
P_2
Eltoo
Boomerang
$T_0+\Delta_{\mathrm{fwd}}$ P_1
$H(p_i)$ P_2
$T_0+\Delta_{\mathrm{fwd}}$ P_1
$H(p_i)$ $P^{\mathrm{boom}}_{1,\mathrm{tmp}}, P^{\mathrm{boom}}_{2,\mathrm{tmp}}$
$\mathsf{TX}^{\mathrm{boom}}_{\mathrm{retaliate}}$
$P(i)$
$P^{\mathrm{boom}}_{1,\mathrm{tmp}}$
$P^{\mathrm{boom}}_{2,\mathrm{tmp}}$
$H(\alpha_0)$ P_1
$T_0+\Delta_{\mathrm{rev}}$ P_2

Fig. 9: A Boomerang contract is deployed as outputs and a thereon depending TX on top of the settlement mechanism of an Eltoo PC. (Legend: Boxes are Bitcoin transactions, an arrow leaving/entering a box indicates an output/input, respectively, labels along arrows indicate spending conditions, forks of arrows indicate alternative spending conditions, labels near TX inputs indicate redemption witnesses, 🔒 'requires signature of', 🖋 'signed by', 🧩 'preimage challenge', 🆗 'preimage solution', 📆 'absolute time-lock', ⏳ 'relative time-lock'.)

4.3 Deployment of Boomerang on an Eltoo Payment Channel

Fig. 9 shows a Boomerang contract for the i-th TX with preimage challenge $H(p_i)$ deployed on top of an Eltoo PC. Transactions and cryptographic identities belonging to Eltoo are marked with $(.)^{\mathrm{el2}}$, those belonging to Boomerang are marked with $(.)^{\mathrm{boom}}$. Fig. 9 also illustrates Figs 11 and 12.

5 Experimental Evaluation

We evaluate Boomerang in a low-fidelity prototype implementation[1] of the routing components inspired by the testbed of Flash [35]. First, we outline our experimental setup, then we specify the three contending schemes, and finally we present and discuss the observed performance.

[1] The source code is available on: https://github.com/tse-group/boomerang

5.1 Experimental Setup

The PN topology is drawn from the Watts–Strogatz ensemble [36] where initially $N = 100$ nodes in a regular ring lattice are connected to their 8 nearest neighbors, and subsequently each initial edge is rewired randomly with probability 0.8. For each PC endpoint, its initial liquidity is drawn log-uniformly in $[\log(100), \log(1000)]$. The topology is static and known to all nodes; each PC's balance is only known to the PC's endpoints. 50000 TFs are generated as follows. Source and destination are sampled uniformly from the N nodes. The amounts are drawn from the Ripple dataset used in SpeedyMurmurs [31]. As in previous works, each node has a backlog of TFs it attempts to route one by one. We report sample mean and sample standard deviation of 10 PNs and TF traces.

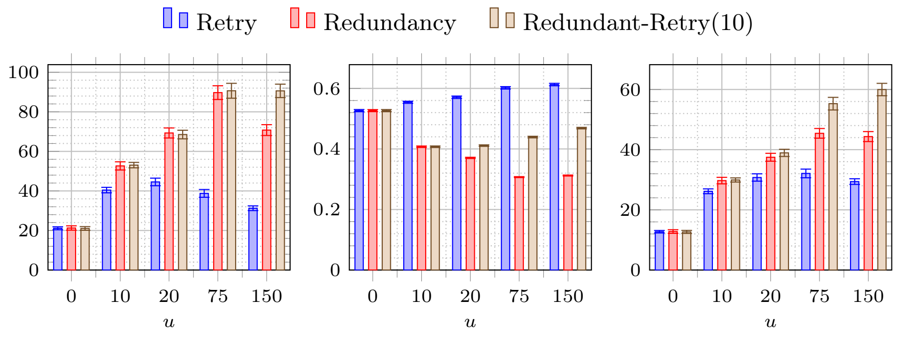
Our implementation is a low-fidelity prototype of the routing and communications tasks. An abstract PN protocol accommodates different routing schemes (cf. Flash [35]). A TX takes place in two phases: First, in the RESERVE phase, Boomerang contracts are set up along a path from A to B if PC liquidity permits. A is notified whether RESERVE was successful. Second, in the ROLLBACK/EXECUTE phase, the chain of Boomerang contracts is either dismantled (ROLLBACK) or the funds are delivered (EXECUTE). An ABORT message can stop an ongoing RESERVE attempt. Purely informational messages (*e.g.*, outcome of RESERVE) are relayed without delay. Operations on the PC (*e.g.*, deploying a Boomerang contract) are simulated by a uniform delay from 50 ms to 150 ms. Note that PC balances are only discovered implicitely through failed/successful TX attempts, similar to the current state of affairs in Lightning.

The software is written in Golang 1.12.7. Every node is an independent process on a machine with 4x AMD Opteron 6378 processors (total 64 cores). The scenario is chosen such that the CPUs are never fully utilized to avoid distortion from computational limitations. The nodes communicate via 'localhost'.

5.2 Three Simple Routing Protocols

We compare three multi-path routing protocols (pseudo code given in Section D). 'Retry' (cf. Alg. 2) initially attempts v TXs and reattempts up to u of them. 'Redundancy' (cf. Alg. 3) attempts $v+u$ TXs from the start. 'Redundant-Retry(10)' (cf. Alg. 4) is a combination of the two, which starts out with $v + \min(u, 10)$ TXs and reattempts up to $u - \min(u, 10)$ of them. The aim is to trade off the lower TTC of 'Redundancy' with the adaptivity of 'Retry'. TXs are routed on paths chosen randomly from a set of precomputed edge-disjoint shortest paths. All three schemes use atomic multi-path (AMP, [25]), *i.e.*, they only EXECUTE once enough successful TX attempts have been made to satisfy the full TF (cf. Alg. 1). If the TF cannot be fully satisfied, all TXs are ROLLBACKed.

The baseline protocol 'Retry' is rather simplistic compared to recent developments [29,30,21,31,16,34,10,35]. We choose it because it resembles what currently can and is being done on Lightning. We conjecture that redundancy as a generic technique can boost AMP routing algorithms across the board.



(a) Throughput $[\frac{\mathrm{XRP}}{\mathrm{s}\cdot\mathrm{node}}]$ (b) TTC [s] (c) Volume [XRP]

Fig. 10: (a) Average success throughput per node, (b) Average time-to-completion for successful TFs, (c) Average volume for successful TFs.

5.3 Simulation Results

Our results are shown in Fig. 10, supplemental plots can be found in Section E. Previous works have gauged the performance of PN routing algorithms in terms of success volume or success count, *i.e.*, for a given trace of TFs, what total amount or number of TFs gets satisfied. However, these metrics are problematic. If TFs are spread out in time then more liquidity is available in the PN. This renders it easier to satisfy a TF and inflates these metrics.

Instead, we consider success throughput, *i.e.*, total amount of successfully transferred funds *per runtime*. The results for $v = 25$ are shown in Fig. 10(a) (cf. Fig. 13(a)). 'Redundancy' and 'Redundant-Retry(10)' show a 2x increase in throughput compared to 'Retry'.

Another relevant figure of merit is the average time-to-completion (TTC) of a successful TF, *i.e.*, how long is the delay between when the execution of a TF starts and when AMP is finalized. This determines the latency experienced by the user and for how long liquidity is tied up in pending AMP TFs. The results for $v = 25$ in Fig. 10(b) (cf. Fig. 13(b)) show a 40 % reduction in TTC for 'Redundancy' over 'Retry'. The larger u, the longer 'Retry' takes but the quicker 'Redundancy' completes. This plot also demonstrates how 'Redundant-Retry(10)' trades off between 'Retry' and 'Redundancy'. For $u \leq 10$, 'Redundant-Retry(10)' is identical with 'Redundancy' and hence follows the performance improvement of 'Redundancy'. For $u > 10$, the retry aspect weighs in more and more and 'Redundant-Retry(10)' follows a similar trajectory as 'Retry'.

Finally, Fig. 10(c) (cf. Fig. 13(c)) shows the average size of a successful TF, where 'Redundant-Retry(10)' outperforms 'Redundancy' by 30 % which in turn outperforms 'Retry' by 40 %. Note that the throughput of 'Redundant-Retry(10)' and 'Redundancy' are comparable. Thus, weighting between adaptively retrying and upfront redundancy trades off TTC and average successful TF volume, at a constant throughput.

Acknowledgments

We thank Giulia Fanti and Lei Yang for fruitful discussions. VB and DT are supported by the Center for Science of Information, an NSF Science and Technology Center, under grant agreement CCF-0939370. JN is supported by a Reed-Hodgson Stanford Graduate Fellowship. Icons from 'Twemoji v12.0' (https://github.com/twitter/twemoji) by Twitter, Inc and other contributors, licensed under CC BY 4.0.

References

1. Lightning Strikes, But Select Hubs Dominate Network Funds (Jun 2018), https://diar.co/volume-2-issue-25/#1
2. It's Getting Harder to Send Bitcoin's Lightning Torch – Here's Why (Mar 2019), https://www.coindesk.com/its-getting-harder-to-send-bitcoins-lightning-torch-heres-why
3. What are atomic multi path payments (AMPs) and why/how is it being implemented in Lightning Network? (Jul 2019), https://bitcoin.stackexchange.com/q/89475
4. Aktas, M.F., Soljanin, E.: Straggler Mitigation at Scale (Jun 2019), http://arxiv.org/abs/1906.10664
5. Benaloh, J.C.: Secret Sharing Homomorphisms: Keeping Shares of a Secret Secret. In: Proc. CRYPTO '86. pp. 251–260. Santa Barbara, CA, USA (1987)
6. Byers, J.W., Luby, M., Mitzenmacher, M., Rege, A.: A Digital Fountain Approach to Reliable Distribution of Bulk Data. In: Proc. ACM SIGCOMM. pp. 56–67. Vancouver, B.C., Canada (1998). https://doi.org/10.1145/285237.285258
7. Dean, J., Barroso, L.A.: The Tail at Scale. Commun. ACM **56**(2), 74–80 (Feb 2013). https://doi.org/10.1145/2408776.2408794
8. Decker, C., Russell, R., Osuntokun, O.: eltoo: A Simple Layer2 Protocol for Bitcoin. Tech. rep. (Apr 2018), https://blockstream.com/2018/04/30/en-eltoo-next-lightning/
9. Decker, C., Wattenhofer, R.: A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. In: Stabilization, Safety, and Security of Distributed Systems. pp. 3–18. LNCS, Springer (2015)
10. Di Stasi, G., Avallone, S., Canonico, R., Ventre, G.: Routing Payments on the Lightning Network. In: Proc. IEEE iThings/GreenCom/CPSCom/SmartData. pp. 1161–1170 (Jul 2018). https://doi.org/10.1109/Cybermatics_2018.2018.00209
11. Dziembowski, S., Eckey, L., Faust, S., Malinowski, D.: Perun: Virtual Payment Hubs over Cryptocurrencies (2017), https://eprint.iacr.org/2017/635
12. Dziembowski, S., Faust, S., Hostáková, K.: General State Channel Networks. In: Proc. ACM SIGSAC. pp. 949–966. Toronto, Canada (2018). https://doi.org/10.1145/3243734.3243856
13. Elias, P.: Coding for Two Noisy Channels. In: Information Theory, pp. 61–74. Academic Press (1956)
14. Feldman, P.: A practical scheme for non-interactive verifiable secret sharing. In: 28th Annual Symposium on Foundations of Computer Science (sfcs 1987). pp. 427–438 (Oct 1987). https://doi.org/10.1109/SFCS.1987.4
15. Gudgeon, L., Moreno-Sanchez, P., Roos, S., McCorry, P., Gervais, A.: SoK: Off The Chain Transactions (2019), https://eprint.iacr.org/2019/360
16. Hoenisch, P., Weber, I.: AODV–Based Routing for Payment Channel Networks. In: Blockchain – ICBC 2018. pp. 107–124. LNCS, Springer (2018)

17. Jourenko, M., Kurazumi, K., Larangeira, M., Tanaka, K.: SoK: A Taxonomy for Layer-2 Scalability Related Protocols for Cryptocurrencies (2019), https://eprint.iacr.org/2019/352
18. Khalil, R., Gervais, A.: Revive: Rebalancing Off-Blockchain Payment Networks (2017), https://eprint.iacr.org/2017/823
19. Lee, K., Lam, M., Pedarsani, R., Papailiopoulos, D., Ramchandran, K.: Speeding Up Distributed Machine Learning Using Codes. IEEE Transactions on Information Theory **64**(3), 1514–1529 (Mar 2018). https://doi.org/10.1109/TIT.2017.2736066
20. Luby, M., Shokrollahi, A., Watson, M., Stockhammer, T., Minder, L.: RaptorQ Forward Error Correction Scheme for Object Delivery. RFC **6330** (2011). https://doi.org/10.17487/RFC6330
21. Malavolta, G., Moreno-Sanchez, P., Kate, A., Maffei, M.: SilentWhispers: Enforcing Security and Privacy in Decentralized Credit Networks (2016), https://eprint.iacr.org/2016/1054
22. Maxwell, G., Poelstra, A., Seurin, Y., Wuille, P.: Simple Schnorr Multi-Signatures with Applications to Bitcoin (2018), https://eprint.iacr.org/2018/068
23. Miller, A., Bentov, I., Kumaresan, R., Cordi, C., McCorry, P.: Sprites and State Channels: Payment Networks that Go Faster than Lightning (Feb 2017), http://arxiv.org/abs/1702.05812
24. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. Tech. rep. (Oct 2008), https://bitcoin.org/bitcoin.pdf
25. Osuntokun, O.: AMP: Atomic Multi-Path Payments over Lightning (Feb 2018), https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-February/000993.html
26. Pedersen, T.P.: Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In: Proc. CRYPTO '91. pp. 129–140. Santa Barbara, CA, USA (1992)
27. Piatkivskyi, D., Nowostawski, M.: Split Payments in Payment Networks. In: Data Privacy Management, Cryptocurrencies and Blockchain Technology. pp. 67–75. LNCS, Springer (2018)
28. Poelstra, A.: Scriptless Scripts (2018), https://download.wpsoftware.net/bitcoin/wizardry/mw-slides/2018-05-18-l2/slides.pdf
29. Poon, J., Dryja, T.: The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. Tech. rep. (Jan 2016), https://lightning.network/docs/
30. Prihodko, P., Zhigulin, S., Sahno, M., Ostrovskiy, A., Osuntokun, O.: Flare: An Approach to Routing in Lightning Network (Jul 2016)
31. Roos, S., Moreno-Sanchez, P., Kate, A., Goldberg, I.: Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions (Sep 2017), http://arxiv.org/abs/1709.05748
32. Schoenmakers, B.: A Simple Publicly Verifiable Secret Sharing Scheme and Its Application to Electronic Voting. In: Proc. CRYPTO '99. pp. 148–164. Santa Barbara, CA, USA (1999)
33. Shamir, A.: How to Share a Secret. Commun. ACM **22**(11), 612–613 (Nov 1979). https://doi.org/10.1145/359168.359176
34. Sivaraman, V., Venkatakrishnan, S.B., Alizadeh, M., Fanti, G., Viswanath, P.: Routing Cryptocurrency with the Spider Network (Sep 2018), http://arxiv.org/abs/1809.05088
35. Wang, P., Xu, H., Jin, X., Wang, T.: Flash: Efficient Dynamic Routing for Offchain Networks (Feb 2019), http://arxiv.org/abs/1902.05260
36. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**(6684), 440–442 (1998)

A Cryptographic Preliminaries

We briefly recapitulate the cryptographic tools used throughout the paper. Let $\mathbb{G}$ be a cyclic multiplicative group of prime order $q \geq 2^{2\lambda}$ with a generator $g \in \mathbb{G}$, where λ is a security parameter. Let $H\colon \mathbb{Z}_q \to \mathbb{G}$ with $H(x) \triangleq g^x$, where $\mathbb{Z}_q$ is the finite field of size q (*i.e.*, integers modulo q). We require that H be difficult to invert, which is formalized in the following two definitions:

Definition 1 (Negligible Function). *A function $\varepsilon\colon \mathbb{N} \to \mathbb{R}^+$ is* negligible *if*

$$\forall c > 0\colon \exists k_0\colon \forall k > k_0\colon \qquad \varepsilon(k) < \frac{1}{k^c}. \tag{8}$$

In other words, negligible is what decays faster than every polynomial.

Definition 2 (Discrete Logarithm (DL) Assumption). *Given a generator g of a group $\mathbb{G}$, and an $x \overset{\mathrm{R}}{\leftarrow} \mathbb{Z}_q$ chosen uniformly at random in $\mathbb{Z}_q$, for every probabilistic polynomial time (with respect to λ) algorithm $\mathcal{A}_{\mathrm{DL}}$,*

$$\Pr[\mathcal{A}_{\mathrm{DL}}(g, g^x) = x] = \varepsilon(\lambda). \tag{9}$$

The discrete logarithm problem (DLP) is said to be hard for generator g in group $\mathbb{G}$, if the DL assumption holds for g and $\mathbb{G}$, *i.e.*, no computationally bounded adversary can compute $\log_g(g^x)$ except for a negligible fraction of $\mathbb{G}$. It is commonly assumed that the DLP is hard in certain elliptic curves (ECs), which are hence widely used in cryptographic applications, *e.g.*, in Bitcoin.

The DL assumption makes H a one-way function. It has the following homomorphic property which we make extensive use of:

$$\forall n \geq 1\colon \forall c_1, \ldots, c_n\colon \quad H\left(\sum_{i=1}^{n} c_i x_i\right) = \prod_{i=1}^{n} H(x_i)^{c_i} \tag{10}$$

For the digital signature scheme employed by Bitcoin, an identity P consists of a public key pk_P and a private (secret) key sk_P. Using sk_P, a signature $\sigma \triangleq \mathsf{sig}_P(x)$ for string x can be created, and using pk_P, a purported signature σ' for string x can be verified, $\mathsf{ver}_P(x, \sigma') \in \{0, 1\}$.

B Implementation of Boomerang Contract in Bitcoin Script via Elliptic Curve Scalar Multiplication

See Figs 11 and 12.

C Background on Adaptor Signatures

We briefly summarize Schnorr signatures [22]. Let $\tilde{H}$ be a cryptographic hash function, and $x \| y$ denote the concatenation of x and y. We continue to assume that $\mathbb{G}$ is a multiplicative group with group operation '$\cdot$'. For Schnorr signatures,

Bitcoin Script implementation of output on $\mathsf{TX}_{\text{settle}}$:

```
1  IF
2      ECEXP⟨$g$⟩  PUSH⟨$H(p_i)$⟩  EQUALVERIFY
3      PUSH⟨$\mathsf{pk}_{P_2}$⟩  CHECKSIGVERIFY
4  ELSE
5      PUSH⟨$T_0 + \Delta_{\text{fwd}}$⟩  CHECKLOCKTIMEVERIFY  DROP
6      PUSH⟨$\mathsf{pk}_{P_1}$⟩  CHECKSIGVERIFY
7  ENDIF
```

Redemption for P_1 ('no' branch):

```
1  $\mathsf{sig}_{P_1}(\mathsf{TX}_{\text{payout},P_1})$
2  FALSE
```

Redemption for P_2 ('yes' branch):

```
1  $\mathsf{sig}_{P_2}(\mathsf{TX}_{\text{payout},P_2})$
2  $\hat{p}_i$
3  TRUE
```

Fig. 11: Bitcoin Script implementation of the output concerning δ TX fee (cf. Fig. 5(b)), and witness stacks for redemption in favor of P_1 and P_2, respectively: l. 2 enforces revelation of $\hat{p}_i$ such that $H(\hat{p}_i) = H(p_i)$, l. 3 requires a signature of P_2, l. 5 enforces the timelock until $T_0 + \Delta_{\text{fwd}}$, l. 6 requires a signature of P_1.

every identity is composed of a secret key x and a public key $P \triangleq g^x$. To sign a message m, draw $r \overset{\text{R}}{\leftarrow} \mathbb{Z}_q$, then compute $R \triangleq g^r$ and $s = r + \tilde{H}(P\|R\|m)x$. The signature is $\sigma \triangleq (s, R)$. To verify a signature $\sigma \triangleq (s, R)$ for m by P, check

$$g^s \overset{?}{=} R \cdot P^{\tilde{H}(P\|R\|m)}. \tag{11}$$

An adaptor signature σ' has the property that given σ', knowledge of a proper signature σ is equivalent to knowledge of a precommitted value t [28]. Consider parties P_1 and P_2 with secret keys x_i and public keys $P_i \triangleq g^{x_i}$. Both know a commitment $T \triangleq g^t$ to a (potentially unknown) value t. To create an adaptor signature σ' for m, both draw $r_i \overset{\text{R}}{\leftarrow} \mathbb{Z}_q$, compute $R_i \triangleq g^{r_i}$, and exchange (P_i, R_i). Then, they compute and exchange

$$s'_i = r_i + \tilde{H}(P_1 \cdot P_2 \| R_1 \cdot R_2 \cdot T \| m)x_i. \tag{12}$$

The adaptor signature is $\sigma' = (R_1 \cdot R_2 \cdot T, s'_1 + s'_2)$. If either P_i gets to know t, they can produce a valid total signature $\sigma = (R_1 \cdot R_2 \cdot T, s'_1 + s'_2 + t)$. Vice versa, if either P_i learns a valid total signature $\sigma = (R_1 \cdot R_2 \cdot T, s)$, they can compute $t = s - s'_1 - s'_2$. For instance, suppose m is a transaction that benefits P_2 and requires a signature from $P_1 \cdot P_2$ with nonce $R_1 \cdot R_2 \cdot T$. Furthermore, suppose P_2 obtains t. Then it can use the adaptor signature σ' to produce a valid total signature σ and claim its funds. In this case, P_1 can recover t from σ and σ'.

D Pseudo Code of Evaluated Routing Schemes

See Algs 1, 2, 3 and 4.

E Supplemental Plots for Experimental Evaluation

See Fig. 13.

Bitcoin Script implementation of output on $\mathsf{TX}_{\text{settle}}$:

```
1  IF
2      ECEXP⟨$g$⟩  PUSH⟨$H(p_i)$⟩  EQUALVERIFY
3      2  PUSH⟨$\mathsf{pk}_{P_{1,\text{tmp}}}$⟩  PUSH⟨$\mathsf{pk}_{P_{2,\text{tmp}}}$⟩  2  CHECKMULTISIGVERIFY
4  ELSE
5      PUSH⟨$T_0 + \Delta_{\text{fwd}}$⟩  CHECKLOCKTIMEVERIFY  DROP
6      PUSH⟨$\mathsf{pk}_{P_1}$⟩  CHECKSIGVERIFY
7  ENDIF
```

Redemption for P_1 (‘no’ branch):

```
1  $\mathsf{sig}_{P_1}(\mathsf{TX}_{\text{payout},P_1})$
2  FALSE
```

Redemption for P_2 (‘yes’ branch):

```
1  $\mathsf{sig}_{P_{1,\text{tmp}}}(\mathsf{TX}_{\text{retaliate}})$
2  $\mathsf{sig}_{P_{2,\text{tmp}}}(\mathsf{TX}_{\text{retaliate}})$
3  $\hat{p}_i$
4  TRUE
```

Bitcoin Script implementation of output on $\mathsf{TX}_{\text{retaliate}}$:

```
1  IF
2      ECEXP⟨$g$⟩  PUSH⟨$H(\alpha_0)$⟩  EQUALVERIFY
3      PUSH⟨$\mathsf{pk}_{P_1}$⟩  CHECKSIGVERIFY
4  ELSE
5      PUSH⟨$T_0 + \Delta_{\text{rev}}$⟩  CHECKLOCKTIMEVERIFY  DROP
6      PUSH⟨$\mathsf{pk}_{P_2}$⟩  CHECKSIGVERIFY
7  ENDIF
```

Redemption for P_1 (‘yes’ branch):

```
1  $\mathsf{sig}_{P_1}(\mathsf{TX}_{\text{payout},P_1})$
2  $\hat{\alpha}_0$
3  TRUE
```

Redemption for P_2 (‘no’ branch):

```
1  $\mathsf{sig}_{P_2}(\mathsf{TX}_{\text{payout},P_2})$
2  FALSE
```

Fig. 12: Bitcoin Script implementation of the outputs concerning the \$1 unit amount (cf. Fig. 5(a)), and witness stacks for redemption in favor of P_1 and P_2, respectively, for the two staggered transactions $\mathsf{TX}_{\text{settle}}$ and $\mathsf{TX}_{\text{retaliate}}$ (cf. Fig. 9). For $\mathsf{TX}_{\text{settle}}$: l. 2 enforces revelation of $\hat{p}_i$ such that $H(\hat{p}_i) = H(p_i)$, l. 3 requires signatures of the temporary one-time identities $P_{1,\text{tmp}}$ and $P_{2,\text{tmp}}$ (created separately for each instance of the contract, and used to sign $\mathsf{TX}_{\text{retaliate}}$ as part of the commitment to a forwarding), l. 5 enforces the timelock of $T_0 + \Delta_{\text{fwd}}$, l. 6 requires a signature of P_1. For $\mathsf{TX}_{\text{retaliate}}$: l. 2 enforces revelation of $\hat{\alpha}_0$ such that $H(\hat{\alpha}_0) = H(\alpha_0)$, l. 3 requires a signature of P_1, l. 5 enforces the timelock of $T_0 + \Delta_{\text{rev}}$, l. 6 requires a signature of P_2.

Algorithm 1 Finalize AMP TF

```
1: procedure FinalizeAMP($\mathcal{O}, \mathcal{P}, v$)        ▷ Aborts outstanding TX attempts $\mathcal{O}$,
   Executes/Rollbacks successful TXs $\mathcal{P}$ depending on number of required TXs $v$
2:     SendAbort($\mathcal{O}$)
3:     if $|\mathcal{P}| = v$ then
4:         SendExecute($\mathcal{P}$)
5:     else
6:         SendRollback($\mathcal{P}$)
7:     end if
8:     ReceiveReserveResponsesAndSendRollback($\mathcal{O}$)
9: end procedure
```

Algorithm 2 ‘Retry’ routing scheme

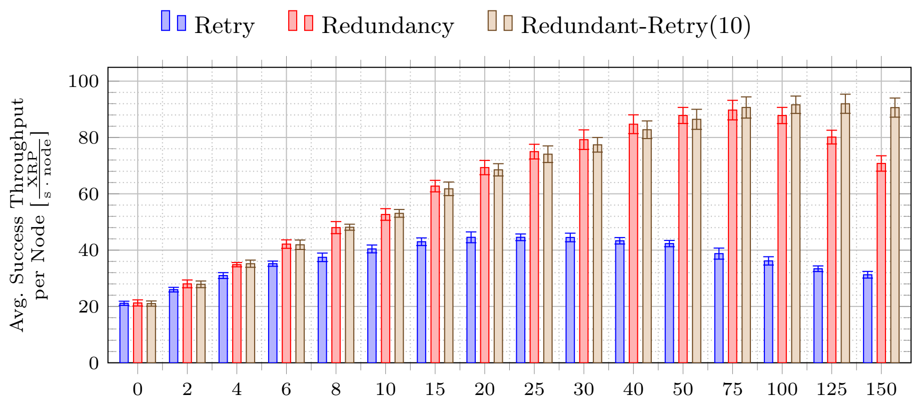
```
1: procedure Retry($\mathcal{T}, v, u$)        ▷ Split each TF $t \in \mathcal{T}$ into $v$ TXs and retry $u$ TXs
2:     for $t \in \mathcal{T}$ do        ▷ Source $t.s$, destination $t.d$, amount $t.v$
3:         $\mathcal{O}, \mathcal{P}, \mathcal{N} \leftarrow \emptyset, \emptyset, \emptyset$        ▷ TX attempts: outstanding $\mathcal{O}$, successful $\mathcal{P}$, failed $\mathcal{N}$
4:         $u' \leftarrow u$        ▷ Unused TX retries
5:         for $i = 1, \ldots, v$ do        ▷ Attempt $v$ TXs
6:             $\mathcal{O} \leftarrow \mathcal{O} \cup$ SendReserve(RandomPath($t.s, t.d$), $t.v/v$)
7:         end for
8:         while $|\mathcal{P}| < v \wedge |\mathcal{O}| > 0 \wedge |\mathcal{P}| + |\mathcal{O}| + u' \geq v$ do        ▷ TF is contingent
9:             ReceiveAndClassifyReserveResponses($\mathcal{O}, \mathcal{P}, \mathcal{N}$)
10:            for new elements $r \in \mathcal{N}$ do
11:                SendRollback($r$)
12:                if $u' > 0$ then
13:                    $\mathcal{O} \leftarrow \mathcal{O} \cup$ SendReserve(RandomPath($t.s, t.d$), $t.v/v$)
14:                    $u' \leftarrow u' - 1$
15:                end if
16:            end for
17:        end while
18:        FinalizeAMP($\mathcal{O}, \mathcal{P}, v$)
19:    end for
20: end procedure
```

Algorithm 3 ‘Redundancy’ routing scheme

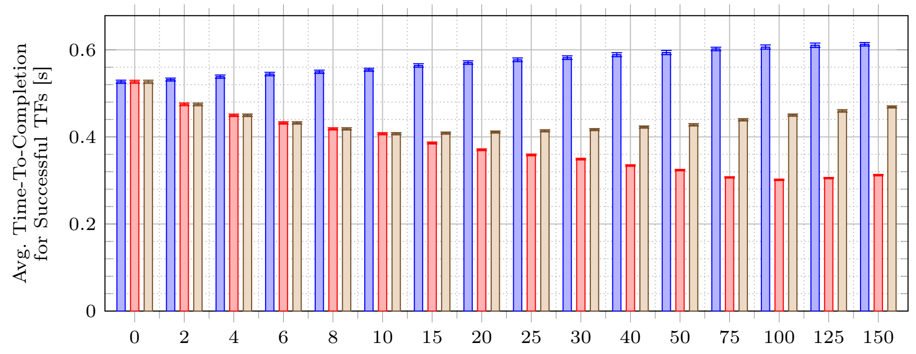
```
1: procedure Redundancy($\mathcal{T}, v, u$)        ▷ Split TFs $t \in \mathcal{T}$ into $v$ plus $u$ redundant TXs
2:     for $t \in \mathcal{T}$ do        ▷ Source $t.s$, destination $t.d$, amount $t.v$
3:         $\mathcal{O}, \mathcal{P}, \mathcal{N} \leftarrow \emptyset, \emptyset, \emptyset$        ▷ TX attempts: outstanding $\mathcal{O}$, successful $\mathcal{P}$, failed $\mathcal{N}$
4:         for $i = 1, \ldots, v + u$ do        ▷ Attempt $v$ TXs
5:             $\mathcal{O} \leftarrow \mathcal{O} \cup$ SendReserve(RandomPath($t.s, t.d$), $t.v/v$)
6:         end for
7:         while $|\mathcal{P}| < v \wedge |\mathcal{O}| > 0 \wedge |\mathcal{P}| + |\mathcal{O}| \geq v$ do        ▷ TF is contingent
8:             ReceiveAndClassifyReserveResponses($\mathcal{O}, \mathcal{P}, \mathcal{N}$)
9:             for new elements $r \in \mathcal{N}$ do
10:                SendRollback($r$)
11:            end for
12:        end while
13:        FinalizeAMP($\mathcal{O}, \mathcal{P}, v$)
14:    end for
15: end procedure
```

Algorithm 4 ‘Redundant-Retry(10)’ routing scheme

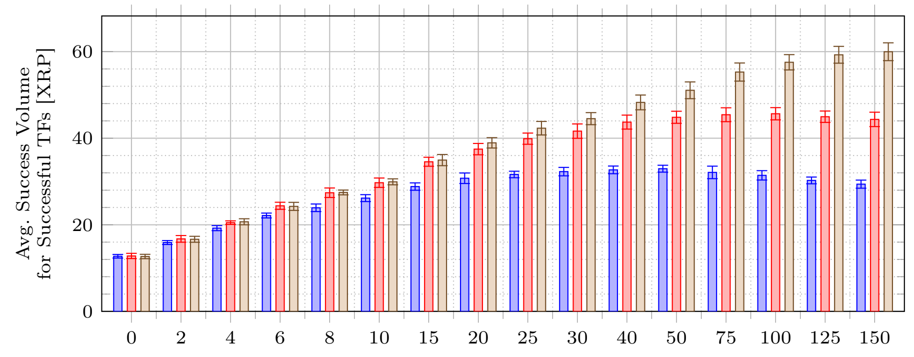
```
1: procedure RedundantRetry($\mathcal{T}, v, u, 10$)
2:     for $t \in \mathcal{T}$ do        ▷ Source $t.s$, destination $t.d$, amount $t.v$
3:         $\mathcal{O}, \mathcal{P}, \mathcal{N} \leftarrow \emptyset, \emptyset, \emptyset$        ▷ TX attempts: outstanding $\mathcal{O}$, successful $\mathcal{P}$, failed $\mathcal{N}$
4:         $u' \leftarrow u - \min(u, 10)$        ▷ Unused TX retries
5:         for $i = 1, \ldots, v + \min(u, 10)$ do        ▷ Attempt $v + \min(u, 10)$ TXs
6:             $\mathcal{O} \leftarrow \mathcal{O} \cup$ SendReserve(RandomPath($t.s, t.d$), $t.v/v$)
7:         end for
8:         while $|\mathcal{P}| < v \wedge |\mathcal{O}| > 0 \wedge |\mathcal{P}| + |\mathcal{O}| + u' \geq v$ do        ▷ TF is contingent
9:             ReceiveAndClassifyReserveResponses($\mathcal{O}, \mathcal{P}, \mathcal{N}$)
10:            for new elements $r \in \mathcal{N}$ do
11:                SendRollback($r$)
12:                if $u' > 0$ then
13:                    $\mathcal{O} \leftarrow \mathcal{O} \cup$ SendReserve(RandomPath($t.s, t.d$), $t.v/v$)
14:                    $u' \leftarrow u' - 1$
15:                end if
16:            end for
17:        end while
18:        FinalizeAMP($\mathcal{O}, \mathcal{P}, v$)
19:    end for
20: end procedure
```



(a) 'Redundancy' and 'Redundant-Retry(10)' show a 2x increase in throughput compared to 'Retry'.



(b) 'Redundancy' shows a 40 % reduction in TTC over 'Retry'. 'Redundant-Retry(10)' interpolates in TTC between 'Retry' and 'Redundancy'.



(c) 'Redundant-Retry(10)' satisfies 30 % larger TFs than 'Redundancy' which in turn satisfies 40 % larger TFs than the 'Retry' baseline.

Fig. 13: Performance as a function of maximum number of additional TXs (u)