TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub

Ethan Heilman*, Leen AlShenibr*, Foteini Baldimtsi†, Alessandra Scafuro‡ and Sharon Goldberg*
*Boston University {heilman, leenshe}@bu.edu, goldbe@cs.bu.edu
†George Mason University foteini@gmu.edu
‡North Carolina State University ascafur@ncsu.edu

***Abstract*—This paper presents *TumbleBit*, a new unidirectional unlinkable payment hub that is fully compatible with today's Bitcoin protocol. TumbleBit allows parties to make fast, anonymous, off-blockchain payments through an untrusted intermediary called the Tumbler. TumbleBit's anonymity properties are similar to classic Chaumian eCash: no one, not even the Tumbler, can link a payment from its payer to its payee. Every payment made via TumbleBit is backed by bitcoins, and comes with a guarantee that Tumbler can neither violate anonymity, nor steal bitcoins, nor "print money" by issuing payments to itself. We prove the security of TumbleBit using the real/ideal world paradigm and the random oracle model. Security follows from the standard RSA assumption and ECDSA unforgeability. We implement TumbleBit, mix payments from 800 users and show that TumbleBit's off-blockchain payments can complete in seconds.**

I. Introduction

One reason for Bitcoin's initial popularity was the perception of anonymity. Today, however, the sheen of anonymity has all but worn off, dulled by a stream of academic papers [40], [52], and a blockchain surveillance industry [34], [28], that have demonstrated weaknesses in Bitcoin's anonymity properties. As a result, a new market of anonymity-enhancing services has emerged [45], [24], [1]; for instance, 1 million USD in bitcoins are funneled through JoinMarket each month [45]. These services promise to mix bitcoins from a set of *payers* (*aka,* input Bitcoin addresses $\mathcal{A}$) to a set of *payees* (*aka,* output bitcoin addresses $\mathcal{B}$) in a manner that makes it difficult to determine which payer transferred bitcoins to which payee.

To deliver on this promise, anonymity must also be provided in the face of the anonymity-enhancing service itself—if the service knows exactly which payer is paying which payee, then a compromise of the service leads to a total loss of anonymity. Compromise of anonymity-enhancing technologies is not unknown. In 2016, for example, researchers found more than 100 Tor nodes snooping on their users [47]. Moreover, users of mix services must also contend with the potential risk of "exit scams", where an established business takes in new payments but stops providing services. Exit scams have been known to occur in the Bitcoin world. In 2015, a Darknet Marketplace stole 11.7M dollars worth of escrowed customer bitcoins [54], while btcmixers.com mentions eight different scam mix services. Thus, it is crucial that anonymity-enhancing services be designed in a manner that prevents bitcoin theft.

TumbleBit: An unlinkable payment hub. We present TumbleBit, a *unidirectional unlinkable payment hub* that uses an *untrusted* intermediary, the *Tumbler* $\mathcal{T}$, to enhance anonymity. Every payment made via TumbleBit is backed by bitcoins. We use cryptographic techniques to guarantee Tumbler $\mathcal{T}$ can neither violate anonymity, nor steal bitcoins, nor "print money" by issuing payments to itself. TumbleBit allows a payer Alice $\mathcal{A}$ to send fast off-blockchain payments (of denomination one bitcoin) to a set of payees $(\mathcal{B}_1, ..., \mathcal{B}_Q)$ of her choice. Because payments are performed off the blockchain, TumbleBit also serves to scale the volume and velocity of bitcoin-backed payments. Today, on-blockchain bitcoin transactions suffer a latency of $\approx$ 10 minutes. Meanwhile, TumbleBit payments are sent off-blockchain, via the Tumbler $\mathcal{T}$, and complete in seconds. (Our implementation[1] completed a payment in 1.2 seconds, on average, when $\mathcal{T}$ was in New York and $\mathcal{A}$ and $\mathcal{B}$ were in Boston.)

TumbleBit Overview. TumbleBit replaces on-blockchain payments with off-blockchain puzzle solving, where Alice $\mathcal{A}$ pays Bob $\mathcal{B}$ by providing $\mathcal{B}$ with the solution to a puzzle. The puzzle z is generated through interaction between $\mathcal{B}$ and $\mathcal{T}$, and solved through an interaction between $\mathcal{A}$ and $\mathcal{T}$. Each time a puzzle is solved, 1 bitcoin is transferred from Alice $\mathcal{A}$ to the Tumbler $\mathcal{T}$ and finally on to Bob $\mathcal{B}$.

The protocol proceeds in three phases; see Figure 1. In the on-blockchain *Escrow Phase*, each payer Alice

Foteini Baldimtsi and Alessandra Scafuro performed this work while at Boston University.

This is the authors' full version, last updated on February 17, 2017, of a paper that appears at NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
http://dx.doi.org/10.14722/ndss.2017.23086

[1] https://github.com/BUSEC/TumbleBit/

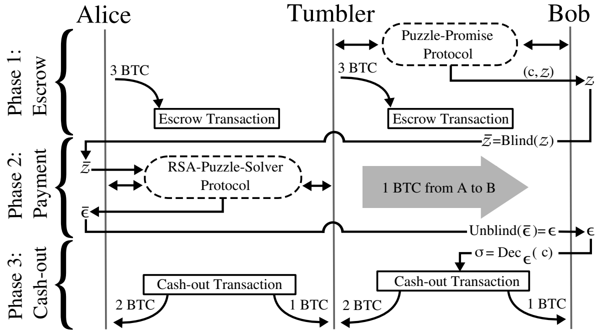

Fig. 1. Overview of the TumbleBit protocol.

$\mathcal{A}$ opens a payment channel with the Tumbler $\mathcal{T}$ by escrowing Q bitcoins on the blockchain. Each payee Bob $\mathcal{B}$ also opens a channel with $\mathcal{T}$. This involves (1) $\mathcal{T}$ escrowing Q bitcoins on the blockchain, and (2) $\mathcal{B}$ and $\mathcal{T}$ engaging in a *puzzle-promise* protocol that generates up to Q puzzles for $\mathcal{B}$. During the off-blockchain *Payment Phase*, each payer $\mathcal{A}$ makes up to Q off-blockchain payments to any set of payees. To make a payment, $\mathcal{A}$ interacts with $\mathcal{T}$ to learn the solution to a puzzle $\mathcal{B}$ provided. Finally, the *Cash-Out Phase* closes all payment channels. Each payee $\mathcal{B}$ uses his Q' solved puzzles (*aka,* TumbleBit payments) to create an on-blockchain transaction that claims Q' bitcoins from $\mathcal{T}$'s escrow. Each payer $\mathcal{A}$ also closes her escrow with $\mathcal{T}$, recovering bitcoins not used in a payment.

Anonymity properties. TumbleBit provides *unlinkability*: Given the set of escrow transactions and the set of cash-out transactions, we define a valid configuration as a set of payments that explains the transfer of funds from Escrow to Cash-Out. Unlinkability ensures that if the Tumbler $\mathcal{T}$ does not collude with other TumbleBit users, then $\mathcal{T}$ cannot distinguish the true configuration (*i.e.,* the set of payments actually sent during the Payment Phase) from any other valid configuration.

TumbleBit is therefore similar to classic Chaumian eCash [17]. With Chaumian eCash, a payee $\mathcal{A}$ first withdraws an eCash coin in exchange for money (*e.g.,* USD) at an intermediary Bank, then uses the coin to pay a payee $\mathcal{B}$. Finally $\mathcal{B}$ redeems the eCash coin to the Bank in exchange for money. Unlinkability ensures that the Bank cannot link the withdrawal of an eCash coin to the redemption of it. TumbleBit provides unlinkability, with Tumbler $\mathcal{T}$ playing the role of the Chaumian Bank. However, while Tumbler $\mathcal{T}$ need not be trusted, the Chaumian Bank is trusted to not (1) "print money" (*i.e.,* issue eCash coins to itself) or (2) steal money (*i.e.,* refuse to exchange coins for money).

TumbleBit: As a classic tumbler. TumbleBit can also be used as a classic Bitcoin tumbler, mixing together the transfer of one bitcoin from $\aleph$ distinct *payers* (Alice $\mathcal{A}$) to $\aleph$ distinct payees (Bob $\mathcal{B}$). In this mode, TumbleBit is run as in Figure 1 with the payment phase shrunk to 30 seconds, so the protocol runs in *epochs* that require two blocks added to the blockchain. As a classic tumbler, TumbleBit provides *k-anonymity within an epoch*—no one, not even the Tumbler $\mathcal{T}$, can link one of the k transfers that were successfully completed during the epoch to a specific pair of payer and payee $(\mathcal{A}, \mathcal{B})$.

RSA-puzzle solving. At the core of TumbleBit is our new "RSA puzzle solver" protocol that may be of independent interest. This protocol allows Alice $\mathcal{A}$ to pay one bitcoin to $\mathcal{T}$ in *fair exchange*[2] for an RSA exponentiation of a "puzzle" value z under $\mathcal{T}$'s secret key. Fair exchange prevents a cheating $\mathcal{T}$ from claiming $\mathcal{A}$'s bitcoin without solving the puzzle. Our protocol is interesting because it is fast—solving 2048-bit RSA puzzles faster than [39]'s fair-exchange protocol for solving 16x16 Sudoku puzzles (Section VIII))—and because it supports RSA. The use of RSA means that blinding can be used to break the link between the user providing the puzzle (*i.e.,* Bob $\mathcal{B}$) and the user requesting its solution (*e.g.,* payer Alice $\mathcal{A}$).

Cryptographic protocols. TumbleBit is realized by interleaving the RSA-puzzle-solver protocol with another fair-exchange *puzzle-promise* protocol. We formally prove that each protocol is a fair exchange. Our proofs use the real/ideal paradigm in the random oracle model (ROM) and security relies on the standard RSA assumption and the unforgeability of ECDSA signatures.

A. TumbleBit Features

Bitcoin compatibility. TumbleBit is fully compatible with today's Bitcoin protocol. We developed (off-blockchain) cryptographic protocols that work with the very limited set of (on-blockchain) instructions provided by today's Bitcoin scripts. Bitcoin scripts can only be used to perform two cryptographic operations: (1) validate the preimage of a hash, or (2) validate an ECDSA signature on a Bitcoin transaction. The limited functionality of Bitcoin scripts is likely here to stay; indeed, the recent "DAO" theft [49] has highlighted the security risks of complex scripting functionalities. Moreover, the Bitcoin community is currently debating [20] whether to deploy a solution ("segregated witnesses" [59]) that corrects Bitcoin's transaction malleability issue 1). TumbleBit, however, remains secure even if this solution is not deployed as explained in Appendix I.

No coordination. In contrast to earlier work [37], [53], if Alice $\mathcal{A}$ wants to pay Bob $\mathcal{B}$, she need not interact with any other TumbleBit users. Instead, $\mathcal{A}$ and $\mathcal{B}$ need only interact with the Tumbler and each other. This lack of coordination between TumbleBit users makes it possible to scale our system.

[2] True fair exchange is impossible in the standard model [48] and thus alternatives have been proposed, such as gradual release mechanisms, optimistic models, or use of a trusted third party. We follow prior works that use Bitcoin for fair exchange [4], [32], [33] and treat the blockchain as a trusted public ledger. Other works use the term Contingent Payment or Atomic Swaps [36], [6].

Scheme	Prevents Theft	Anonymous	Resists DoS	Resists Sybils	Minimum Mixing Time	Bitcoin Compatible	No Coordination?
Coinjoin [37]	✓	small set	×	×	1 block	✓	×
Coinshuffle [53], [43]	✓	small set	×	×	1 block	✓	× (p2p network)
Coinparty [60]	$2/3$ users honest	✓	some[1]	✓ (fees)	2 blocks	✓	×
XIM [13]	✓	✓	✓	✓ (fees)	hours	✓	× (uses blockchain)
Mixcoin [15]	TTP accountable	× (TTP)	✓	✓ (fees)	2 blocks	✓	✓
Blindcoin [58]	TTP accountable	✓	✓	✓ (fees)	2 blocks	✓	✓
CoinSwap [38]	✓	× (TTP)[2]	✓	✓ (fees)	2 blocks	✓	✓
BSC [27]	✓	✓	✓	✓ (fees)	3 blocks	×	✓
TumbleBit	✓	✓	✓	✓ (fees)	2 blocks	✓	✓

TABLE I. A comparison of Bitcoin Tumbler services. TTP stands for Trusted Third Party. We count minimum mixing time by the minimum number of Bitcoin blocks. Any mixing service inherently requires at least one block. [1]Coinparty could achieve some DoS resistance by forcing parties to solve puzzles before participating.

Performance. We have implemented our TumbleBit system in C++ and python, using LibreSSL as our cryptographic library. We have tumbled payments from 800 payers to 800 payees; the relevant transactions are visible on the blockchain. (See Section VIII-C).Our protocol requires 327 KB of data on the wire, and 0.6 seconds of computation on a single CPU. Thus, performance in classic tumbler mode is limited only by the time it takes for two blocks to be confirmed on the blockchain and the time it takes for transactions to be confirmed; currently, this takes ≈ 20 minutes. Meanwhile, off-blockchain payments can complete in seconds (Section VIII).

B. Related Work

TumbleBit is related to work proposing new anonymous cryptocurrencies (*e.g.,* Zerocash [42], [10], Monero [2] or Mimblewimble [30]). While these are very promising, they have yet to be as widely adopted as Bitcoin. On the other hand, TumbleBit is an anonymity service for Bitcoin's *existing* user base.

Off-blockchain payments. When used as an unlinkable payment hub, TumbleBit is related to micropayment channel networks, notably Duplex Micropayment Channels [18] and the Lightning Network [50]. These systems also allow for Bitcoin-backed fast off-blockchain payments. Payments are sent via paths of intermediaries with pre-established on-blockchain pairwise escrow transactions. TumbleBit (conceptually) does the same. However, while the intermediaries in micropayment channel network can link payments from $\mathcal{A}$ to $\mathcal{B}$, TumbleBit's intermediary $\mathcal{T}$ cannot.

Our earlier workshop paper [27] proposed a protocol that adds anonymity to micropayment channel networks. TumbleBit is implemented and Bitcoin compatible, while [27] is not. Moreover, [27] requires both Alice $\mathcal{A}$ and Bob $\mathcal{B}$ to interact with the Tumbler $\mathcal{T}$ as part of every off-blockchain payment. Thus, the Tumbler could correlate the timing of its interactions with $\mathcal{A}$ and $\mathcal{B}$ in order to link their payments. Meanwhile, TumbleBit eliminates this timing channel by only requiring interaction between $\mathcal{A}$ and $\mathcal{T}$ (and $\mathcal{A}$ and $\mathcal{B}$) during an off-blockchain payment (see Figure 1 and Section VII-B).

TumbleBit is also related to concurrent work proposing Bolt [25], an off-blockchain unlinkable payment channel. While TumbleBit is implemented and Bitcoin compatible, Bolt has not been implemented and employs scripting functionalities that are not available in Bitcoin. Instead, Bolt runs on top of Zerocash [42], [10].

Bolt operates in several modes, including a unidirectional payment channel (where Alice can pay Bob), a bidirectional payment channel (where Alice and Bob can pay each other), and bidirectional payment hub (where Alice and Bob can pay each other through an Intermediary). The latter mode is most relevant to TumbleBit, and offers different unlinkability properties. First, Bolt hides the denomination of the payment from the Intermediary; meanwhile, TumbleBit payments all have the same denomination, which is revealed to the Tumbler. Second, off-blockchain Bolt payments hide the identity of the payer and payee; meanwhile, off-blockchain TumbleBit payments reveals the identity of the payee (but not the payer) to the Tumbler. Finally, if a party aborts a payment via Bolt's bidirectional payment hub, then the identity of the payer and payee is revealed. In this case, Bolt must fall back on the anonymity properties of Zerocash, which ensures that on-blockchain identities are anonymous. Meanwhile, abort attacks on TumbleBit are less damaging (see Section VII-C). This is important because TumbleBit cannot fall back on Zerocash's anonymity properties.

Bitcoin Tumblers. Prior work on classic Bitcoin Tumblers is summarized in Table I-A.

Blindcoin [58], and its predecessor Mixcoin [15], use a trusted third party (TTP) to mix Bitcoin addresses. However, this third party can steal users' bitcoins; theft is detected but not prevented. In Mixcoin, the TTP can also violate anonymity. CoinSwap [38] is a fair-exchange mixer that allows two parties to anonymously send bitcoins through an intermediary. Fair exchange prevents the CoinSwap intermediary from stealing funds. Unlike TumbleBit, however, CoinSwap does not provide anonymity against even an honest-but-curious intermediary. Coinparty [60] is another decentralized solution, but it is secure only if $2/3$ of the users are honest.

CoinShuffle [53] and CoinShuffle++[43] build on CoinJoin [37] to provide a decentralized tumbler that prevents bitcoin theft. Their anonymity properties are analyzed in [41]. CoinShuffle(++) [53], [43] both perform a mix in a single transaction. Bitcoin's maximum transaction size (100KB) limits CoinShuffle(++) to 538 users per mix. These systems are also particularly vulnerable to DoS attacks, where a user joins the mix and then aborts, disrupting the protocol for all other users. Decentralization also requires mix users to interact via a peer-to-peer network in order to identify each other and mix payments. This coordination between users causes communication to grow quadratically [13], [14], limiting scalability; neither [53] nor [43] performs a mix with more than 50 users. Decentralization also makes it easy for an attacker to create many Sybils and trick Alice $\mathcal{A}$ into mixing with them in order to deanonymize her payments [14], [57]. TumbleBit sidesteps these scalability limitations by not requiring coordination between mix users.

XIM [13] builds on fair-exchange mixers like [8]. XIM prevents bitcoin theft, and uses fees to resist DoS and Sybil attacks—users must pay to participate in a mix, raising the bar for attackers that disrupt the protocol by joining the mix and then aborting. We use fees in TumbleBit as well. Also, an abort by a single XIM user does not disrupt the mix for others. TumbleBit also has this property. One of XIM's key innovations is a method for finding parties to participate in a mix. However, this adds several hours to the protocol, because users must advertise themselves as mix partners on the blockchain. TumbleBit is faster; a tumble requires only two blocks on the blockchain.

When used as a classic tumbler, TumbleBit and [27] shares the same fair-exchange properties and anonymity properties. However, unlike TumbleBit, [27] is not compatible with Bitcoin and does not provide an implementation. Also, [27] requires three blocks to be confirmed on the blockchain, while TumbleBit requires two.

After this paper was first posted, Dorier and Ficsor began an independent TumbleBit implementation.[3]

II. Bitcoin Scripts and Smart Contracts

In designing TumbleBit, our key challenge was ensuring compatibility with today's Bitcoin protocol. We therefore start by reviewing Bitcoin transactions and Bitcoin's non-Turing-complete language *Script*.

Transactions. A Bitcoin user Alice $\mathcal{A}$ is identified by her bitcoin address (which is a public ECDSA key), and her bitcoins are "stored" in *transactions*. A single transaction can have multiple *outputs* and multiple *inputs*. Bitcoins are transferred by sending the bitcoins held in the output of one transaction to the input of a different transaction. The blockchain exists to provide a public record of all valid transfers. The bitcoins held in a transaction output can only be transferred to a single transaction input. A transaction input T_3 *double-spends* a transaction input T_2 when both T_2 and T_3 *point to* (*i.e.,* attempt to transfer bitcoins from) the same transaction output T_1. The security of the Bitcoin protocol implies that double-spending transactions will not be confirmed on the blockchain. Transactions also include a *transaction fee* that is paid to the Bitcoin miner that confirms the transaction on the blockchain. Higher fees are paid for larger transactions. Indeed, fees for confirming transactions on the blockchain are typically expressed as "Satoshi-per-byte" of the transaction.

Scripts. Each transaction uses Script to determine the conditions under which the bitcoins held in that transaction can be moved to another transaction. We build "smart contracts" from the following transactions:
- T_{offer}: One party $\mathcal{A}$ offers to pay bitcoins to any party that can sign a transaction that meets some condition $\mathcal{C}$. The T_{offer} transaction is signed by $\mathcal{A}$.
- T_{fulfill}: This transaction points to T_{offer}, meets the condition $\mathcal{C}$ stipulated in T_{offer}, and contains the public key of the party $\mathcal{B}$ receiving the bitcoins.
T_{offer} is posted to the blockchain first. When T_{fulfill} is confirmed by the blockchain, the bitcoins in T_{fulfill} flow from the party signing transaction T_{offer} to the party signing T_{fulfill}. Bitcoin scripts support two types of conditions that involve cryptographic operations:

Hashing condition: The condition $\mathcal{C}$ stipulated in T_{offer} is: "T_{fulfill} must contain the preimage of value y computed under the hash function H." Then, T_{fulfill} collects the offered bitcoin by including a value x such that $H(x) = y$. (We use the `OP_RIPEMD160` opcode so that H is the RIPEMD-160 hash function.)

Signing condition: The condition $\mathcal{C}$ stipulated in T_{offer} is: "T_{fulfill} must be digitally signed by a signature that verifies under public key PK." Then, T_{fulfill} fulfills this condition if it is validly signed under PK. The signing condition is highly restrictive: (1) today's Bitcoin protocol requires the signature to be ECDSA over the Secp256k1 elliptic curve [51]—no other elliptic curves or types of signatures are supported, and (2) the condition specifically requires T_{fulfill} *itself* to be signed. Thus, one could not use the signing condition to build a contract whose condition requires an *arbitrary message* m to be signed by PK.[4] (TumbleBit uses the `OP_CHECKSIG` opcode, which requires verification of a single signature, and the "2-of-2 multisignature" template '`OP_2 key1 key2 OP_2 OP_CHECKMULTISIG`' which requires verification of a signature under `key1` AND a signature under `key2`.)[5]

Script supports composing conditions under "IF" and "ELSE". Script also supports *timelocking* (`OP_CHECKLOCKTIMEVERIFY` opcode [56]), where T_{offer} also stipulates that T_{fulfill} is timelocked to time window tw. (Note that tw is an absolute block height.) This allows the party that posted T_{fulfill} to reclaim their bitcoin if T_{fulfill} is unspent and the block height is

[3] https://github.com/NTumbleBit/NTumbleBit

[4] This is why [27] is not Bitcoin-compatible. [27] requires a blind signature to be computed over an arbitrary message. Also, ECDSA-Secp256k1 does not support blind signatures.

[5] Unlike cryptographic multisignatures, a Bitcoin 2-of-2 multisignature is a tuple of two distinct signatures and not a joint signature.

higher than tw. Section VIII-A details the scripts used in our implementation. See also Appendix I.

2-of-2 escrow. TumbleBit relies heavily on the commonly-used *2-of-2 escrow* smart contract. Suppose that Alice $\mathcal{A}$ wants to put Q bitcoin in escrow to be redeemed under the condition $\mathcal{C}_{2of2}$: "the fulfilling transaction includes two signatures: one under public key PK_1 AND one under PK_2."

To do so, $\mathcal{A}$ first creates a multisig address $PK_{(1,2)}$ for the keys PK_1 and PK_2 using the Bitcoin `createmultisig` command. Then, $\mathcal{A}$ posts an *escrow transaction* T_{escr} on the blockchain that sends Q bitcoin to this new multisig address $PK_{(1,2)}$. The T_{escr} transaction is essentially a T_{offer} transaction that requires the fulfilling transaction to meet condition $\mathcal{C}_{2of2}$. We call the fulfilling transaction T_{cash} the *cash-out transaction*. Given that $\mathcal{A}$ doesn't control both PK_1 and PK_2 (*i.e.,* doesn't know the corresponding secret keys), we also timelock the T_{escr} transaction for a time window tw. Thus, if a valid T_{cash} is not confirmed by the blockchain within time window tw, the escrowed bitcoins can be reclaimed by $\mathcal{A}$. Therefore, $\mathcal{A}$'s bitcoins are escrowed until either (1) the time window expires and $\mathcal{A}$ reclaims her bitcoins or (2) a valid T_{cash} is confirmed. TumbleBit uses 2-of-2 escrow to establish pairwise payment channels, per Figure 1.

III. TumbleBit: An Unlinkable Payment Hub

Our goal is to allow a *payer*, Alice $\mathcal{A}$, to unlinkably send 1 bitcoin to a *payee*, Bob $\mathcal{B}$. Naturally, if Alice $\mathcal{A}$ signed a regular Bitcoin transaction indicating that $Addr_A$ pays 1 bitcoin to $Addr_B$, then the blockchain would record a link between Alice $\mathcal{A}$ and Bob $\mathcal{B}$ and anonymity could be harmed using the techniques of [40], [52], [12]. Instead, TumbleBit funnels payments from multiple payer-payee pairs through the Tumbler $\mathcal{T}$, using cryptographic techniques to ensure that, as long as $\mathcal{T}$ does not collude with TumbleBit's users, then no one can link a payment from payer $\mathcal{A}$ to payee $\mathcal{B}$.

A. Overview of Bob's Interaction with the Tumbler

We overview TumbleBit's phases under the assumption that Bob $\mathcal{B}$ receives a single payment of value 1 bitcoin. TumbleBit's Anonymity properties require all payments made in the system to have the same denomination; we use 1 bitcoin for simplicity. Appendix A shows how Bob can receive multiple payments of denomination 1 bitcoin each.

TumbleBit has three phases (Fig 1). Off-blockchain TumbleBit payments take place during the middle *Payment Phase*, which can last for hours or even days. Meanwhile, the first *Escrow Phase* sets up payment channels, and the last *Cash-Out Phase* closes them down; these two phases require on-blockchain transactions. All users of TumbleBit know exactly when each phase begins and ends. One way to coordinate is to use block height; for instance, if the payment phase lasts for 1 day (*i.e.,* ≈ 144 blocks) then the Escrow Phase is when block height is divisible by 144, and the Cash-Out Phase is when blockheight$+1$ is divisible by 144.

1: Escrow Phase. Every Alice $\mathcal{A}$ that wants to send payments (and Bob $\mathcal{B}$ that wants to receive payments) during the upcoming Payment Phase runs the escrow phase with $\mathcal{T}$. The escrow phase has two parts:

(a) Payee $\mathcal{B}$ asks the Tumbler $\mathcal{T}$ to set up a payment channel. $\mathcal{T}$ escrows 1 bitcoin on the blockchain via a 2-of-2 escrow transaction (Section II) denoted as $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ stipulating that 1 bitcoin can be claimed by any transaction signed by both $\mathcal{T}$ and $\mathcal{B}$. $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ is timelocked to time window tw_2, after which $\mathcal{T}$ can reclaim its bitcoin. Similarly, the payeer $\mathcal{A}$ escrows 1 bitcoin in a 2-of-2 escrow with $\mathcal{T}$ denoted as $T_{\text{escr}(\mathcal{A},\mathcal{T})}$, timelocked for time window tw_1 such that $tw_1 < tw_2$. Upon conclusion of the puzzle-promise protocol both the escrows are established by confirming $T_{\text{escr}(\mathcal{A},\mathcal{T})}$, $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ on Bitcoin's blockchain.

(b) Bob $\mathcal{B}$ obtains a puzzle z through an off-blockchain cryptographic protocol with $\mathcal{T}$ which we call the *puzzle-promise protocol*. Conceptually, the output of this protocol is a promise by $\mathcal{T}$ to pay 1 bitcoin to $\mathcal{B}$ in exchange for the solution to a puzzle z. The puzzle z is just an RSA encryption of a value ϵ

$$z = f_{RSA}(\epsilon, e, N) = \epsilon^e \mod N \tag{1}$$

where (e, N) is the TumbleBit RSA public key of the Tumbler $\mathcal{T}$. "Solving the puzzle" is equivalent to decrypting z and thus obtaining its "solution" ϵ. Meanwhile, the "promise" c is a symmetric encryption under key ϵ

$$c = \text{Enc}_\epsilon(\sigma)$$

where σ is the Tumbler's ECDSA-Secp256k1 signature on the transaction $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ which transfers the bitcoin escrowed in $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ from $\mathcal{T}$ to $\mathcal{B}$. (We use ECDSA-Secp256k1 for compatibility with the Bitcoin protocol.) Thus, the solution to a puzzle z enables $\mathcal{B}$ to claim 1 bitcoin from $\mathcal{T}$. To prevent misbehavior by the Tumbler $\mathcal{T}$, our puzzle-promise protocol requires $\mathcal{T}$ to provide a proof that the puzzle solution ϵ is indeed the key which decrypts the promise ciphertext c. The details of this protocol, and its security guarantees, are in Section VI.

2: Payment Phase. Once Alice $\mathcal{A}$ indicates she is ready to pay Bob $\mathcal{B}$, Bob $\mathcal{B}$ chooses a random blinding factor $r \in Z_N^*$ and blinds the puzzle to

$$\overline{z} = r^e z \mod N. \tag{2}$$

Blinding ensures that even $\mathcal{T}$ cannot link the original puzzle z to its blinded version $\overline{z}$. Bob $\mathcal{B}$ then sends $\overline{z}$ to $\mathcal{A}$. Next, $\mathcal{A}$ solves the blinded puzzle $\overline{z}$ by interacting with $\mathcal{T}$. This *puzzle-solver protocol* is a fair exchange that ensures that $\mathcal{A}$ transfers 1 bitcoin to T iff $\mathcal{T}$ gives a valid solution to the puzzle $\overline{z}$. Finally, Alice $\mathcal{A}$ sends the solution to the blinded puzzle $\overline{\epsilon}$ back to Bob $\mathcal{B}$. Bob unblinds $\overline{\epsilon}$ to obtain the solution

$$\epsilon = \overline{\epsilon}/r \mod N \tag{3}$$

and accepts Alice's payment if the solution is valid, *i.e.,* $\epsilon^e = z \mod N$.

3: Cash-Out Phase. Bob $\mathcal{B}$ uses the puzzle solution ϵ to decrypt the ciphertext c. From the result $\mathcal{B}$ can create a transaction $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ that is signed by both $\mathcal{T}$ and $\mathcal{B}$. $\mathcal{B}$ posts $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ to the blockchain to receive 1 bitcoin from $\mathcal{T}$.

Our protocol crucially relies on the algebraic properties of RSA, and RSA blinding. To make sure that the Tumbler is using a valid RSA public key (e, N), TumbleBit also has an one-time setup phase:

0: Setup. Tumbler $\mathcal{T}$ announces its RSA public key (e, N) and Bitcoin address $Addr_{\mathcal{T}}$, together with a non-interactive zero-knowledge proof of knowledge π[6] of the corresponding RSA secret key d. Every user of TumbleBit validates (e, N) using π.

B. Overview of Alice's Interaction with the Tumbler

We now focus on the puzzle-solving protocol between $\mathcal{A}$ and the Tumbler $\mathcal{T}$ to show how TumbleBit allows $\mathcal{A}$ to make many off-blockchain payments via only *two* on-blockchain transactions (aiding scalability).

During the Escrow Phase, Alice opens a payment channel with the Tumbler $\mathcal{T}$ by escrowing Q bitcoins in an on-blockchain transaction $T_{\text{escr}(\mathcal{A},\mathcal{T})}$. Each escrowed bitcoin can pay $\mathcal{T}$ for the solution to one puzzle. Next, during the off-blockchain Payment Phase, $\mathcal{A}$ makes off-blockchain payments to $j \leq Q$ payees. Finally, during the Cash-Out Phase, Alice $\mathcal{A}$ pays the Tumbler $\mathcal{T}$ by posting a transaction $T_{\text{cash}(\mathcal{A},\mathcal{T})}(j)$ that reflects the new allocation of bitcoins; namely, that $\mathcal{T}$ holds j bitcoins, while $\mathcal{A}$ holds $Q - j$ bitcoins. The details of Alice $\mathcal{A}$'s interaction with $\mathcal{T}$, which are based on a technique used in micropayment channels [46, p. 86], are as follows:

1: Escrow Phase. Alice $\mathcal{A}$ posts a 2-of-2 escrow transaction $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ to the blockchain that escrows Q of Alice's bitcoins. If no valid transaction $T_{\text{cash}(\mathcal{A},\mathcal{T})}$ is posted before time window tw_1, then all Q escrowed bitcoins can be reclaimed by $\mathcal{A}$.

2: Payment Phase. Alice $\mathcal{A}$ uses her escrowed bitcoins to make off-blockchain payments to the Tumbler $\mathcal{T}$. For each payment, $\mathcal{A}$ and $\mathcal{T}$ engage in an off-blockchain puzzle-solver protocol (see Sections V-B,V-D).

Once the puzzle is solved, Alice signs and gives $\mathcal{T}$ a new transaction $T_{\text{cash}(\mathcal{A},\mathcal{T})}(i)$. $T_{\text{cash}(\mathcal{A},\mathcal{T})}(i)$ points to $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ and reflects the new balance between $\mathcal{A}$ and $\mathcal{T}$ (*i.e.,* that $\mathcal{T}$ holds i bitcoins while $\mathcal{A}$ holds $Q - i$ bitcoins). $\mathcal{T}$ collects a new $T_{\text{cash}(\mathcal{A},\mathcal{T})}(i)$ from $\mathcal{A}$ for each payment. If Alice refuses to sign $T_{\text{cash}(\mathcal{A},\mathcal{T})}(i)$, then the Tumbler refuses to help Alice solve further puzzles. Importantly, each $T_{\text{cash}(\mathcal{A},\mathcal{T})}(i)$ for $i = 1...j$ (for $j < Q$) is signed by Alice $\mathcal{A}$ but *not* by $\mathcal{T}$, and is *not* posted to the blockchain. At the end of the Payment Phase, $\mathcal{A}$ has made j payments, and the Tumbler $\mathcal{T}$ has transaction $T_{\text{cash}(\mathcal{A},\mathcal{T})}(j)$, signed by Alice $\mathcal{A}$, reflecting a balance of j bitcoins for $\mathcal{T}$ and $Q - j$ bitcoins for $\mathcal{T}$.

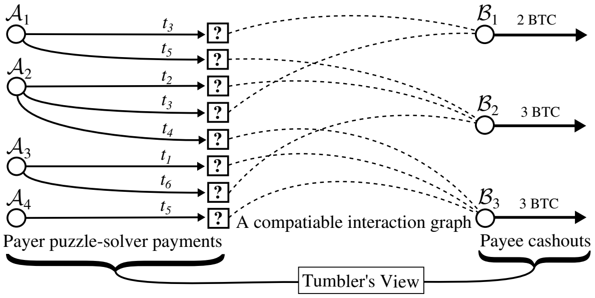

Fig. 2. Our unlinkability definition: The Tumblers view and a compatible interaction multi-graph.

3: Cash-Out Phase. The Tumbler $\mathcal{T}$ claims its bitcoins from $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ by signing $T_{\text{cash}(\mathcal{A},\mathcal{T})}(j)$ and posting it to the blockchain. This fulfills the condition in $T_{\text{escr}(\mathcal{A},\mathcal{T})}$, which stipulated that the escrowed coins be claimed by a transaction signed by *both* $\mathcal{A}$ and $\mathcal{T}$. (Notice that all the $T_{\text{cash}(\mathcal{A},\mathcal{T})}(i)$ point to the *same* escrow transaction $T_{\text{escr}(\mathcal{A},\mathcal{T})}$. The blockchain will therefore only confirm one of these transactions; otherwise, double spending would occur. Rationally, the Tumbler $\mathcal{T}$ always prefers to confirm $T_{\text{cash}(\mathcal{A},\mathcal{T})}(j)$ since it transfers the maximum number of bitcoins to $\mathcal{T}$.) Because $T_{\text{cash}(\mathcal{A},\mathcal{T})}(j)$ is the only transaction signed by the Tumbler $\mathcal{T}$, a cheating Alice cannot steal bitcoins by posting a transaction that allocates fewer than j bitcoins to the Tumbler $\mathcal{T}$.

Remark: Scaling Bitcoin. A similar (but more elaborate) technique can be applied between $\mathcal{B}$ and $\mathcal{T}$ so that only *two* on-blockchain transactions suffice for Bob $\mathcal{B}$ to receive an arbitrary number of off-blockchain payments. Details are in Appendix A. Given that each party uses *two* on-blockchain transactions to make multiple off-blockchain playments, Tumblebit helps Bitcoin scale.

C. TumbleBit's Security Properties

Unlinkability. We assume that the Tumbler $\mathcal{T}$ does not collude with other users. The *view* of $\mathcal{T}$ consists of (1) the set of escrow transactions established between (a) each payer $\mathcal{A}_j$ and the Tumbler ($\mathcal{A}_j \overset{\text{escrow},a_i}{\longrightarrow} \mathcal{T}$) of value a_i and (b) the Tumbler and each payee $\mathcal{B}_i$ ($\mathcal{T} \overset{\text{escrow},b_i}{\longrightarrow} \mathcal{B}_i$), (2) the set of puzzle-solver protocols completed with each payer $\mathcal{A}_j$ at time t during the Payment Phase, and (3) the set of cashout transactions made by each payer $\mathcal{A}_j$ and each payee $\mathcal{B}_i$ during the Cash-Out Phase.

[6] Such a proof could be provided using the GQ identification protocol [26] made non-interactive using the Fiat-Shamir heuristic [21] in the random oracle model.

An *interaction multi-graph* is a mapping of payments from payers to payees (Figure 2). For each successful puzzle-solver protocol completed by payer $\mathcal{A}_j$ at time t, this graph has an edge, labeled with time t, from $\mathcal{A}_j$ to some payee $\mathcal{B}_i$. An interaction graph is *compatible* if it explains the view of the Tumbler $\mathcal{T}$, *i.e.,* the number of edges incident on $\mathcal{B}_i$ is equal to the total number of bitcoins cashed out by $\mathcal{B}_i$. Unlinkability requires all compatible interaction graphs to be equally likely. Anonymity therefore depends on the number of compatible interaction graphs.

Notice that payees $\mathcal{B}_i$ have better anonymity than payers $\mathcal{A}_j$. (This follows because the Tumbler $\mathcal{T}$ knows the time t at which payer $\mathcal{A}_j$ makes each payment. Meanwhile, the Tumbler $\mathcal{T}$ only knows the aggregate amount of bitcoins cashed-out by each payee $\mathcal{B}_i$.)

A high-level proof of TumbleBit's unlinkability is in Section VII, and the limitations of unlinkability are discussed in Section VII-C.

Balance. The system should not be exploited to print new money or steal money, *even when parties collude*. As in [25], we call this property *balance*, which establishes that no party should be able to cash-out more bitcoins than what is dictated by the payments that were successfully completed in the Payment Phase. We discuss how TumbleBit satisfies balance in Section VII.

DoS and Sybil protection. TumbleBit uses transaction fees to resist DoS and Sybil attacks. Every Bitcoin transaction can include a *transaction fee* that is paid to the Bitcoin miner who confirms the transaction on the blockchain as an incentive to confirm transactions. However, because the Tumbler $\mathcal{T}$ does not trust Alice $\mathcal{A}$ and Bob $\mathcal{B}$, $\mathcal{T}$ should not be expected to pay fees on the transactions posted during the Escrow Phase. To this end, when Alice $\mathcal{A}$ establishes a payment channel with $\mathcal{T}$, she pays for both the Q escrowed in transaction $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ and for its transaction fees. Meanwhile, when the Tumbler $\mathcal{T}$ and Bob $\mathcal{B}$ establish a payment channel, the Q escrowed bitcoins in $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ are paid in the Tumbler $\mathcal{T}$, but the transaction fees are paid by Bob $\mathcal{B}$ (Section III-A). Per [13], fees raise the cost of an DoS attack where $\mathcal{B}$ starts and aborts many parallel sessions, locking $\mathcal{T}$'s bitcoins in escrow transactions. This similarly provides Sybil resistance, making it expensive for an adversary to harm anonymity by tricking a user into entering a run of TumbleBit where all other users are Sybils under the adversary's control.

IV. TumbleBit: Also a Classic Tumbler.

We can also operate TumbleBit as classic Bitcoin Tumbler. As a classic Tumbler, TumbleBit operates in epoches, each of which (roughly) requires two blocks to be confirmed on the blockchain ($\approx$ 20 mins). During each epoch, there are exactly $\aleph$ distinct bitcoin addresses making payments (payers) and $\aleph$ bitcoin addresses receiving payments (payees). Each payment is of denomination 1 bitcoin, and the mapping from payers to payees is a bijection. During one epoch, the protocol itself is identical to that in Section III with the following changes: (1) the duration of the Payment Phase shrinks to seconds (rather than hours or days); (2) each payment channel escrows exactly $Q = 1$ bitcoin; and (3) every payee Bob $\mathcal{B}$ receives payments at an ephemeral bitcoin address $Addr_B$ chosen freshly for the epoch.

A. Anonymity Properties

As a classic tumbler, TumbleBit has the same *balance* property, but stronger anonymity: *k-anonymity within an epoch* [27], [13]. Specifically, while the blockchain reveals which payers and payees participated in an epoch, no one (not even the Tumbler $\mathcal{T}$) can tell which payer paid which payee during that specific epoch. Thus, if k payments successfully completed during an epoch, the anonymity set is of size k. (This stronger property follows directly from our unlinkability definition (Section III-C): there are k compatible interaction graphs because the interaction graph is bijection.)

Recovery from anonymity failures. It's not always the case that $k = \aleph$. The exact anonymity level achieved in an epoch can be established only after its Cash-Out Phase. For instance, anonymity is reduced to $k = \aleph - 1$ if $\mathcal{T}$ aborts an payment made by payer $\mathcal{A}_j$. We deal with this by requiring $\mathcal{B}$ to uses an *ephemeral* Bitcoin address $Addr_B$ in each epoch. Consider first a malicious Tumbler $\mathcal{T}$ that behaves itself during the Escrow Phase of some epoch, but then refuses to help some payer $\mathcal{A}$ solve a puzzle during the Payment Phase. The payment from $\mathcal{A}$ to its payee $\mathcal{B}$ will fail. As such, the payee $\mathcal{B}$ will not be able to claim a bitcoin from $\mathcal{T}$ during the Cash-Out Phase. It follows that the Tumbler can trivially link $\mathcal{A}$ and $\mathcal{B}$ by identifying the payee that failed to cash out. To recover from this, we follow [27] and require $\mathcal{B}$ to discard his ephemeral address and never use it again if $\mathcal{T}$ aborts the protocol. Note that both $\mathcal{B}$ loses nothing in this case, since no funds have been transferred from $\mathcal{T}$ to $\mathcal{B}$. Also, $\mathcal{A}$ loses nothing, since by the fair-exchange property of the puzzle-solver protocol (Theorem 1) $\mathcal{T}$ only obtains a bitcoin from $\mathcal{A}$ if it cooperated in solving the puzzle.

Let us now consider a non-aborting epoch with a small anonymity set. If $\mathcal{B}$ is comfortable with the size of his anonymity set, he can use standard Bitcoin transactions to move the bitcoin from his ephemeral address to his long-lived Bitcoin address. Otherwise, if he thinks that the anonymity set is too small, $\mathcal{B}$ can *remix*, *i.e.,* choose a new fresh ephemeral address $Addr'_B$ and rerun the protocol where his old ephemeral address $Addr_B$ pays his new ephemeral address $Addr'_B$. Remixing can continue until $\mathcal{B}$ is happy with the size of his anonymity set, and he transfers his funds to his long-lived address.

Remark: Intersection attacks. While this notion of k-anonymity is commonly used in Bitcoin tumblers (*e.g.,* [13], [27]), it does suffer from the following weakness. Any adversary that observes the transactions posted to the blockchain within one epoch can learn which payers and payees participated in that epoch. Then, this information can be correlated to de-anonymize users across epochs (*e.g.,* using frequency analysis or techniques used to break k-anonymity [22]). These 'intersection attacks' follow because k-anonymity is composed across epochs; see also [13], [41] for discussion.

DoS and Sybil Attacks. We use fees to resist DoS and Sybil attacks. Alice again pays for both the Q escrowed in transaction $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ and for its transaction fees. However, we run into a problem if we want Bob $\mathcal{B}$ to pay the fee on the escrow transaction $T_{\text{escr}(\mathcal{T},\mathcal{B})}$. Because Bob $\mathcal{B}$ uses a freshly-chosen Bitcoin address $Addr_B$, that is not linked to any prior transaction on the blockchain, $Addr_B$ cannot hold any bitcoins. Thus, Bob $\mathcal{B}$ will have to pay the Tumbler $\mathcal{T}$ out of band. The anonymous fee vouchers described in [27] provide one way to address this, which also has the additional feature that payers $\mathcal{A}$ cover all fees.

An anonymous fee voucher is a blind signature $\overline{\sigma}$ that $\mathcal{T}$ provides to A in exchange for a small out-of-band payment; $\mathcal{A}$ could pre-purchase these vouchers in bulk, before she begins participating in TumbleBit. Then, when $\mathcal{A}$ is ready to participate, she unblinds $\overline{\sigma}$ to σ and provides it to $\mathcal{B}$ who passes it along to $\mathcal{T}$. The protocol begins once $\mathcal{T}$ is sure that it was paid for its efforts.

V. A Fair Exchange for RSA Puzzle Solving

We now explain how to realize a Bitcoin-compatible *fair-exchange* where Alice $\mathcal{A}$ pays Tumbler $\mathcal{T}$ one bitcoin iff the $\mathcal{T}$ provides a valid solution to an RSA puzzle. The Tumbler $\mathcal{T}$ has an RSA secret key d and the corresponding public key (e, N). The RSA puzzle y is provided by Alice, and its solution is an RSA secret-key exponentiation

$$\epsilon = f_{RSA}^{-1}(y, d, N) = y^d \bmod N \tag{4}$$

The puzzle solution is essentially an RSA decryption or RSA signing operation.

This protocol is at the heart of TumbleBit's Payment Phase. However, we also think that this protocol is of independent interest, since there is also a growing interest in techniques that can fairly exchange a bitcoin for the solution to a computational "puzzle". We therefore start by surveying the literature in Section V-A. Section V-B presents our RSA-puzzle-solver protocol as a stand-alone protocol that requires two blocks to be confirmed on the blockchain. Our protocol is fast—solving 2048-bit RSA puzzles faster than [39]'s protocol for solving 16x16 Sudoku puzzles (Section VIII)). Also, the use of RSA means that our protocol supports solving *blinded* puzzles (see equation (2)), and thus can be used to create an unlinkable payment scheme. Section V-D shows how our protocol is integrated into TumbleBit's Payment Phase. Implementation results are in Table II of Section VIII-B.

A. Approaches from the Literature

Contingent payments. Maxwell described a protocol for "zero-knowledge contingent payments" (ZKCP) [36]. The scheme in [36] swaps one bitcoin from Alice $\mathcal{A}$ in exchange for having $\mathcal{T}$ compute any agreed-upon function f on an input of $\mathcal{A}$'s choosing. The idea is as follows. After $\mathcal{T}$ computes the result $f(y)$ on Alice's input y, it encrypts the result under a randomly chosen key k to obtain a ciphertext c, and hashes the encryption key to obtain $h = H(k)$. $\mathcal{T}$ then sends Alice $\mathcal{A}$ the ciphertext c and hash h along with a zero-knowledge (ZK) proof that they were formed correctly. (This proof must been done in zero knowledge, because $\mathcal{T}$ should not reveal the key k that decrypts $f(y)$ to $\mathcal{A}$ before being paid with $\mathcal{A}$'s bitcoin.) After $\mathcal{A}$ verifies the proof, $\mathcal{A}$ posts a transaction T_{puzzle} offering one bitcoin under condition: "T_{solve} must contain the hash preimage of h". $\mathcal{T}$ claims the bitcoin by posting a transaction T_{solve} containing k. Now $\mathcal{A}$ can use k to decrypt c to obtain her desired output $f(y)$. This realizes a fair exchange because the offered bitcoin reverts back to $\mathcal{A}$ if $\mathcal{T}$ fails to post a valid T_{solve} in a timely manner.

The limitations of using ZKCP in this setting arise due to the inefficiency of the instantiations of ZK proofs. Two main approaches exist:

ZKCP via ZK-Snarks. Recently, [39] showed how to instantiate the ZK proofs used in this protocol with ZK-Snarks [11]. The function f was a 16x16 Sudoku puzzle and the resulting protocol was run and completed within 20 seconds. We could use this approach in our setting by (1) letting f be f_{RSA}^{-1}, an RSA decryption/signature, and (2) using [39]'s ZK-snark but replacing the verification of the Sudoku puzzle with an RSA verification f_{RSA}. One disadvantage of this approach is that RSA verification within a ZK-Snark is likely to be slower than Sudoku puzzle verification because state-of-the-art ZK-Snarks operate in prime order fields of (roughly) 254 bits. Since a 2048-bit RSA verification deals with 2048-bit numbers, each such number has to be split up and expressed as an array of smaller ones, making arithmetic operations far more complicated [19]. In any case, our protocol for RSA exponentiation is faster than [39]'s protocol for 16x16 Sudoku puzzles (Section VIII). Also, ZK-Snarks are only secure under less standard cryptographic assumptions. Meanwhile, our protocol's security follows from the standard RSA assumption (in the random oracle model).

ZKCP via Garbled Circuits. As an alternative to ZK-Snarks, one could use more generic ZK proofs based on zero-knowledge garbled circuits (GC) as shown in [29]. While GC-based ZK proofs work reasonably well for evaluating hash functions, they are computationally heavier for modular exponentiations (like RSA verification) because the latter do not have a short Boolean-circuit representation [31].

ZKCP via Proof-of-Knowledge. A concurrent work [7] proposes proposing ZKCP using Bitcoin Script's Signing Condition (Section II) rather than its Hashing Condition. [7] aims to be generic to all problems that have zero-knowledge proof-of-knowledge protocols. Our approach is customized to RSA puzzles. [7] needs about 2 minutes to sell a 1024-bit RSA factorization with a cheating probability of 2^{-48}. Our approach is faster, solving 2048-bit RSA puzzles in seconds with cheating probability of $\approx 2^{-80}$.

Incentivizing correct computation. [32] proposed a different approach that also uses GCs. Two parties use GCs to compute an arbitrary function $g(a, b)$ without revealing their respective inputs a and b. [32]'s protocol has the added feature that if one party aborts before the output is revealed, the other party is automatically compensated with bitcoins. To use this protocol in our setting, the function g should be f_{RSA}^{-1}, input a is the RSA secret key d of $\mathcal{T}$, and b becomes the input y chosen by $\mathcal{A}$. Then, if $\mathcal{T}$ aborts the protocol before $\mathcal{A}$ learns the output, the bitcoin offered by $\mathcal{A}$ can be reclaimed by $\mathcal{A}$. Again, however, the efficiency of this approach is limited by the computational overhead of performing modular exponentiations inside a GC.

Our protocol sidesteps these issues by avoiding ZK proofs and GCs altogether.

B. Our (Stand-Alone) RSA-Puzzle-Solver Protocol

The following stand-alone protocol description assumes Alice $\mathcal{A}$ wants to transfer 1 bitcoin in exchange for one puzzle solution. Section V-D shows how to support the transfer of up to Q bitcoins for Q puzzle solutions (where each solution is worth 1 bitcoin).

The core idea is similar to that of contingent payments [36]: Tumbler $\mathcal{T}$ solves Alice's $\mathcal{A}$'s puzzle y by computing the solution $y^d \mod N$, then encrypts the solution under a randomly chosen key k to obtain a ciphertext c, hashes the key k under bitcoin's hash as $h = H(k)$ and finally, provides (c, h) to Alice. Alice $\mathcal{A}$ prepares T_{puzzle} offering one bitcoin in exchange for the preimage of h. Tumbler $\mathcal{T}$ earns the bitcoin by posting a transaction T_{solve} that contains k, the preimage of h, and thus fulfills the condition in T_{puzzle} and claims a bitcoin for $\mathcal{T}$. Alice $\mathcal{A}$ learns k from T_{solve}, and uses k to decrypt c and obtain the solution to her puzzle.

Our challenge is to find a mechanism that allows $\mathcal{A}$ to validate that c is the encryption of the correct value, without using ZK proofs. We do so by applying the *cut-and-choose technique* and exploiting the blinding properties of RSA. (We follow the blueprint of Lindell's recent work [35]. Roughly, a malicious party can only cheat if *all* of the "opened" values are correct and *all* of the "hidden" ones are incorrect. This allows us to use fewer values in order to more efficiently achieve a better security level.)

Thus, instead of asking $\mathcal{T}$ to provide just *one* (c, h) pair, $\mathcal{T}$ will be asked to provide $m + n$ pairs (Step 3). Then, we use cut and choose: $\mathcal{A}$ asks $\mathcal{T}$ to "open" n of these pairs, by revealing the randomly-chosen keys k_i's used to create each of the n pairs (Step 7). For a malicious $\mathcal{T}$ to successfully cheat $\mathcal{A}$, it would have to correctly guess all the n "challenge" pairs and form them properly (so it does not get caught), while at the same time malforming *all* the m unopened pairs (so it can claim a bitcoin from $\mathcal{A}$ without actually solving the puzzle). Since $\mathcal{T}$ cannot predict which pairs $\mathcal{A}$ asks it to open, $\mathcal{T}$ can only cheat with very low probability $1/\binom{m+n}{n}$.

However, we have a problem. Why should $\mathcal{T}$ agree to open *any* of the (c, h) values that it produced? If $\mathcal{A}$ received the opening of a correctly formed (c, h) pair, she would be able to obtain a puzzle solution without paying a bitcoin. As such, we introduce the notion of "fake values". Specifically, the n (c, h)-pairs that $\mathcal{A}$ asks $\mathcal{T}$ to open will open to "fake values" rather than "real" puzzles. Before $\mathcal{T}$ agrees to open them (Step 7), $\mathcal{A}$ must prove that these n values are indeed fake (Step 6).

We must also ensure that $\mathcal{T}$ cannot distinguish "real puzzles" from "fake values". We do this with RSA blinding. The real puzzle y is blinded m times with different RSA-blinding factors (Step 1), while the n fake values are RSA-blinded as well (Step 2). Finally, $\mathcal{A}$ randomly permutes the real and fake values (Step 3).

Once Alice confirms the correctness of the opened "fake" (c, h) values (Step 7), she signs a transaction T_{puzzle} offering one bitcoin for the keys k that open all of the m "real" (c, h) values (Step 8). But what if Alice cheated, so that each of the "real" (c, h) values opened to the solution to a *different* puzzle? This would not be fair to $\mathcal{T}$, since $\mathcal{A}$ has only paid for the solution to a single puzzle, but has tricked $\mathcal{T}$ into solving multiple puzzles. We solve this problem in Step 9: once $\mathcal{A}$ posts T_{puzzle}, she proves to $\mathcal{T}$ that all m "real" values open to *the same* puzzle y. This is done by revealing the RSA-blinding factors blinding puzzle y. Once $\mathcal{T}$ verifies this, $\mathcal{T}$ agrees to post T_{solve} which reveals m of the k values that open "real" (c, h) pairs (Step 10). $\mathcal{A}$ gets a valid solution to puzzle y if at *least one* of the real (c, h) pairs is validly formed (Step 11).

C. Fair Exchange

Fair exchange exchange entails the following: (1) *Fairness for $\mathcal{T}$:* After one execution of the protocol $\mathcal{A}$ will learn the correct solution $y^d \mod N$ to at most one puzzle y of her choice. (2) *Fairness for $\mathcal{A}$:* $\mathcal{T}$ will earn 1 bitcoin iff $\mathcal{A}$ obtains a correct solution.

Public input: (e, N).

π proves validity of (e, N) in a one-time-only setup phase.

Alice $\mathcal{A}$ Input: Puzzle y		Tumbler $\mathcal{T}$ **Secret** input: d
1. Prepare Real Puzzles R For $j \in [m]$, pick $r_j \in \mathbf{Z}_N^*$ $d_j \leftarrow y \cdot (r_j)^e \mod N$		
2. Prepare Fake Values F For $i \in [n]$, pick $\rho_i \in \mathbf{Z}_N^*$ $\delta_i \leftarrow (\rho_i)^e \mod N$		
3. Mix Sets. Randomly permute $\{d_1 \dots d_m, \delta_1 \dots \delta_n\}$ to $\{\beta_1 \dots \beta_{m+n}\}$ Let R be the indices of the d_i Let F be the indices of the δ_i	$\xrightarrow{\beta_1 \dots \beta_{m+n}}$	4. Evaluation For $i = 1 \dots m+n$ Evaluate β_i: $s_i = \beta_i^d \mod N$ Encrypt the result s_i: – Choose random $k_i \in \{0,1\}^{\lambda_1}$ – $c_i = H^{\mathsf{prg}}(k_i) \oplus s_i$ Commit to the keys: $h_i = H(k_i)$
	$\xleftarrow{c_1, \dots, c_{m+n}}$ $\xleftarrow{h_1, \dots, h_{m+n}}$	
5. Identify Fake Set F	$\xrightarrow{F, \rho_i\ \forall i \in F}$	6. Check Fake Set F For all $i \in F$: Verify $\beta_i = (\rho_i)^e \mod N$, If yes, reveal $k_i\ \forall i \in [F]$. Else abort.
7. Check Fake Set F For all $i \in F$, Verify that $h_i = H(k_i)$ Decrypt $s_i = H^{\mathsf{prg}}(k_i) \oplus c_i$ Verify $s_i = \rho_i \mod N$ Abort if any check fails.	$\xleftarrow{k_i\ \forall i \in F}$	
8. Post transaction T_{puzzle} T_{puzzle} offers 1 bitcoin within timewindow tw_1 under condition "the fulfilling transaction is signed by $\mathcal{T}$ and has preimages of $h_j\ \forall j \in R$".		
	$\xrightarrow{y,\ \ r_j \forall j \in R}$	9. Check β_j unblind to $y\ \forall j \in R$ For all $j \in R$ Verify $\beta_j = y \cdot (r_j)^e \mod N$ If not, abort.
		10. Post transaction T_{solve} T_{solve} contains $k_j \forall j \in R$
11. Obtain Puzzle Solution For $j \in R$: Learn k_j from T_{solve} Decrypt c_j to $s_j = H^{\mathsf{prg}}(k_j) \oplus c_j$ If s_j is s.t. $(s_j)^e = \beta_j \mod N$, Obtain solution $s_j / r_j \mod N$ which is $y^d \mod N$.		

Fig. 3. RSA puzzle solving protocol. H and H^{prg} are modeled as random oracles. In our implementation, H is RIPEMD-160, and H^{prg} is ChaCha20 with a 128-bit key, so that $\lambda_1 = 128$.

We prove this using the real-ideal paradigm [23]. We call the ideal functionality $\mathcal{F}_{\textsf{fair-RSA}}$ and present it in Appendix C. $\mathcal{F}_{\textsf{fair-RSA}}$ acts like a trusted party between $\mathcal{A}$ and $\mathcal{T}$. $\mathcal{F}_{\textsf{fair-RSA}}$ gets a puzzle-solving request (y, 1 bitcoin) from $\mathcal{A}$, and forwards the request to $\mathcal{T}$. If $\mathcal{T}$ agrees to solve puzzle y for $\mathcal{A}$, then $\mathcal{T}$ gets 1 bitcoin and $\mathcal{A}$ gets the puzzle solution. Otherwise, if $\mathcal{T}$ refuses, $\mathcal{A}$ gets 1 bitcoin back, and $\mathcal{T}$ gets nothing. Fairness for $\mathcal{T}$ is captured because $\mathcal{A}$ can request a puzzle solution only if she sends 1 bitcoin to $\mathcal{F}_{\textsf{fair-RSA}}$. Fairness for $\mathcal{B}$ is captured because $\mathcal{T}$ receives 1 bitcoin only if he agrees to solve the puzzle. The following theorem is proved in Appendix E:

Theorem 1: Let λ be the security parameter, m, n be statistical security parameters, let $N > 2^{\lambda}$. Let π be a publicly verifiable zero-knowledge proof of knowledge in the random oracle model. If the RSA assumption holds in $\mathbf{Z}_N^*$, and if functions $H^{\textsf{prg}}, H$ are independent random oracles, there exists a negligible function ν, such that protocol in Figure 3 securely realizes $\mathcal{F}_{\textsf{fair-RSA}}$ in the random oracle model with the following security guarantees. The security for $\mathcal{T}$ is $1-\nu(\lambda)$ while security for $\mathcal{A}$ is $1 - \frac{1}{\binom{m+n}{n}} - \nu(\lambda)$.

D. Solving Many Puzzles and Moving Off-Blockchain

To integrate the protocol in Figure 3 into TumbleBit, we have to deal with three issues. First, if TumbleBit is to scale Bitcoin (Section III-B), then Alice $\mathcal{A}$ must be able to use only *two* on-blockchain transactions $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ and $T_{\text{cash}(\mathcal{A},\mathcal{T})}$ to pay for the an *arbitrary* number of Q puzzle solutions (each worth 1 bitcoin) during the Payment Phase; the protocol in Figure 3 only allows for the solution to a single puzzle. Second, per Section III-B, the puzzle-solving protocol should occur entirely off-blockchain; the protocol in Figure 3 uses two *on-blockchain* transactions T_{puzzle} and T_{solve}. Third, the T_{solve} transactions are longer than typical transactions (since they contain m hash preimages), and thus require higher transaction fees.

To deal with these issues, we now present a fair-exchange protocol that uses only *two* on-blockchain transactions to solve an *arbitrary* number of RSA puzzles.

Escrow Phase. Before puzzle solving begins, Alice posts a 2-of-2 escrow transaction $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ to the blockchain that escrows Q bitcoins, (per Section III-B). $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ is timelocked to time window tw_1, and stipulates that the escrowed bitcoins can be transferred to a transaction signed by both $\mathcal{A}$ and $\mathcal{T}$.

Payment Phase. Alice $\mathcal{A}$ can buy solutions for up to Q puzzles, paying 1 bitcoin for each. Tumbler $\mathcal{T}$ keeps a counter of how many puzzles it has solved for $\mathcal{A}$, making sure that the counter does not exceed Q. When $\mathcal{A}$ wants her i^{th} puzzle solved, she runs the protocol in Figure 3 with the following modifications after Step 8 (so that it runs entirely off-blockchain):

(1) Because the Payment Phase is off-blockchain, transaction T_{puzzle} from Figure 3 is *not posted* to the blockchain. Instead, Alice $\mathcal{A}$ forms and signs transaction T_{puzzle} and sends it to the Tumbler $\mathcal{T}$. Importantly, Tumbler $\mathcal{T}$ does *not* sign or post this transaction yet.

(2) Transaction T_{puzzle} points to the escrow transaction $T_{\text{escr}(\mathcal{A},\mathcal{T})}$; T_{puzzle} changes its balance so that $\mathcal{T}$ holds i bitcoin and Alice $\mathcal{A}$ holds $Q-i$ bitcoins. T_{puzzle} is timelocked to time window $tw1$ and stipulates the same condition in Figure 3: "the fulfilling transaction is signed by T and has preimages of $h_j \forall j \in R$."

(Suppose that $\mathcal{T}$ deviates from this protocol, and instead immediately signs and post T_{puzzle}. Then the bitcoins in $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ would be transferred to T_{puzzle}. However, these bitcoins would remain locked in T_{puzzle} until either (a) the timelock tw expired, at which point Alice $\mathcal{A}$ could reclaim her bitcoins, or (b) $\mathcal{T}$ signs and posts a transaction fulfilling the condition in T_{puzzle}, which allows Alice to obtain the solution to her puzzle.)

(3) Instead of revealing the preimages $k_j \forall j \in R$ in an on-blockchain transaction T_{solve} as in Figure 3, the Tumbler $\mathcal{T}$ just sends the preimages directly to Alice.

(4) Finally, Alice $\mathcal{A}$ checks that the preimages open a valid puzzle solution. If so, Alice signs a regular cash-out transaction $T_{\text{cash}(\mathcal{A},\mathcal{T})}$ (per Section III-B). $T_{\text{cash}(\mathcal{A},\mathcal{T})}$ points to the escrow transaction $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ and reflects the new balance between $\mathcal{A}$ and $\mathcal{T}$.

At the end of the i^{th} payment, the Tumbler $\mathcal{T}$ should have two new signed transactions from Alice: $T_{\text{puzzle}}(i)$ and $T_{\text{cash}(\mathcal{A},\mathcal{T})}(i)$, each reflecting the (same) balance of bitcoins between $\mathcal{T}$ (holding i bitcoins) and $\mathcal{A}$ (holding $Q-i$ bitcoins). However, Alice $\mathcal{A}$ already has her puzzle solution at this point (step (4) modification above). What if she refuses to sign $T_{\text{cash}(\mathcal{A},\mathcal{T})}(i)$?

In this case, the Tumbler immediately begins to cash out, even without waiting for the Cash-Out Phase. Specifically, Tumbler $\mathcal{T}$ holds transaction $T_{\text{puzzle}}(i)$, signed by $\mathcal{A}$, which reflects a correct balance of i bitcoins to $\mathcal{T}$ and $Q - i$ bitcoins to $\mathcal{A}$. Thus, $\mathcal{T}$ signs $T_{\text{puzzle}}(i)$ and posts it to the blockchain. Then, $\mathcal{T}$ claims the bitcoins locked in $T_{\text{puzzle}}(i)$ by signing and posting transaction T_{solve}. As in Figure 3, T_{solve} fulfills T_{puzzle} by containing the m preimages $k_j \forall j \in R$. The bitcoin in $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ will be transferred to T_{puzzle} and then to T_{solve} and thus to the Tumbler $\mathcal{T}$. The only harm done is that $\mathcal{T}$ posts two longer transactions $T_{\text{puzzle}}(i)$, $T_{\text{solve}}(i)$ (instead of just $T_{\text{cash}(\mathcal{A},\mathcal{T})}$), which require higher fees to be confirmed on the blockchain. (Indeed, this is why we have introduced the $T_{\text{cash}(\mathcal{A},\mathcal{T})}(i)$ transaction.)

Cash-Out Phase. Alice has j puzzle solutions once the the Payment Phase is over and the Cash-Out Phase begins. If the Tumbler $\mathcal{T}$ has a transaction $T_{\text{cash}(\mathcal{A},\mathcal{T})}(j)$ signed by Alice, the Tumbler $\mathcal{T}$ just signs and post this transaction to the blockchain, claiming its j bitcoins.

We implement this protocol and evaluate its performance in Table II and Section VIII-B.

VI. Puzzle-Promise Protocol

We present the puzzle-promise protocol run between $\mathcal{B}$ and $\mathcal{T}$ in the Escrow Phase. Recall from Section III-A, that the goal of this protocol is to provide Bob $\mathcal{B}$ with a puzzle-promise pair (c, z). The "promise" c is an encryption (under key ϵ) of the Tumbler's ECDSA-Secp256k1 signature σ on the transaction $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ which transfers the bitcoin escrowed in $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ from $\mathcal{T}$ to $\mathcal{B}$. Meanwhile the RSA-puzzle z hides the encryption key ϵ per equation (1).

If Tumbler $\mathcal{T}$ just sent a pair (c, z) to Bob, then Bob has no guarantee that the promise c is actually encrypting the correct signature, or that z is actually hiding the correct encryption key. On the other hand, $\mathcal{T}$ cannot just reveal the signature σ directly, because Bob could use σ to claim the bitcoin escrowed in $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ without actually being paid (off-blockchain) by Alice $\mathcal{A}$ during TumbleBit's Payment Phase.

To solve this problem, we again use cut and choose: we ask $\mathcal{T}$ to compute many puzzle-promise pairs (c_i, z_i), and have Bob $\mathcal{B}$ test that some of the pairs are computed correctly. As in Section V-B, we use "fake" transactions (that will be "opened" and used only to check if the other party has cheated) and "real" transactions (that remain "unopened" and result in correctly-formed puzzle-promise pairs). Cut-and-choose guarantees that $\mathcal{B}$ knows that *at least one* of the unopened pairs is correctly formed. However, how does $\mathcal{B}$ know which puzzle z_i is correctly formed? Importantly, $\mathcal{B}$ can only choose one puzzle z_i that he will ask Alice $\mathcal{A}$ to solve during TumbleBit's Payment Phase (Section III-A). To deal with this, we introduce an *RSA quotient-chain technique* that ties together all puzzles z_i so that solving one puzzle z_{j1} gives the solution to all other puzzles.

In this section, we assume that $\mathcal{B}$ wishes to obtain only a single payment of denomination 1 bitcoin; the protocol as described in Figure 4 and Section VI-A suffices to run TumbleBit as a classic tumbler. We discuss its security properties in Section VI-B and implementation in Section VIII-B. In Appendix A and Figure 6, we show how to modify this protocol so that it allows $\mathcal{B}$ to receive arbitrary number of Q off-blockchain payments using only two on-blockchain transactions.

A. Protocol Walk Through

$\mathcal{B}$ prepares μ distinct "real" transactions and η "fake" transactions, hides them by hashing them with H' (Step 2-3), permutes them (Step 4), and finally sends them to $\mathcal{T}$ as $\beta_1, ..., \beta_{m+n}$. In Step 5, $\mathcal{T}$ signs each β_i to create an ECDSA-Secp256k1 signature σ_i. Each σ_i is then hidden inside an promise c_i which can decrypted with key ϵ_i. Finally, $\mathcal{T}$ hides each ϵ_i (the encryption keys) in an RSA puzzle z_i per equation (1). As each ϵ_i is uniformly chosen at random, puzzle z_i computationally hides its solution ϵ_i, under the RSA assumption[7].

Next, $\mathcal{B}$ needs to check that the η "fake" (c_i, z_i) pairs are correctly formed by $\mathcal{T}$ (Step 8). To do this, $\mathcal{B}$ needs $\mathcal{T}$ to provide the solutions ϵ_i to the puzzles z_i in fake pairs. $\mathcal{T}$ reveals these solutions only after $\mathcal{B}$ has proved that the η pairs really are fake (Step 7). Once this check is done, $\mathcal{B}$ knows that $\mathcal{T}$ can cheat with probability less than $1/\binom{\mu+\eta}{\eta}$.

Now we need our new trick. We want to ensure that if *at least one* of the "real" (c_i, z_i) pairs opens to a valid ECDSA-Secp256k1 signature σ_i, then just one puzzle solution ϵ_i with $i \in R$, can be used to open this pair. (We need this because $\mathcal{B}$ must decide which puzzle z_i to give to the payer $\mathcal{A}$ for decryption without knowing which pair (c_i, z_i) is validly formed.) We solve this by having $\mathcal{T}$ provide $\mathcal{B}$ with $\mu - 1$ quotients (Step 9).

$$q_2 = \frac{\epsilon_{j_2}}{\epsilon_{j_1}}, \quad \ldots, \quad q_\mu = \frac{\epsilon_{j_\mu}}{\epsilon_{j_{\mu-1}}} \mod N$$

where $\{j_1, \ldots, j_\mu\} = R$ are the indices for the "real" values. This solves our problem since knowledge of $\epsilon = \epsilon_{j_1}$ allows $\mathcal{B}$ to recover of *all* other ϵ_{j_i}, since

$$\epsilon_{j_i} = \epsilon_1 \cdot q_2 \cdot, \ldots, \cdot q_i$$

On the flip side, what if $\mathcal{B}$ obtains *more than one* valid ECDSA-Secp256k1 signatures by opening the (c_i, z_i) pairs? Fortunately, however, we don't need to worry about this. The escrow transaction $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ offers 1 bitcoin in exchange for a ECDSA-Secp256k1 signature under an *ephemeral* key $PK_{\mathcal{T}}^{eph}$ used *only once* during this protocol execution with this specific payee $\mathcal{B}$. Thus, even if $\mathcal{B}$ gets many signatures, only one can be used to form the cash-out transaction $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ that redeems the bitcoin escrowed in $T_{\text{escr}(\mathcal{T},\mathcal{B})}$.

B. Security Properties

We again capture the security requirements of the puzzle-promise protocol using real-ideal paradigm [23]. The ideal functionality $\mathcal{F}_{\text{promise-sign}}$ is presented in Appendix D. $\mathcal{F}_{\text{promise-sign}}$ is designed to guarantee the following properties: (1) *Fairness for $\mathcal{T}$:* Bob $\mathcal{B}$ learns nothing except signatures on *fake* transactions. (2) *Fairness for $\mathcal{B}$:* If $\mathcal{T}$ agrees to complete the protocol, then Bob $\mathcal{B}$ obtains at least one puzzle-promise pair. To do this, $\mathcal{F}_{\text{promise-sign}}$ acts a trusted party between $\mathcal{B}$ and $\mathcal{T}$. Bob $\mathcal{B}$ sends the "real" and "fake" transactions to $\mathcal{F}_{\text{promise-sign}}$. $\mathcal{F}_{\text{promise-sign}}$ has access to an oracle that can compute the Tumbler's $\mathcal{T}$ signatures on any messages. (This provides property (2).) Then, if Tumbler $\mathcal{T}$ agrees, $\mathcal{F}_{\text{promise-sign}}$ provides Bob $\mathcal{B}$ with signatures on each "fake" transaction only. (This provides property (1).) The following theorem is proved in Appendix F:

[7] Since we model hash functions as random oracles we can prove ϵ_i is computationally hidden when the hash of ϵ_i encrypts σ_i to c_i.

Public input: ($e, N, PK^{eph}_{\mathcal{T}}, \pi$).
Tumbler $\mathcal{T}$ chooses fresh ephemeral ECDSA-Secp256k1 key, *i.e.,* bitcoin address ($SK^{eph}_{\mathcal{T}}, PK^{eph}_{\mathcal{T}}$).
π proves validity of (e, N) in a one-time-only setup phase.

Bob $\mathcal{B}$		Tumbler $\mathcal{T}$. **Secret** input: d
		1. Set up $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ Sign but do not post transaction $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ timelocked for tw_2 offering one bitcoin under the condition: "the fulfilling transaction must be signed under key $PK^{eph}_{\mathcal{T}}$ and under key $PK_{\mathcal{B}}$."
2. Prepare μ Real Unsigned $T_{\text{cash}(\mathcal{T},\mathcal{B})}$. For $i \in 1, \ldots, \mu$: Choose random pad $\rho_i \leftarrow \{0,1\}^\lambda$ Set $T_{\text{cash}(\mathcal{T},\mathcal{B})}{}^i = \text{CashOutTFormat}(\rho_i)$ $ht_i = H'(T_{\text{fulfill}}{}^i)$.	$\xleftarrow{T_{\text{escr}(\mathcal{T},\mathcal{B})}}$	
3. Prepare Fake Set. For $i \in 1, \ldots, \eta$: Choose random pad $r_i \leftarrow \{0,1\}^\lambda$ $ft_i = H'(\text{FakeFormat}\Vert r_i)$.		
4. Mix Sets. Randomly permute $\{ft_1, ..., ft_\eta, ht_1, ..., ht_\mu\}$ to obtain $\{\beta_1, ...\beta_{\mu+\eta}\}$ Let R be the indices of the ht_i Let F be the indices of the ft_i	$\xrightarrow{\beta_1 \ldots \beta_{\mu+\eta}}$	
Choose salt $\in \{0,1\}^\lambda$ Compute: $h_R = H(\text{salt}\Vert R)$ $h_F = H(\text{salt}\Vert F)$	$\xrightarrow{h_R, h_F}$	5. Evaluation. For $i = 1, \ldots, \mu + \eta$: ECDSA sign β_i to get $\sigma_i = \text{Sig}(SK^{eph}_{\mathcal{T}}, \beta_i)$ Randomly choose $\epsilon_i \in Z_N$. Create promise $c_i = H^{\mathsf{shk}}(\epsilon_i) \oplus \sigma_i$ Create puzzle $z_i = f_{RSA}(\epsilon_i, e, N)$ *i.e.,* $z_i = (\epsilon_i)^e \mod N$
	$\xleftarrow{(c_1,z_1),\ldots(c_{\mu+\eta},z_{\mu+\eta})}$	
6. Identify Fake Set.	$\xrightarrow{R,F}$ $\xrightarrow{r_i\ \forall i \in F}$ $\xrightarrow{\text{salt}}$	
		7. Check Fake Set. Check $h_R = H(\text{salt}\Vert R)$ and $h_F = H(\text{salt}\Vert F)$ For all $i \in F$: verify $\beta_i = H'(\text{FakeFormat}\Vert r_i)$ Abort if any check fails
8. Check Fake Set. For all $i \in F$ - Validate that $\epsilon_i < N$ - Validate RSA puzzle $z_i = (\epsilon_i)^e \mod N$ - Validate promise c_i: (a) Decrypt $\sigma_i = H^{\mathsf{prg}}(\epsilon_i) \oplus c_i$ (b) Verify σ_i, *i.e.,* $\mathsf{ECDSA\text{-}Ver}(PK^{eph}_{\mathcal{T}}, H'(ft_i), \sigma_i) = 1$ Abort if any check fails	$\xleftarrow{\epsilon_i\ \forall i \in F}$	
		9. Prepare Quotients. For $R = \{j_1, ..., j_\mu\}$: set $q_2 = \frac{\epsilon_{j_2}}{\epsilon_{j_1}}, ..., q_\mu = \frac{\epsilon_{j_\mu}}{\epsilon_{j_{\mu-1}}}$
	$\xleftarrow{q_2,\ldots,q_\mu}$	
10. Quotient Test. For $R = \{j_1, ..., j_\mu\}$ check equalities: $z_{j_2} = z_{j_1} \cdot (q_2)^e \mod N$ ... $z_{j_\mu} = z_{j_{\mu-1}} \cdot (q_\mu)^e \mod N$ Abort if any check fails		11. Post transaction $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ on blockchain
12. Begin Payment Phase. Set $z = z_{j_1}$. Send $\bar{z} = z \cdot (r)^e$ to Payer $\mathcal{A}$		

Fig. 4. Puzzle-promise protocol when $Q = 1$. $(d, (e, N))$ are the RSA keys for the tumbler $\mathcal{T}$. (Sig, ECDSA-Ver) is an ECDSA-Secp256k1 signature scheme. We model H, H' and H^{shk} as random oracles. In our implementation, H is HMAC-SHA256 (keyed with salt) . H' is 'Hash256', *i.e.,* SHA-256 cascaded with itself, which is the hash function used in Bitcoin's "hash-and-sign" paradigm with ECDSA-Secp256k1. H^{shk} is SHA-512. CashOutTFormat is shorthand for the unsigned portion of a transaction that fulfills $T_{\text{escr}(\mathcal{T},\mathcal{B})}$. The protocol uses ρ_i to ensure the output of CashOutTFormat contains sufficient entropy. FakeFormat is a distinguishable public string.

Theorem 2: Let λ be the security parameter. If RSA trapdoor function is hard in $\mathbf{Z}_N^*$, if H, H', H^{shk} are independent random oracles, if ECDSA is strong existentially unforgeable signature scheme, then the puzzle-promise protocol in Figure 4 securely realizes the $\mathcal{F}_{\mathsf{promise\text{-}sign}}$ functionality. The security for $\mathcal{T}$ is $1 - \nu(\lambda)$ while security for $\mathcal{B}$ is $1 - \frac{1}{\binom{\mu+\eta}{\eta}} - \nu(\lambda)$.

VII. TumbleBit Security

We discuss TumbleBit's unlinkability and balance properties. See Section III-C for DoS/Sybil resistance.

A. Balance

The balance was defined, at high-level, in Section III-C. We analyze balance in several cases.

Tumbler $\mathcal{T}^*$ is corrupt. We want to show that all the bitcoins paid to $\mathcal{T}$ by all $\mathcal{A}_j$'s can be later claimed by the $\mathcal{B}_i$'s. (That is, a malicious $\mathcal{T}^*$ cannot refuse a payment to Bob after being paid by Alice.) If $\mathcal{B}_i$ successfully completes the puzzle-promise protocol with $\mathcal{T}^*$, fairness for this protocol guarantees that $\mathcal{B}_i$ gets a correct "promise" c and puzzle z. Meanwhile, the fairness of the puzzle-solver protocol guarantees that each $\mathcal{A}_j$ gets a correct puzzle solution in exchange for her bitcoin. Thus, for any puzzle z solved, some $\mathcal{B}_i$ can open promise c and form the cash-out transaction $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ that allows $\mathcal{B}_i$ to claim one bitcoin. Moreover, transaction $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ has timelock tw_1 and transaction $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ has timelock tw_2. Since $tw_1 < tw_2$, it follows that either (1) $\mathcal{T}^*$ solves $\mathcal{A}$'s puzzle or (2) $\mathcal{A}$ reclaims the bitcoins in $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ (timelock tw_1), before $\mathcal{T}$ can (3) steal a bitcoin by reclaiming the bitcoins in $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ (timelock tw_2).

Case $\mathcal{A}_j^*$ and $\mathcal{B}_i^*$ are corrupt. Consider colluding payers $\mathcal{B}_i^*$ and payees $\mathcal{A}_j^*$. We show that the sum of bitcoins cashed out by all $\mathcal{B}_i^*$ is no more than the number of puzzles solved by $\mathcal{T}$ in the Payment Phase with all $\mathcal{A}_j^*$.

First, the fairness of the puzzle-promise protocol guarantees that any $\mathcal{B}_i^*$ learns only (c, z) pairs; thus, by the unforgeability of ECDSA signatures and the hardness of solving RSA puzzles, $\mathcal{B}^*$ cannot claim any bitcoin at the end of the Escrow Phase. Next, the fairness of the puzzle-solver protocol guarantees that if $\mathcal{T}$ completes SP_j successful puzzle-solver protocol executions with $\mathcal{A}_j^*$, then $\mathcal{A}_j^*$ gets the solution to exactly SP_j puzzles. Payees $\mathcal{B}_i^*$ use the solved puzzles to claim bitcoins from $\mathcal{T}$. By the unforgeability of ECDSA signatures (and assuming that the blockchain prevents double-spending), all colluding $\mathcal{B}_i^*$ cash-out no more than $\min(t, \sum_j SP_j)$ bitcoin in total, where t is the total number of bitcoins escrowed by $\mathcal{T}$ across all $\mathcal{B}_i^*$.

Case $\mathcal{B}_i^*$ and $\mathcal{T}$ collude. Now suppose that $\mathcal{B}_i^*$ and $\mathcal{T}^*$ collude to harm $\mathcal{A}_j$. Fairness for $\mathcal{A}_j$ still follows directly from the fairness of the puzzle-solver protocol. This follows because the only interaction between $\mathcal{A}_j$ and $\mathcal{B}_i^*$ is the exchange of a puzzle (and its solution). No other secret information about $\mathcal{A}_j$ is revealed to $\mathcal{B}_i^*$. Thus, $\mathcal{B}_i^*$ cannot add any additional information to the view of $\mathcal{T}$, that $\mathcal{T}$ can use to harm fairness for $\mathcal{A}_j$.

We do note, however, that an irrational Bob $\mathcal{B}_i^*$ can misbehave by handing Alice $\mathcal{A}_j$ an incorrect puzzle z^*. In this case, the fairness of the puzzle-solver protocol ensures that Alice $\mathcal{A}_j$ will pay the Tumbler $\mathcal{T}$ for a correct solution ϵ^* to puzzle z^*. As such, Bob $\mathcal{B}_i$ will be expected to provide Alice $\mathcal{A}_j$ with the appropriate goods or services in exchange for the puzzle solution ϵ^*. However, the puzzle solution ϵ^* will be of no value to Bob $\mathcal{B}_i$, *i.e.,* Bob cannot use ϵ^* to claim a bitcoin during the Cash-Out Phase. It follows that the only party harmed by this misbehavior is Bob $\mathcal{B}_i$ himself. As such, we argue that such an attack is of no importance.

Case $\mathcal{A}_j^*$ and $\mathcal{T}$ collude. Similarly, even if $\mathcal{A}_j^*$ and $\mathcal{T}$ collude, fairness for an honest $\mathcal{B}_i$ still follows from the fairness of the puzzle-promise protocol. This is because $\mathcal{A}_j^*$'s interaction with $\mathcal{B}_i$ is restricted in receiving a puzzle z, and handing back a solution. While $\mathcal{A}_j^*$ can always give $\mathcal{B}_i$ an invalid solution ϵ^*, $\mathcal{B}_i$ can easily check that the solution is invalid (since $(\epsilon^*)^e \neq z \mod N$) and refuse to provide goods or services.

Case $\mathcal{A}_j^*$, $\mathcal{B}_i^*$ and $\mathcal{T}$ collude. Suppose $\mathcal{A}_j^*$, $\mathcal{B}_i^*$ and $\mathcal{T}$ all collude to harm some other honest $\mathcal{A}$ and/or $\mathcal{B}$. This can be reduced to one of the two cases above because an honest $\mathcal{A}$ will only interact with $\mathcal{B}_i^*$ and $\mathcal{T}^*$, while an honest $\mathcal{B}$ will only interact with $\mathcal{A}_j^*$ and $\mathcal{T}$.

B. Unlinkability

Unlinkability is defined in Section III-C and must hold against a $\mathcal{T}$ that *does not collude* with other users. We show that *all* interaction multi-graphs $\mathcal{G}$ compatible with $\mathcal{T}$'s view are equally likely.

First, note that all TumbleBit payments have the same denomination (1 bitcoin). Thus, $\mathcal{T}^*$ cannot learn anything by correlating the values in the transactions. Next, recall from Section III-A, that all users of TumbleBit coordinate on phases and epochs. Escrow transactions are posted at the same time, during the Escrow Phase only. All $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ cash-out transactions are posted during the Cash-Out Phase only. All payments made from $\mathcal{A}_i$ and $\mathcal{B}_j$ occur during the Payment Phase only, and payments involve no direct interaction between $\mathcal{T}$ and $\mathcal{B}$. This rules out timing attacks where the Tumbler purposely delays or speeds up its interaction with some payer $\mathcal{A}_j$, with the goal of distinguishing some behavior at the intended payee $\mathcal{B}_i$. Even if the Tumbler $\mathcal{T}^*$ decides to cash-out with $\mathcal{A}_j$ before the Payment Phase completes (as is done in Section V-D when $\mathcal{A}_j$ misbehaves), all the $\mathcal{B}_i$ still cash out at the same time, during the Cash-Out Phase.

Next, observe that transcripts of the puzzle-promise and puzzle-solver protocols are information-theoretically unlinkable. This follows because the puzzle $\overline{z}$ used by any $\mathcal{A}_j$ in the puzzle-solver protocol is *equally likely* to be the blinding of *any* of the puzzles z that appear in the puzzle-promise protocols played with any $\mathcal{B}_i$ (see Section III-A, equation (2)).

Finally, we assume secure channels, so that $\mathcal{T}^*$ cannot eavesdrop on communication between $\mathcal{A}_j$'s and $\mathcal{B}_i$'s, and that $\mathcal{T}^*$ cannot use network information to correlate $\mathcal{A}_j$'s and $\mathcal{B}_i$'s (by *e.g.,* observing that they share the same IP address). Then, the above two observations imply that all interaction multi-graphs, that are compatible with the view of $\mathcal{T}^*$, are equally likely.

C. Limitations of Unlinkability

TumbleBit's unlinkability (see Section III-C) is inspired by Chaumian eCash [17], and thus suffers from similar limitations. (Section VII-D discusses the limitations of Chaumian eCash [17] in more detail.) In what follows, we assume that Alice has a single Bitcoin address $Addr_{\mathcal{A}}$, and Bob has Bitcoin address $Addr_{\mathcal{B}}$.

Alice/Tumbler collusion. Our unlinkability definition assumes that the Tumbler does not collude with other TumbleBit users. However, collusion between the Tumbler and Alice can be used in a *ceiling attack*. Suppose that some Bob has set up a TumbleBit payment channel that allows him to accept up to Q TumbleBit payments, and suppose that Bob has already accepted Q payments at time t_0 of the Payment Phase. Importantly, the Tumbler, working alone, cannot learn that Bob is no longer accepting payments after time t_0. (This follows because the Tumbler and Bob do not interact during the Payment Phase.) However, the Tumbler can learn this by colluding with Alice: Alice offers to pay Bob at time t_0, and finds that Bob cannot accept her payment (because Bob has "hit the ceiling" for his payment channel). Now the Tumbler knows that Bob has obtained Q payments at time t_0, and he can rule out any compatible interaction graphs that link any payment made after time t_0 to Bob.

The ceiling attack assumes that Bob allows any Alice to pay him whenever she wants. However, we can rule out ceiling attacks if every Bob always initiates every interaction with Alice. Alternatively, Bob's can ensure that his payment channel allows him to accept significantly more payments that was he expects to receive during an epoch. Another idea is to stagger TumbleBit epochs, so that multiple epochs take place simultaneously. Then, if Bob's escrow from one epoch runs out, he can start making payments to his escrow in the next epoch.

Thus, suppose we eliminate ceiling attacks. Then, the puzzle z provided by Bob *cannot* be linked to *any* payee Bitcoin address $Addr_{\mathcal{B}_1}, ..., Addr_{\mathcal{B}_\iota}$ that has escrowed Bitcoins with the Tumbler, even if Alice and the Tumbler collude.[8] This property is useful when Alice is able to pay Bob without learning Bob's true identity, *e.g.,* when Bob is a Tor hidden service.

Bob/Tumbler collusion. Bob and the Tumbler can collude to learn the true identity of Alice. Importantly, this collusion attack is useful only if Bob can be paid by Alice without learning her true identity (*e.g.,* if Alice is a Tor user). The attack is simple. Bob reveals the blinded puzzle value $\overline{z}$ to the Tumbler. Now, when Alice asks that Tumbler to solve puzzle $\overline{z}$, the Tumbler knows that this Alice is attempting to pay Bob. Specifically, the Tumbler learns that Bob was paid by the Bitcoin address $Addr_{\mathcal{A}}$ that paid for the solution to puzzle $\overline{z}$.

There is also a simple way to mitigate this attack. Alice chooses a fresh random blinding factor $r' \in Z_N^*$ and asks the Tumbler to solve the double-blinded puzzle

$$\overline{\overline{z}} = (r')^e \cdot \overline{z} \mod N. \tag{5}$$

Once the Tumbler solves the double-blinded puzzle $\overline{\overline{z}}$, Alice can unblind it by dividing by r' and recovering the solution to single-blinded puzzle $\overline{z}$. This way, the Tumbler cannot link the double-blinded puzzle $\overline{\overline{z}}$ from Alice to the single-blinded puzzle $\overline{z}$ from Bob.

However, even with double blinding, there is still a timing channel. Suppose Bob colludes with the Tumbler, and sends the blinded puzzle value $\overline{z}$ to both Alice and the Tumbler at time t_0. The Tumbler can rule out the possibility that any payment made by any Alice prior to time t_0 should be linked to this payment to Bob. Returning to the terminology of our unlinkability definition (Section III-C), this means that Bob and the Tumbler can collude to use timing information to rule out some compatible interaction graphs.

Potato attack. Our definition of unlinkability does not consider external information. Suppose Bob sells potatoes that costs exactly 7 bitcoins, and the Tumbler knows that no other payee sells items that cost exactly 7 bitcoins. The Tumbler can use this external information rule out compatible interaction graphs. For instance, if Alice made 6 TumbleBit payments, the Tumbler infers that Alice could not have bought Bob's potatoes. Similarly, if Alice made 7 TumbleBit payments in a short time, the Tumbler might infer that Alice was buying Bob's potatoes.

Potato attacks could be mitigated by aggregating payments (*e.g.,* buying additional goods at the same time as buying potatoes) or adding noise (*i.e.,* by having Alice set up an a TumbleBit payee address, and paying

[8] To see why, recall that the only interaction between Alice and Bob consists of Alice receiving the puzzle from Bob and then handing back its solution. Since we assumed that Alice always gets a puzzle from Bob, the only information that Alice gets from Bob is a blinded puzzle z (which is information-theoretically unrelated to any puzzle generated by Tumbler). But, z is the same puzzle that is used by Alice in the interaction with Tumbler, and so the combined view of Tumbler and Alice is the same as the view of Tumbler alone. Therefore, the anonymity of Bob follows from the anonymity against a malicious Tumbler only.

that address each time she buys potatoes). This problem seems similar to those tackled with differential-privacy.

Intersection attacks. Our definition of unlinkability applies only to a single epoch. Thus, as mentioned in Section IV-A and [13], [41], our definition does not rule out the correlation of information across epochs.

Abort attacks. Our definition of unlinkability applies to payments that complete during an epoch. It does not account for information gained by strategically aborting payments. As an example, suppose that the Tumbler notices that during several TumbleBit epochs, (1) Alice always makes a single payment, and (2) Bob hits the ceiling for his payment channel. Now in the next epoch, the Tumbler aborts Alice's payment and notices that Bob no longer hits his ceiling. The Tumbler might guess that Alice was trying to pay Bob.

D. Limitations of Unlinkability for Chaumian eCash.

Attacks analogous to those in Section VII-C are also possible for classic Chaumian eCash [17].

Chaumian eCash. With Chaumian eCash, a payee Alice interacts with the Bank to withdraws Q eCash coins. Each coin is an RSA blind signature $\overline{\sigma}$ (signed by the Bank) on a blinded serial number $\overline{sn}$ (selected by Alice). The blindness of the blind signature ensures that the Bank does not learn the serial number sn at the time the coin is withdrawn. To use the coin, Alice unblinds the serial number to sn and signature to σ and sends them to Bob. Upon receipt of the coin (sn, σ), Bob must immediately interact with the Bank to deposit and confirm the coin's validity. To validate the coin, the Bank simply checks that no coin deposited earlier had the same serial number sn. Notice that the view of the Chaumian Bank includes (1) the time and value of each payment made to Bob, and (2) the time and value of each withdrawal made by Alice. This is slightly different than the view of our TumbleBit Tumbler, which sees (1) the time and value of each payment made by Alice (rather than by Bob), and (2) value of the total number of payments received by Bob.

We now point out how the attacks in Section VII-C can be launched on Chaumian eCash. Analogous to the first attack in Section VII-C, Alice and the Bank can collude to unmask Bob, as follows: Alice simply reveals the serial number sn to the Bank, and the Bank links sn to the deposit made by Bob. The Chaumian Bank can also collude with Bob to perform a ceiling attack, as follows: Suppose that Bob offers to sell Alice an item worth 1 coin at time t_0, but Alice has no unspent coins. If Alice either (1) refuses to buy the item, or (2) withdraws an additional coin, then the Bank learns that Alice has no unspent coins at time t_0 and the ceiling attack follows. To launch a potato attack, the Bank observes that Alice withdrew only 6 coins after opening her account with the Bank. However, a potato costs 7 coins, and so the Bank learns that Alice could not have bought a potato. Finally, the Chaumian bank can launch an abort attack identical to that of Section VII-C by refusing to allow Alice to withdraw a coin.

VIII. Implementation

To show that TumbleBit is performant and compatible with Bitcoin, we implemented TumbleBit as a classic tumbler. (That is, each payer and payee can send/receive $Q = 1$ payment/epoch.) We then used TumbleBit to mix bitcoins from 800 payers (Alice $\mathcal{A}$) to 800 payees (Bob $\mathcal{B}$). We describe how our implementation instantiates our TumbleBit protocols. We then measure the off-blockchain performance, *i.e.,* compute time, running time, and bandwidth consumed. Finally, we describe two on-blockchain tests of TumbleBit.

A. Protocol Instantiation

We instantiated our protocols with 2048-bit RSA. The hash functions and signatures are instantiated as described in the captions to Figure 3 and Figure 4.[9]

Choosing m and n in the puzzle-solving protocol. Per Theorem 1, the probability that $\mathcal{T}$ can cheat is parameterized by $1/\binom{m+n}{m}$ where m is the number of "real" values and n is the number of "fake" values in Figure 3. From a security perspective, we want m and n to be as large as possible, but in practice we are constrained by the Bitcoin protocol. Our main constraint is that m RIPEMD-160 hash outputs must be stored in T_{puzzle} of our puzzle-solver protocol. Bitcoin P2SH scripts (as described below) are limited in size to 520 bytes, which means $m \leq 21$. Increasing m also increases the transaction fees. Fortunately, n is not constrained by the Bitcoin protocol; increasing n only means we perform more off-blockchain RSA exponentiations. Therefore, we chose $m = 15$ and $n = 285$ to bound $\mathcal{T}$'s cheating probability to 2^{-80}. (2^{-80} equals RIPEMD-160's collision probability.)

Choosing μ and η in the puzzle-promise protocol. Theorem 2 also allows $\mathcal{T}$ to cheat with probability $1/\binom{\mu+\eta}{\mu}$. However, this protocol has no Bitcoin-related constraints on μ and η. Thus, we take $\mu = \eta = 42$ to achieve a security level of 2^{-80} while minimizing the number of off-blockchain RSA computations performed in Figure 4 (which is $\mu + \eta$).

Scripts. By default, Bitcoin clients and miners only operate on transactions that fall into one of the five standard Bitcoin transaction templates. We therefore conform to the Pay-To-Script-Hash (P2SH) [3] template. To format transaction T_{offer} (per Section II) as

[9] There were slight difference between our protocols as described in this paper and the implementation used in some of the tests. In Figure 3, $\mathcal{A}$ reveals blinds $r_j \forall j \in R$ to $\mathcal{T}$, our implementation instead reveals an encrypted version $r_j^e \forall j \in R$. This change does not affect performance, since $\mathcal{A}$ hold both r_j and r_j^e. Also, our implementation omits the index hashes h_R and h_F from Figure 4; these are two 256-bit hash outputs and thus should not significantly affect performance. We have since removed these differences.

a P2SH, we specify a *redeem script* (written in Script) whose condition $\mathcal{C}$ must be met to fulfill the transaction. This redeem script is hashed and stored in transaction T_{offer}. To transfer funds out of T_{offer}, a transaction T_{fulfill} is constructed. T_{fulfill} includes (1) the redeem script and (2) a set of input values that the redeem script is run against. To programmatically validate that T_{fulfill} can fulfill T_{offer}, the redeem script T_{fulfill} is hashed, and the resulting hash value is compared to the hash value stored in T_{offer}. If these match, the redeem script is run against the input values in T_{fulfill}. T_{fulfill} fulfills T_{offer} if the redeem script outputs true. All our redeem scripts include a time-locked refund condition, that allows the party offering T_{offer} to reclaim the funds after a time window expires. To do so, the party signs and posts a *refund transaction* T_{refund} that points to T_{offer} and reclaims the funds locked in T_{offer}. We reproduce our scripts in Appendix I and Figure 9.

B. *Off-Blockchain Performance Evaluation*

We evaluate the performance for a run of our protocols between one payer Alice $\mathcal{A}$, one payee Bob $\mathcal{B}$ and the Tumbler $\mathcal{T}$. We used several machines: an EC2 t2.medium instance in Tokyo (2 Cores at 2.50 GHz, 4 GB of RAM), a MacBook Pro in Boston (2.8 GHz processor, 16 GB RAM), and Digital Ocean nodes in New York, Toronto and Frankfurt (1 Core at 2.40 GHz and 512 MB RAM).

Puzzle-solver protocol (Table II). The total network bandwidth consumed by our protocol was 269 Kb, which is roughly 1/8th the size of the "average webpage" per [55] (2212 Kb). Next, we test the total (off-blockchain) computation time for our puzzle-solver protocol (Section V-B) by running both parties ($\mathcal{A}$ and $\mathcal{T}$) on the Boston machine. We test the impact of network latency by running $\mathcal{A}$ in Boston and $\mathcal{T}$ in Tokyo, and then with $\mathcal{T}$ in New York. (The average Boston-to-Tokyo Round Trip Times (RTT) was 187 ms and the Boston-to-New York RTT was 9 ms.) From Table II, we see the protocol completes in < 4 seconds, with running time dominated by network latency. Indeed, even when $\mathcal{A}$ and $\mathcal{T}$ are very far apart, our 2048-bit RSA puzzle solving protocol is still faster than [39]'s 16x16 Sudoku puzzle solving protocol, which takes 20 seconds.

TumbleBit as a classic tumbler (Table II). Next, we consider classic Tumbler mode (Section IV). We consider a scenario where $\mathcal{A}$ and $\mathcal{B}$ use the same machine, because Alice $\mathcal{A}$ wants anonymize her bitcoin by transferring it to a fresh ephemeral bitcoin address that she controls. Thus, we run (1) $\mathcal{A}$ and $\mathcal{B}$ in Boston and $\mathcal{T}$ in Tokyo, and (2) $\mathcal{A}$ and $\mathcal{B}$ in Boston and $\mathcal{T}$ in New York. To prevent the Tumbler $\mathcal{T}$ for linking $\mathcal{A}$ and $\mathcal{B}$ via their IP address, we also tested with (a) $\mathcal{B}$ connecting to $\mathcal{T}$ over Tor, and (b) both $\mathcal{A}$ and $\mathcal{B}$ connected through Tor. Per Table II, running time is bound by network latency, but is < 11 seconds even with when both parties connect to Tokyo over Tor. Connecting to New York (in clear) results in ≈ 1 second running time. Compute time is only 0.6 seconds, again measured by running $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{T}$ on the Boston machine. Thus, TumbleBit's performance, as a classic Tumbler, is bound by the time it takes to confirm 2 blocks on the blockchain ($\approx$ 20 minutes).

Performance of TumbleBit's Phases. (Table III) Next, we break out the performance of each of TumbleBit's phases when $Q = 1$. We start by measuring compute time by running all $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{T}$ on the Boston machine. Then, we locate each party on different machines. We first set A in Toronto, B in Boston and $\mathcal{T}$ in New York and get RTTs to be 22 ms from Boston to New York, 23 ms from New York to Toronto, and 55 ms from Toronto to Boston. Then we set A in Frankfurt, B in Boston and $\mathcal{T}$ in Tokyo and get RTTs to be 106 ms from Boston to Frankfurt, 240 ms from Frankfurt to Tokyo, and 197 ms from Tokyo to Boston. An off-blockchain payment in the Payment Phase completes in under 5 seconds and most of the running time is due to network latency.

C. *Blockchain Tests*

Our on-blockchain tests use TumbleBit as a classic tumbler, where payers pay themselves into a fresh ephemeral Bitcoin address. All transactions are visible on the blockchain. Transaction IDs (TXIDs) are hyperlinked below. The denomination of each TumbleBit payment (*i.e.,* the price of puzzle solution) was 0.0000769 BTC (roughly $0.04 USD on 8/15/2016). Table IV details the size and fees[10]used for each transaction.

Test where everyone behaves. In our first test, all parties completed the protocol without aborting. We tumbled 800 payments between $\aleph = 800$ payers and $\aleph = 800$ payees, resulting in 3200 transactions posted to the blockchain and a k-anonymity of $k = 800$.The puzzle-promise escrow transactions $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ are all funded from this TXID and the puzzler-solver escrow transactions $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ are all funded from this TXID. This test completed in 23 blocks in total, with Escrow Phase completing in 16 blocks, Payment Phase taking 1 block, and Cash-Out Phase completing in 6 blocks.

We note, however, that our protocol could also have completed much faster, *e.g.,* with 1 block for the Escrow

[10] We use a lower transaction fee rate of 15 Satoshi/byte (see Table IV) for T_{puzzle} and T_{solve} because we are in less of hurry to have them confirmed. Specifically, if $\mathcal{A}$ refuses to sign $T_{\text{cash}(\mathcal{A},\mathcal{T})}$, then $\mathcal{T}$ ends the Payment Phase with $\mathcal{A}$ early (even before the Cash-Out Phase begins), and immediately posts T_{puzzle} and then T_{solve} to the blockchain. See Section V-D.

[10] This test, all escrow transactions $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ and $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ had the same timelock tw_2 and T_{puzzle} had a timelock of tw_1, where $tw_1 < tw_2$. Also, we also modify the protocol description in in Step (2) of Section V-D to have *both* $\mathcal{A}$ and $\mathcal{T}$ sign T_{puzzle} during the Payment Phase without posting it to the blockchain. (We can do this because Alice is only making a single payment in this epoch (*i.e.,* $Q = 1$).) Then, if a malicious Tumbler tried to from steal bitcoins (per the 'Tumbler is corrupt' case of Section VII-A), $\mathcal{A}$ could protect herself by posting T_{puzzle} to the blockchain, and reclaim the bitcoins locked in T_{puzzle} after its timelock tw_1 expires, but prior to $tw2$.

TABLE II. AVERAGE PERFORMANCE OF RSA-PUZZLE-SOLVER AND CLASSIC TUMBLER, IN SECONDS. (100 TRIALS).

	Compute Time	Running Time (Boston-New York)	RTT (Boston-New York)	Running Time (Boston-Tokyo)	RTT (Boston-Tokyo)	Bandwidth
RSA-puzzle-solving protocol	0.398	0.846	0.007949	4.18	0.186	269 KB
Classic Tumbler (in clear)	0.614	1.190	0.008036	5.99	0.187	326 KB
Classic Tumbler (B over Tor)	0.614	3.10	0.0875	8.37	0.273	342 KB
Classic Tumbler (Both over Tor)	0.614	6.84	0.0875	10.8	0.273	384 KB

TABLE III. AVERAGE OFF-BLOCKCHAIN RUNNING TIMES OF TUMBLEBIT'S PHASES, IN SECONDS. (100 TRIALS)

	Compute Time	Running Time (Boston-New York-Toronto)	Running Time (Boston-Frankfurt-Tokyo)
Escrow	0.2052	0.3303	1.5503
Payment	0.3878	1.1352	4.3455
Cash-Out	0.0046	0.0069	0.0068

TABLE IV. TRANSACTION SIZES AND FEES IN OUR TESTS.

Transaction	Size	Satoshi/byte	Fee (in BTC)
T_{escr}	190B	30	0.000057
T_{cash}	447B	30	0.000134
T_{refund} for T_{escr}	373B	30	0.000111
T_{puzzle}	447B	15	0.000067
T_{solve}	907B	15	0.000136
T_{refund} for T_{puzzle}	651B	20	0.000130

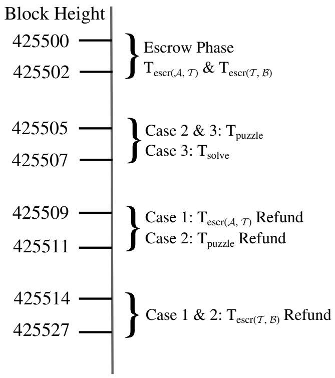
Block Height
425500
425502
Escrow Phase
$T_{\text{escr}(\mathcal{A},\mathcal{T})}$ & $T_{\text{escr}(\mathcal{T},\mathcal{B})}$
425505
425507
Case 2 & 3: T_{puzzle}
Case 3: T_{solve}
425509
425511
Case 1: $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ Refund
Case 2: T_{puzzle} Refund
425514
425527
Case 1 & 2: $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ Refund

Fig. 5. Timeline of test with uncooperative behavior, showing block height when each transaction was confirmed.

Phase, and 1 block for the Cash Out Phase. A Bitcoin block can typically hold ≈ 5260 of our 2-of-2 escrow transactions T_{escr} and ≈ 2440 of our cash-out transaction T_{cash}. We could increase transaction fees to make sure that our Escrow Phase and Cash-Out phase (each confirming 2×800 transactions) occur within one block. In our tests, we used fairly conservative transaction fees (Table IV). While the exact fees needed vary from minute to minute, doubling our fees to 60 Satoshi per Byte should be sufficient under standard transaction volume.[11] As a classic Tumbler, we therefore expect TumbleBit to have a higher denomination than the 0.0000769 BTC we used for our test. For instance, transaction fees of 60 Satoshi per Byte (0.0007644 BTC/user) are $\approx 1/1000$ of a denomination of 0.5 BTC.

Test with uncooperative behavior. Our second run of only 10 users (5 payers and 5 payees) demonstrates how fair exchange is enforced in the face of uncooperative or malicious parties. Transactions $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ and T_{puzzle} were timelocked for 10 blocks and $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ was timelocked for 15 blocks. All escrow transactions $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ are funded by TXID and all escrow transactions $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ are funded by TXID. Two payer-payee pairs completed the protocol successfully. For the remaining three pairs, some party aborted the protocol:

Case 1: The Tumbler $\mathcal{T}$ (or, equivalently, Alice $\mathcal{A}_1$) refused to cooperate after the Escrow Phase. Alice $\mathcal{A}_1$ reclaims her bitcoins from escrow transaction $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ via a refund transaction after the timelock expires. $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ was timelocked for 10 blocks, and the refund transaction was confirmed 8 blocks after $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ was confirmed. The Tumbler $\mathcal{T}$ reclaims its bitcoins from his payment channel with Bob $\mathcal{B}_1$ escrow transaction $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ via a refund transaction after the timelock expires. $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ was timelocked for 15 blocks, and the refund transaction was confirmed 25 blocks after $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ was confirmed.

Case 2: The Tumbler aborts the puzzle-solver protocol by posting the transaction T_{puzzle} but refusing to provide the transaction T_{solve}. (Per Section V-D, to meet the condition in T_{puzzle} and claim its bitcoins, the Tumbler $\mathcal{T}$ has to post T_{solve} that reveal a set of preimages. Because the Tumbler refuses to post T_{solve}, thus refusing to solve Alice's puzzle, Alice's bitcoins are locked in T_{puzzle} until its timelock expires.) No payment completes from $\mathcal{A}_2$ to $\mathcal{B}_2$. Instead, $\mathcal{A}_2$ reclaims her bitcoin from T_{puzzle} via a refund transaction after the timelock in T_{puzzle} expires. The refund transaction was confirmed 4 blocks after T_{puzzle} was confirmed. Tumbler reclaims its bitcoins from its payment channel with Bob $\mathcal{B}_2$ via a refund transaction after the timelock on the escrow transaction $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ expires. The refund transaction was confirmed 25 blocks after $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ was confirmed.

Case 3: The Tumbler provides Alice $\mathcal{A}_3$ the solution to her puzzle in the puzzle-solver protocol, and the Tumbler has an T_{puzzle} signed by $\mathcal{A}$ (Section V-D). However, Alice refuses to sign the cash-out transaction $T_{\text{cash}(\mathcal{A},\mathcal{T})}$ to pay out from her escrow with the Tumbler. Then, the Tumbler signs and posts the transaction T_{puzzle} and its fulfilling transaction T_{solve} and claims its bitcoin. Payment from $\mathcal{A}_3$ to $\mathcal{B}_3$ completes but the Tumbler has to pay more in transaction fees. This is because the Tumbler has to post *both* transactions T_{puzzle} and T_{solve}, rather than just $T_{\text{cash}(\mathcal{A},\mathcal{T})}$; see Table IV.

Remark: Anonymity when parties are uncooperative. Notice that in Case 1 and Case 2, the protocol aborted without completing payment from Alice to Bob. k-anonymity for this TumbleBit run was therefore $k = 3$. By aborting, the Tumbler $\mathcal{T}$ learns that payers $\mathcal{A}_1, \mathcal{A}_2$

[11] For instance, in a 24 hour window starting on Aug 12 2016, all 188K transactions with a fee ≥ 41 Satoshi/Byte were confirmed in the next block. A precise model of current Bitcoin miner behavior, under different fees rates and transaction volumes, remains an open research question. [44] analyzes transaction priority and fee rates but uses older data which no longer reflects current trends.

were trying to pay payees $\mathcal{B}_1, \mathcal{B}_2$. However, anonymity of $\mathcal{A}_1, \mathcal{A}_2, B_1, B_2$ remains unharmed, since $\mathcal{B}_1$ and $\mathcal{B}_2$ were using ephemeral Bitcoin addresses they now discard to safeguard their anonymity (see Section IV-A).

Acknowledgements

We thank Ethan Donowitz for assistance with the preliminary stages of this project, and Nicolas Dorier, Adam Ficsor, Gregory Galperin, Omer Paneth, Dimitris Papadopoulos, Leonid Reyzin, Ann Ming Samborski, Sophia Yakoubov, the anonymous reviewers and many members of the Bitcoin community for useful discussions and suggestions. This work was supported by NSF grants 1012910, 1414119, and 1350733.

References

[1] Bitcoin Fog. *Wikipedia*, 2016.

[2] Monero, https://getmonero.org/home. 2016.

[3] Gavin Andresen. BIP-0016: Pay to Script Hash. *Bitcoin Improvement Proposals*, 2014.

[4] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *IEEE S&P*, pages 443–458, 2014.

[5] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. On the malleability of bitcoin transactions. In *International Conference on Financial Cryptography and Data Security*, pages 1–18. Springer, 2015.

[6] A Back, G Maxwell, M Corallo, M Friedenbach, and L Dashjr. Enabling blockchain innovations with pegged sidechains. *Blockstream, https://blockstream.com/sidechains.pdf*, 2014.

[7] Wacław Banasik, Stefan Dziembowski, and Daniel Malinowski. Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts. Cryptology ePrint Archive, Report 2016/451, 2016.

[8] Simon Barber, Xavier Boyen, Elaine Shi, and Ersin Uzun. Bitter to Better - How to Make Bitcoin a Better Currency. In *Financial Cryptography and Data Security*. Springer, 2012.

[9] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM-CCS*, pages 62–73, 1993.

[10] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Security and Privacy (SP)*, pages 459–474, 2014.

[11] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In *CRYPTO*, pages 90–108, 2013.

[12] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. Deanonymisation of Clients in Bitcoin P2P Network. In *ACM-CCS*, pages 15–29, 2014.

[13] George Bissias, A Pinar Ozisik, Brian N Levine, and Marc Liberatore. Sybil-resistant mixing for bitcoin. In *Workshop on Privacy in the Electronic Society*, pages 149–158, 2014.

[14] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *IEEE - SP*, 2015.

[15] Joseph Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, JoshuaA. Kroll, and EdwardW. Felten. Mixcoin: Anonymity for bitcoin with accountable mixes. In *Financial Cryptography and Data Security*, 2014.

[16] Ran Canetti. Universally composable signature, certification, and authentication. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, pages 219–233. IEEE, 2004.

[17] David Chaum. Blind signature system. In *CRYPTO*, 1983.

[18] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Stabilization, Safety, and Security of Distributed Systems*, pages 3–18. Springer, 2015.

[19] Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. Cinderella: Turning Shabby X. 509 Certificates into Elegant Anonymous Credentials with the Magic of Verifiable Computation. *IEEE Symposium on Security and Privacy 2016*, 2016.

[20] Corin Faife. Will 2017 bring an end to bitcoin's great scaling debate? http://www.coindesk.com/2016-bitcoin-protocol-block-size-debate/, January 5 2017.

[21] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, 1986.

[22] Srivatsava Ranjit Ganta, Shiva Prasad Kasiviswanathan, and Adam Smith. Composition attacks and auxiliary information in data privacy. In *ACM SIGKDD*, pages 265–273, 2008.

[23] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*. ACM, 1987.

[24] Grams. Helixlight: Helix made simple. https://grams7enufi7jmdl.onion.to/helix/light.

[25] Matthew Green and Ian Miers. Bolt: Anonymous Payment Channels for Decentralized Currencies. *Cryptology ePrint Archive 2016/701*, 2016.

[26] Louis C. Guillou and Jean-Jacques Quisquater. A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory. In *EUROCRYPT*, 1988.

[27] Ethan Heilman, Foteini Baldimtsi, and Sharon Goldberg. Blindly Signed Contracts: Anonymous On-Blockchain and Off-Blockchain Bitcoin Transactions. In *Workshop on Bitcoin and Blockchain Research at Financial Crypto*, February 2016.

[28] Chainalysis Inc. Chainalysis: Blockchain analysis, 2016. https://www.chainalysis.com/.

[29] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *ACM-CCS*, 2013.

[30] Tom Elvis Jedusor. Mimblewimble. 2016.

[31] Benjamin Kreuter, Abhi Shelat, Benjamin Mood, and Kevin RB Butler. Pcf: A portable circuit format for scalable two-party secure computation. In *USENIX Security*, 2013.

[32] Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In *ACM-CCS*, 2014.

[33] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In *ACM-CCS*, 2015.

[34] Elliptic Enterprises Limited. Elliptic: The global standard for blockchain intelligence, 2016. https://www.elliptic.co/.

[35] Yehuda Lindell. Fast cut-and-choose-based protocols for malicious and covert adversaries. *Journal of Cryptology*, 29(2):456–490, 2016.

[36] Gregory Maxwell. Zero Knowledge Contingent Payment. *Bitcoin Wiki*, 2011.

[37] Gregory Maxwell. CoinJoin: Bitcoin privacy for the real world. *Bitcoin-talk*, 2013.

[38] Gregory Maxwell. CoinSwap: transaction graph disjoint trustless trading. *Bitcoin-talk*, 2013.

[39] Gregory Maxwell. The first successful Zero-Knowledge Contingent Payment. Bitcoin Core, February 2016.

[40] S Meiklejohn, M Pomarole, G Jordan, K Levchenko, GM Voelker, S Savage, and D McCoy. A fistful of bitcoins:

Characterizing payments among men with no names. In *ACM-SIGCOMM Internet Measurement Conference, IMC*, 2013.

[41] Sarah Meiklejohn and Claudio Orlandi. Privacy-Enhancing Overlays in Bitcoin. In *Lecture Notes in Computer Science*, volume 8976. Springer Berlin Heidelberg, 2015.

[42] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *IEEE Security and Privacy (SP)*, pages 397–411, 2013.

[43] Pedro Moreno-Sanchez, Tim Ruffing, and Aniket Kate. P2P Mixing and Unlinkable P2P Transactions. *Draft*, June 2016.

[44] Malte Möser and Rainer Böhme. Trends, tips, tolls: A longitudinal study of Bitcoin transaction fees. In *FC*, 2015.

[45] Malte Möser and Rainer Böhme. Join Me on a Market for Anonymity. *Workshop on Privacy in the Electronic Society*, 2016.

[46] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and cryptocurrency technologies*. Princeton University Pres, 2016.

[47] Guevara Noubir and Amirali Sanatinia. Honey onions: Exposing snooping tor hsdir relays. In *DEF CON 24*, 2016.

[48] Henning Pagnia and Felix C. Grtner. On the impossibility of fair exchange without a trusted third party, 1999.

[49] Morgen Peck. DAO May Be Dead After $60 Million Theft. *IEEE Spectrum, Tech Talk Blog*, 17 June 2016.

[50] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. Technical report, Technical Report (draft). https://lightning. network, 2015.

[51] Certicom Research. Sec 2: Recommended elliptic curve domain parameters, 2010.

[52] Dorit Ron and Adi Shamir. Quantitative analysis of the full bitcoin transaction graph. In *Financial Cryptography and Data Security*, pages 6–24. Springer, 2013.

[53] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. Coinshuffle: Practical decentralized coin mixing for bitcoin. In *ESORICS*, pages 345–364. Springer, 2014.

[54] Jeff Stone. Evolution Downfall: Insider 'Exit Scam' Blamed For Massive Drug Bazaar's Sudden Disappearance. *international business times*, 2015.

[55] the Internet Archive. Http Archive: Trends, 2015. http://httparchive.org/trends.php.

[56] Peter Todd. BIP-0065: OP CHECKLOCKTIMEVERIFY. *Bitcoin Improvement Proposal*, 2014.

[57] F. Tschorsch and B. Scheuermann. Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies. *IEEE Communications Surveys Tutorials*, PP(99), 2016.

[58] Luke Valenta and Brendan Rowan. Blindcoin: Blinded, accountable mixes for bitcoin. In *FC*, 2015.

[59] Pieter Wuille. Segregated witness and its impact on scalability https://www.youtube.com/watch?v=NOYNZB5BCHM, December 14 2015.

[60] Jan Henrik Ziegeldorf, Fred Grossmann, Martin Henze, Nicolas Inden, and Klaus Wehrle. Coinparty: Secure multi-party mixing of bitcoins. In *CODASPY*, 2015.

Appendix

A. Puzzle-Promise Protocol: Extending to Q payments.

We now extend the puzzle-promise protocol between Bob $\mathcal{B}$ and Tumbler $\mathcal{T}$ from its "base case" of allowing a single payment of denomination 1 bitcoin (Figure 4) to allowing Q payments of denomination 1 bitcoin. The extended protocol is in Figure 6. The extended protocol combines some new cryptographic techniques with the ideas we used in Section III-B (to extend the $\mathcal{A}$-to-$\mathcal{T}$ puzzle-solver protocol to handle Q payments.

The extended puzzle-promise protocol produces Q puzzles $z_1, ..., z_Q$ for Bob $\mathcal{B}$, where the solution to the j^{th} puzzle allows Bob to "open" a promise c_j. The promise c_j contains the Tumbler's ECDSA signature on cash-out transaction $T_{\text{cash}(\mathcal{T},\mathcal{B})}(j)$ that allocates j bitcoins to Bob and $Q - j$ bitcoins for the Tumbler. Each transaction $T_{\text{cash}(\mathcal{T},\mathcal{B})}(j)$ for $j = 1, ..., Q$ points to the *same* 2-of-2 escrow transaction $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ where the Tumbler escrowed Q bitcoins during the Escrow Phase. During the Payment Phase, Bob $\mathcal{B}$ asks the j^{th} payer to solve puzzle z_j; this puzzle solution "opens" promise c_j and provides Bob with the Tumbler's signature on transaction $T_{\text{cash}(\mathcal{T},\mathcal{B})}(j)$. As in Section III-B, Bob does *not* sign this transaction $T_{\text{cash}(\mathcal{T},\mathcal{B})}(j)$ and also does *not* post it to the blockchain during the Payment Phase. Instead, Bob waits until the Escrow Phase starts, and then signs and posts the single cash-out transaction allowing him to claim the maximum number of bitcoins, *i.e.,* the $T_{\text{cash}(\mathcal{T},\mathcal{B})}(j)$ for the last payment Bob received during the Payment Phase.

How do we ensure that Bob $\mathcal{B}$ can open the promise c_j (and thus obtain $T_{\text{cash}(\mathcal{T},\mathcal{B})}(j)$) only *after* he has opened all prior promises $c_{j-1}, ..., c_1$? (This is crucial, because otherwise a cheating Bob claim Q bitcoins from his very first payment, by asking his first payee for the solution to puzzle z_Q.)

We solve this problem by requiring that the solutions to *all* of the puzzle $z_1, ..., z_j$ be used to open the j^{th} promise c_j. To do this, we repeat the steps of the puzzle-promise protocol Q times in parallel. We refer to the Q parallel executions as Q *levels*. In the j-th level, $\mathcal{B}$ prepares $\eta + \mu$ transactions $T_{\text{cash}(\mathcal{T},\mathcal{B})}(j)$, each of which transfers j bitcoins to $\mathcal{B}$. Let $z_{j,\ell}$ denote a puzzle and its solution $\epsilon_{j,\ell}$ at level j, and let $\ell \in [\eta + \mu]$ denote the index for the cut-and-choose as in the base puzzle-promise protocol in Section VI-A. The promise is encrypted under the j puzzle solutions $\epsilon_{1,\ell} ... \epsilon_{j,\ell}$ as:

$$c_{j,\ell} = H(j|\epsilon_{j,\ell}||\epsilon_{j-1,\ell}|| \dots, \epsilon_{1,\ell}) \oplus \sigma_{j,\ell}$$

where $\sigma_{j,\ell}$ is the Tumbler's signature on the cash-out transaction $T_{\text{cash}(\mathcal{T},\mathcal{B})}(j)$ that allocates j bitcoins to Bob.

Now that we have Q levels, we need to extend the cut-and-choose to check the behavior of Tumbler across *all* Q levels. Recall that for the base case of 1 bitcoin, $\mathcal{B}$ prepares $\eta + \mu$ transactions (η of which are fake) of 1 bitcoin each, each of which will be evaluated by $\mathcal{T}$ to obtain a $\eta + \mu$ puzzle-promise pairs. (See Step 5 in Figure 4.) We can visualize this as a $1 \times (\eta + \mu)$ vector, among which $\mathcal{B}$ will check the column positions $\ell \in F$ that correspond to fake values (Step 8 in Figure 4). Now, for the case of Q bitcoins, instead of having a $1 \times (\eta + \mu)$ vector, we have a *matrix* of $Q \times (\eta + \mu)$ elements (Step 5 in Figure 4). $\mathcal{B}$ still checks the *same* column positions $i \in F$, but instead of checking a single puzzle-promise

Public input: $(e, N, PK^{eph}_{\mathcal{T}}, \pi)$.

Tumbler $\mathcal{T}$ chooses fresh ephemeral ECDSA-Secp256k1 key, *i.e.,* bitcoin address $(SK^{eph}_{\mathcal{T}}, PK^{eph}_{\mathcal{T}})$.

π proves validity of (e, N) in a one-time-only setup phase.

Payee Bob $\mathcal{B}(Q)$ | Voucher Promise Protocol | Tumbler $\mathcal{T}(Q, d, (e, N))$

Tumbler:

```
1. Set up $T_{\text{escr}(\mathcal{T},\mathcal{B})}$
Sign but do not post transaction $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ timelocked for $tw_2$ offering $Q$ bitcoins under the condition: "the fulfilling transaction must be signed under key $PK^{eph}_{\mathcal{T}}$ and under $PK_{\mathcal{B}}$."
```

$$\xleftarrow{T_{\text{escr}(\mathcal{T},\mathcal{B})}}$$

Bob:

```
2. Prepare Real Unsigned $T_{\text{cash}(\mathcal{T},\mathcal{B})}$.
For $j \in [Q]$ and $i \in 1, \ldots, \mu$:
    Choose random pad $\rho_{j,i} \leftarrow \{0,1\}^{\lambda}$
    Set ${T_{\text{cash}(\mathcal{T},\mathcal{B})}}^{j,i} = \text{CashOutFormat}(j, \rho_{j,i})$
    $ht_{j,i} = H'({T_{\text{cash}(\mathcal{T},\mathcal{B})}}^{j,i})$.

3. Prepare Fake Set.
For $j \in [Q]$ and $i \in 1, \ldots, \mu$:
    Choose random pad $r_{j,i} \leftarrow \{0,1\}^{\lambda}$
    $ft_{j,i} = H'(\text{FakeFormat}||r_{j,i})$.

4. Mix Sets.
Let $R$ be $\mu$ random indices in $[\mu + \eta]$.
Let $F$ be remaining indices $F = [\mu + \eta] \backslash R$.
For $i = 1, \ldots, \mu + \eta$ and $j \in [Q]$:
    if $i \in R$: $\beta_{j,i} = ht_{j,i}$ (real).
    else $i \in F$: $\beta_{j,i} = ft_{j,i}$ (fake).
```

$$\xrightarrow{\beta_{1,1} \cdots \beta_{1,\mu+\eta}}$$

$$\vdots$$

$$\xrightarrow{\beta_{Q,1} \cdots \beta_{Q,\mu+\eta}}$$

```
Choose salt $\in \{0,1\}^{\lambda}$
Compute: $h_R = H(\text{salt}||R)$, $h_F = H(\text{salt}||F)$
```

$$\xrightarrow{h_R, h_F}$$

Tumbler:

```
5. Evaluation.
For $j \in [Q]$:
    For $i \in [\mu + \eta]$:
        ECDSA sign $\beta_{j,i}$ to get $\sigma_{j,i} = \text{Sig}(SK^{eph}_{\mathcal{T}}, \beta_{j,i})$
        Randomly choose $\epsilon_{j,i} \in Z_N$.
        Compute puzzle $z_{j,i}$=$f_{RSA}(\epsilon_{j+1,i}, e, N)$
        Compute $c_{j,i} = H^{\mathsf{shk}}(j|i|\epsilon_{j,i}||\epsilon_{j-1,i}|| \ldots, \epsilon_{1,i}) \oplus \sigma_{j,i}$.
```

$$\xleftarrow{\{(z_{j,i}, c_{j,i})\}\ \forall i \in [\mu+\eta]\ \forall j \in [Q]}$$

Bob:

```
6. Reveal Real and Fake Set.
```

$$\xrightarrow{R, F, \text{salt}}$$

$$\xrightarrow{\{r_{j,i}, \rho_{j,l}\} i \in R, l \in F, j \in [Q]}$$

Tumbler:

```
7. Check Real and Fake Set.
Check $h_R = H(\text{salt}||R)$ and $h_F = H(\text{salt}||F)$
For all $j \in [Q]$:
    For all $\ell \in F$: Verify $\beta_{j,\ell} = H(\text{FakeFormat}||r_{j,\ell})$.
    For all $i \in R$, Verify $\beta_{j,i} = H(\text{CashOutFormat}(j, \rho_{j,i}))$.

Abort if any check fails
```

$$\xleftarrow{\epsilon_{j,\ell}, \forall \ell \in F, j \in [Q]}$$

Bob:

```
8. Check Fake Set.
If all $\epsilon_{j,\ell} < N$, For $j \in [Q]$; for $\ell \in F$:
    Validate puzzle $z_{j,\ell} = (\epsilon_{j,\ell})^e \mod N$
    Validate promise $c_{j,\ell}$:
    (a) Decrypt $\sigma_{j,\ell} = c_{j,\ell} \oplus H(j|i|\epsilon_{j,\ell}||\epsilon_{j-1,\ell}|| \ldots, \epsilon_{1,\ell})$
    (b) Verify $\mathsf{ECDSA\text{-}Ver}(PK^{eph}_{\mathcal{T}}, H(ft_{j,\ell}), \sigma_{j,\ell}) = 1$
Abort if any check fails.
```

Tumbler:

```
9. Prepare quotients.
For $j \in [Q]$ and for $R = \{\ell_1, ..., \ell_\mu\}$:
    Set $q_{j,2} = \frac{\epsilon_{j,\ell_2}}{\epsilon_{j,\ell_1}}, ..., q_{j,\mu} = \frac{\epsilon_{j,\ell_\mu}}{\epsilon_{j,\ell_{\mu-1}}}$
```

$$\xleftarrow{q_{1,2}, \ldots, q_{1,\mu}}$$

$$\vdots$$

$$\xleftarrow{q_{Q,2}, \ldots, q_{Q,\mu}}$$

Bob:

```
10. Quotient Test.
Let $R = \{\ell_1, ..., \ell_\mu\}$ .
For each $j \in [Q]$ check equalities:
        $z_{j,\ell_2} = z_{j,\ell_1} \cdot (q_{j,2})^e \mod N$
        ...
        $z_{j,\ell_\mu} = z_{j,\ell_{\mu-1}} \cdot (q_{j,\mu})^e \mod N$
Abort if any check fails.
```

Tumbler:

```
11. Post transaction $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ on blockchain
```

Bob:

```
12. Begin Payment Phase.
For $j \in [Q]$, the $j^{th}$ puzzle is $z_j = z_{j,\ell_1}$
```

Fig. 6. Puzzle promise protocol that allows Bob $\mathcal{B}$ to obtain up to Q payments. $(d, (e, N))$ are the RSA keys for the tumbler $\mathcal{T}$. (Sig, ECDSA-Ver) is an ECDSA-Secp256k1 signature scheme. We model H, H' and H^{shk} as random oracles. In our implementation, H is HMAC-SHA256 (keyed with salt), H' is 'Hash256', *i.e.,* SHA-256 cascaded with itself, which is the hash function used in Bitcoin's "hash-and-sign" paradigm with ECDSA-Secp256k1. CashOutFormat is shorthand for the unsigned portion of a transaction that fulfills $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ and transfers j bitcoins to $\mathcal{B}$ and $Q - j$ bitcoins to $\mathcal{T}$. The protocol uses ρ_i to ensure the output of CashOutFormat contains sufficent entropy. FakeFormat is a distinguishable string known to all parties.

pair (c_ℓ, z_ℓ), $\mathcal{B}$ will check a *column* of Q puzzle-promise pairs $[(c_{1,\ell}, z_{1,\ell}), \ldots, (c_{Q,\ell}, z_{Q,\ell})]$ (Step 8 in Figure 6).

Finally, recall that in the base case, $\mathcal{B}$ additionally checks that the full level of puzzles $z_{l_1}, \ldots, z_{l_\mu}$ is consistent with the quotients $q_1, \ldots, q_\mu$ (Step 10 in Figure 4). Similarly, in the case of Q bitcoins, $\mathcal{B}$ will obtain one set of quotient chain for each of the Q levels. Bob checks each level individually (Step 10 in Figure 6) to ensure that by solving puzzle z_j for level j, Bob can also solve all puzzles $z_{j,l_1}, \ldots, z_{j,l_\mu}$ for level j.

Security. The prove security for this protocol, we extend the definition of the ideal functionality $\mathcal{F}_{\textsf{promise-sign}}$ in Section D to the case of Q payments. At high level, this extended definition has the following additional goal: to make sure that Bob $\mathcal{B}$ cannot cheat by claiming more than j bitcoins using his j^{th} puzzle solution. That is, a real puzzle-promise pair $(c_{j,\ell}, z_{j,\ell})$ for level j must contain a signature on a cash-out transaction that transfers exactly j bitcoins to Bob, and no more. Our extended ideal functionality $\mathcal{F}_{\textsf{promise-sign}}$ enforces this as follows: When Bob $\mathcal{B}$ submits a request to receive the promise of a signature on a real message, then the ideal functionality $\mathcal{F}_{\textsf{promise-sign}}$ checks that the real messages confirms to a correct format—namely, that for the j^{th} level it confirms to the format CashOutFormat which transfers exactly j bitcoins to Bob—before sending it off to $\mathcal{T}$. Thus, the security of our protocol follows from the theorem below, which is proved in Appendix G:

Theorem 3: Let λ be the security parameter. If RSA trapdoor function is hard in $\mathbf{Z}_N^*$, if $H, H', H^{\textsf{shk}}$ are independent random oracles, if ECDSA is strong existentially unforgeable signature scheme, then the puzzle-promise protocol in Figure 6 securely realizes the extended $\mathcal{F}_{\textsf{promise-sign}}$ functionality for the case of Q payments. The security for $\mathcal{T}$ is $1-\nu(\lambda)$ while security for $\mathcal{B}$ is $1 - \frac{1}{\binom{\mu+\eta}{\eta}} - \nu(\lambda)$.

B. Ideal Functionalities

We analyze each of the fair-exchange protocols used in TumbleBit in isolation. For each protocol, we identify the security (fairness) properties that we require for the players involved in that phase.

In the Escrow Phase, we consider only interactions between players $\mathcal{B}$ and $\mathcal{T}$ in the puzzle-promise protocol (Section VI-A). We identify the functionality and the security requirements that we expect by this interaction and we formally capture them through and ideal functionality $\mathcal{F}_{\textsf{promise-sign}}$ that we describe in the details in Section D.

In the Payment Phase, we consider only interactions between players $\mathcal{A}$ and $\mathcal{T}$ in the puzzle-solver protocol (Section V-B). We capture the functionality and security requirements of this interaction in the ideal functionality $\mathcal{F}_{\textsf{fair-RSA}}$ described in Section C.

We follow the standard ideal/real world paradigm. To prove that a protocol π *securely realizes* an ideal functionality $\mathcal{F}$, one must show that the view obtained by a real world adversary Adv, corrupting either one of the parties, and running protocol π, can be simulated by a PPT simulator S that only has access to the interface of $\mathcal{F}$. Let us denote by $\textsf{view}_{real}^{\pi,\textsf{Adv}}$ the view that the adversary Adv, corrupting party P_i and playing protocol π, with party P_j, playing with input x_j. Let us denote by $\textsf{view}_{ideal}^{\mathcal{F},\textsf{Adv}}$ the view generated by simulator S, interacting with $\mathcal{F}$ and having black-box access to Adv. Security is defined as follows.

Definition 1 (Secure realization of $\mathcal{F}$.): A two-party protocol π securely realizes $\mathcal{F}$ if, for every PPT static and malicious adversary Adv corrupting either party P_1 or party P_2, there exists a PPT Simulator S such that the view $\textsf{view}_{real}^{\pi,\textsf{Adv}}$ and $\textsf{view}_{ideal}^{\mathcal{F},\textsf{Adv}}$ are computationally indistinguishable.

The Random Oracle Model [9]. Our security proofs are in the *Random Oracle (RO) model* [9]; hash functions are modeled as perfectly random functions, and in the security proof the simulator can *program* their answers.

C. Ideal functionality $\mathcal{F}_{\textsf{fair-RSA}}$

The puzzle-solver protocol allows Alice $\mathcal{A}$ to obtain the solution to a *single* RSA-puzzle y (chosen by $\mathcal{A}$), from the Tumbler $\mathcal{T}$ (who posses the RSA secret key d), in exchange for a bitcoin. Fair exchange for this protocol entails the following: (1) *Fairness for $\mathcal{T}$:* After one execution of the protocol $\mathcal{A}$ will learn the correct solution $y^d \mod N$ to at most one puzzle y of her choice. (2) *Fairness for $\mathcal{A}$:* $\mathcal{T}$ will earn 1 bitcoin iff $\mathcal{A}$ obtains a correct solution.

We model the above two requirements with an ideal functionality, that we call $\mathcal{F}_{\textsf{fair-RSA}}$, shown in Figure 7. $\mathcal{F}_{\textsf{fair-RSA}}$ is a trusted party between $\mathcal{A}$ and $\mathcal{T}$. $\mathcal{F}_{\textsf{fair-RSA}}$ receives a puzzle-solving request of the form (y, 1 bitcoin) from $\mathcal{A}$, and forwards the request to $\mathcal{T}$. If $\mathcal{T}$ agrees to solve the puzzle y for $\mathcal{A}$, then $\mathcal{T}$ receives 1 bitcoin while $\mathcal{A}$ receives the puzzle solution. Otherwise, if $\mathcal{T}$ refuses, $\mathcal{A}$ will get 1 bitcoin back, and $\mathcal{T}$ gets nothing. Fairness for $\mathcal{T}$ is captured because $\mathcal{A}$ can request a puzzle solution only if she sends 1 bitcoin to $\mathcal{F}_{\textsf{fair-RSA}}$. Fairness for $\mathcal{B}$ is captured because $\mathcal{T}$ receives 1 bitcoin only if he agrees to reveal the puzzle solution.

Remark 1. Note that $\mathcal{A}$ can *always* learn solution to RSA puzzles that she generates herself without interacting with $\mathcal{F}_{\textsf{fair-RSA}}$. That is, $\mathcal{A}$ can always choose a random $x \in \mathbf{Z}_N^*$ and generate the puzzle $y = x^e \mod N$; in this case, she trivially knows the puzzle solution is x. This is not a problem because TumbleBit requires Alice $\mathcal{A}$ to solve puzzles that were provided to her by Bob $\mathcal{B}$, and generated through Bob $\mathcal{B}$'s interaction with $\mathcal{T}$ during the puzzle-promise protocol.

Remark 2. Note that the functionality $\mathcal{F}_{\textsf{fair-RSA}}$ does *not provide any privacy* for $\mathcal{A}$. Indeed, $\mathcal{T}$ learns $\mathcal{A}$'s puzzle

$\mathcal{F}_{\textsf{fair-RSA}}$ is parameterized by a function (f_{RSA}, f_{RSA}^{-1}), key verification algorithm KeyVer and and expiration time $tw \in \mathbb{N}$.

Parties. $\mathcal{A}$, $\mathcal{T}$, and adversary S.
Setup. On receiving ($\texttt{Setup}$, (e, N, d)) from $\mathcal{T}$.
If $\textsf{KeyVer}(e, N, d) = 0$ then do nothing.
Else, send ($\texttt{Setup}$, e, N) to $\mathcal{A}$ and S.
Evaluation.

On input $(\texttt{request}, \texttt{sid}, y, 1BTC)$ from $\mathcal{A}$:
If y is in the range of f_{RSA}, send
($\texttt{request}$, $\texttt{sid}$, $\mathcal{A}$, y) to $\mathcal{T}$.
Start counter $tw_{\texttt{sid}} = 0$.

On input $(\texttt{evaluate}, \texttt{sid}, \mathcal{A})$ from $\mathcal{T}$:
Send $(\texttt{sid}, f_{RSA}^{-1}(d, y))$ to $\mathcal{A}$.
Send $(\texttt{payment}, \texttt{sid}, \text{1BTC})$ to $\mathcal{T}$.

If $tw_{\texttt{sid}} = tw$, send ($\texttt{refund}$,$\texttt{sid}$, 1BTC) to $\mathcal{A}$.

Fig. 7. Ideal Functionality $\mathcal{F}_{\textsf{fair-RSA}}$.

y even if $\mathcal{T}$ refuses to solve the puzzle. To use $\mathcal{F}_{\textsf{fair-RSA}}$ in our unlinkable TumbleBit scheme, users will have to first blind their inputs to $\mathcal{F}_{\textsf{fair-RSA}}$.

D. Ideal functionality $\mathcal{F}_{\textsf{promise-sign}}$

The security requirements for the puzzle-promise protocol (Figure 4), which is run in the Escrow Phase of TumbleBit, are captured by an ideal functionality $\mathcal{F}_{\textsf{promise-sign}}$ that we describe in Figure 8.

$\mathcal{F}_{\textsf{promise-sign}}$ acts a trusted party between $\mathcal{B}$ and $\mathcal{T}$. Bob $\mathcal{B}$ sends the "real" and "fake" transactions to $\mathcal{F}_{\textsf{promise-sign}}$. $\mathcal{F}_{\textsf{promise-sign}}$ has access to an oracle that can compute the Tumbler's $\mathcal{T}$ signatures on any messages. $\mathcal{F}_{\textsf{promise-sign}}$ is designed to guarantee the following two properties:

(1) *Fairness for* $\mathcal{B}$*:* If $\mathcal{T}$ agrees to complete the protocol, then Bob $\mathcal{B}$ obtains at least one promise that contains a valid signature on a real transaction. This property follows because $\mathcal{F}_{\textsf{promise-sign}}$ has access to an oracle that computes the $\mathcal{T}$'s signature. Specifically, upon receipt of a real message m_j from $\mathcal{B}$, functionality $\mathcal{F}_{\textsf{promise-sign}}$ keeps a record $(m_j, PK_{\mathcal{T}}^{eph}, \textsf{promise})$ that promises to return a valid signature on m_j to $\mathcal{B}$. Importantly, however, $\mathcal{F}_{\textsf{promise-sign}}$ does not reveal the actual signature on m_j, but only a promise that this signature will be revealed in the future.

(2) *Fairness for* $\mathcal{T}$*:* Bob $\mathcal{B}$ learns nothing except signatures on *fake* transactions. This property follows for three reasons. First, $\mathcal{F}_{\textsf{promise-sign}}$ will only ask its signing oracle to sign fake transaction, *i.e.,* to sign messages that conform to the fake transaction format 'FakeFormat'. Second, when $\mathcal{F}_{\textsf{promise-sign}}$ is asked to verify signatures, only signatures computed on fake transactions will be valid and all others will be invalid. This follows because the only signatures $\mathcal{F}_{\textsf{promise-sign}}$ considers to be valid are those that had previously been computed by its signing oracle. Third, $\mathcal{F}_{\textsf{promise-sign}}$ does not reveal the actual signature on a real message m_j, but only a promise that the signature will be revealed

Functionality $\mathcal{F}_{\textsf{promise-sign}}$

The functionality is parameterized by a format specification FakeFormat, and parameters μ and η.

Parties. $\mathcal{B}$, $\mathcal{T}$, and adversary S.

Setup. Inform $\mathcal{F}_{\textsf{promise-sign}}$ if $\mathcal{T}$ is corrupt or honest.

Key Generation. Upon receiving message $(\texttt{KeyGen}, \mathcal{B})$ from party $\mathcal{B}$, send it to S and receive response $(PK_{\mathcal{T}}^{eph}, \textsf{Sig})$. Sig is a signing algorithm.
Send $(\texttt{Setup}, PK_{\mathcal{T}}^{eph})$ to $\mathcal{B}$ and record the pair $(PK_{\mathcal{T}}^{eph}, \textsf{Sig})$.

Signature Request. Upon receiving this message from $\mathcal{B}$:

$$(\texttt{sign-request}, PK_{\mathcal{T}}^{eph'}, \{\textsf{FkTxn}_i\}_{i\in[\eta]}, \{m_i\}_{i\in[\mu]})$$

If $PK_{\mathcal{T}}^{eph'} \neq PK_{\mathcal{T}}^{eph}$, then do nothing.
If $\forall i$, $\textsf{FkTxn}_i$ complies with FakeFormat then send to $\mathcal{T}$

$$(\texttt{sign-request}, \mathcal{B}, PK_{\mathcal{T}}^{eph}, \{\textsf{FkTxn}_i\}_{i\in[\eta]})$$

Else, do nothing.

Promise. Upon receiving $(\texttt{promise}, \mathcal{B}, \textsf{ANS}, \textsf{Set})$ from $\mathcal{T}$.
If ANS = NO, then set all signatures to $\perp$.
Else, if $\textsf{Set} \neq \emptyset$, compute signatures as follows:

- If $\mathcal{T}$ is honest:
  Set $\textsf{FkSign}_i$= $\textsf{Sig}(\textsf{FkTxn}_i, PK_{\mathcal{T}}^{eph})$ for $i \in [\eta]$.
- Else $\mathcal{T}$ is corrupt:
  Send $(\texttt{Sign}, \textsf{FkTxn}_i, \mathcal{B})$ to adversary S, and obtain respective signatures.
- Abort if there is a recorded entry $(\textsf{FkTxn}_i, \textsf{FkSign}_i, PK_{\mathcal{T}}^{eph}, 0)$.
- Record entries $(\textsf{FkTxn}_i, \textsf{FkSign}_i, PK_{\mathcal{T}}^{eph}, 1)$ and $(m_j, PK_{\mathcal{T}}^{eph}, \textsf{promise})$.

Send $(\texttt{Sign-promise}, \textsf{ANS})$ to $\mathcal{B}$.

Signature Verification. Upon receiving $(\texttt{Verify}, sid, m, \sigma, PK_{\mathcal{T}}^{eph'})$ from any party P:

- If $PK_{\mathcal{T}}^{eph'} \neq PK_{\mathcal{T}}^{eph}$, do nothing.
- Else, if $\mathcal{T}$ is honest:
  ♢ If there is a recorded entry $(m, \sigma, PK_{\mathcal{T}}^{eph}, 1)$, then set $ver = 1$ (**completeness condition**).
  ♢ If there is no recorded entry $(m, \sigma, PK_{\mathcal{T}}^{eph}, 1)$, then set $ver = 0$ and set the entry $(m, \sigma, PK_{\mathcal{T}}^{eph}, 0)$ (**unforgeability condition**).
- Else, if $\mathcal{T}$ is corrupt:
  then let ver be set by S. (**Corrupt signer case**).

Send $(\texttt{Verified}, sid, m, \sigma, ver)$ to party P.

Fig. 8. Ideal Functionality $\mathcal{F}_{\textsf{promise-sign}}$.

in the future.

Discussion. In the ideal world, S will be the signing oracle for $\mathcal{F}_{\textsf{promise-sign}}$. This is follows the definition of the ideal functionality for signatures, per [16]. We stress that this does not mean that S has an additional power. The reason being that in the ideal world, the only signatures that are verified by $\mathcal{F}_{\textsf{promise-sign}}$ are the ones on fake messages. Thus, towards ensuring indistinguishability between the real world and ideal world, we just need to make sure that (when $\mathcal{T}$ is honest), no party can (in both the ideal and real world) produce a signature on a real message without breaking unforgeability of the signature scheme.

$\mathcal{F}_{\textsf{promise-sign}}$ and the case of Q payments. For the case of Q payments, we provide an extended version of the $\mathcal{F}_{\textsf{promise-sign}}$ functionality that deals with Q sets of signatures. The main modification that we need to make—beside having $\mathcal{F}_{\textsf{promise-sign}}$ provide Q sets of fake signatures rather than one set—is to additionally check the format of real messages. We therefore extend $\mathcal{F}_{\textsf{promise-sign}}$ with parameters Q and RealFormat$(\cdot)$, and a validation test that checks that any real message $m_{j,i}$ sent for the j-th set, complies with format RealFormat(j). This is important because in our application we need to control the type of messages that $\mathcal{B}$ gets signed. In our application, RealFormat(j) = CashOutFormat.

Concretely, we extend Figure 8 as follows. First, $\mathcal{F}_{\textsf{promise-sign}}$ is parameterized by two format specifications: FakeFormat and RealFormat, and 3 parameters Q, μ and η. Second, in the **Signature Request** step $\mathcal{B}$ one of the following tuples for every $j \in [Q]$:

$$(\texttt{sign-request}, PK_{\mathcal{T}}^{eph'}, \{\textsf{FkTxn}_{j,i}\}_{i\in[\eta]}, \{m_{j,i}\}_{i\in[\mu]})$$

In other words, $\mathcal{B}$ sends a $Q\times\mu$-matrix of fake messages $\textsf{FkTxn}_{j,i}$ and a $Q\times\mu$-matrix of real messages $m_{j,i}$. The ideal functionality will additionally check that:

$$m_{j,i} = \text{RealFormat}(j)$$

Then, when $\mathcal{T}$ chooses the $\textsf{Set} \subset [\mu]$ in the **Promise** step, then index $i \in \textsf{Set}$ means that $\mathcal{B}$ is promised that $m_{j,\ell}$ will be signed *for every* $j \in [Q]$. In other words, if column index i is in Set, it follows that signatures are promised for the entire column of real messages $m_{j,i}$ $\forall j \in [Q]$.

E. Proof of Theorem 1

The proof is divided into two cases.

1) Case: $\mathcal{A}$ is corrupted: We start with intuition, and then present the formal proof.

Intuition. We want to prove that, by participating in (and arbitrarily deviating from) the protocol in Figure 3, any corrupted $\mathcal{A}$ does not learn anything more than the solution x to the puzzle y, *i.e.,* $\mathcal{A}$ learns only the RSA pre-image $x = f^{-1}(y, d) = y^d$. The transcript obtained by $\mathcal{A}$ in the protocol executions contains: (1) pre-images for all the fake values β_i with $i \in F$, that is $f^{-1}(\beta_i, d)$; (2) encryptions c_i of the pre-images of all "real" β_i and hash h_i of the keys used to encrypt these c_i. Informally, such transcript does not leak any information to $\mathcal{A}$ for the following reasons:

1) $\mathcal{A}$ learns nothing from the answers to the fake set: For all β_i in the fake set F, $\mathcal{A}$ must provide the pre-images $f^{-1}(\beta_i, \cdot)$ to $\mathcal{T}$ *before* $\mathcal{T}$ decrypts c_i. Therefore, $\mathcal{A}$ does not learn anything new from $\mathcal{T}$'s decryptions.
2) For the real β_i, $\mathcal{A}$ is computationally-bound to a single puzzle y because in $\mathcal{A}$ must provide values $r_1, \ldots, r_n$ that demonstrate that, for all i,
$$\beta_i = y \cdot (r_i)^e \mod N$$
Intuitively, due to the hardness of inverting RSA trapdoor function, such values can be provided only if β_is where honestly computed.
3) $\mathcal{A}$ does not learn the puzzle solution y^d unless $\mathcal{T}$ reveals k_i used to encrypted the puzzle solution inside c_i. Encryptions c_i are statistically hiding (and in fact, equivocable[12]) in the Random Oracle model.

Overview of proof. Formally, we shall prove this by showing that there exists a PPT simulator S that is able to simulate the transcript between $\mathcal{A}$ and $\mathcal{T}$, having in input only the puzzle solution $f^{-1}(y, d) = y^d$. If this is possible, it means that the transcript reveals nothing more than the puzzle solution y^d to $\mathcal{A}$.

We heavily use the programmability of the RO. In a nutshell, the S computes all ciphertexts using random values (instead of by encrypting the actual values) and will later ensure that they decrypt to the correct values by programming the random oracle (RO). The key observation is that, at any point in the protocol, $\mathcal{T}$ "decrypts" his encryptions only after $\mathcal{A}$ has sent some crucial information. Indeed, $\mathcal{T}$ sees the pre-images (i.e., ρ_i) of the fake set *before* he sends the keys to decrypt his ciphertext. This allows the simulator to learn how to program the RO to decrypt the ciphertext with the values ρ_i that $\mathcal{A}$ reveals. Similarly, S learns the original puzzle y in the second phase of the protocol, and S will query the ideal functionality $\mathcal{F}_{\textsf{fair-RSA}}$ with $(y, 1btc)$ to obtain the puzzle solution $x = y^d$. Finally, S will program the RO so that he can equivocate the remaining ciphertexts so that they decrypt to the correct puzzle solution x.

Proof. The formal proof consists of two steps. First, we describe the simulator S, then we prove that the transcript generated by S is indistinguishable from the transcript generated in the real world execution. Let Adv denote the adversary corrupting player $\mathcal{A}$.

[12] Equivocable means that the encryptions can be later decrypted by S as *any* value, by programming the output of the random oracle.

Simulator S.

S internally runs adversary Adv. Let session identifier be $\mathtt{sid}$. Let $\mathcal{Q}_H$ be the set of queries made to the H random oracle, and $\mathcal{Q}_{H^{\mathsf{prg}}}$ be the queries made to the H^{prg} random oracle.

1) Setup. Extract (e, N, d) from proof π published by Adv. Send $(\mathtt{Setup},\ (e, N, d))$ to $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$.
2) Upon receiving $\beta_1, \ldots, \beta_{m+n}$ from Adv; randomly pick c_i in $\{0,1\}^{\lambda}$ and $h_i \in \{0,1\}^{\lambda_2}$ for $i \in [m+n]$ and send it to Adv.
3) Upon receiving F and ρ_i for $i \in F$, check if $\beta_i = (\rho_i)^e \bmod N, \forall i$. If check fails, output whatever Adv outputs and halt. Else, run procedure $\mathsf{Equivocate}(c_i,\ \rho_i,\ h_i)$ to obtains keys k_i'. Send k_i' for $i \in F$ to Adv.
4) Upon receiving y, r_i for $i \in R$ from Adv. If $\beta_i = y \cdot (r_i)^e \mod N$ for all $i \in R$ and transaction T_{puzzle} is correctly formed, do as follows:
   – Send $(\mathtt{request},\mathtt{sid}, y, 1btc)$ to $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$ and obtain $x = y^d \mod N$.
   – Run $\mathsf{Equivocate}(c_i,\ x \cdot r_i, h_i)$ and obtain k_i' with $i \in R$.
   – Send transaction T_{solve} with values k_i'.
   Else, checks have failed so output whatever Adv outputs and halt.

Procedure RO: Random Oracle Simulation for H, H^{prg} proceeds as follows. Upon receiving query q for H (resp., H^{prg}):

1) if query $q \in \mathcal{Q}_H$ (resp., $\mathcal{Q}_{H^{\mathsf{prg}}}$), retrieve entry (q, a) from the set and output a.
2) Else pick a random $a \in \{0,1\}^{\lambda_2}$ (resp. λ_1), add (q, a) to $\mathcal{Q}_H$ (resp., $\mathcal{Q}_{H^{\mathsf{prg}}}$) and output a.

Procedure $\mathsf{Equivocate}(c_i, m_i, h_i)$ is as follows:

1) Pick a random $k_i' \in \{0,1\}^{\lambda_1}$. If $k_i' \in \mathcal{Q}_H$ or $\mathcal{Q}_{H^{\mathsf{prg}}}$, output $\mathtt{Collision}$ and abort.
2) Compute $a_i = c_i \oplus m_i$, then add (k_i', a_i) to $\mathcal{Q}_{H^{\mathsf{prg}}}$.
3) Add (k_i', h_i) to $\mathcal{Q}_H$.
4) Output k_i'.

Indistinguishability proof.

We prove by hybrid arguments that view $\mathsf{view}_{\mathsf{Adv},\mathcal{T}}(\mathcal{T}, d)$ obtained by Adv interacting with $\mathcal{T}$ playing with secret input d is indistinguishable from the view S interacting with $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$.

Lemma 1: Assume π is a zero-knowledge *proof of knowledge* in the random oracle model. Assume RSA assumption holds in $\mathbf{Z}_N^*$, and $H^{\mathsf{prg}} : \{0,1\}^{\lambda_1} \to \{0,1\}^{\lambda}$ and $H : \{0,1\}^{\lambda_1} \to \{0,1\}^{\lambda_2}$ are independent random oracles. Then, the view generated by S is computationally indistinguishable from the view $\mathsf{view}_{\mathsf{Adv},\mathcal{T}}(\mathcal{T}, d)$ obtained by Adv in the real world.

Proof: We use a hybrid argument.

H_0: This is the real game. The transcript is generated precisely using the procedure of $\mathcal{T}$ with secret input d. The view generated in this hybrid experiment is $\mathsf{view}_{\mathsf{Adv}}(\mathcal{T}, d)$.

H_1: In this hybrid we change the way encryptions c_i are decrypted and pre-image of hash values h_i are computed. Instead of sending keys k_i, as computed in the protocol, send k_i' computed by running procedure $\mathsf{Equivocate}(c_i, (\beta_i)^d, h_i)$. Note that, in this hybrid we are still using d (which S does not know), but we are programming the answers to the RO. The difference between H_0 and H_1 is in the way keys k_i' are computed.

Due to the security properties assumed in the RO, the values c_i, h_i statistically hide the message encrypted and the hash-preimage; and probability of event $\mathtt{Collision}$ is negligible. Therefore, the view generated in hybrid H_0 is statistically indistinguishable from the view generated in H_1.

H_2: This hybrid experiment is exactly as H_1 with the only difference that procedure $\mathsf{Equivocate}$ is run with input $\mathsf{Equivocate}(c_i, \rho_i, h_i)$ where ρ_i is taken directly from the transcript, and $\mathsf{Equivocate}(c_i, y^d \cdot r_i, h_i)$. where y^d is taken from $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$ only *after* Adv hast send T_{puzzle}. The view generated in this hybrid experiment is identical to view generated hybrid H_1, and correspond to the simulation strategy S.Note that in this hybrid argument the entire view is generated only with a single evaluation y^d. ■

2) Case: Tumbler $\mathcal{T}$ is corrupted: We want to show that any adversary Adv corrupting $\mathcal{T}$ will earn 1 bitcoin if and only if she provides a correct solution to $\mathcal{A}$'s puzzle. This follows from the following arguments.

Define BAD the following event: (1) $\mathcal{T}$ passes the Fake Set Check (Step 7 in Figure 3), therefore providing n correct decryptions to c_i for $i \in F$, AND (2) all encryptions c_i in the real set $i \in R$ are incorrect, *i.e.,* do not decrypt to a valid puzzle solution.

Given that (1) permuted $\beta_1, \ldots, \beta_{m+n}$ information theoretically do not reveal any information about sets F, R and (2) H, H^{prg} are modeled as random oracles, then the probability of event BAD amounts to the probability of guessing set F, which is

$$\Pr[\mathsf{BAD}] = \frac{1}{\binom{m+n}{n}} + \frac{1}{2^{\lambda_1}}$$

Simulator S.

Suppose Adv corrupts Tumbler $\mathcal{T}$. S internally runs Adv. S, playing in the ideal world, receives $(\mathtt{request}$, $\mathtt{sid}$, y, 1 bitcoin) from $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$. Using input y, S simulates the transcript that Adv expects to see in the real world, by honestly following $\mathcal{A}$'s procedure on input y. Upon receiving $\{k_i\}$ for all $i \in F$ from Adv, S checks if all $\{c_i\}_{i \in F}$ are correct. If so, S sends message $(\mathtt{evaluate},\mathtt{sid}, \mathcal{A})$ to $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$ which then passes the puzzle solution x to the ideal world player $\mathcal{A}$. Meanwhile, S sends T_{puzzle} to Adv. Upon receiving T_{solve} from Adv: if all keys $\{k_i\}$ for all $i \in R$ decrypt ciphertexts c_i that do not contain valid puzzle solutions, then S outputs `BAD` and aborts. Else, S outputs whatever Adv outputs and halts.

Indistinguishability Proof.
Real world and ideal world execution are distinguishable only in the case that "`BAD`" event happens. To see why, note that, when S outputs "`BAD`" it means that in the ideal world S sent message (`evaluate`,`sid`, $\mathcal{A}$) to $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$, so that ideal player $\mathcal{A}$ receives her output. However, in the real world, $\mathcal{A}$ will not get any valid output. So the two worlds will be distinguishable. The two worlds are therefore distinguishable with probability $\Pr[\mathsf{BAD}]$.

F. Proof of Theorem 2

In this section we provide the formal proof of Theorem 2. We prove that the puzzle-promise protocol in Figure 4 securely realizes functionality $\mathcal{F}_{\mathsf{promise\text{-}sign}}$ (Figure 8) in the random oracle model.

The proof consists of analyzing two cases: (1) Case $\mathcal{B}$ is corrupted, where we argue that any malicious $\mathcal{B}^*$ does not learn anything besides signatures of *fake* transactions; (2) Case $\mathcal{T}$ is corrupted, where we argue that, if the protocol successfully terminates, then $\mathcal{B}$ will be able to retrieve a signature (on a real cash-out transaction $T_{\mathsf{cash}(\mathcal{T},\mathcal{B})}$) from a puzzle-promise pair (c_i, z_i) for some $i \in R$.

1) Case $\mathcal{B}$ is corrupted: The proof consists of showing that for any corrupted $\mathcal{B}^*$ there exists a PPT simulator S that corrupts $\mathcal{B}$ in the ideal world, and can generate the entire view of $\mathcal{B}^*$ while having access only to the information provided by the ideal functionality $\mathcal{F}_{\mathsf{promise\text{-}sign}}$. Recall that, in the ideal world, Bob $\mathcal{B}$ only receives signatures for the fake messages. With only this information in hand, S will have to simulate the view that the real world adversary $\mathcal{B}^*$ has during its interaction with $\mathcal{T}$. The idea is that, if we can prove that the transcript generated by S is indistinguishable from the one generated by $\mathcal{T}$, then it follows that any $\mathcal{B}^*$ learns no more that what $\mathcal{B}$ learns from $\mathcal{F}_{\mathsf{promise\text{-}sign}}$.

We stress that the following analysis only consider the puzzle-promise protocol run during the Escrow Phase. The analysis works in the *stand-alone setting, where no other protocol, except the puzzle-promise protocol is executed.*

We start with the intuition behind the proof.

Proof Intuition. We have to prove that the transcript of the protocol between $\mathcal{B}^*$ and $\mathcal{T}$, reveals nothing more than signatures on fake messages, *i.e.,* $\sigma_\ell = \mathrm{Sig}(SK^{eph}_{\mathcal{T}}, \beta_\ell)$ for $\ell \in F$; and a "promise" of at least one valid signatures on a real messages β_i for $i \in R$. In the real world, the promise is the set of puzzle-promise pairs (c_i, z_i) for $i \in R$, where c_i is an encryption of a valid signatures on β_i, and z_i is an RSA puzzle whose solution can be used to decrypt c_i. The point of the proof is to show that $\mathcal{B}$ learns nothing else beyond the guarantee that for $i \in R$ there is at least one pair (c_i, z_i) that has an encryption of a valid ECDSA signatures on a real message β_i.

We prove this by showing that if (1) the encryption scheme is perfectly secure in the RO model, and (2) that RSA trapdoor function is hard to invert, then the transcript obtained by $\mathcal{B}$ from the interaction with $\mathcal{T}$ reveals nothing but the signatures on fake messages. We will show that the entire transcript can be simulated by a simulator S that only gets the signatures on fake messages as input.

To build intuition, we list the information that $\mathcal{B}^*$ obtains from the transcript, and we explain why it gives no information on the signatures of valid messages. $\mathcal{B}^*$ obtains the following values:

1. Encryptions $(c_1, \ldots, c_{\mu+\eta})$ computed as a one-time pad of the output of the Random Oracle H^{shk} queried with secret values $\epsilon_1, \ldots, \epsilon_{\mu+\eta}$.

As we work in the (programmable) random oracle, we assume that each encryption perfectly hides the message. Also, the simulator can equivocate each decryption, by programming the random oracle. This means that, encryptions c_i alone do not reveal any information to $\mathcal{B}^*$; indeed the simulator S could generate such c_i by just sending a random value. Here we are using the unpredictability property of the RO, as well its programmability.

2. RSA puzzles $(z_1, \ldots, z_{\mu+\eta})$ where $z_i = \mathcal{F}_{\mathsf{fair\text{-}RSA}}(\epsilon_i, e, N)$. Recall that each ϵ_i is *randomly chosen* group element, and that RSA parameters are computed using the correct procedure. Therefore, under the assumption that RSA trapdoor function is hard in the group determined by the chosen parameters, a PPT $\mathcal{B}^*$ cannot learn any ϵ_i from z_i.

3. Quotients $(q_2, ..., q_\mu)$, where $q_j = \frac{\epsilon_{j_i}}{\epsilon_{j_{i-1}}}$ for $j_i \in R$. This is a sequence of connected divisions of the secret keys $\epsilon_{j_1}, \ldots, \epsilon_{j_\mu}$. Intuitively, to see that these quotients do not give any more information than what can be learnt from values $z_{j_1}, \ldots, z_{j_\mu}$, we show that one can compute z_{j_i} and q_{j_i} that pass the "quotient test", without knowing *any* ϵ_{j_i}.

To see why, note that the quotient test checks that for each each $i = 2, ..., \mu$,

$$z_{j_i} = z_{j_{i-1}} \cdot (q_{j_i})^e$$

where $R = \{j_1, ..., j_\mu\}$. This means that one can fix arbitrary $z_{j_i}, q_{j_i} \in \mathbf{Z}_N^*$ and compute $z_{j_{i-1}}$ as $z_{j_i}/(q_{j_i})^e$. In this way one can generate z_{j_i}, q_{j_i} that pass the test without knowing the RSA inverse of any of the z_{j_i}.

This observation will be crucial in the proof, because it allows us to show that if there is an adversary $\mathcal{B}^*$ that is able to learn some ϵ_{j_i} for $j_i \in R$, then we can build a reduction $\mathcal{A}_{\mathsf{RSA}}$ that can solve an RSA puzzle z^*.

Looking ahead, in order to carry out the reduction, we need to make sure that adversary $\mathcal{A}_{\mathsf{RSA}}$ can identify the set R in advance, so that he can place his challenge value as $z^* = z_{j_i}$ for some j_i in the real set. To achieve this, we exploit the *observability* of the RO. Namely, the reduction $\mathcal{A}_{\mathsf{RSA}}$ can obtain the set F and R by observing the RO queries made by $\mathcal{B}^*$ to obtain the values h_R and h_F.

Formal Proof. The formal proof consists of two steps. First we show a PPT simulator S generates a simulated transcript for $\mathcal{B}^*$, by using only the information that $\mathcal{B}$ would get in the ideal world (that is, only signatures of fake values obtained through interaction with $\mathcal{F}_{\mathsf{promise\text{-}sign}}$). S exploits the extractability/programmability properties of the RO. Second, we prove that the view generated by the simulator S in the ideal world is computationally indistinguishable from the transcript in the real world.

Simulator S.
S, interacting with $\mathcal{F}_{\mathsf{promise\text{-}sign}}$, internally runs adversary $\mathcal{B}^*$ and simulates the messages that $\mathcal{B}^*$ expects from $\mathcal{T}$ as follows.

First, inform $\mathcal{F}_{\mathsf{promise\text{-}sign}}$ that $\mathcal{T}$ is honest. Then, compute $(PK^{eph}_{\mathcal{T}}, SK^{eph}_{\mathcal{T}}) = \mathsf{GenKey}(1^\lambda)$, and send $PK^{eph}_{\mathcal{T}}$ to $\mathcal{B}^*$ and to $\mathcal{F}_{\mathsf{promise\text{-}sign}}$.

(**A**) Upon receiving h_R, h_F, and $\{\beta_1 \dots \beta_{\mu+\eta}\}$ from $\mathcal{B}^*$, do the following:

1) Extract sets F and R from RO. To do this, look at the set of queries $\mathcal{Q}_H$ made by $\mathcal{B}^*$ and extract the pairs $(\text{salt}||R, h_R)$ and $(\text{salt}||F, h_F)$. If there is no pair with $(\cdot, h_R)$ or $(\cdot, h_F)$, then set $R = F = \perp$.
2) Send pairs (c_i, z_i) to $\mathcal{B}^*$ prepared as:
(1) For all i, $c_i \xleftarrow{\$} \{0,1\}^s$;
(2) For $i \in F$, $z_i = (\epsilon_i)^e$ where $\epsilon_i \xleftarrow{\$} \mathbf{Z}^*_N$;
(3) For $R = \{j_1, \dots, j_\mu\}$, $z_{j_i} = (q_i)^e \cdot z_{j_{i-1}}$ where $z_{j_1}, q_2, \dots q_\mu \xleftarrow{\$} \mathbf{Z}^*_N$.

(**B**) Upon receiving (F', R', r_i) from $\mathcal{B}^*$, do the following:

1) If $F' \neq F$ or $R \neq R'$, then abort.
2) If any $(\text{FakeFormat}||r_i, \beta_i) \notin \mathcal{Q}_{H'}$ for $i \in F$ then abort. For $j \in R$, set $m_j = \gamma$ if there exists $(\gamma,\, \beta_i) \in \mathcal{Q}_{H'}$. Else set $m_j = \perp$.
3) Send to $\mathcal{F}_{\mathsf{promise\text{-}sign}}$ the message $(\texttt{sign-request}, PK^{eph}_{\mathcal{T}}, \{\text{FakeFormat}||r_i\}_{i \in F}, \{m_j\}_{j \in R})$
Obtain response $(\texttt{promise}, |B^*, \mathsf{ANS}, \{\mathsf{FkSign}_i\}_{i \in [\eta]})$.
If $\mathsf{ANS} = \mathsf{NO}$, then halt and output whatever $\mathcal{B}^*$ outputs.
4) Compute $h_{j_\ell} = c_{j_\ell} \oplus \mathsf{FkSign}_\ell$.
Store in $\mathcal{Q}_{H^{\mathsf{shk}}}$ the pair $(\epsilon_{j_\ell, h_{j_\ell}})$.
5) Send ϵ_i for $i \in F$ and quotients $q_{j_1}, \dots, q_{j_\mu}$.

(**C**) Finally, output whatever $\mathcal{B}^*$ outputs and halt.

Procedure RO1: Random Oracle simulation for H proceeds as follows. Upon receiving query γ for H:

1) If query $\gamma \in \mathcal{Q}_H$, retrieve (γ, a) from $\mathcal{Q}_H$.
2) Else pick random $a \in \{0,1\}^{\lambda_2}$. Add tuple (γ, a) to $\mathcal{Q}_H$.
3) Output a.

Procedure RO2: Random Oracle simulation for H^{shk} proceeds as follows. Upon receiving query γ for H^{shk}:

1) If query $\gamma \in \mathcal{Q}_{H^{\mathsf{shk}}}$, retrieve (γ, a) from $\mathcal{Q}_{H^{\mathsf{shk}}}$.
2) If $(\gamma)^e = (z_i)$ for some $i \in R$ and no pair (γ, a) has been recorded yet in $\mathcal{Q}_{H^{\mathsf{shk}}}$, then output `RSA failure`.
3) Else, pick a random $a \in \{0,1\}^{\lambda_2}$. Add tuple (γ, a) to $\mathcal{Q}_{H^{\mathsf{shk}}}$.
4) Output a.

Indistinguishability Proof.
We now show that the transcript generated by S is indistinguishable from the transcript generated by $\mathcal{T}$ in the real world. This is done via a sequence of hybrid experiments. We start with the real world transcript, hybrid H_0, where the transcript of the protocol is computed following algorithm $\mathcal{T}$ (Figure 4). Then, in a sequence of hybrid experiments we change the way we compute the values β_i, c_i, z_i, q_i until we reach the final hybrid experiment where all values are computed following the algorithm S defined above.

H_0. This is the real world. The transcript is computed according to Protocol in Figure 4. Namely, the simulator follows exactly the same steps as the Tumbler $\mathcal{T}$.

$H_{0.5}$ (Learn R, F using observability of RO). In this hybrid experiment the simulator uses the observability of the RO H during the protocol execution. Namely, upon receiving message (h_F, h_R, β_i) from $\mathcal{B}^*$, we extract the queries $(\text{salt}||F, h_F)$ and $(\text{salt}||R, h_R)$ made to $\mathcal{Q}_H$ to identify the real and fake sets R, F. If no such query is found, but later $\mathcal{B}^*$ sends a well formed message, the simulator aborts.

The difference between the distribution of the transcript obtained in H_0 and that in $H_{0.5}$ is that the simulator aborts in $H_{0.5}$ if the RO H was not queried when forming h_R, h_R). The probability of aborting corresponds to the probability of correctly guessing the output of H. As H is modeled as a RO, this probability amounts to $1/2^{\lambda_2}$. Therefore experiments H_0 and $H_{0.5}$ are statistically close.

H_1 (Equivocate encryptions using programmability of RO). In this hybrid we change the way encryptions c_i are computed. Instead of computing

$$c_i = H^{\mathsf{shk}}(\epsilon_i) \oplus \sigma_i$$

In H_1, the simulator sets $c_i \overset{\$}{\leftarrow} \{0,1\}^s$ and stores the pair $(\epsilon_i, c_i \oplus \sigma_i)$ in $\mathcal{Q}_{H^{\mathsf{shk}}}$. Hybrids $H_{0.5}$ and H_1 are statistically close due to the unpredictability of the RO H^{shk} (when answers to RO H^{shk} are processed as in procedure RO2 of S above).

H_2 (Change computation of values for real set R using RSA security). In this hybrid the simulator computes z_{j_i}, q_i for $i \in R$, following the algorithm S described above. The differences are the following. For $j_i \in R$, in H_1 we have that

$$z_{j_i} = (\epsilon_{j_i})^e$$

while in H_2 we have that

$$z_{j_i} = (q_i)^e \cdot z_{j_{i-1}}$$

where $z_{j_1}, q_2, \dots q_\mu \overset{\$}{\leftarrow} \mathbf{Z}_N^*$. Note that in H_2, ϵ_{j_i} is neither computed nor stored in $\mathcal{Q}_{H^{\mathsf{shk}}}$. Thus, H_2 is different from H_1 because in H_2 procedure RO2 can trigger a `RSA failure` event and abort. (Because the `RSA failure` event happens when $\mathcal{B}^*$ queries oracle $\mathcal{Q}_{H^{\mathsf{shk}}}$ with the pre-image of a real puzzle z_{j_i}, it follows that the probability of an abort in H_2 is related to the probability of $\mathcal{B}^*$ of (RSA)-inverting z_{j_i} for some $j_i \in R$.) Therefore, to argue that H_1 and H_2 are computationally indistinguishable, we need to show that the distinguishing event – event `RSA failure` – happens only with probability that is negligible in λ.

Lemma 2: Assuming that RSA is hard in $\mathbf{Z}_N^*$, with $N > 2^\lambda$ then

$$Pr[\texttt{RSA failure}] \leq \nu(\lambda)$$

Proof: We can construct a reduction $\mathsf{Adv}_{\mathsf{RSA}}$ to the hardness of RSA trapdoor function using an adversary $\mathcal{B}^*$ that causes hybrid H_2 to abort due to an `RSA failure` event.

$\mathsf{Adv}_{\mathsf{RSA}}$ plays the RSA game, receiving values $(e, N), z^*$ from a challenger. The goal of $\mathsf{Adv}_{\mathsf{RSA}}$ is to output a the pre-image $x = (z^*)^d$ with non-negligible probability.

Meanwhile, the reduction's high-level goal is to place the challenge value z^* among the values $z_{j_1}, \dots, z_{j_\mu}$ with $j_i \in R$. Because B^* causes hybrid H_2 to abort due to `RSA failure` event, there exists some i such that $\mathcal{B}^*$ queries H^{shk} with $\epsilon_{j_i} = (z_{j_i})^d$, with non-negligible probability. Thus, if $\mathsf{Adv}_{\mathsf{RSA}}$ places z^* in position j_i, then $\mathsf{Adv}_{\mathsf{RSA}}$ wins the game with the same probability (discounted by a $1/\mu$ polynomial factor of guessing j_i correctly). The crux of the reduction is to show how $\mathsf{Adv}_{\mathsf{RSA}}$ generates the entire transcript for $\mathcal{B}^*$—and in particular the quotients $q_2, \dots, q_\mu$—without knowing the the pre-image of z_{j_i}. To do this, we have $\mathsf{Adv}_{\mathsf{RSA}}$ generate *all* z_{j_ℓ} for $j_\ell \in R$, without knowing their pre-image.

Reduction $\mathsf{Adv}_{\mathsf{RSA}}$.
$\mathsf{Adv}_{\mathsf{RSA}}$ receives (e, N, z^*) from the RSA challenger. $\mathsf{Adv}_{\mathsf{RSA}}$ chooses ECDSA ephemeral key $(SK_{\mathcal{T}}^{eph}, PK_{\mathcal{T}}^{eph})$. $\mathsf{Adv}_{\mathsf{RSA}}$ activates $\mathcal{B}^*$ on input $((e, N), PK_{\mathcal{T}}^{eph})$ and follow procedure run in H_2 by computing (z_{j_i}, q_i) as follows. $\mathsf{Adv}_{\mathsf{RSA}}$ first randomly picks an index $j_i \in R$ and sets $z_{j_i} = z^*$. Then she chooses values $q_{j_1}, \dots, q_{j_\mu}$ and remaining $z_{j_1}, \dots, z_{j_\mu}$ as follows.

1) For values preceding z_{j_i} (*i.e.,* for $0 < \ell < i$), pick $q_{\ell+1} \in \mathbf{Z}_N^*$; compute $z_\ell = \frac{z_{\ell+1}}{(q_{\ell+1})^e}$.
2) For values following z_{j_i} (*i.e.,* for $i < \ell \leq \mu$), pick $q_\ell \in \mathbf{Z}_N^*$; compute $z_\ell = (q_\ell)^e \cdot z_{\ell-1}$

If event `RSA failure` occurs, then procedure RO2 has observed an RSA pre-image some some z_i. If $z_i = z^*$ then $\mathsf{Adv}_{\mathsf{RSA}}$ outputs it and win the game. Else, she halts. ■

Summing up, in hybrid H_2, tuples $(\epsilon_i, c_i \oplus \sigma_i)$ for all $i \in R$ are not recorded in $Q_{H^{\mathsf{shk}}}$. In other words, neither ϵ_i, nor the signature σ_i for real messages m_i with $i \in R$ are computed in this hybrid.

H_3 (Obtaining signatures from $\mathcal{F}_{\mathsf{promise\text{-}sign}}$.) In this hybrid experiment the signatures σ_i for $i \in F$ are computed using $\mathcal{F}_{\mathsf{promise\text{-}sign}}$. That is, S sends $\mathcal{F}_{\mathsf{promise\text{-}sign}}$ the message
$(\texttt{sign-request}, \mathcal{B}, PK_{\mathcal{T}}^{eph}, \{\text{FakeFormat}||r_i\}_{i \in F},$
$\{m_j\}_{j \in R})$
If ANS =*yes*, S uses answers $\sigma_i = \mathsf{FkSign}_i$ to set $(\epsilon_i, c_i \oplus \sigma_i)$ in $\mathcal{Q}_{H^{\mathsf{shk}}}$. From $\mathcal{B}^*$'s the point of view, hybrid H_2 and H_3 are identical. Experiment H_3 thus corresponds to the exact simulation strategy S described above. This conclude the proof.

In the above proof we have shown that any PPT $\mathcal{B}^*$ does not learn anything from the transcript obtained in Protocol in Figure 4. We now show that if $\mathcal{B}^*$ does indeed output a valid signature σ^* for a valid message $T_{\text{cash}(\mathcal{T},\mathcal{B})}$, then $\mathcal{B}^*$ (who is not getting any information from the transcript), must have produced a signature forgery. Define event E_{forge} as the event where, a PPT $\mathcal{B}^*$ runs the protocol in Figure 4 and outputs a pair $(T_{\text{cash}(\mathcal{T},\mathcal{B})}, \sigma)$ where $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ is a real message rather than a fake message (*i.e.,* $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ is a valid cash-out transaction for $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ that does not conform to FakeFormat). We now prove the following.

Lemma 3: If ECDSA is an existentially unforgeable signature scheme, $Pr[E_{\mathsf{forge}}]$ is negligible.

Proof: We can construct an adversary $\mathsf{Adv}_{\mathsf{ecdsa}}$ that forges a signature on a new message $T_{\text{escr}(\mathcal{T},\mathcal{B})}{}^i$ using adversary $\mathcal{B}^*$.

$\mathsf{Adv}_{\mathsf{ecdsa}}$ plays the signature game and has oracle access to the signing algorithm $\mathcal{O}$, and has verification key $PK_{\mathcal{T}}^{eph}$. The goal of $\mathsf{Adv}_{\mathsf{ecdsa}}$ is to use $\mathcal{B}^*$ to produce a signatures σ^* on a message that was never queried to $\mathcal{O}$. $\mathsf{Adv}_{\mathsf{ecdsa}}$ simulates the interaction between

$\mathcal{B}$ and $\mathcal{T}$ using algorithm S. Recall that S obtains the signatures by interacting with the ideal functionality $\mathcal{F}_{\mathsf{promise\text{-}sign}}$ and, in particular, S only queries $\mathcal{F}_{\mathsf{promise\text{-}sign}}$ for signatures on fake messages, *i.e.,* , with messages in $fk_i \in$ FakeFormat. Thus, the reduction $\mathsf{Adv}_{\mathsf{ecdsa}}$ will simply run S's algorithm, and when S queries $\mathcal{F}_{\mathsf{promise\text{-}sign}}$, $\mathsf{Adv}_{\mathsf{ecdsa}}$ will use its access to $\mathcal{O}$ to generate the correct signatures. It follows from the previous hybrid arguments that $\mathcal{B}^*$ cannot distinguish whether she is talking to $\mathcal{T}$ or S. Therefore, the probability of $\mathcal{B}^*$ generating a forgery when interacting with $\mathcal{T}$ is close (up to a negligible factor) to the probability of $\mathcal{B}^*$ generating a forgery when interacting with S and therefore $\mathsf{Adv}_{\mathsf{ecdsa}}$.

If $\mathcal{B}^*$ outputs the pair (σ, m), and $m \notin$ FakeFormat, then $\mathsf{Adv}_{\mathsf{ecdsa}}$ has obtained her forgery (σ, m). Thus, $Pr[E_{\mathsf{forge}}] = Pr[E_{\mathsf{forge-ECDSA}}] - \nu(\lambda)$. Which is negligible assuming EDCSA signature scheme is secure. ■

2) $\mathcal{T}$ is corrupted: We now show that the view of any corrupted $\mathcal{T}^*$, playing with an honest $\mathcal{B}$, can be simulated by a simulator $S_{\mathcal{T}}$ that only has access to the ideal functionality $\mathcal{F}_{\mathsf{promise\text{-}sign}}$.

Proof Intuition. In the ideal world, $\mathcal{T}$ needs to decide whether to grant signatures to $\mathcal{B}$ (that is, set ANS to *yes* or *no*, and the indexes in Set) in a committing manner: if ANS =*yes* then $\mathcal{T}$ has no power to prevent $\mathcal{B}$ from getting the promised signatures later. This is because ideal functionality $\mathcal{F}_{\mathsf{promise\text{-}sign}}$ has access to the algorithm Sig, and when ANS =*yes* the ideal functionality proceeds with the computation of the required signatures for the fake messages, and has the ability to sign the real messages in the future.

Now, the goal of the simulator $S_{\mathcal{T}}$, is twofold: (1) To decide whether $\mathcal{T}$ should set ANS to *yes* or *no*, and to choose Set in the ideal world. (2) To correctly compute the signatures requested by $\mathcal{F}_{\mathsf{promise\text{-}sign}}$ via Sig. To this end, $S_{\mathcal{T}}$ will do as follows. $S_{\mathcal{T}}$ interacts with real world $\mathcal{T}^*$, and if $\mathcal{T}^*$ provides an accepting transcript, then $S_{\mathcal{T}}$ will play ANS =*yes*. Then, by using the observability of the RO H^{shk}, $S_{\mathcal{T}}$ will extract the signatures σ_i for $i \in F \cup$ Set and uses these signatures to produce the output of the signing algorithm Sig.

At high-level, a bad case for the simulator $S_{\mathcal{T}}$ is when (1) the transcript is accepting[13] and $S_{\mathcal{T}}$ sent promise 'ANS =*yes*, Set' to $\mathcal{F}_{\mathsf{promise\text{-}sign}}$, but (2) the real $\mathcal{T}^*$ did not make RO queries that allow $S_{\mathcal{T}}$ to recover σ_i from c_i for all $i \in F \cup$ Set. That is, the bad case happens when the promise is fulfilled in the ideal world, but not in the real world.

Thus, the crux of the proof is to show that the probability of the bad event is negligible if $\mathcal{T}^*$ provides an accepting transcript. That is, when the transcript is accepting and $S_{\mathcal{T}}$ plays ANS =*yes* in the ideal world, also real world $\mathcal{B}$ is guaranteed that will receive the promised signature. At a high level, this holds due to the following reasons.

1. Fake-Set Test. Due to the *perfect hiding* of the RO, sets F and R are information theoretically hidden for $\mathcal{T}^*$. Thus the probability that $\mathcal{T}^*$ successfully passes the cut-and-choose phase, (*i.e.,* the Fake Set Check in Figure 4) and that there is no $i \in R$ such that (c_i, z_i) is correctly formed, corresponds to the probability of correctly guessing the set F. This happens with probability: $\frac{1}{\binom{\eta+\mu}{\eta}}$

2. Quotient Test. The quotients $q_2, \ldots, q_\mu$ guarantees that knowledge of $\epsilon_{j_1} = (z_{j_1})^d$ for $j_1 \in R$, allows $\mathcal{B}$ to learn *all* remaining keys $\epsilon_{j_2}, \ldots, \epsilon_{j_\mu}$. To see why, notice that if $\mathcal{T}^*$ passes the Quotient Test, it means that for each i, $z_{j_i} = q_i \cdot z_{j_{i-1}}$. Thus unlocking z_{j_1} recovers ϵ_{j_1} that in turns unlocks z_{j_2} which recovers ϵ_{j_2} and so on. Therefore, even if only one ciphertext c_{j_i} contains a valid signature, $\mathcal{B}$ will be able to decrypt c_{j_i} and recover that signature.

Formal proof. We now proceed with the formal argument. We present the simulator $S_{\mathcal{T}}$, the algorithm Sig (which is part of $\mathcal{F}_{\mathsf{promise\text{-}sign}}$, see Figure 8), and finally argue that the transcript generated by the simulator in the ideal world is indistinguishable from that in the real world.

Simulator S.
S runs $\mathcal{T}^*$ internally.

Upon receiving $(\texttt{KeyGen}, \mathcal{B})$, send request to $\mathcal{T}^*$ and obtain $PK_{\mathcal{T}}^{eph}$. Send $(PK_{\mathcal{T}}^{eph}, \mathsf{Sig})$ to $\mathcal{F}_{\mathsf{promise\text{-}sign}}$ where Sig is defined as below.

Upon receiving $(\texttt{sign-request}, \mathcal{B}, PK_{\mathcal{T}}^{eph}, \{\mathsf{FkTxn}_i\}_{i \in [\eta]})$: Randomly pick R, F with $R \cap T = \emptyset$, compute the RO outputs h_F, h_R and $\beta_1, \ldots, \beta_{\eta+\mu}$, and send them to $\mathcal{T}^*$.

Upon receiving a pair (c_i, z_i) from $\mathcal{T}^*$:

1) Extract ϵ_i by observing queries to H^{shk}.
2) Let σ_i be the signature decrypted from c_i using ϵ_i.
3) If $c_i, z_i, \epsilon_i, \beta_i, \sigma_i$ for $i \in F$ pass all validity checks, record tuple $(\mathsf{FkTxn}_i, \beta_i, \sigma_i)$ in L_{fake}.
4) If $c_i, z_i, \epsilon_i, \beta_i, \sigma_i$ for $i \in R$ pass all validity checks, add i to set Set, and add (β_i, σ_i) to L_{real}. If no such i exists, set $\mathsf{real} =$ *no*. (Note that, for the real set, it is sufficient that one a single ϵ_l, with $l \in R$ is correct. This is because the quotient chain guarantees that knowledge of ϵ_{j_1} allows $\mathcal{B}$ to obtain all ϵ_{j_i} for $j_i \in R$.)

For all $i \in F$, pick randomness r_i, and send it $\mathcal{T}^*$. Add pair $(\mathsf{FkTxn}_i || r_i, \beta_i)$ to $Q_{H'}$.

Upon receiving the 'openings' ϵ'_i to fake messages $i \in F$, use ϵ'_i to obtain σ'_i from c_i. If

[13] A transcript is accepting if the honest player $\mathcal{B}$ completes the protocol without aborting.

any $(i, c_i, z_i, \epsilon'_i, \beta_i, \sigma'_i)$ fails any validity check, send $(\texttt{promise}, \mathcal{B}, NO, \bot)$ to $\mathcal{F}_{\textsf{promise-sign}}$. Else, if all checks pass, send $(\texttt{promise}, \mathcal{B}, yes, \textsf{Set})$.

Now we have two cases:

Case 1: Suppose there exists $i \in F$ such that tuple $(i, \epsilon'_i, \beta_i, \sigma'_i)$ passes all validity checks but $(\cdot, \beta_i, \sigma'_i)$ is *not* recorded in L_{fake}. Then, abort and output `binding-fail!`.

Case 2: Otherwise, for all $i \in F$, we have that $(i, \epsilon'_i, \beta_i, \sigma'_i)$ pass all validity tests and $(\cdot, \beta_i, \sigma'_i)$ is recorded in L_{fake}.

1) If $\textsf{real} = no$, abort and output `cut-and-choose-fail!`.
2) Else, set variables L_{fake} and L_{real} for algorithm $\textsf{Sig}$.

Algorithm $\textsf{Sig}(m_i, PK^{eph}_{\mathcal{T}})$.
Internal variable L_{fake} and L_{real}.
If there is tuple (m_i, β_i, σ_i) in L_{fake}, output signature (β_i, σ_i).
Else if there is tuple (β_i, σ_i) in L_{real}, then add tuple (m_i, β) to $Q_{H'}$, and output signature (β_i, σ_i).
Else, abort.

Indistinguishability Proof.
The protocol messages generated by $S_{\mathcal{T}}$ interacting with $\mathcal{T}^*$ are distributed identically to the transcript produced by a real $\mathcal{B}$. The only difference between the distribution of the output of the real and ideal world is that the simulator aborts more often. Thus, proving indistinguishability between the two distributions, amounts to proving that events `binding-fail!` and `cut-and-choose-fail!` happen with negligible probability.

Let us look at each event, and argue why they occur with negligible probability

Event `cut-and-choose-fail!` This event happens when the transcript is accepting and all the fake values are computed correctly, but *all* the real values are incorrect (*i.e.,* $S_{\mathcal{T}}$ set $real = no$). Namely $\texttt{cut} - \texttt{and} - \texttt{choose} - \texttt{fail}!$ happens when:

1) For all $i \in F$, $\mathcal{T}^*$ provides consistent responses ϵ_i, which were queried to $H^{\textsf{shk}}$.
2) For all $j \in R$, there exists no $\epsilon_j \in \mathcal{Q}_{H^{\textsf{shk}}}$ that can be used to decrypt c_j to a valid signature σ_j on β_j.

By the hiding of H, H', probability of $\texttt{cut} - \texttt{and} - \texttt{choose} - \texttt{fail}!$ corresponds to the probability that $\mathcal{T}^*$ guesses the set F which is:

$$\frac{1}{\binom{\mu+\eta}{n}} + \frac{1}{2^{\lambda_1}}$$

.

Event `binding-fail!` This event happens when (1) $\mathcal{T}^*$ did not query the random oracle $H^{\textsf{shk}}$ with string ϵ'_i, but (2) later he sends ϵ'_i that passes all validity checks and is such that

$$c_i = H^{\textsf{shk}}(\epsilon'_i) \oplus \sigma_i$$

where σ_i is a valid signature on β_i. This event happens if somehow $\mathcal{T}^*$ was able to predict the output of $H^{\textsf{shk}}$ without actually querying $H^{\textsf{shk}}$, or if $\mathcal{T}^*$ finds two ϵ_i, ϵ'_i such that $z_i = \epsilon_i$ and $z_i = \epsilon'_i$. Due to the fact that RSA is a permutation and that $\mathcal{B}$ checks that $\epsilon_i < N$, the latter event happens with probability 0. Due to the one-wayness of the random oracle, the first event happens when $\mathcal{T}^*$ guesses the output of H, which happens with negligible probability $1/2^{\lambda_2}$

G. Proof of Theorem 3: the case of Q payments

We now prove the Theorem 3 for puzzle-promise protocol that allows Bob $\mathcal{B}$ to obtain Q payments. The proof follows similar arguments used to prove Theorem 2 for the "base case" where $Q = 1$ in Appendix F.

1) Case $\mathcal{B}$ is corrupt: We outline the key differences w.t.r to the simulator and the indistinguishability proof provided in Appendix F.

Simulator.
The simulator for protocol in Figure 6, that we denote by S^Q, follows the same steps as the simulator S shown for the base case, with the following modifications:

Step (**A**): S receives h_R, h_F and $\beta_{j,i}$ for $j \in Q, i \in [\eta + \mu]$.

Step (**B**.2): S additionally receives $\rho_{j,i}$ for $j \in [Q], i \in R$, checks whether

$$(\text{CashOutFormat}(j, \rho_{j,i}) \in \mathcal{Q}_{H'}$$

and, if so, sets the real messages $m_{j,i} = \text{CashOutFormat}(j, \rho_{j,i})$, and the fake messages $\textsf{FkTxn}_{j,i} = \textsf{FakeFormat}||r_{j,i}$. Else, it aborts. The difference here is that for the case of Q payments we also check the *semantics* of the real messages. That is, a real message for level j must be a cash-out transaction that transfers exactly j bitcoins to Bob.

Step (**B**.3): For each $j \in Q$, S sends

$$(\texttt{sign-request}, PK^{eph'}_{\mathcal{T}}, \{\textsf{FkTxn}_{j,\ell}\}_{\ell \in F}, \{m_{j,i}\}_{i \in R})$$

Step (**B**.4): For $j \in Q$, $\ell \in F$, S stores in $\mathcal{Q}_{H^{\textsf{shk}}}$ the pair

$$([j, \ell, \epsilon_{j,\ell}||\epsilon_{j-1,\ell}|| \ldots, \epsilon_{1,\ell}], h_{j,\ell})$$

Procedure RO2: The difference here is that in the case of Q payments, when decrypting a ciphertext at level j, $\mathcal{B}^*$ needs to query the RO $H^{\textsf{shk}}$ with all the $\epsilon_{j'}$ with $j' < j$. Thus, Procedure RO2 is modified as follows. A query γ is parsed as $(j|\ell|\gamma_j|\gamma_{j-1}| \ldots, \gamma_1)$, and the procedure aborts and outputs `RSA failure` if there exits a $j^* \in [j]$ such that $(\gamma_{j^*})^e = z_{j^*,\ell}$ with

$\ell \in R$, and no pair $([j^*, \ell, \ldots ||\gamma_{j^*}|| \ldots], a)$ has been recorded yet in $\mathcal{Q}_{H^{\mathsf{shk}}}$.

Indistinguishability Proof.
The indistinguishability proof for the output of S^Q follows the same hybrid experiments shown for arguing indistinguishability of the output of S in AppendixF. In particular, experiments $H_{0.5}$ and H_3 can be directly extended for the case of Q payments. Denote the extended experiments by $H_{0.5}^Q$ and H_3^Q.

In experiment H_2, S changes the way the real RSA puzzles are computed by not recording the RSA-solutions ϵ_{i_ℓ} for $i_\ell \in R$ in $\mathcal{Q}_{H^{\mathsf{shk}}}$. This change potentially triggers event `RSA failure` in the RO2 procedure. The distribution of hybrid H_2 is only computationally indistinguishable from hybrid H_1 in the proof of the base case in of Appendix F. So, when we deal with of Q payments we use a sequence of sub-hybrids H_2^Q, H_2^{Q-1}, …, H_2^1. In sub-hybrid H_2^j we change the j-th *row of* real RSA puzzles. Hence, first we define hybrid H_2^j as the experiment where: (1) the j-th row of puzzles $z_{j,i_1}, \ldots, z_{j,i_\mu}$ is computed as:

$$z_{j,i_l} = (q_l)^e \cdot z_{j,i_{l-1}}, \forall l \in [\mu]$$

(2) queries $(j||\ell|\epsilon_{j,\ell}, \epsilon_{j-1,\ell}, \ldots, \epsilon_{1,\ell})$ for $\ell \in R$ are *not* recorded in $\mathcal{Q}_{H^{\mathsf{shk}}}$.

Then, indistinguishability of experiment H_2^j and H_2^{j-1} can be argued by following the same argument as Lemma 2: the RSA reduction $\mathsf{Adv}_{\mathsf{RSA}}$ will place its challenge z^* among the puzzles of the j-th row $z_{j,i_1}, \ldots, z_{j,i_\mu}$ ($i_l \in R$), while computing all the remaining rows as in H_2^j.

H. Case $\mathcal{T}$ is corrupt

Here it will be most convenient to present the full simulator $S_{\mathcal{T}}^Q$ for the case of Q payment. We then provide an indistinguishability proof by presenting its key differences w.r.t. the base case shown in Appendix F.

Simulator.
Conceptually, there is no difference between $S_{\mathcal{T}}^Q$ and $S_{\mathcal{T}}$ (the simulator for the base case in Appendix F). The both send ANS =*yes* to $\mathcal{F}_{\mathsf{promise\text{-}sign}}$ if the transcript is accepting, and they both try to extract keys $\epsilon_{j,i}$ from the RO H^{shk}, for all $i \in F$ and for some $i \in R$. For both simulators, the bad event corresponds to case when the transcript is accepting but the simulator does not observes a RO query that contains an ϵ_i that allows σ_i to be recovered from c_i for all $i \in F \cup \mathsf{Set}$.

The only difference between $S_{\mathcal{T}}$ and $S_{\mathcal{T}}^Q$ is what constitutes a good key, that is, a key that allows the simulator to decrypt a signature. In the base case of 1 payment, a good key is a string ϵ_i such that $H^{\mathsf{shk}}(\epsilon_i) \oplus c_i = \sigma_i$ and σ_i is a valid signature and $z_i = (\epsilon_i)^e$. In the case of Q payment, we need Q good keys, and the j-th good key is a chain $(\epsilon_{j,i}, \epsilon_{j-1,i}, \ldots, \epsilon_{1,i})$ of good keys.

Simulator $S_{\mathcal{T}}^Q$.
$S_{\mathcal{T}}^Q$ runs $\mathcal{T}^*$ internally.

Upon receiving $(\mathtt{KeyGen}, \mathcal{B})$ send request to $\mathcal{T}^*$ and obtain $PK_{\mathcal{T}}^{eph}$. Send $(PK_{\mathcal{T}}^{eph}, \mathsf{Sig})$ to $\mathcal{F}_{\mathsf{promise\text{-}sign}}$ where Sig is defined below.

Upon receiving

$$(\mathtt{sign\text{-}request}, PK_{\mathcal{T}}^{eph'}, \{\mathsf{FkTxn}_{j,i}\}_{i \in [\eta]}, \{m_{j,i}\}_{i \in [\mu]})$$

for all $j \in [Q]$. Randomly pick R, F with $R \cap F = \emptyset$, compute the RO outputs h_F, h_R and $\beta_{j,i}$ for $j \in Q$ and $i \in [\eta + \mu]$, and send them to $\mathcal{T}^*$.

Upon receiving pairs $(c_{j,i}, z_{j,i})$ from $\mathcal{T}^*$. Then, do the following for $j = 1, \ldots, Q$:

1) Extract key $\epsilon_{j,i}$ by observing queries to H^{shk} that have format $(j|i|\epsilon_{j,i}|\epsilon_{j-1,i}| \ldots |\star)$, where $\epsilon_{j-1,i}, \ldots \epsilon_{1,i}$ are the keys extracted previously.
2) Let $\sigma_{j,i}$ be the signature decrypted from $c_{j,i}$ using $\epsilon_{j,i}, \ldots, \epsilon_{1,i}$.
3) If $c_{j,i}, z_{j,i}, \epsilon_{j,i}, \beta_{j,i}, \sigma_{j,i}$ for $i \in F$ pass all validity checks, record tuple $(\mathsf{FkTxn}_{j,i}, \beta_{j,i}, \sigma_{j,i})$ in L_{fake}.
4) If $c_{j,i}, z_{j,i}, \epsilon_{j,i}, \beta_{j,i}, \sigma_{j,i}$ for $i \in R$ pass all validity checks add pair (j, i) to a temporary set $\mathsf{TempSet}$. Else add $(j, \perp)$ in $\mathsf{TempSet}$.

If there exist no i such that for all j, tuple $(j, i) \in \mathsf{TempSet}$ then set $\mathsf{real} = $ *no*. Else, store in Set the indexes i such that $(j, i) \in \mathsf{TempSet}$ for all $j \in Q$. Then, only for $i \in \mathsf{Set}$, record tuple $(m_{j,i}, \beta_{j,i}, \sigma_{j,i}) \in L_{real}$ for all $j \in [Q]$.

For all $j \in [Q]$ and $i \in F$, pick randomness $r_{j,i}$, and send it $\mathcal{T}^*$. Add pair $(\mathsf{FkTxn}_{j,i}||r_{j,i}, \beta_{j,i})$ to $\mathcal{Q}_{H'}$.

For all $j \in [Q]$ and $i \in R$, pick randomness $\rho_{j,i}$, and send it $\mathcal{T}^*$. Add pair $(m_{j,i}||\rho_{j,i}, \beta_{j,i})$ to $\mathcal{Q}_{H'}$.

For all $j \in [Q]$, upon receiving the 'openings' $\epsilon'_{j,i}$ to fake messages $i \in F$, use $\epsilon'_{1,i}, \ldots, \epsilon'_{j,i}$ to obtain $\sigma'_{j,i}$ from $c_{j,i}$. If any $(j, i, c_{j,i}, z_{j,i}, \epsilon'_{j,i}, \beta_{j,i}, \sigma'_{j,i})$ fails any validity check, send $(\mathtt{promise}, \mathcal{B}, NO, \perp)$ to $\mathcal{F}_{\mathsf{promise\text{-}sign}}$. Else, if all checks pass, send $(\mathtt{promise}, \mathcal{B}, YES, \mathsf{Set})$.

Now we have two cases:

Case 1: Suppose there exists $i \in F$ and $j \in [Q]$ such that tuple $(i, \epsilon'_{j,i}, \beta_{j,i}, \sigma'_{j,i})$ passes all validity checks but $(\cdot, \beta_{j,i}, \sigma'_{j,i})$ is *not* recorded in L_{fake}. Then, abort and output `binding-fail!`.

Case 2: Otherwise, for all $i \in F$ and $j \in [Q]$, we have that $(j, i, \epsilon'_{j,i}, \beta_{j,i}, \sigma'_{j,i})$ pass all validity tests and $(\cdot, \beta_{j,i}, \sigma'_{j,i})$ is recorded in L_{fake}.

1) If $\mathsf{real} = no$, abort and output `cut-and-choose-fail!`.
2) Else, set variables L_{fake} and L_{real} for algorithm Sig.

Algorithm $\mathsf{Sig}(m_i, PK^{eph}_{\mathcal{T}})$.
Variables: L_{fake} and L_{real}.
Search for record $(m_{j,i}, \beta_{j,i}, \sigma_{j,i})$ in L_{fake} and L_{real} and output $(\beta_{j,i}, \sigma_{j,i})$. Else, abort.
Note that $m_{i,j}$ cannot be found in both lists. This is because L_{fake} is populated only with signatures of messages in FakeFormat, and L_{real} only collects signatures of messages in RealFormat, and a message $m_{j,i}$ cannot be both fake and real.

Indistinguishability Proof.
As for the base case, the indistinguishability proof amounts to show that the probability that $S^Q_{\mathcal{T}}$ aborts in the simulation is negligible. As in the base case (Appendix F), $S^Q_{\mathcal{T}}$ will abort if the test phase passes successfully, but later he finds that $\mathcal{T}^*$ did not compute any $i \in R$ correctly, therefore not delivering any promise. We focus only on the event `cut-and-choose-fail!`(event `binding-fail!` depends only on the failure of the random oracle, and is independent of the number of payments). We argue that probability of such event is negligible, due to the same arguments as the base case. Too see why, consider the following observations.

In the base case, a coordinate $i \in F$ passes the validity check if the tuple z_i, c_i, σ_i is correctly computed. In the Q-payments case, a coordinate $i \in F$ passes the check if the column $z_{j,i}, c_{j,i}, \sigma_{j,i}$ is correctly computed for all $j \in [Q]$. That is, starting from $j = 1$, it holds that $c_{1,i}$ is correctly computed using keys $\epsilon_{1,i}$, and for $j = 2$, $c_{2,i}$ is correctly computed using both keys $(\epsilon_{2,i}|\epsilon_{1,i})$ and so on and so forth. Thus, for column i we have that $\mathcal{B}$ is guaranteed that he can decrypt all signatures for all $j = [Q]$.

Now, recall that this check is performed for all $i \in F$. This means that for η coordinates $\mathcal{T}^*$ correctly computed *for all levels* $j \in [Q]$. Hence, the probability that all columns $i \in F$ are computed correctly in all Q levels, but there is no column $i \in R$ that is completely correct in all Q levels, amounts to the probability of guessing F. Thus, it follows that there exists at one column $i \in R$, where values $(z_{j,i} c_{j,i}, \sigma_{j,i})$ are computed correctly *for all levels* j, except with probability $\frac{1}{\binom{\mu+\eta}{n}}$.

However, we need a second argument that shows that, even one good index $i \in R$ suffices for $\mathcal{B}$ to recover his required signatures. To argue this, we use the quotient chains provided for each $j \in Q$. The quotient chain $q_{j,1}, \ldots, q_{j,\mu}$ guarantees that for level j, solving puzzle z_{j,i_1} for $i_1 \in R$ is sufficient to then unlock *all* puzzles in level j. This guarantees that for every level j, $\mathcal{B}$ will be able to unlock keys $\epsilon_{j,i}$ for all $i \in R$. Thus, if there is at least on $i \in R$ where the entire column is correctly formed, then $\mathcal{B}$ is guaranteed to decrypt at least one entire column (and $S^Q_{\mathcal{T}}$ is guaranteed to not abort) except with probability $\frac{1}{\binom{\mu+\eta}{n}}$.

I. Details of our Bitcoin Scripts

Figure 9 overviews the relationships between the transactions used in the TumbleBit protocol. We walk through the details of our transactions, explain why they conform to the Pay-To-Script-Hash (P2SH) [3] template, and discuss why TumbleBit protocol is not affected by the *transaction malleability* issue [5] of the current Bitcoin protocol:

Transaction malleability. The transaction malleability issue is roughly explained as follows. If one bitcoin transaction T_{fulfill} fulfills another bitcoin transaction T_{offer}, then T_{fulfill} must contain a *pointer* to T_{offer}. The pointer is the TXID, which is the hash of entire T_{offer} transaction, *including any signatures on that transaction.* Now, bitcoin uses ECDSA signatures over the Secp256k1 elliptic curve. It is well known that ECDSA signatures are not deterministic. First, a party that holds the secret signing key can easily produce multiple valid signatures on a single message m. Second, even a party that does not know the secret signing key can take a valid signature on a message m, and maul it to produce a different valid signature on m. Now, because TXID is the hash of the *entire* T_{offer} transaction, including all signatures on that transaction, mauling these signatures results to a different TXID of T_{offer}. Such mauling attacks are not a problem for a transaction that is already in a blockchain, but they can cause problems to transactions that are still unconfirmed.

The Bitcoin community is currently considering patching transaction malleability using a solution called *segregated witness* [59], but as of this writing it has not been fully deployed [20]. TumbleBit, however, remains secure even in the absence of segregated witness.

1) Interaction between Tumbler and Bob.: The right side of the Figure 9 presents the transactions used for the interaction between the Tumbler $\mathcal{T}$ and Bob $\mathcal{B}$. The script in the $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ transaction offers 1 bitcoin to a fulfilling transaction satisfies the condition $(\mathcal{B} \wedge \mathcal{T}) \vee (\mathcal{T} \wedge tw_2)$, *i.e.,* a transaction that is either (1) signed by both $\mathcal{B}$ and $\mathcal{T}$, or (2) signed by $\mathcal{T}$ and posted to the blockchain after timewindow tw_2. Condition (2) is timelocked refund condition which is scripted as follows:

```
locktime
OP_CHECKLOCKTIMEVERIFY
OP_DROP
payer_pubkey
OP_CHECKSIG
```

where `locktime` is a timewindow (*i.e.,* an absolute block height). All subsequent descriptions of our scripts use `refund_condition` as a placeholder for the script above. For $T_{\text{escr}(\mathcal{T},\mathcal{B})}$, the refund condition script has `locktime` is set to tw_2 and `payer_pubkey` is set to the Tumbler's public key.

Now, the full *redeem script* for the two-of-two escrow transaction $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ is as follows:

```
OP_IF
OP_2
payer_pubkey
redeemer_pubkey
OP_2
OP_CHECKMULTISIG,
OP_ELSE
refund_condition
OP_ENDIF
```

where `payer_pubkey` is the Tumbler's public key, `redeemer_pubkey` is Bob's public key, and the `refund_condition` is scripted as described above with `locktime` set equal to tw_2. Note that instructions up to and including `OP_CHECKMULTISIG` checks for the condition $\mathcal{T} \wedge \mathcal{B}$—checking if a valid $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ has been posted that is signed by both Tumbler $\mathcal{T}$ and Bob $\mathcal{B}$. The redeem script above is hashed and its hash is stored in $T_{\text{cash}(\mathcal{T},\mathcal{B})}$. (This ensures that the transaction conform to the to the Pay-To-Script-Hash (P2SH) [3] template.)

If $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ is posted to the blockchain, it contains (1) the redeem script above and (2) the following *input values* that include the required two signatures:

```
OP_FALSE
payer_signature
redeemer_signature
OP_TRUE
```

To programmatically validate that $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ can fulfill $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ (per the P2SH template), the redeem script in $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ is hashed, and the resulting hash value is compared to the hash value stored in $T_{\text{escr}(\mathcal{T},\mathcal{B})}$. If these match, the redeem script is run against the input values in $T_{\text{cash}(\mathcal{T},\mathcal{B})}$. $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ fulfills $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ if the redeem script outputs true.

Meanwhile, if Bob $\mathcal{B}$ refuses to post $T_{\text{cash}(\mathcal{T},\mathcal{B})}$ before the timewindow tw_2 ends, then the Tumbler $\mathcal{T}$ can reclaim the bitcoin escrowed in $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ by posting a refund transaction $T_{\text{refund}(\mathcal{T},\mathcal{B})}$. (See the right side of Figure 9.) When $T_{\text{refund}(\mathcal{T},\mathcal{B})}$ is posted to the blockchain, it contains (1) the redeem script above and (2) the following input values, where `signature` is a signature that verifies under `payer_pubkey`:

```
Signature
OP_FALSE
```

$T_{\text{refund}(\mathcal{T},\mathcal{B})}$ fulfills $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ if the hash of the redeem script in $T_{\text{refund}(\mathcal{T},\mathcal{B})}$ matches the hash value stored in $T_{\text{escr}(\mathcal{T},\mathcal{B})}$, and if the redeem script in $T_{\text{refund}(\mathcal{T},\mathcal{B})}$ outputs true when run against the input values in $T_{\text{refund}(\mathcal{T},\mathcal{B})}$.

Notice that Bob $\mathcal{B}$ is not involved in constructing the refund transaction $T_{\text{refund}(\mathcal{T},\mathcal{B})}$; indeed, $T_{\text{refund}(\mathcal{T},\mathcal{B})}$ need only be signed by the Tumbler $\mathcal{T}$. There are two reasons why this is crucial.

First, $T_{\text{refund}(\mathcal{T},\mathcal{B})}$ must be posted when $\mathcal{B}$ becomes uncooperative. Thus, Tumbler $\mathcal{T}$ can singlehandedly post $T_{\text{refund}(\mathcal{T},\mathcal{B})}$, and reclaim his bitcoin, even in cases where Bob refuses to interact with $\mathcal{T}$.

The second reason is the *transaction malleablity* issue [5] of the current Bitcoin protocol. Suppose that Bob $\mathcal{B}$ mauls[14] his signature on transaction $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ before it is posted to the blockchain, causing the TXID for $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ to change from the TXID value expected by $\mathcal{T}$. This has no effect on the Tumbler's ability to post the refund transaction $T_{\text{refund}(\mathcal{T},\mathcal{B})}$. Specifically, before posting $T_{\text{refund}(\mathcal{T},\mathcal{B})}$, the Tumbler need only find $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ on the blockchain, hash $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ to obtain its TXID, and use this TXID when it forming his refund transaction $T_{\text{refund}(\mathcal{T},\mathcal{B})}$. By contrast, suppose our protocol had instead somehow required Bob to participate in forming $T_{\text{refund}(\mathcal{T},\mathcal{B})}$ *before* $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ had been posted to the blockchain. Then a malicious Bob could give the Tumbler a valid $T_{\text{refund}(\mathcal{T},\mathcal{B})}$ on $T_{\text{escr}(\mathcal{T},\mathcal{B})}$. Then, Bob could maul the signatures on $T_{\text{escr}(\mathcal{T},\mathcal{B})}$, and then post the mauled $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ to the blockchain. $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ would still be a valid transaction, but the $T_{\text{refund}(\mathcal{T},\mathcal{B})}$ held by the Tumbler would be useless, because $T_{\text{refund}(\mathcal{T},\mathcal{B})}$ no longer points to $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ (because Bob has mauled the TXID of $T_{\text{escr}(\mathcal{T},\mathcal{B})}$).

2) Interaction between Tumbler and Bob.: The left side of the Figure 9 presents the transactions used for the interaction between Alice $\mathcal{A}$ and the Tumbler $\mathcal{T}$. The script in the $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ transaction offers 1 bitcoin to a fulfilling transaction satisfies the condition $(\mathcal{A} \wedge \mathcal{T}) \vee (\mathcal{T} \wedge tw_1)$. The redeem script for this transaction is identical to the one used in $T_{\text{escr}(\mathcal{T},\mathcal{B})}$, except that now `payer_pubkey` is Alice's public key, `redeemer_pubkey` is the Tumbler's public key, and the `locktime` in the `refund_condition` is set equal to tw_1. $T_{\text{cash}(\mathcal{A},\mathcal{T})}$ from Figure 9 is formed analogously to $T_{\text{cash}(\mathcal{T},\mathcal{B})}$, and the refund $T_{\text{refund}(\mathcal{A},\mathcal{T})}$ pointing to $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ is formed analogously to $T_{\text{refund}(\mathcal{T},\mathcal{B})}$.

Recall from Section V-D, that in the puzzle-solver protocol, Alice forms and signs T_{puzzle} and sends it to the Tumbler. Transaction T_{puzzle} fulfils $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ via the condition $(\mathcal{A} \wedge \mathcal{T})$. Thus, (just like $T_{\text{cash}(\mathcal{A},\mathcal{T})}$), a valid transaction T_{puzzle} should contain (1) a hash of redeem script for $T_{\text{escr}(\mathcal{T},\mathcal{B})}$, and (2) input values that include the required signatures from $\mathcal{A}$ and $\mathcal{T}$. If a valid T_{puzzle} is posted to the blockchain, Alice's bitcoin escrowed in $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ is transferred to T_{puzzle}. However, this bitcoin remains locked up in T_{puzzle} until T_{puzzle} fulfilled by a

[14] In fact, this mauling could even be done by the Bitcoin miner that confirms $T_{\text{escr}(\mathcal{T},\mathcal{B})}$ on the blockchain!

transaction the meets the condition $(\mathcal{T} \wedge \mathcal{A} \forall j \in R : h_j = H(k_j)) \vee (\mathcal{A} \wedge tw1)$ as specified to the following redeem script:

```
OP_IF
OP_RIPEMD160, h1, OP_EQUALVERIFY
OP_RIPEMD160, h2, OP_EQUALVERIFY
...
OP_RIPEMD160, h15, OP_EQUALVERIFY
redeemer_pubkey
OP_CHECKSIG
OP_ELSE
refund_condition
OP_ENDIF
```

The `redeemer_pubkey` is the Tumbler $\mathcal{T}$ public key, and the refund condition has `payer_pubkey` as Alice's public key and `locktime` as tw_1. This redeem script checks that either (1) the fulfilling transaction has input values that contain the correct preimages ($h_1, \ldots, h_{15}$ from Figure 3) and is signed by $\mathcal{T}$'s public key, or (2) the fulfilling transaction is a refund transaction signed by Alice and posted to the blockchain after timewindow tw_1. This redeem script is hashed and its hash is stored in T_{puzzle}. To fulfil T_{puzzle}, the transaction T_{solve} contains (1) the redeem script whose hash is stored in T_{puzzle}, and (2) the following input values:

```
signature
k15
...
k1
OP_TRUE
```

where `signature` is a signature under the the Tumbler $\mathcal{T}$'s public key. The preimages $k_1, ..., k_{15}$ are such that $H(k_\ell) = h_\ell$ per Figure 3.

Per Section V-D, however, if all parties are cooperative, the Tumbler $\mathcal{T}$ just holds on to T_{puzzle} and never signs or posts T_{puzzle} to the blockchain. However, it is important to note that once Alice $\mathcal{A}$ provides T_{puzzle} to the Tumbler $\mathcal{T}$, the Tumbler can claim the bitcoin escrowed in $T_{\text{escr}(\mathcal{A},\mathcal{T})}$. To do this, $\mathcal{T}$ just signs and posts T_{puzzle} to the blockchain, and then forms, signs and posts T_{solve} to the blockchain. No involvement from Alice $\mathcal{A}$ is required to do this, and thus the Tumbler $\mathcal{T}$ can claim his bitcoin even if Alice stops communicating with $\mathcal{T}$. Notice, however, if $\mathcal{T}$ decides to unilaterally claims a bitcoin by posting T_{solve}, the Tumbler $\mathcal{T}$ necessarily reveals the puzzle solution (see Section V-B). Therefore, Alice gets what she paid for even if she stops cooperating with the Tumbler $\mathcal{T}$. As a final note, Alice cannot use transaction malleability to steal her bitcoin from the Tumbler; when Alice $\mathcal{A}$ gives T_{puzzle} to the Tumbler $\mathcal{T}$, then T_{puzzle} points to the $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ transaction which is already confirmed by the blockchain and thus cannot be mauled.

Finally, recall from Section V-D that if $\mathcal{T}$ becomes uncooperative, $\mathcal{T}$ could sign and post T_{puzzle} to the blockchain, and then refuse to sign and post T_{solve}. In this case, Alice never obtains her puzzle solution, and must reclaim her bitcoin which is locked in T_{puzzle} by posting a refund transaction $T_{\text{refund}(\mathcal{A},\mathcal{T})}$ that points at T_{puzzle}. (See Figure 9.) Specifically, $T_{\text{refund}(\mathcal{A},\mathcal{T})}$ points at T_{puzzle} and (1) contains the redeem redeem script whose hash is stored in T_{puzzle} and (2) and the following input values values, where `signature` is a signature that verifies under Alice's public key:

```
Signature
OP_FALSE
```

Once again, Alice can post $T_{\text{refund}(\mathcal{A},\mathcal{T})}$ without any help from the Tumbler. Once again, this matters because the refund transactions must be posted when $\mathcal{T}$ becomes uncooperative, and must still be valid even in the face of transaction malleability (*i.e.,* if $\mathcal{T}$ mauls the TXID for the transaction fulfilled by $T_{\text{refund}(\mathcal{A},\mathcal{T})}$.)

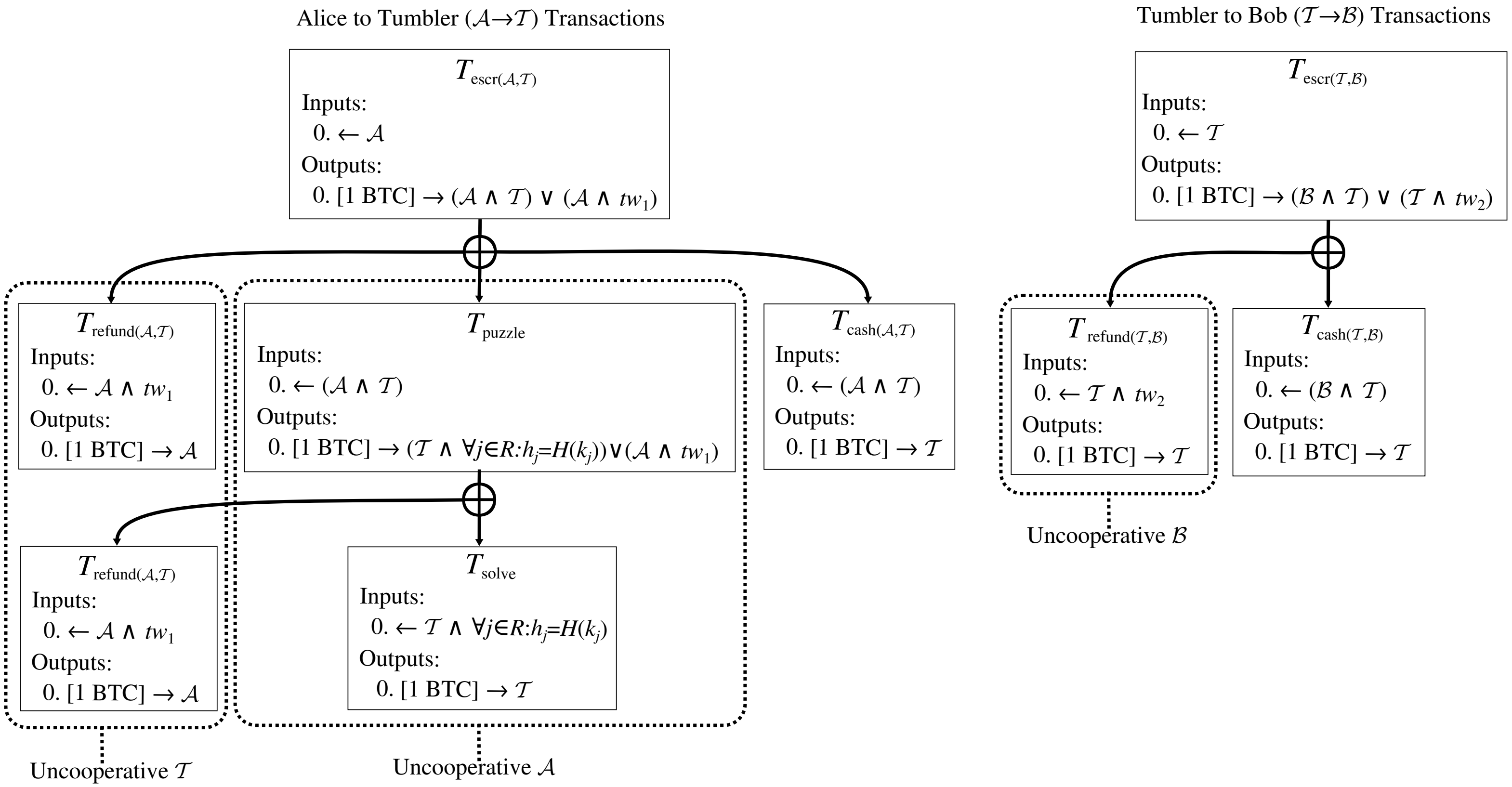
Alice to Tumbler ($\mathcal{A}\rightarrow\mathcal{T}$) Transactions
$T_{\text{escr}(\mathcal{A},\mathcal{T})}$
Inputs:
0. $\leftarrow \mathcal{A}$
Outputs:
0. [1 BTC] $\rightarrow (\mathcal{A} \wedge \mathcal{T}) \vee (\mathcal{A} \wedge tw_1)$
$T_{\text{refund}(\mathcal{A},\mathcal{T})}$
Inputs:
0. $\leftarrow \mathcal{A} \wedge tw_1$
Outputs:
0. [1 BTC] $\rightarrow \mathcal{A}$
T_{puzzle}
Inputs:
0. $\leftarrow (\mathcal{A} \wedge \mathcal{T})$
Outputs:
0. [1 BTC] $\rightarrow (\mathcal{T} \wedge \forall j \in R{:}h_j{=}H(k_j)) \vee (\mathcal{A} \wedge tw_1)$
$T_{\text{cash}(\mathcal{A},\mathcal{T})}$
Inputs:
0. $\leftarrow (\mathcal{A} \wedge \mathcal{T})$
Outputs:
0. [1 BTC] $\rightarrow \mathcal{T}$
$T_{\text{refund}(\mathcal{A},\mathcal{T})}$
Inputs:
0. $\leftarrow \mathcal{A} \wedge tw_1$
Outputs:
0. [1 BTC] $\rightarrow \mathcal{A}$
T_{solve}
Inputs:
0. $\leftarrow \mathcal{T} \wedge \forall j \in R{:}h_j{=}H(k_j)$
Outputs:
0. [1 BTC] $\rightarrow \mathcal{T}$
Uncooperative $\mathcal{T}$
Uncooperative $\mathcal{A}$
Tumbler to Bob ($\mathcal{T}\rightarrow\mathcal{B}$) Transactions
$T_{\text{escr}(\mathcal{T},\mathcal{B})}$
Inputs:
0. $\leftarrow \mathcal{T}$
Outputs:
0. [1 BTC] $\rightarrow (\mathcal{B} \wedge \mathcal{T}) \vee (\mathcal{T} \wedge tw_2)$
$T_{\text{refund}(\mathcal{T},\mathcal{B})}$
Inputs:
0. $\leftarrow \mathcal{T} \wedge tw_2$
Outputs:
0. [1 BTC] $\rightarrow \mathcal{T}$
$T_{\text{cash}(\mathcal{T},\mathcal{B})}$
Inputs:
0. $\leftarrow (\mathcal{B} \wedge \mathcal{T})$
Outputs:
0. [1 BTC] $\rightarrow \mathcal{T}$
Uncooperative $\mathcal{B}$

Fig. 9. Transaction relationships when $Q = 1$. Arrows indicate spending. Transactions in dotted line boxes denote transactions that are only published if a party is uncooperative.