TumbleBit: An Untrusted Tumbler for Bitcoin-Compatible Anonymous Payments

Ethan Heilman, Foteini Baldimtsi, Leen Alshenibr, Alessandra Scafuro, Sharon Goldberg
Boston University

ABSTRACT

This paper presents *TumbleBit*, a new anonymous payments scheme that is fully compatible with today's Bitcoin protocol. TumbleBit allows parties to make payments through an untrusted Tumbler. No-one, not even the Tumbler, can tell which payer paid which payee during a TumbleBit epoch. TumbleBit consists of two interleaved fair-exchange protocols that prevent theft of bitcoins by cheating users or a malicious Tumbler. Our protocol combines fast cryptographic computations (performed off the blockchain) with standard bitcoin scripting functionalities (on the blockchain). We prove the security of TumbleBit using the ideal/real world paradigm and the random oracle model. Security follows from the standard RSA assumption. We have implemented our protocol and used it to mix payments from several participants on the blockchain. Because our off-blockchain computations run in less than a second, TumbleBit's performance is limited only by the time it takes to confirm three blocks on the blockchain.

1. INTRODUCTION

One original reason for Bitcoin's popularity was a perception of anonymity. Indeed, Satoshi originally claimed that bitcoins could be spent *"without information linking the transaction to anyone"* [31]. However, it was subsequently shown that the movement of bitcoins could be traced through the blockchain [28, 35]. It has since become clear that anonymity is not guaranteed for the average Bitcoin user. Thus, two research directions have since emerged. The first proposes new anonymous and decentralized cryptocurrencies (*e.g.,* Zerocash [30, 9]). The second develops anonymity mechanisms that are compatible with Bitcoin, so-called *mixing* or *tumbler* services [6, 15, 36, 41, 26, 9, 37, 42, 12, 19]. While the first direction is very promising, its not clear if it will be widely adopted by users. Thus, we need robust anonymity solutions for Bitcoin's existing user base.

In this paper, we design, prove secure, and implement TumbleBit, a new anonymity system that is compatible with today's Bitcoin protocol. While several works have already been proposed to provide anonymity on top of Bitcoin, TumbleBit presents a new point in the design space that outperforms these works in several respects. We compare TumbleBit to prior work in Section 1.1.

TumbleBit uses a untrusted intermediary, the *Tumbler* $\mathcal{T}$, to mix together payments from $\aleph$ distinct *payers* (Alice $\mathcal{A}$) to $\aleph$ distinct payees (Bob $\mathcal{B}$). Each *epoch* of the protocol requires three blocks to be confirmed on the blockchain. TumbleBit provides *set-anonymity within an epoch*—no one other than $\mathcal{A}$ or $\mathcal{B}$, including the Tumbler, can determine which payer $\mathcal{A}$ paid which payee $\mathcal{B}$ during a given epoch. Like earlier schemes, we use fees to resist Sybil and Dos attacks [12], and ephemeral keys to recover from a malicious Tumbler that tries to link Alice to Bob by aborting their payment [19].

TumbleBit prevents a malicious Tumbler $\mathcal{T}$ from stealing users' bitcoins. To do this, TumbleBit consists of two interleaved *fair exchange protocols*; the first allows Alice $\mathcal{A}$ to swap a bitcoin for an anonymous voucher from $\mathcal{T}$, and the second allows Bob $\mathcal{B}$ to provide $\mathcal{T}$ with an anonymous voucher in exchange for a bitcoin. The fair exchange property prevents $\mathcal{T}$ from stealing bitcoins by refusing to issue/redeem a voucher. We formally prove that our protocols satisfy this property.[1] The security of our scheme relies on the standard RSA assumption and an equivocal encryption scheme in the random oracle model (ROM).

Most of our effort is spent on making TumbleBit compatible with the today's Bitcoin protocol. Specifically, we implement both of TumbleBit's fair-exchange protocols using smart contracts [38] via Bitcoin's current scripting functionality. Bitcoin scripts offer a very limited set of instructions—they allow us to implement contracts that exchange a bitcoin for (1) the preimage of a hash, or (2) an ECDSA signature on a Bitcoin transaction. Nevertheless, we combine these limited scripting

[1]True fair exchange is impossible in the standard model [32] 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 [22, 23] and treat the blockchain as a trusted public ledger. Other works use the term Contingent Payment or Atomic Swaps [25, 4].

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

functionalities (on the blockchain) with a cryptographic protocol (performed off the blockchain) to obtain our desired anonymity system.

Our cryptographic protocols are specially tailored to be fast; they run in seconds because they rely on simple RSA blind signatures, symmetric encryption, and cut-and-choose techniques. At the core of TumbleBit is an "RSA evaluation as a service" protocol that may be of independent interest. This protocol allows Alice $\mathcal{A}$ to pay one bitcoin to $\mathcal{T}$ in fair exchange for an RSA evaluation of an input x (chosen by $\mathcal{A}$) under $\mathcal{T}$'s secret key. While our idea is similar in spirit to "zero knowledge contingent payments" [25] or the concurrent work in [5], we do not need expensive zero-knowledge proofs or ZK-Snarks [10]; see Section 4.1.

We have implemented our TumbleBit system in 7032 lines of code in C++ and python, using OpenSSL as our cryptographic library. Our protocol requires only 430K bytes of data on the wire and a combined 0.8 seconds of compute time on a single CPU. Thus, the performance of our protocol is currently limited only by the time it takes for three blocks to be added to Bitcoin's blockchain; currently, this takes $\approx$ 30 minutes. We have used our TumbleBit system to mix payments from five payers to five payees; the relevant transactions are visible on the blockchain (Appendix A).

1.1 Related Work

A cryptocurrency tumbler or mixing service provides a way to mix potentially identifiable or "tainted" coins with others, such that it becomes hard to trace the trail to the coin's original source. We summarize the properties of prior works in this area in Figure 1.

CoinShuffle [36] builds on CoinJoin [24] to provide a decentralized tumbler that prevents bitcoin theft. The anonymity properties of CoinShuffle are rigourously analyzed in [29]. However, CoinShuffle's anonymity set is thought to be small as communication overhead increases quadratically [12, 14]. Because CoinShuffle and CoinJoin perform their mix in a single transaction, they are particularly vulnerable to DoS attacks (where a user joins the mix and then aborts, disrupting the protocol for all other users) and Sybil attacks (where an adversary deanonymizes a user by forcing it to mix with Sybil identities secretly under her control) [14, 40].

XIM [12] builds on fair-exchange mixers like [6]. XIM prevents theft, and uses fees to resist DoS and Sybil attacks—users must pay to participate in a mix, raising the bar for attacker that seeks to disrupt the protocol by joining the mix and then aborting. Moreover, an abort by a single user does not disrupt the mix for others. One of XIM's key innovations is a secure method for finding parties to participate in a mix. However, this also adds several hours to the protocol, because users must advertise themselves as mix partners on the blockchain.

Blindcoin [41] 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 [26] 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 our scheme, however, CoinSwap does not provide anonymity against even a honest but curious intermediary. Coinparty [42] offers another decentralized solution, but it is secure only if 2/3 of the mixing parties are honest.

Our work builds on the recent work of [19]. As described in Section 2, TumbleBit shares the same anonymity and fair exchange properties as [19]. Both [19] and TumbleBit uses an untrusted intermediary that cannot harm anonymity or steal bitcoins. Both TumbleBit and [19] use anonymous fee vouchers to resist DoS and Sybil attacks (Section 5.4), building on ideas from XIM [12]. However, while [19] requires scripting functionality that is not currently supported by bitcoin, TumbleBit is fully compatible with bitcoin. Moreover, while [19] does not provide an implementation, TumbleBit has been implemented (Section 6) and has been used to mix payments on the blockchain (Appendix A).

The work related to our 'RSA evaluation as a service' protocol is reviewed in Section 4.1.

2. BACKGROUND AND OVERVIEW

Our goal is to allow a *payer*, Alice $\mathcal{A}$, to anonymously send 1 bitcoin to a *payee* Bob $\mathcal{B}$. Alice has her (potentially-long-term) Bitcoin address $Addr_A$, and Bob chooses a fresh ephemeral address $Addr_B$. Achieving this goal implies that we also have a Tumbler, since the payer and payee also could be the same person.

Naturally, if Alice signed a regular Bitcoin transaction indicating that $Addr_A$ pays one 1 bitcoin to $Addr_B$, then the blockchain would record a link between $\mathcal{A}$ and $\mathcal{B}$ and anonymity could be harmed using the techniques of [28, 35, 11]. An approach to break this linkability is to introduce a *trusted* third party that receives bitcoins from payers, shuffles them and then presents them to the payees. However, the the trusted third party knows who is paying who, and can potentially also steal their bitcoins. Thus, a natural question is

> How can we build a system that allows $\mathcal{A}$ to anonymously pay $\mathcal{B}$, without requiring a *trusted* third party?

We start by reviewing a trivial solution that uses electronic cash (eCash) and a third party called the Tumbler $\mathcal{T}$. We then review why this trivial solution does not prevent bitcoin theft by an *untrusted* Tumbler, and explain how to deal with this using "smart contracts" [38].

2.1 Trivial solution with eCash.

In [19], it was observed that it might be possible to solve this problem using a trivial online eCash scheme based on [16]. Multiple payer-payee pairs pay through the Tumbler $\mathcal{T}$. The eCash scheme ensures that if enough payments are made simultaneously through $\mathcal{T}$, then noone, not even $\mathcal{T}$, can link a payer A to its payee $\mathcal{B}$. This *unlinkability* is achieved via *blind signatures.*

Blind signatures. In a blind signature scheme, the signer signs a "blinded" message that it cannot read. Blind signatures offer the usual signature *unforgeability* property, and also *blindness*— the signer cannot link a

Scheme	Prevents Coin Theft	Anonymous	Resists DoS	Resists Sybils	Min Mixing Time	Compatible with Bitcoin
Coinjoin [24]	✓	small set	×	×	1 block	✓
Coinshuffle [36]	✓	small set	×	×	1 block	✓
Coinparty [42]	2/3 users honest	✓	some[1]	✓ (fees)	2 blocks	✓
XIM [12]	✓	✓	✓	✓ (fees)	hours	✓
Mixcoin [15]	TTP accountable	× (TTP)	✓	✓ (fees)	2 blocks	✓
Blindcoin [41]	TTP accountable	✓	✓	✓ (fees)	2 blocks	✓
CoinSwap [26]	✓	× (TTP)[2]	✓	✓ (fees)	2 blocks	✓
BSC [19]	✓	✓	✓	✓ (fees)	3 blocks	×
TumbleBit	✓	✓	✓	✓ (fees)	3 blocks[*]	✓

Table 1: 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. [2]CoinSwap is not anonymous when the TTP is malicious. [*]TumbleBit can be modified to run in 2 blocks, see Section 6.4.

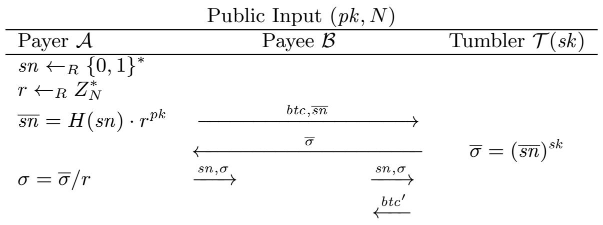
Public Input (pk, N)
Payer $\mathcal{A}$
Payee $\mathcal{B}$
Tumbler $\mathcal{T}(sk)$
$sn \leftarrow_R \{0,1\}^*$
$r \leftarrow_R Z_N^*$
$\overline{sn} = H(sn) \cdot r^{pk}$
$btc, \overline{sn}$
$\overline{\sigma}$
$\overline{\sigma} = (\overline{sn})^{sk}$
$\sigma = \overline{\sigma}/r$
sn, σ
sn, σ
btc'

Figure 1: Trivial protocol for anonymous payments, allows tumbler $\mathcal{T}$ to steal bitcoins. All computations are done modulo N.

particular message/signature pair to a particular execution of the protocol. A simple blind signature can be instantiated with RSA (as suggested by Chaum [16]). The signer has a standard RSA secret key sk and public key (pk, N) where N is an RSA modulus, and (sk, pk) are RSA exponents. Let G be an RSA full-domain hash function [8]. Suppose Alice $\mathcal{A}$ wants the signer to sign a message sn. Alice $\mathcal{A}$ produces a *blinded message* $\overline{sn}$ by choosing a random "blind" $r \leftarrow Z_N^*$ and letting

$$\overline{sn} = r^{pk} G(sn) \mod N.$$

The signer produces the blind signature as

$$\overline{\sigma} = \overline{sn}^{sk} \mod N$$

and finally Alice unblinds the signature to obtain

$$\sigma = \overline{\sigma}/r = (G(sn))^{sk} \mod N.$$

Notice that the final σ is just a regular full-domain-hash RSA signature on message sn and can be verified as such. Blind signatures can also be constructed from other cryptographic assumptions, *e.g.,* [33, 13, 3].

Tumbler based on eCash In Figure 1 we show how blind signatures are applied to this setting. Payer $\mathcal{A}$ sends $\mathcal{T}$ a bitcoin btc and a blinded serial number $\overline{sn}$. In return, $\mathcal{A}$ obtains a blind signature $\overline{\sigma}$. Payer $\mathcal{A}$ unblinds these values to create an anonymous voucher (sn, σ) which she uses to pay $\mathcal{B}$. Finally, $\mathcal{B}$ obtains a bitcoin by redeeming (sn, σ) with the tumbler $\mathcal{T}$. *Blindness* provides anonymity, by ensuring that $\mathcal{T}$ cannot link a blind signature $\overline{\sigma}$ that $\mathcal{T}$ issued with an unblinded signature σ that $\mathcal{B}$ redeemed. *Unforgeability* prevents a cheating user from issuing a voucher to herself.

2.2 The need for fair exchange

However, this trivial scheme is not robust to a malicious tumbler. That is, a malicious tumbler could steal bitcoins by (1) refusing to issue $\mathcal{A}$ a blind signature $\overline{\sigma}$ in exchange for her bitcoin, or (2) refusing to redeem $\mathcal{B}$'s anonymous voucher (sn, σ) in exchange for a bitcoin. [19] solves this problem using Bitcoin *scripts* that enforce a *fair exchange* mechanism: in a first fair exchange $\mathcal{A}$ swaps his bitcoin for a blind signature $\overline{\sigma}$ that is provided by $\mathcal{T}$ and in the second fair exchange $\mathcal{B}$ swaps an anonymous voucher (sn, σ) for a bitcoin provided by $\mathcal{T}$.

[19] builds these fair exchanges using scripting functionalities that are just not available in Bitcoin today. The authors of [19] argue that these functionalities could be added to Bitcoin via a *soft fork*. (A fork occurs when the Bitcoin protocol alters its interpretation of what blocks can be considered valid.) However, a soft fork requires consensus from a majority of Bitcoin miners, and thus can be very challenging to achieve in practice. As such, in this work we build two fair exchanges that are analogous to those in [19] and compatible with today's Bitcoin scripts.

2.3 Bitcoin scripts and smart contacts.

Recall that bitcoins are 'stored' in transactions, and transferred by sending the bitcoins in one transaction to a new transaction. The blockchain exists to provide a public record of all such valid transfers.

Each transaction determines the conditions under which the bitcoins held in that transaction can be moved to another transaction. These rules are specified by a simple non-Turing-complete scripting language called *Script*. We use Script to build "smart contracts" [38] that can be constructed from the following two transactions:

- The T_{offer} transaction, where one party offers to pay his bitcoin to any party that can sign a transaction that meets some condition $\mathcal{C}$.
- The $T_{fulfill}$ transaction, which meets the condition $\mathcal{C}$ stipulated in T_{offer}.

T_{offer} is posted to the blockchain first, and once $T_{fulfill}$ is confirmed by the blockchain, the bitcoins in $T_{fulfill}$ flow from the party signing transaction T_{offer} to the party signing $T_{fulfill}$. Script also supports *time-locking* for these contracts (CHECKLOCKTIMEVERIFY feature [39]), where T_{offer} additionally stipulates that a valid $T_{fulfill}$ must be confirmed by the blockchain within a specific time window tw. If $T_{fulfill}$ is not confirmed in time, then the bitcoins in T_{offer} revert back to the party signing T_{offer} and are no longer on offer to other parties. Two conditions (and some variations on them) are currently supported by Bitcoin:

Hashing condition (OP_RIPEMD160). The condition $\mathcal{C}$ stipulated in T_{offer} is: "$T_{fulfill}$ must contain the preimage of value y computed under the hash function H." (In this case H would be the hash function RIPEMD-160.) Then, $T_{fulfill}$ fulfills this condition by including a value x such that $H(x) = y$.

Signing condition (OP_CHECKSIGVERIFY). 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 by the secret key corresponding to public key PK. This particular condition is the most restrictive. First, today's bitcoin protocol requires the signature to be ECDSA over the Secp256k1 elliptic curve [34]. No other elliptic curves or types of signatures are supported. Second, the condition specifically requires $T_{fulfill}$ *itself* to be signed. Thus, one could not use this functionality to build a contract whose condition requires an *arbitrary message* m to be signed by PK.

These conditions can be composed under "AND" and "OR" operators. TumbleBit only requires contracts based on ANDs and ORs of the two conditions described above.

2.4 Blindly-signed contracts

We now describe the protocol in [19], and explain why the contracts it requires are not compatible with the scripting functionalities we just described.

2.4.1 Bitcoin for blind signature.

In the following fair exchange, $\mathcal{A}$ offers a bitcoin to $\mathcal{T}$ in exchange for $\mathcal{T}$ providing a blind signature σ on serial number sn, where sn is randomly chosen by $\mathcal{A}$.

Protocol overview. $\mathcal{A}$ chooses random sn and then blinds it to $\overline{sn}$. Then, $\mathcal{A}$ signs and posts a transaction T_{offer} that offers one bitcoin under condition "$T_{fulfill}$ must contain a blind signature on blinded message $\overline{sn}$ that validates under the tumbler's $\mathcal{T}$ public key PK." T_{offer} is also timelocked to time window tw (Section 2.3). Once T_{offer} is confirmed by the blockchain, $\mathcal{T}$ obtains the offered bitcoin by posting a transaction $T_{fulfill}$ containing $\overline{\sigma}$ a valid blind signature on $\overline{sn}$. Importantly, the validity of $\overline{\sigma}$ in $T_{fulfill}$ is programmatically enforced by the bitcoin protocol—namely, $T_{fulfill}$ will be confirmed only if miners can use PK to validate that $\overline{\sigma}$ is indeed a valid blind signature on $\overline{sn}$.

Why is this a fair exchange? This follows because the bitcoin offered by $\mathcal{A}$ only flows to the tumbler $\mathcal{T}$ once $\mathcal{T}$ produces a $T_{fulfill}$ that contains a valid blind signature $\overline{\sigma}$. If $\mathcal{T}$ fails to do this in time, the timelock ensures that $\mathcal{A}$ loses nothing because the bitcoin offered in T_{offer} reverts back to $\mathcal{A}$.

Lack of bitcoin support. Notice, however, that the condition in T_{offer} is not currently supported by Bitcoin. Firstly, $\mathcal{T}$ is required to sign $\overline{sn}$, a *blinded message* with has a specific structure induced by the blind signature scheme; today, Bitcoin Script only supports the verification of ECDSA signatures over Secp256k1. Given that T_{offer} needs to verify a blind signature, and we do not have secure constructions of blind signature from ECDSA, checking the condition in T_{offer} with Script is impossible without the introduction of a new opcode.

In Section 4.2, we realize this protocol in bitcoin-compatible manner using bitcoin's *hashing condition*.

2.4.2 Wrapper protocol from [19].

[19] uses a second fair exchange, where $\mathcal{B}$ offers a valid anonymous voucher (sn, σ) to $\mathcal{T}$ in exchange for a bitcoin. To prevent $\mathcal{T}$ from stealing bitcoins, this fair exchange (1) forces $\mathcal{T}$ to commit to redeeming the anonymous voucher (sn, σ) *even before he issues it*, and is thus (2) wrapped around the bitcoin-for-blind-signature fair exchange between $\mathcal{A}$ and $\mathcal{T}$ that we just described. This way, $\mathcal{T}$ cannot honestly complete the bitcoin-for-blind-signature fair exchange with $\mathcal{A}$ and then refuse to redeem $\mathcal{B}$'s corresponding anonymous voucher.

Protocol description. The protocol is as follows.

1. The payer $\mathcal{A}$ chooses her serial number sn, and hashes it to $hsn = H(sn)$ where H is the Bitcoin hash function. $\mathcal{A}$ sends hsn to $\mathcal{B}$.

2. $\mathcal{B}$ asks the tumbler $\mathcal{T}$ to post a transaction T_{offer} offering one bitcoin under condition: "$T_{fulfill}$ must be signed by the public key of $\mathcal{B}$ and contain (1) the hash preimage of hsn under H and (2) a valid signature on that preimage that verifies under the public key of $\mathcal{T}$." T_{offer} is timelocked to time window $tw' > tw$.

3. After the blockchain confirms T_{offer}, the payer $\mathcal{A}$ and tumbler $\mathcal{T}$ engage in the bitcoin-for-blind-signature fair exchange described above. At the end of this fair exchange, $\mathcal{A}$ learns a blind signature $\overline{\sigma}$ on her blinded serial number $\overline{sn}$.

4. $\mathcal{A}$ unblinds these values to obtain the anonymous voucher (sn, σ) which she provides to $\mathcal{B}$.

5. Finally, $\mathcal{B}$ completes the fair exchange by posting a transaction $T_{fulfill}$ which contains (sn, σ). Notice $T_{fulfill}$ fulfills the condition in $\mathcal{T}$ because (1) sn is a valid preimage for hsn (*i.e.*, $hsn = H(sn)$) and (2) σ is valid signature on sn. Once $T_{fulfill}$ is confirmed by the blockchain, the bitcoin in T_{offer} flows from $\mathcal{T}$ to $\mathcal{B}$. To speed up the protocol, this transaction can be confirmed in the same block as the $T_{fulfill}$ from the bitcoin-for-blind-signature fair exchange.

The whole protocol requires four transactions that can be confirmed in three blocks of the bitcoin blockchain. [19] calls this three-block window an "epoch".

Why is this a fair exchange? This follows because by signing the value hsn in T_{offer}, the tumbler $\mathcal{T}$ commits to redeeming the voucher (sn, σ). Meanwhile, $\mathcal{B}$ cannot steal this bitcoin from $\mathcal{T}$ because $\mathcal{B}$ cannot forge the signature σ on the serial number sn. Instead, $\mathcal{B}$ obtains a bitcoin from $\mathcal{T}$ iff $\mathcal{B}$ posts a $T_{fulfill}$ containing valid voucher (sn, σ) where $hsn = H(sn)$; if $\mathcal{B}$ fails to do this, the bitcoin offered in T_{fulfill} reverts back to $\mathcal{T}$.

Lack of bitcoin support. However, this fair-exchange mechanism also cannot be constructed from the scripting functionalities currently supported by bitcoin. The problem arises in the second condition of T_{offer}, which requires "a valid signature on that preimage that verifies under the public key of $\mathcal{T}$". First, we are requiring a signature on an arbitrary message (namely on the randomly-chosen serial number *sn* that is the hash preimage of *hsn*) rather than on a message structured as a Bitcoin transaction. Second, Bitcoin scripts only support ECDSA signatures over Secp256k1, and we lack instantiations of ECDSA blind signatures.

In Section 5, we solve this problem by using bitcoin-compatible protocol that fairly exchanges a bitcoin for a new type of anonymous voucher.

3. ANONYMITY AND SECURITY

All payers $\mathcal{A}$ and payees $\mathcal{B}$ run our protocol in lockstep during a three-block epoch. All users know the start of an epoch (because *e.g.,* it has a starting block of height divisible by three). We also require payer $\mathcal{A}$ and payee $\mathcal{B}$ to trust each other; if they didn't, a malicious $\mathcal{A}$ could always publicly proclaim that she paid $\mathcal{B}$.

Set-anonymity within an epoch. Assume payers only make one anonymous payment per epoch, and payees only accept one payment per epoch. Then, if $\aleph$ payments completed during the epoch, the probability of linking any chosen payer $\mathcal{A}$ to a payee $\mathcal{B}$ should not be more than $\frac{1}{\aleph}$ plus some negligible function [19, 12]. This holds for any adversary that inspects the blockchain, and even for a potentially malicious $\mathcal{T}$.

Remark: Intersection attacks. While this notion of anonymity is commonly used in Bitcoin tumblers (*e.g.,* [12, 19]), 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 [18]). These 'intersection attacks' follow because set-anonymity is composed across epochs; see also [12, 29] for discussion.

Recovery from anonymity failures. We also require TumbleBit users to be able to recover from anonymity failures that result from (1) payments that *do not* complete during an epoch, or (2) epochs where the anonymity set is too small (*e.g.,* $\aleph = 1$).

DoS Resistance. The Tumbler $\mathcal{T}$ should resist Denial of Service (DoS) attacks where a malicious user starts and aborts many parallel runs of the protocol.

Sybil Resistance. The protocol should resist Sybils (*i.e.,* identities that are under the control of single user) that attempt to deanonymize a target user.

We explain why TumbleBit satisfies these properties in Section 5.4.

Fair exchange. We formally specify the fair exchange properties of our schemes in Sections 4.2.1 and 5.4. At a high level, we require that the Tumbler $\mathcal{T}$ cannot steal bitcoins from payer $\mathcal{A}$ or payee $\mathcal{B}$. $\mathcal{A}$ and $\mathcal{B}$ should not be able to steal either: $\mathcal{B}$ should not be able to claim a bitcoin from $\mathcal{T}$ unless his corresponding payer really sent $\mathcal{B}$ a payment through $\mathcal{T}$. $\mathcal{A}$ should not use a single bitcoin to pay multiple payers $\mathcal{B}$. Finally, $\mathcal{A}$ should not be able to convince $\mathcal{B}$ that she paid him without actually spending a bitcoin.

4. BITCOIN FOR RSA EXPONENTIATION FAIR EXCHANGE

We now show how to realize a *Bitcoin-compatible* fair exchange where $\mathcal{A}$ pays one bitcoin to $\mathcal{T}$ in exchange for $\mathcal{T}$ computing an RSA exponentiation to its RSA secret key *sk*. This protocol is a crucial building block for our TumbleBit scheme, but is also of independent interest. Essentially, such a protocol allows a user to "purchase" two specific operations from $\mathcal{T}$: (a) an RSA signature and, (b) an RSA decryption. Both these operations could be done "blind" (*i.e.,* blind signature and blind decryption) if the user first blinds the message to be signed/decrypted[2].

Specifically, our protocol fairly exchanges one bitcoin from Alice $\mathcal{A}$ for the computation by T of

$$f^{-1}_{\mathit{RSA}}(y, \mathit{sk}, N) = y^{\mathit{sk}} \bmod N$$

where the input y is chosen by Alice $\mathcal{A}$, *sk* is $\mathcal{T}$'s secret RSA key and N is the RSA modulus. The RSA verification function is

$$f_{\mathit{RSA}}(x, \mathit{pk}, N) = x^{\mathit{pk}} \bmod N$$

and where *pk* is $\mathcal{T}$'s public RSA key.

4.1 Approaches from the literature

Generic solutions exist for this problem.

Contingent payments. Greg Maxwell described a protocol for "zero-knowledge contingent payments" (ZKCP) [25]. The scheme in [25] swaps one bitcoin from Alice $\mathcal{A}$ in exchange for having T compute *any* publicly-verifiable 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 k of $f(y)$ to $\mathcal{A}$ before being paid with $\mathcal{A}$'s bitcoin.) After $\mathcal{A}$ verifies the proof, $\mathcal{A}$ posts a transaction T_{offer} offering one bitcoin under condition: "T_{fulfill} must contain the hash preimage of h". $\mathcal{T}$ claims the bitcoin by posting a transaction T_{fulfill} 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_{fulfill} in a timely manner.

[2]We use RSA because it lets us blind a single message multiple times (*i.e.,* Blind m to $\bar{m}$. Then blind $\bar{m}$ to $\bar{\bar{m}}$. *etc.*) Our technique could be used by other operators that have this property.

ZKCP via ZK-Snarks. Recently, [27] showed how to instantiate the ZP proofs used in this protocol with ZK-Snarks [10]. The function f was a 16x16 Sudoku puzzle and the resulting protocol was run and completed within about 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 [27]'s ZK-snark but replacing the verification of the Sudoku puzzle with an RSA verification f_{RSA}. A disadvantage of this approach is that RSA verification within a ZK-Snark is likely to be slower than Sudoku puzzle verification; one reason for this is that 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 long numbers, each such number has to be split up and expressed as an array of smaller ones, making arithmetic operations much more complicated [17]. In any case, our protocol for RSA exponentiation is faster (running in < 1 second) than [27]'s protocol for 16x16 Sudoku puzzles (Section 6).

ZKCP via Garbled Circuits. As an alternative to ZK-Snarks, one could use more generic ZK proof based on zero-knowledge garbled circuits (GC) as shown in [20]. While GC-based ZK proofs work reasonably well for evaluating hash functions, they are computationally heavier for modular exponentiations (like the one involved in the RSA verification) because the latter do not have a "good" (*i.e.,* short) Boolean circuit representation [21].

Incentivizing correct computation. A different approach that also uses GCs was proposed in [22]. Two parties use GCs to compute an arbitrary function $g(a, b)$ without revealing their respective inputs a and b, and with the added feature that if one party aborts the protocol before the output is revealed, then 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 sk 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, then the bitcoin offered by $\mathcal{A}$ reverts back to $\mathcal{A}$. Again, however, the efficiency of this approach is limited the computational overhead of performing modular exponentiations inside a garbled circuit.

Our protocol sidesteps these issues because it avoids ZK proofs or GCs; instead, we use simple RSA operations along with a custom cut-and-choose technique.

4.2 f_{RSA} evaluation as a service

The goal of this scheme is to allow $\mathcal{A}$ to pay one bitcoin to $\mathcal{T}$ in (fair) exchange for a computation of be f_{RSA}^{-1}, which is essentially an RSA signature/decryption from $\mathcal{T}$ on $\mathcal{A}$ input y. The output of the protocol is $y^{sk} \mod N$. The core idea similar to that of contingent payments: $\mathcal{T}$ evaluates$\mathcal{A}$'s input y by computing y^{sk}, encrypts the result under a randomly chosen key k to obtain a ciphertext c, hashes the key k under bitcoin's hash as $h = H(k)$ and then provides (c, h) to Alice. $\mathcal{A}$ posts a T_{offer} offering one bitcoin in exchange for the preimage of h, and $\mathcal{T}$ earns the bitcoin by publishing k in $T_{fulfill}$. As before, we need to find a way for $\mathcal{A}$ to validate that the preimage of h will really decrypt c to the her desired value. This time, we do it using a cut-and-choose technique and the blinding properties of RSA. The full protocol is shown in Figure 3.

$\mathcal{F}_{\mathsf{fair\text{-}RSA}}$ is parameterized by a function (f_{RSA}, f_{RSA}^{-1}) and expiration time $tw \in \mathbb{N}$.
Parties: $\mathcal{A}$, $\mathcal{T}$, and adversary S.
Setup. Pick (pk, sk) for (f_{RSA}, f_{RSA}^{-1}). Send (`Setup`, pk) to $\mathcal{A}$ and S, and (`Setup`, pk, sk) to $\mathcal{T}$.
Evaluation. On input $(\texttt{request}, \texttt{sid}, y, 1BTC)$ from $\mathcal{A}$: If y is in the range of f_{RSA}, send (`request`, `sid`, $\mathcal{A}$, y) to $\mathcal{T}$. Start counter $tw_{\texttt{sid}} = 0$.
On input (`evaluate`,`sid`, $\mathcal{A}$) from $\mathcal{T}$: Send $(\texttt{sid}, f_{RSA}^{-1}(sk, y))$ to $\mathcal{A}$. Send $(\texttt{payment}, \texttt{sid}, 1BTC)$ to $\mathcal{T}$. If $tw_{\texttt{sid}} = tw$, send (`refund no-service`, `sid`,1BTC) to $\mathcal{A}$.

Figure 2: Ideal Functionality $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$.

4.2.1 Security Definition

Informally, we want to capture the following two fairness requirements:

- (**Fairness for Tumbler** $\mathcal{T}$) After one execution of the protocol $\mathcal{A}$ will learn $f_{RSA}^{-1}(y, sk)$ on exactly one input y of her choice.
- (**Fairness for Alice** $\mathcal{A}$) $\mathcal{T}$ will earn 1 bitcoin iff $\mathcal{A}$ obtains a correct value $f_{RSA}^{-1}(y, sk)$.

We model the above two requirements with an ideal functionality, that we call $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$, shown in Fig. 2. $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$ acts like a trusted party between $\mathcal{A}$ and $\mathcal{T}$. $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$ receives a preimage request of the form (y, 1 bitcoin) from $\mathcal{A}$, and it forwards the request to $\mathcal{T}$. If $\mathcal{T}$ agrees to reveal the preimage of y to $\mathcal{A}$, then $\mathcal{T}$ receives 1 bitcoin while $\mathcal{A}$ receives $x = f_{RSA}^{-1}(y, sk)$. Otherwise, if $\mathcal{T}$ refuses, $\mathcal{A}$ will get 1 bitcoin back, and $\mathcal{T}$ receives nothing. Fairness is modeled by the fact that $\mathcal{A}$ can request a preimage only if she sends 1 bitcoin to $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$; similarly, $\mathcal{T}$ receives 1 bitcoin only if he agrees to reveal $f_{RSA}^{-1}(y, sk)$ to $\mathcal{A}$. Therefore, $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$ captures the fairness requirement that we need in our scheme.

Remark 2. Note that, when f_{RSA}, f_{RSA}^{-1} is the RSA trapdoor function, $\mathcal{A}$ can *always* learn pre-images of random values (i.e., values that are not of her choice) without interacting with $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$. That is, $\mathcal{A}$ can compute $y = x^{pk} \mod N$, in which case, she trivially"knows" that $y^{sk} \mod N = x$.

Remark. Note that functionality $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$ does *not provide any privacy* for $\mathcal{A}$. Indeed, $\mathcal{T}$ learns $\mathcal{A}$'s input y (even if she refuses to provide her with the service). To use $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$ in our TumblerBit scheme, users will have to first blind their inputs to $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$.

We follow the standard ideal/real world paradigm. To prove that a protocol securely realizes $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$, one must show that the view obtained by a real world adversary Adv, corrupting either $\mathcal{T}$ or $\mathcal{A}$ and playing the real protocol where $\mathcal{T}$ participates with the secret sk, can be simulated by the ideal-world adversary (which is a PPT simulator S) that interacts only with $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$ (and thus learns no information or bitcoins apart from what is allowed by the fair functionality).

Definition 1 (Secure realization of $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$). *A protocol π securely realizes $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$ if, for every PPT adversary* Adv *corrupting either $\mathcal{A}$ or $\mathcal{T}$, there exists a PPT Simulator S such that the view* view_{real} *of* Adv *participating in the real protocol π is computationally indistinguishable from the view* view_{ideal} *generated by S. Where* view_{real} *contains the randomness and the input of* Adv *together with the transcript of the protocol.*

Our proof is in the *Random Oracle (RO) model* [7]; hash functions are modeled as perfectly random functions, and in the security proof the simulator can *program* their answers.

4.2.2 Protocol Description

We overview the protocol in Figure 3. Instead of asking $\mathcal{T}$ to provide just *one* (c, h) pair, $\mathcal{T}$ will be asked to provide $n + m$ such pairs. Then, we use the cut-and-choose idea: $\mathcal{A}$ will ask $\mathcal{T}$ to "open" n of these pairs, by revealing the randomly-chosen keys k_i's used to create each of the n pairs. In order for a malicious $\mathcal{T}$ to successfully cheat $\mathcal{A}$, it would have to correctly identify all the n "challenge" pairs and form them properly (so it does not get caught cheating), while at the same time malforming *all* the m unopened pairs (so it can claim a bitcoin from $\mathcal{A}$ without actually providing a blind signature in return). However, because $\mathcal{T}$ cannot predict which pairs $\mathcal{A}$ asks it to open, it will succeed at its goal 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? Indeed, if $\mathcal{A}$ received the opening of a correctly formed (c, h) pair, she would be able to recover her desired blind signature without paying her bitcoin (note that serial numbers are random values, so any valid signature on any *sn* should have the value of 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 decrypt to "fake values" rather than blind signatures. Before $\mathcal{T}$ agrees to open them, $\mathcal{A}$ must prove that these n values are indeed fake. Specifically, the "fake" inputs to the protocol are

$$\delta_i = (\rho_i)^{pk} \mod N$$

for a randomly chosen $\rho_i \leftarrow_R Z_N^*$. $\mathcal{A}$ proves the fakeness of an input by providing its ρ_i to $\mathcal{T}$. The idea is that when $\mathcal{T}$ evaluates f_{RSA}^{-1} on a fake input δ^i, the result is

$$(\delta_i)^{sk} = ((\rho_i)^{pk})^{sk} = \rho_i \mod N$$

which is utterly useless to $\mathcal{A}$, since $\mathcal{A}$ already knows ρ_i.

For this reason, $\mathcal{T}$ will agree to open the (c, h) pairs corresponding to fake values, as long as $\mathcal{A}$ is able to first prove they are fake. Once Alice confirms the correctness of the opened "fake" (c, h) values, she posts a T_{offer} offering one bitcoin for the keys k that open all of the m "real" (c, h) values that remain unopened. (Specifically, T_{offer} offers one bitcoin under the condition: "$T_{fulfill}$ contains the hash preimages of $h_1, ..., h_m$".)

Public input: pk, N

Alice $\mathcal{A}$		Tumbler $\mathcal{T}$
Input: y		**Secret** input: sk
<u>1. Prepare Real Set R</u> For $i \in [n]$, pick $r_i \in \mathbf{Z}_n^*$ $d_i \leftarrow y \cdot (r_i)^{pk} \mod N$		
<u>2. Prepare Fake set F</u> For $i \in [m]$, pick $\rho_i \in \mathbf{Z}_n^*$ $\delta_i \leftarrow (\rho_i)^{pk} \mod N$		
<u>3. Mix Sets.</u> Randomly permute $\{d_1 \dots d_n, \delta_1 \dots \delta_m\}$ to obtain $\{\beta_1 \dots \beta_{n+m}\}$ Let R be the indices of the d_i Let F be the indices of the δ_i	$\xrightarrow{\beta_1 \dots \beta_{n+m}}$	**<u>Evaluation</u>** For $i = 1 \dots n+m$ Evaluate β_i: $s_i = \beta_i^{sk} \mod N$ 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: $h_i = H(k_i)$
	$\xleftarrow{c_1, \dots, c_{n+m}}$	
	$\xleftarrow{(h_1, \dots, h_{n+m})}$	
	$\xrightarrow{F, \rho_i \ \forall i \in F}$	**<u>Check validity of Fake Set F.</u>** For all $i \in F$: Verify that $\beta_i = (\rho_i)^{pk} \mod N$, *i.e.*, $\mathcal{A}$ knows all pre-images for values in Fake Set F. If test passes, reveal the keys for the $c_i \forall i \in [F]$. Else, abort.
<u>Check validity of $\mathcal{T}$'s answers</u> For all $i \in F$, Verify that $h_i = H(k_i)$ Decrypt $s_i = H^{\mathsf{prg}}(k_i) \oplus c_i$ Verify that *i.e.*, $(s_i)^{pk} = (\rho_i)^{pk} \mod N$	$\xleftarrow{r_i', k_i \ \forall i \in F}$	
<u>Post transaction T_{offer}</u> Offer 1 bitcoin under condition "the spending transaction provides preimages of h_j for each $j \in R$."	$\xrightarrow{y, (r_j)^{pk} S \forall j \in R}$	**<u>$\mathcal{T}$ checks on $\mathcal{A}$</u>** Verify $\beta_j = y \cdot (r_j)^{pk} \mod N \quad \forall j \in R$
		<u>Post transaction $T_{fulfill}$</u> Post k_j for each $j \in R$
<u>Obtain Output</u> For $j \in R$: Learn k_j from $T_{fulfill}$. Decrypt c_j to get $s_j = H^{\mathsf{prg}}(k_j) \oplus c_j$ If s_j is s.t. $(s_j)^{pk} = \beta_j \mod N$, Obtain $y^{sk} = \frac{s_j}{(r_j)} \mod N$.		

Figure 3: f_{RSA} Function Evaluation as a Service. H and H^{prg} are modeled as random oracles, and instantiated in our implementation as H with RIPEMD-160, and H^{prg} is ChaCha20 with a 128-bit key, so that $\lambda_1 = 128$.

However, we now have another problem. If each one of the "real" (c, h) values opened to a *different* value, then $\mathcal{A}$ will obtain m blind signatures instead of one. This will not be fair to $\mathcal{T}$. We solve this problem by adding an extra step: once $\mathcal{A}$ posts T_{offer}, she proves to $\mathcal{T}$ that all m "real" values open to *the same* input. Once $\mathcal{T}$ verifies this, she agrees to post $T_{fulfill}$ to contains all m of the k values that open the "real" (c, h) pairs. We do this as follows. The m real inputs that $\mathcal{A}$ sends to $\mathcal{T}$ at the start of the protocol are

$$d_j = y(r_j)^{pk} \mod N$$

where each d_j is the same message y RSA-blinded under a different blind $r_j \leftarrow_R Z_N^*$. Then, when $\mathcal{T}$ signs (computes the f_{RSA}^{-1} of the real input d_j, the result is

$$(d_j)^{sk} = (y(r_j)^{pk})^{sk} = (y)^{sk} r_j \mod N$$

each of which is an evaluation of the same value y.

Thus, just after $\mathcal{A}$'s transaction T_{offer} is confirmed by the blockchain, $\mathcal{A}$ proves to $\mathcal{T}$ that her real inputs d_j are correctly formed. To do this, $\mathcal{A}$ will reveal all blinds r_j

to $\mathcal{T}$. As this point, $\mathcal{T}$ has committed to all its values, and thus $\mathcal{T}$ might as well redeem its bitcoin from $\mathcal{A}$. Thus, $\mathcal{T}$ posts $T_{fulfill}$ containing the keys k needed to open the real (c, h) pairs. Alice can obtain the output of f^{-1} on her input y as long as at *least one* of these (c, h) is validly formed and thus can be opened by a k value in $T_{fulfill}$.

THEOREM 1. *If the RSA assumption holds, and if functions $H^{\mathsf{prg}} : \{0,1\}^{\lambda_1} \to \{0,1\}^{\lambda_2}$ and $H : \{0,1\}^{s_1} \to \{0,1\}^{s_3}$ are independent random oracles, protocol in Fig. 3 securely realizes $\mathcal{F}_{\mathsf{fair\text{-}RSA}}$ in the random oracle model.*

The proof is in the full version.

5. TUMBLEBIT PROTOCOL

We now show how to use f_{RSA} evaluation as a service (Section 4.2) to build our TumbleBit protocol.

5.1 A new type of "voucher".

Our approach is inspired by [19], which 'wraps' its bitcoin-to-blind-signature fair exchange in another protocol that realizes a tumbler. In [19], the tumbler $\mathcal{T}$ posts a T_{offer} committing to the redemption of a valid anonymous voucher in exchange for a bitcoin. Then, the payee $\mathcal{B}$ redeems the bitcoin by posting a $T_{fulfill}$ containing the valid voucher. The voucher is (sn, σ), where sn is a serial number randomly chosen by the payer $\mathcal{A}$, and σ is an signature on sn under the tumbler $\mathcal{T}$'s public key. Recall also the σ is the unblinding of a blind signature.

The problem with the voucher in [19] is that validating (sn, σ) requires a signature-verification functionality this is not supported by bitcoin scripts. We now deal with this by introducing a new 'voucher' that can be validated using only the bitcoin-supported "signature condition". (Recall from Section 2.3 that this condition is "$T_{fulfill}$ must be digitally signed by an ECDSA-Secp256k1 signature that verifies under public key PK.") Our new voucher is a *transaction* $T_{fulfill}$ signed by bitcoin's ECDSA-Secp256k1 signature. This signature will be computed under an *ephemeral* public key $(SK_{\mathcal{T}}^{eph}, PK_{\mathcal{T}}^{eph})$ freshly chosen by the tumbler $\mathcal{T}$ for this specific run of the protocol. We shall shortly see why this key needs to be ephemeral.

5.2 Overview of the TumbleBit protocol.

Like [19], our TumbleBit protocol uses *wrapper protocol* wrapped around our f_{RSA} evaluation as a service. The full wrapper protocol is in Figure 4. We overview the protocol here.

1. The protocol starts by having $\mathcal{T}$ post a timelocked T_{offer} offering one bitcoin in exchange for a valid "voucher" from $\mathcal{B}$. More precisely, T_{offer} offers one bitcoin under the condition "$T_{fulfill}$ must be digitally signed by an ECDSA-Secp256k1 signature that verifies under public key $PK_{\mathcal{T}}^{eph}$ and $PK_{\mathcal{B}}$." Notice that we are requiring *two* ECDSA-Secp256k1 signatures on $T_{fulfill}$; the signature under the ephemeral $PK_{\mathcal{T}}^{eph}$ which realizes a valid 'voucher", and a signature under the long term $PK_{\mathcal{B}}$ that ensures that $\mathcal{T}$ can will redeem the voucher if it is submitted by this specific payee $\mathcal{B}$.

2. $\mathcal{B}$ convinces $\mathcal{T}$ to sign the required voucher. Specifically, the bulk of of the wrapper protocol allows $\mathcal{B}$ to obtain
$$z = f_{RSA}(\epsilon, pk, N)$$
The value ϵ allows $\mathcal{B}$ to open an encryption c_ℓ to a valid 'voucher'. Specifically, c_ℓ encrypts an ECDSA-Secp256k1 signature σ_ℓ on a transaction $T_{fulfill}$.

3. The payee $\mathcal{B}$ asks the payer $\mathcal{A}$ to "invert" the value z. To do this, the payer $\mathcal{A}$ interacts with $\mathcal{T}$ to have $\mathcal{T}$ *blindly invert* z (using $\mathcal{T}$'s secret RSA sk), in exchange for $\mathcal{A}$'s bitcoin. Blindless here means that $\mathcal{T}$ (or anyone) cannot link $\mathcal{T}$'s current interaction with $\mathcal{A}$ with the original value z (and thus $\mathcal{T}$'s interaction with $\mathcal{B}$). This will realize our anonymity goal, because it means that $\mathcal{T}$ will not be able to determine which $\mathcal{A}$ was paying which $\mathcal{B}$.

   How will this blind inversion be performed? This is the point where we use our protocol for f_{RSA}^{-1} evaluation as a service. Thus, $\mathcal{A}$ will obtain the desired pre-image of z by engaging in the protocol of Figure 3 with $\mathcal{T}$. Specifically, $\mathcal{A}$ first blinds z to
$$y = z \cdot r^{pk} \mod N$$
   where $r \leftarrow_R Z_N^*$ where r is a randomly chosen blind value[3]. Next, $\mathcal{A}$ performs the f_{RSA}^{-1} evaluation with the tumbler $\mathcal{T}$ on input y. At end of this fair exchange, $\mathcal{A}$ swaps her bitcoin in exchange for value $(y)^{sk}$ where
$$(y)^{sk} = (z \cdot r^{pk})^{sk} = (z)^{sk} \cdot r = \epsilon \cdot r \mod N$$
   The result is actually the *inversion* of z, blinded with r. Thus, $\mathcal{A}$ can unblind the result as
$$(y)^{sk}/r = \epsilon \cdot r/r = \epsilon \mod N$$
   thus unveiling ϵ.

4. $\mathcal{A}$ provides ϵ to $\mathcal{B}$. $\mathcal{B}$ uses ϵ to open the commitment c_ℓ that contains the ECDSA-Secp256k1 signature σ_ℓ on a transaction $T_{fulfill}$. Finally, $\mathcal{B}$ ECDSA-Secp256k1 signs $T_{fulfill}$ under his own key $SK_{\mathcal{B}}$, and posts the result to the blockchain in order to claim his bitcoin from $\mathcal{T}$.

5.3 Details of the wrapper protocol.

We describe Step 2 of the wrapper in more detail. How does $\mathcal{B}$ obtain the required z and encryption c_ℓ from T? We use cut-and-choose with "real" and "fake" values as in Section 4.2, along with several new tricks.

$\mathcal{B}$ prepares μ distinct "real" transactions and η "fake" transactions, and then hides these values by hashing them under H'. (H' is the 'Hash256' hash function (*i.e.,* SHA256 cascaded with itself) that bitcoin uses as part of its "hash-and-sign" paradigm for ECDSA-Secp256k1 signatures.) $\mathcal{B}$ now creates a set of $\mu + \eta$ hash values β_ℓ for $\ell = 1...\mu + \eta$ by permuting the hashes of the "real" and "fake" transactions. These β_ℓ are sent to $\mathcal{T}$.

[3] $\mathcal{B}$ could also blind the value z before handing it to $\mathcal{A}$. In that case even if $\mathcal{A}$ doesn't care about her anonymity and neglects to blind $\mathcal{B}$ would still be protected.

Next, $\mathcal{T}$ signs each β_ℓ to obtain an ECDSA-Secp256k1 signature σ_ℓ. Each σ_ℓ is then hidden inside an encryption c_ℓ which can decrypted with key ϵ_ℓ. Finally, $\mathcal{T}$ hides each ϵ_ℓ (the encryption keys) by using RSA trapdoor function:

$$z_\ell = f_{RSA}(\epsilon_\ell, pk, N)$$

All κ pairs of (c_ℓ, z_ℓ) are sent back to $\mathcal{B}$. As each ϵ_ℓ is uniformly chosen at random, z_ℓ computationally hides ϵ_ℓ, under the RSA assumption [4].

Now we are back to the trick in Section 4.2. $\mathcal{B}$ needs to check that the η "fake" (c_ℓ, z_ℓ) pairs are correctly formed by $\mathcal{T}$. To do this, $\mathcal{B}$ asks $\mathcal{T}$ to provide the keys ϵ_ℓ to the fake pairs. As before, $\mathcal{T}$ revels these keys only after $\mathcal{B}$ has proved that the η pairs really are fake. Once this is done, $\mathcal{B}$ knows that $\mathcal{T}$ can cheat with probability no larger than $1/\binom{\mu+\eta}{\eta}$.

Now we need a new trick. We want to ensure that if *at least one* of the "real" (c_ℓ, z_ℓ) pairs opens to a valid ECDSA-Secp256k1 signatures σ_ℓ, then just one key ϵ can be used to open this pair. We need this because the payee $\mathcal{B}$ must decide which encryption of ϵ_ℓ to give to the payee $\mathcal{A}$ for decryption *before* $\mathcal{B}$ knows which of the (c_ℓ, z_ℓ) pairs is validly formed. We solve this problem by "chaining" together the ϵ_ℓ values that corresponds to "real" values. Specifically, $\mathcal{T}$ also provides $\mathcal{B}$ with the following $\mu - 1$ quotients

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

where $\{j_1, ..., j_\mu\} = R$ 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_ℓ}, since $\epsilon_{j_\ell} = \epsilon_1 \cdot q_2 \cdot ... \cdot q_\ell$.

On the flip side of this coin, what happens if $\mathcal{B}$ obtains *more than one* valid ECDSA-Secp256k1 signature by opening the (c_ℓ, z_ℓ) pairs? Fortunately, however, we don't need to worry about this. The T_{offer} transaction posted by $\mathcal{T}$ offers just one bitcoin in exchange for a ECDSA-Secp256k1 signature under an *ephemeral* key $PK^{eph}_{\mathcal{T}}$ that $\mathcal{T}$ uses *only once* during this protocol execution with this specific payee $\mathcal{B}$. Thus, even if $\mathcal{B}$ gets many such ECDSA-Secp256k1 signatures, only one of them can be used to redeem the bitcoin offered in the T_{offer} transaction, and the rest are useless.

5.4 Fair exchange and anonymity.

The wrapper protocol should satisfy the following:

Completeness for $\mathcal{B}$. At the end of the protocol, $\mathcal{B}$ obtains $z_i = f_{RSA}(\epsilon_i, pk, N)$ and encryptions c_i of valid ECDSA-Secp256k1 signatures that can be unlocked with knowledge of ϵ_i.

Completeness for $\mathcal{T}$. At the end of the protocol, $\mathcal{T}$ has issued locked signatures c_i that can be unlocked only if some payer will pay 1 bitcoin to learn $f^{-1}_{RSA}(z_{j_1}, sk, N)$.

Fairness for $\mathcal{B}$. For any arbitrarily malicious Tumbler, there is negligible probability that the protocol successfully ends but $\mathcal{B}$ is not able to unlock c_i.

[4]Each ϵ_ℓ is also used in the encryption c_ℓ, and hashed. As we model hash functions are random oracles, we are able to prove that ϵ_ℓ is still computationally hidden.

Fairness for $\mathcal{T}$. For any PPT malicious $\mathcal{B}$, there is negligible probability that $\mathcal{T}$ accepts a voucher from $\mathcal{B}$ but $\mathcal{T}$ never completed a 'f_{RSA} evaluation as a service' for evaluating z (with some payer $\mathcal{A}$ during the epoch).

The full version gives a proof sketch of why the protocol in Figure 4 satisfies the above fairness and completeness properties.

Set-anonymity within an epoch. We need to make sure that the values passed between payer $\mathcal{A}$ and payee $\mathcal{B}$ cannot be used to link $\mathcal{A}$ to $\mathcal{B}$. Notice that $\mathcal{B}$ provides the value z to payee $\mathcal{A}$, and $\mathcal{A}$ provides ϵ to back to $\mathcal{B}$. As long as the confidentiality of the contents and existence of the communication between $\mathcal{A}$ and $\mathcal{B}$ is preserved, neither $\mathcal{T}$ nor any third party can use use these values to link their payments. This follows because $\mathcal{A}$ information-theoretically blinds z to y before engaging in her fair exchange protocol with $\mathcal{T}$, and then unblinds the result before she passes ϵ to $\mathcal{B}$.

Recovery from anonymity failures. Like [19], the anonymity provided by TumbleBit crucially relies on the fact that $\mathcal{B}$ uses an *ephemeral* bitcoin address (*i.e.,* a new cryptographic public key that is chosen freshly) in each epoch. Let us now see how this is helpful.

Consider first a malicious $\mathcal{T}$ that, during one epoch has posted T_{offer} for its exchanges with all payers $\mathcal{A}$, and the T_{offer} for its exchange with all payees $\mathcal{A}$. Then, $\mathcal{T}$ decides *not* to provide the T_{fulfill} for the exchange with one payer $\mathcal{A}$. It follows that the corresponding payer $\mathcal{B}$ will not be able to provide a valid T_{fulfill} (since this would require forging a valid voucher). As such, the malicious $\mathcal{T}$ could match the aborted exchange with $\mathcal{A}$ with the aborted exchange with $\mathcal{B}$ to learn that $\mathcal{A}$ was trying to pay $\mathcal{B}$. To recover from this, we follow [19] and require $\mathcal{B}$ to discard his ephemeral address and never use it again if $\mathcal{T}$ aborts the protocol. Note that $\mathcal{B}$ loses nothing, since the abort ensures that no funds have been transferred to $\mathcal{B}$'s ephemeral address.

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.

DoS and Sybil protection. Since $\mathcal{T}$ can not trust $\mathcal{B}$ or $\mathcal{A}$, $\mathcal{T}$ should not be required to cover the cost of the transaction fee for first transaction contract T_{offer} that $\mathcal{T}$ posts to the blockchain (see Figure 4). To address this, we apply [12, 19]'s idea of anonymous fee vouchers. An anonymous voucher is a blind signature $\overline{\sigma}$ that $\mathcal{T}$ provides to A in exchange for a small 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 $\mathcal{T}$. All of this is done off the blockchain. The protocol begins once $\mathcal{T}$ is sure that it was paid for its efforts.

Payee Bob $\mathcal{B}(SK_{\mathcal{B}}, PK_{\mathcal{B}})$		Tumbler $\mathcal{T}(sk, (pk, N))$
		Choose ECDSA ephemeral key $(SK^{eph}_{\mathcal{T}}, PK^{eph}_{\mathcal{T}})$ Post transaction T_{offer} timelocked for tw_0 offering one bitcoin under the condition: "T_{fulfill} must be digitally signed by a signature that verifies under $PK^{eph}_{\mathcal{T}}$ and a signature that verifies under $PK_{\mathcal{B}}$."
Create and hash μ unique transactions fulfilling T_{offer}: $T_{\mathit{fulfill}}{}^1, \cdots, T_{\mathit{fulfill}}{}^\mu$ $ht_1 = H(T_{\mathit{fulfill}}{}^1), \cdots, ht_\mu = H(T_{\mathit{fulfill}}{}^\mu)$ Create η fake transaction hashes: Randomly choose $r_1, ..., r_\eta$ $ft_1 = H'(0...0 \parallel r_1), \cdots, ft_\eta = H'(0...0 \parallel r_\eta)$ Randomly permute $\{ht_1 \ldots ht_\mu, ft_1 \ldots ft_\eta\}$ to obtain $\{\beta_1 \ldots \beta_{\mu+\eta}\}$ Let R be the indices of the ht_i Let F be the indices of the ft_i	$\xrightarrow{\beta_1 \cdots \beta_{\mu+\eta}}$	For $\ell = 1 \ldots \mu + \eta$ ECDSA sign β_ℓ to get $\sigma_\ell = \mathrm{Sig}(SK^{eph}_{\mathcal{T}}, \beta_\ell))$ Randomly choose $\epsilon_\ell \in Z_N$. Create Encryption $c_\ell = H^{\mathsf{prg}}(\epsilon_\ell) \oplus \sigma_\ell$ Compute $z_\ell {=} f_{RSA}(\epsilon_\ell, pk, N)$ *i.e.,* $z_\ell = (\epsilon_\ell)^{pk} \mod N$
	$\xleftarrow{(c_1,z_1),\ldots(c_\mu,z_\eta)}$	
	$\xrightarrow{F,R}$	
	$\xrightarrow{r_i\ \forall i \in F}$	For all $i \in F$ Verify fakeness, *i.e.,* $\beta_i = H(0...0 \parallel r_i)$
	$\xleftarrow{\epsilon_i\ \forall i \in F}$	
For all $i \in F$, Validate RSA evaluation $z_i = (\epsilon_i)^{pk} \mod N$ Validate c_i: Decrypt $\sigma_i = H^{\mathsf{prg}}(\epsilon_i) \oplus c_i$ ECDSA-Ver$(PK^{eph}_{\mathcal{T}}, H(0...0 \parallel r_i), \sigma_i) = 1$		Prepare quotients: For $R = \{j_1, ..., j_\mu\}$, set $q_2 = \frac{\epsilon_{j_2}}{\epsilon_{j_1}}, ..., q_\mu = \frac{\epsilon_{j_\mu}}{\epsilon_{j_{\mu-1}}}$
	$\xleftarrow{q_2,\ldots,q_\mu}$	
RSA validate the quotients: For $R = \{j_1, ..., j_\mu\}$ check that $z_{j_2} = z_{j_1} \cdot (q_2)^{pk} \mod N$ ... $z_{j_\mu} = z_{j_{\mu-1}} \cdot (q_\mu)^{pk} \mod N$ Send $z = z_{j_1}$ to Payer $\mathcal{A}$		
$\mathcal{A}$ runs protocol in Figure 3 with $\mathcal{T}$ to blindly obtain $\epsilon = f^{-1}_{RSA}(z, sk, N){=}\ z^{sk} \mod N$. Payer $\mathcal{A}$ provides ϵ to $\mathcal{B}$		
Let $R = \{j_1, j_2, ..., j_\mu\}$ Recover the other epsilons: $\epsilon_{j_1} = \epsilon$ $\epsilon_{j_2} = \epsilon_{j_1} \cdot q_{j_2} \mod N$ ... $\epsilon_{j_\mu} = \epsilon_{j_{\mu-1}} \cdot q_\mu \mod N$ Recover the ECDSA signatures: For $\ell = 1, ..., \mu$: Decrypt $\sigma_\ell = H^{\mathsf{prg}}(\epsilon_{j_\ell}) \oplus c_{j_\ell}$ If $\exists$ an ECDSA signature σ_ℓ that is valid for some real transaction $T_{\mathit{fulfill}}{}^i$ under $PK^{eph}_{\mathcal{T}}$, sign $T_{\mathit{fulfill}}{}^i$ under $SK_{\mathcal{B}}$, and post it to blockchain along with σ_ℓ.		

Figure 4: Wrapper protocol. $(sk, (pk, N))$ are the RSA keys for the tumbler $\mathcal{T}$. $(SK^{eph}_{\mathcal{T}}, PK^{eph}_{\mathcal{T}})$ and $(SK_{\mathcal{B}}, PK_{\mathcal{B}})$ are ECDSA-Secp256k1 keys. (Sig, ECDSA-Ver) is an ECDSA-Secp256k1 signature scheme. We model H, H' and H^{prg} as random oracles. In our implementation, H is SHA256 and 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^{prg} first hashes its 2048-bit input to 256 bits using SHA-256, and then uses the result as a key for ChaCha20.

Per [12], fees vouchers raise the cost of an DoS attack where $\mathcal{B}$ starts and aborts many parallel sessions, locking $\mathcal{T}$'s bitcoin in the blockchain via T_{offer} transactions. This similarly provides Sybil resistance, making it expensive for an adversary to trick a target payer or payee into participating in an epoch where all other users are Sybils under the adversary's control.

6. IMPLEMENTATION

To show that TumbleBit is performant and fully compatible with Bitcoin, we implemented our protocols and tumbled 5 payments between 10 parties (5 payers and 5 payees, *i.e.,* $\aleph = 5$) resulting in 20 transactions posted to the blockchain. See Appendix A for the Bitcoin transaction IDs created during the tumble.

6.1 Instantiation.

We instantiated our protocols with 2048-bit RSA. The hash functions and signature schemes are instantiated as described in the captions to Figure 3 and Figure 4. The only exceptions are the following minor differences.

- For the protocol in Figure 4: (1) Instead of encrypting s_i by XORing with $H^{\mathsf{prg}}(k_i)$, we encrypted s_i with with AES-128 in CBC mode. (2) Instead of choosing "fake" values as $\delta_i = (\rho_i)^{pk}$, we choose a random 256-byte value γ_i, zero out the first 128-bits, and hash the result with full-domain hash G, obtaining a 2048-bit pseudorandom string fsn_i. We then RSA-blind fsn_i by choosing random $\varrho_i \in \mathbf{Z}_n^*$ and letting $\delta_i = fsn_i(\varrho_i)^{pk} \mod N$. The Tumbler $\mathcal{T}$ can later check the validity of the fake set by obtaining ϱ_i, γ_i from $\mathcal{A}$, and confirming that $\delta_i/(\varrho_i)^{pk} = G(0^*||\gamma_i)$.
- For the protocol in Figure 3: Instead of encrypting σ_ℓ by XORing with $H^{\mathsf{prg}}(\epsilon_\ell)$, we encrypted σ_ℓ by first hashing (the 2048-bit) ϵ_i with SHA-256, and then using the first 128 bits of the output to encrypt σ_ℓ with with AES-128 in CBC mode.

These issues have negligible impact on performance, and have been removed from TumbleBit's current implementation.

Scripts. By default, Bitcoin clients and miners only operate on transactions that fall into one of five standard Bitcoin transaction templates. We therefore conform to the Pay-To-Script-Hash(P2SH) [2] transaction template, which is a standard transaction type. The contracts specified in our protocol description are exactly equivalent, in security and functionality, when reformulated as P2SH transactions.

To format T_{offer} as a P2SH transaction we specify a *redeem script* whose conditions must be met to fulfil the transaction. This redeem script is hashed, and its hash is stored in T_{offer}.

To spend 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. For example, the redeem script for our 'f_{RSA} evaluation as a service' protocol checks that the input values in $T_{fulfill}$ contains the correct preimages and that $T_{fulfill}$ is signed by $\mathcal{A}$'s public key:

```
OP_RIPEMD160, h1, OP_EQUALVERIFY,
...
OP_RIPEMD160, h15, OP_EQUALVERIFY,
PubKey, OP_CHECKSIG
```

where `h1`,...,`h15` are $h_1, ..., h_{15}$ from Figure 3 and `PubKey` is the Tumbler $\mathcal{T}$'s permanent bitcoin address (*i.e.,* public key). The input values (that are stored in $T_{fulfill}$) and the redeem script is run against are

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

`signature` is a signature under the the Tumbler $\mathcal{T}$'s permanent bitcoin address. The preimages $k_1, ..., k_{15}$ are such that $H(k_\ell) = h_\ell$ per Figure 3. To programmatically validate that $T_{fulfill}$ can fulfils 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 $T_{fulfill}$. $T_{fulfill}$ fulfils T_{offer} if the redeem script outputs true.

6.2 Choosing number of 'real' and 'fake'.

In our 'f_{RSA}^{-1} evaluation as a service protocol' (Figure 3), the probability that $\mathcal{T}$ could cheat is parameterized by m (the number of "real" values) and n (the number of "fake" values). Recall from Section 4.2 that $\mathcal{T}$ can cheat with probability $1/\binom{m+n}{m}$.

From a security perspective, we want m and n to be as large as possible, but in practice we face constraints from the Bitcoin protocol. Our main constraint on m is that m RIPEMD-160 hash outputs must be stored in the redeem transaction for our 'f_{RSA} Function Evaluation as a Service' protocol. However, P2SH redeem scripts are limited in size to 520 bytes, which means $m \leq 21$. Furthermore, increasing m also increases the transaction fee, since fees paid (to miners) to confirm a transaction on the blockchain scale with the size of the transaction. Fortunately, however, the choice of n is not limited by the constraints of the bitcoin protocol, because "fake" values are dealt with completely off the blockchain. The only drawback to increasing n is that we then have to perform more RSA exponentiations (off the blockchain).

Thus, our strategy is to choose m and n so that the probability that the Tumbler $\mathcal{T}$ cheats without getting caught is bound by 2^{-80}; this probability is smaller than the collision resistance of RIPEMD-160. We also want to minimize m as much as possible while keeping n to a reasonable number. Under these constraints, we chose $m = 15$ and $n = 285$, 'f_{RSA}^{-1} evaluation as a service protocol' (Figure 3).

We used the same values of $\mu = 15$ and $\eta = 285$ in our implementation of the wrapper protocol (Figure 4).[5]

[5]Note, however, that the wrapper protocol has no Bitcoin-related constraints on the choice of μ and η, since μ, η play no role in the number of inputs that must be provided with the $T_{fulfill}$. Thus, we could have minimize the number of RSA computations that we perform (which is $\mu + \eta$) by taking $\mu = \eta = 42$. If we

6.3 Performance evaluation

All tests were performed on EC2-medium t2.medium instances (2 Cores, 4 GB of RAM). We used standard benchmarking software to find that they could perform a 2048-bit RSA signing and verification in $1469\mu s$ and $440\mu s$ respectively. Bitcoin transaction signatures (ECDSA-Secp256k1) were significantly faster, with signing and verification taking $66\mu s$ and $104\mu s$ respectively.

Latency. To test the impact that network latency would have on our 'f_{RSA} Function Evaluation as a Service' protocol, we ran $\mathcal{A}$ on a server in Oregon and $\mathcal{T}$ on a server in Tokyo. The average RTT between servers was 92.4 ms. We ran the off-blockchain portion of this protocol 2000 times. Its average running time was 1.29 seconds.

Compute time. We next reduced RTT to near zero, by allowing all parties to run on the same server. We again performed 2000 tests and found average running times of 0.465 and 0.361 seconds for the off-blockchain computations in 'f_{RSA} Function Evaluation as a Service' and our wrapper protocol, respectively. Therefore, the total compute time for the off-blockchain portion of TumbleBit is 0.826 seconds, on average.

Bandwidth. The combined bandwidth of the full TumbleBit protocol consumed by all parties was 430 KB, which is roughly 1/5th size of an average webpage (2212 Kb [1]).

Thus, we argue that performance of TumbleBit is bound only by the time it takes to confirm blocks on the blockchain; this follows because it typically takes ≈ 10 minutes to confirm a block, and the off-blockchain portions of TumbleBit run in ≈ 1 second.

6.4 Coordinating on epochs

Thus far we have assumed that transactions 'posted to the blockchain' are then confirmed in the next discovered block. This is not unreasonable, but Bitcoin provides no inherent guarantee that a transaction will be confirmed in a timely manner. A transaction might not be confirmed for a variety of reasons. For example, there may be insufficient space in the block, or the block might have been discovered by a miner that had not learned about the transaction. As the number of participants in a TumbleBit epoch grows, the risk that a transaction is confirmed "late" grows as well. If a transaction is confirmed too late (*i.e.,* in a different block that that of the other users), this will abort the anonymous payment for the relevant participant in the TumbleBit epoch.

If TumbleBit is to scale to thousands of participants, more work is required to solve this problem. We plan to address this issue in future versions of TumbleBit. One viable approach involves scaling the epoch window with the number of mix participants, to provide a margin of error for "late" transactions. Another idea is to borrow a technique from [26], to reduce the size an epoch to two blocks.

Two-block epoch. The idea is to have $\mathcal{A}$ and $\mathcal{T}$ set up a '2-of-2 escrow' transaction $T_{\mathcal{A},\mathcal{T}}$ that will act as 'insurance' for the $\mathcal{A}$-to-$\mathcal{T}$ portion of the TumbleBit protocol. Specifically, $T_{\mathcal{A},\mathcal{T}}$ timelocks a Bitcoin from $\mathcal{A}$ under the condition "the spending transaction must be signed by both $\mathcal{A}$ and $\mathcal{T}$". $T_{\mathcal{A},\mathcal{T}}$ may be confirmed in the same block as the T_{offer} transaction that initiates the wrapper protocol in Figure 4. The TumbleBit protocol then runs as in Figure 4, with the following modifications to its $\mathcal{A}$-to-$\mathcal{T}$ portion: (1) The T_{offer} in the $\mathcal{A}$-to-$\mathcal{T}$ protocol (Figure 3) now spends the bitcoin offered in $T_{\mathcal{A},\mathcal{T}}$, and is thus signed by both $\mathcal{A}$ and $\mathcal{T}$. This transaction T_{offer} is *not* posted to the blockchain. (2) $T_{fulfill}$ is as in Figure 3, but $T_{fulfill}$ is also *not* posted to the blockchain. (3) When the $\mathcal{A}$-to-$\mathcal{T}$ protocol completes, then $\mathcal{A}$ and $\mathcal{T}$ both sign a new transaction ${T_{fulfill}}_{\mathcal{A},\mathcal{T}}$ that transfers the bitcoin in $T_{\mathcal{A},\mathcal{T}}$ to the Tumbler $\mathcal{T}$. This final ${T_{fulfill}}_{\mathcal{A},\mathcal{T}}$ transaction is posted to the blockchain, and may be confirmed in the same block as the $T_{fulfill}$ from the wrapper protocol. Thus, the entire protocol completes in only two blocks.

Why does this work? In our original protocol, we required T_{offer} for the $\mathcal{B}$-to-$\mathcal{T}$ portion of the protocol to be confirmed *before* the T_{offer} for the $\mathcal{A}$-to-$\mathcal{T}$ part of the protocol. This protects Payer $\mathcal{A}$. Why? Suppose $\mathcal{A}$ posted the T_{offer} for the $\mathcal{A}$-to-$\mathcal{T}$ part of the protocol, but a cheating $\mathcal{T}$ decides not to post the T_{offer} for the $\mathcal{B}$-to-$\mathcal{T}$ part of protocol. In this case, the cheating $\mathcal{T}$ could claim a bitcoin from Payer $\mathcal{A}$ in exchange for an RSA evaluation, but this RSA evaluation is of no value for the Payee $\mathcal{B}$ (and therefore also of no value for $\mathcal{A}$) if $\mathcal{T}$ refuses to initiate the $\mathcal{B}$-to-$\mathcal{T}$ protocol. However, the two-epoch design prevents $\mathcal{T}$ from cheating this way: if $\mathcal{T}$ refuses to initiate the protocol with $\mathcal{B}$, then $\mathcal{A}$ can refuse to sign any transaction spending the funds in $T_{\mathcal{A},\mathcal{T}}$, and thus $\mathcal{A}$ will not lose her Bitcoins.

Acknowledgements

We thank Ethan Donowitz for assistance with the preliminary stages of this project, and Omer Paneth, Dimitris Papadopoulos, Leonid Reyzin, Ann Ming Samborski, and Sophia Yakobov for useful discussions.

7. REFERENCES

[1] Http archive: Trends, 2015.

[2] Bitcoin wiki: Pay to script hash, 2016.

[3] Masayuki Abe. A secure three-move blind signature scheme for polynomially many signatures. In *EUROCRYPT'01*, pages 136–151, 2001.

[4] Adam Back, G Maxwell, M Corallo, Mark Friedenbach, and L Dashjr. Enabling blockchain innovations with pegged sidechains. 2014.

[5] Wacław Banasik, Stefan Dziembowski, and Daniel Malinowski. Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. Cryptology ePrint Archive, Report 2016/451, 2016. http://eprint.iacr.org/.

[6] 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.

do this, the probability that $\mathcal{T}$ can cheat is still bound by the RIPEMD-160 collision probability of 2^{-80}, but we have to perform $3x$ fewer RSA computations. This change could significantly improve the performance of the wrapper protocol.

[7] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and Communications security*, pages 62–73. ACM, 1993.

[8] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures-how to sign with rsa and rabin. In *EUROCRYPT*, pages 399–416, 1996.

[9] 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.

[10] 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.

[11] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. Deanonymisation of clients in bitcoin p2p network. In *ACM-CCS*, pages 15–29, 2014.

[12] 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. ACM, 2014.

[13] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *PKC*, volume 2567, pages 31–46, 2003.

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

[17] 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.

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

[19] Ethan Heilman, Foteini Baldimtsi, and Sharon Goldberg. Blindly signed contracts: Anonymous on-blockchain and off-blockchain bitcoin transactions.

[20] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *ACM-CCS*, pages 955–966, 2013.

[21] Benjamin Kreuter, Abhi Shelat, Benjamin Mood, and Kevin RB Butler. Pcf: A portable circuit format for scalable two-party secure computation. In *Usenix Security*, volume 13, pages 321–336, 2013.

[22] Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In *ACM-CCS*, pages 30–41. ACM, 2014.

[23] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In *ACM-CCS*, pages 195–206. ACM, 2015.

[24] G Maxwell. Coinjoin: Bitcoin privacy for the real world, 2013.

[25] Greg Maxwell. Zero knowledge contingent payment.

[26] Gregory Maxwell. Coinswap: transaction graph disjoint trustless trading, 2013.

[27] Gregory Maxwell. The first successful zero-knowledge contingent payment. Bitcoin Core https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/, February 2016.

[28] 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*, pages 127–139, 2013.

[29] Sarah Meiklejohn and Claudio Orlandi. Privacy-enhancing overlays in bitcoin. In *Financial Cryptography and Data Security*, volume 8976, pages 127–141. 2015.

[30] 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.

[31] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. (2012):28, 2008.

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

[33] David Pointcheval and Jacques Stern. Provably secure blind signature schemes. In *ASIACRYPT*, pages 252–265, 1996.

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

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

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

[37] Amitabh Saxena, Janardan Misra, and Aritra Dhar. Increasing anonymity in bitcoin. In *Financial Cryptography and Data Security*, pages 122–139. Springer, 2014.

[38] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[39] Peter Todd. Bip 65: Op checklocktimeverify. *Bitcoin improvement proposal*, 2014.

[40] Florian Tschorsch and Björn Scheuermann. Bitcoin and beyond: A technical survey on decentralized digital currencies.

[41] Luke Valenta and Brendan Rowan. Blindcoin: Blinded, accountable mixes for bitcoin. In *Financial Cryptography and Data Security*, pages 112–126. Springer, 2015.

[42] Jan Henrik Ziegeldorf, Fred Grossmann, Martin Henze, Nicolas Inden, and Klaus Wehrle. Coinparty: Secure multi-party mixing of bitcoins. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 75–86. ACM, 2015.

APPENDIX

A. TUMBLEBIT TRANSACTIONS ON BITCOIN'S BLOCKCHAIN

We performed a tumble with five anonymous payments using our TumbleBit implementation ($\aleph = 5$). We provide the transaction IDs (txids) of the transactions created by TumbleBit so that they can be located on the Bitcoin blockchain.

$\mathcal{T}$ publishes five T_{offer} starting the wrapper protocol:

```
bd8cb3c30ca57eefacd0d3823bc3f9a17808a1c10e4dd37a9f9938d2e0133f16
f4e687ab434f9d64fc90613b5ed11febebf8e52db687c5ebf67d02a129772dd4
2499370a7fd576ae06bbafcac56cb9148ad8de22673b72c8a918d9b9c909d544
```

```
d78d4c576c4789b0cc6cfdc5b33b5f696491725bef98b078930b0bde1966d6b1
29e72d4ca23f167d979657a9e4813683258498fe1169d89694baaad965ea1b7c
```

Each $\mathcal{A}$ publishes a T_{offer} starting f_{RSA}:

```
1c8b70af66d73bc75ff1caefde520b4611ebe807eeb74d90901294dd09c6d72d
5172ec41921d0130711c6af22ab083f53965dad266e66a90616890c4387530b5
dc2d000ba61061bf7785b0aa463b51db2cd36d49bcd83adb8329980f77d27a06
781a2013a7ac28e0f7a94e3e3f2e3bef72aa1c33bc4992022da7e19fddc82908
6bd46f1ce8363d86a03376d7ebb84ce49c3d5b3881abbdcf7ad93a9373fb9bba
```

$\mathcal{T}$ publishes five $T_{fulfill}$ spending the T_{offer} each $\mathcal{A}$ published in the last step:

```
dbaa883d210de99b484f9051988dbf31292a69e0a4a5977a483b73089d6b2ed8
89b5f92241ee902b9452cd8f276a929b5ca8cea15cce1e10b2ee51d9f1f196d8
8f427ddf5e260ac2e93962d711b930dc50e631cd954edc525cefea6dff8ed228
ade89b3b90983b93975733aea0a572b5cbba0206c27358408a57346cfa631d77
77a15175f3b6c53c997ce87fe42105920bbccd1f54e35cd9b75f785fb091696a
```

Finally each $\mathcal{B}$ claim their bitcoins by spending the T_{offer} which $\mathcal{T}$ published in the first step.

```
c267220d13c590d8e860078fad1eef4f33981993416997409b9bb604b1870229
e1666ced63c70612f3b2af143278d97a90a669c1cc33594fbcabe415622a9d23
a2fb6f5f558218669e7754540e60db52e22d51a2773f924d76b0edb174e52703
e1e9e61bf27147f0df567c835b01f932ca9d5fb13f8f9f1ba1d6f3b0506bfc78
3019aa5cf743d86e335f993e290efa81c64a7af30e994c3e8bc35e0ddc5d2a01
```