2022

Mirror worlds, eclipse attacks and the security of Bitcoin and the RPKI

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

MIRROR WORLDS, ECLIPSE ATTACKS AND THE SECURITY OF BITCOIN AND THE RPKI

by

ETHAN HEILMAN

B.S., Bridgewater State University, 2007

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

2022

Approved by

First Reader

Leonid Reyzin, PhD
Professor of Computer Science

Second Reader

Sharon Goldberg
Associate Professor of Computer Science

Third Reader

Ran Canetti
Professor of Computer Science

Fourth Reader

Andrew Miller
Assistant Professor of Electrical and Computer Engineering
University of Illinois, Urbana-Champaign

104. *IF a merchant gives to an agent grain, wool, oil, or goods of any kind with which to trade, the agent shall write down the value and return the money to the merchant. The agent shall take a sealed [authenticated] receipt for the money which he gives to the merchant.*

105. *IF the agent is careless and does not take a receipt for the money which he has given to the merchant, the money not receipted for shall not be placed to his account.* The Law Code of Hammurabi

Acknowledgments

First I want to acknowledge and thank my advisors Sharon Goldberg and Leonid Reyzin. Sharon's drive and focus on pragmatic solutions to problems voiced by users put purpose in my heart and always made me feel my work was meaningful. She showed me the value of tunneling to the center of a problem despite the many false starts and turns. I am deeply grateful to have had such a wise researcher and professor as an advisor and mentor. Leonid Reyzin only recently started co-advising me. Even before he was my co-advisor I would often venture into his office seeking advice. He always had the time to give me sage advice and he never set me wrong.

When I was preparing to give my first big research talk both Sharon and Leo had me present what I had, gave me notes on what I could do better and then had me present again and again. It was a serious time commitment on their part but their continuous coaching had a dramatic impact on my public speaking ability. Whenever I present something well, I think back to the time and effort they put in to get me to that point.

I want to thank my parents, my grandparents, and my wife. My Dad, Ward Heilman, who first taught me about mono-alphabetic ciphers and how to break them. My interest in this field comes from the stories he raised me on of hacking, cryptology and mathematical lore. He was always ready to tutor me or help me when I struggled with school. My Mom, Kathleen Cullen-Kortleven, who never missed an opportunity to teach me or show me something. From taking me to see Shakespeare, to quizzing me on the times table up to twelve or her habit of covering the English parts of multi-lingual signs and insisting that I translate the Spanish or the French. She helped teach me the value of using every opportunity to learn. My wife Anna Krohn, who encouraged me to apply to grad school, supported me, and kept me functioning through the tough parts.

As the research in this dissertation is drawn from several research papers, many minds other than my own contributed to it. This dissertation includes contributions from Leen AlShenibr, Foteini Baldimtsi, Danny Cooper, Sharon Goldberg, Alison Kendler, Leonid Reyzin, Alessandra Scafuro, and Aviv Zohar. I want to thank all my other coauthors: Juozas Baltikauskas, Kyle Brogle, Nicolas Christin, Michael Colavita, Tadge Dryja, Lawrence Harman, Henry Heffan, Jason Hennessey, Kevin Lee, Sebastien Lipmann, James Lovejoy, Patrick McCorry, Andrew Miller, Malte Möser, Arvind Narayanan, Neha Narula, Uma Shama, Kyle Soska, Shashvat Srivastava, Garrett Tanzer, and Madars Virza.

I also want to thank Uma Shama and Larry Harman, Magaly Ponce, and Eric LePage at Bridgewater State University who gave me my first research projects, and Eran Tromer and Ron Rivest who functioned as unofficial advisors on my early cryptanalysis research before I started at Boston University.

Thanks for all the fun times and interesting conversations to my friends at Boston University, William Blair, Yilei Chen, Danny Cooper, Jason Hennessey, Kyle Hogan, William Koch, Aanchal Malhotra, Dimitris Papadopoulos, Oxana Poburinnaya, Sachin Vasant, Sarah Scheffler, Abhishek Sharma, Nikolaj Volgushev, AJ Trainor, Allan Wirth, and Sophia Yakoubov. I would be remiss if I did not thank my fellow residents of Fort Gore, my roommate Jen Rich who regularly answers my paper writing grammar questions and my two wonderful cats, Thalia and Clio, who have ensured that I am warm on many cold writing days.

I am grateful to members of this committee, Leonid Reyzin, Sharon Goldberg, Ran Canetti, and Andrew Miller being a part of this defense.

This dissertation was only possible by the efforts of many people, I don't have room to thank them all here. Know that I am grateful for everyone who helped me get to this point.

MIRROR WORLDS, ECLIPSE ATTACKS AND THE SECURITY OF BITCOIN AND THE RPKI

ETHAN HEILMAN

Boston University, Graduate School of Arts and Sciences, 2022

Major Professor: Leonid Reyzin, PhD
Professor of Computer Science

ABSTRACT

While distributed databases offer great promise their decentralized nature poses a number of security and privacy issues. In what ways can parties misbehave? If a database is truly distributed can a malicious actor hide their misdeeds by presenting conflicting views of the database? Can we overcome such deceit and either prevent it by eliminating trust assumptions or detect such perfidy and hold the malicious party to account? We study these questions across two distributed databases: RPKI (Resource Public Key Infrastructure), which is used to authenticate the allocation and announcement of IP prefixes; and Bitcoin, a cryptocurrency that utilizes a permissionless database called a blockchain to track the transfer and ownership of bitcoins.

The first part of this dissertation focuses on RPKI and the potential of RPKI authorities to misbehave. We consider the methods, motivations, and impact of this misbehavior and how an RPKI authority can present inconsistent views to hide this misbehavior. After studying the problem we propose solutions to detect and identify such misbehavior.

Now we turn our attention to Bitcoin. We look at ways an attacker can manipulate Bitcoin's Peer-to-Peer network to cause members of the network to have inconsistent

views of Bitcoin's blockchain and subvert Bitcoin's core security guarantees. We then propose countermeasures to harden Bitcoin against such attacks.

The final part of this dissertation discusses the problem of privacy in Bitcoin. Many of the protocols developed to address Bitcoin's privacy limitations introduce trusted parties. We instead design privacy enhancing protocols that use an untrusted intermediary to mix *aka,* anonymize, bitcoin transactions via blind signatures. To do this we must invent a novel blind signature fair-exchange protocol that runs on Bitcoin's blockchain.

This dissertation favors a dirty slate design process. We work to layer protections on existing protocols and when we must make changes to the underlying protocol we carefully weigh compatibility and deployment considerations. This philosophy has resulted in some of the research described in this dissertation influencing the design of deployed protocols. In the case of Bitcoin our research is currently used to harden a network controlling approximately a trillion dollars.

Contents

List of Tables

List of Figures

List of Abbreviations

ARIN	……	American Registry for Internet Numbers
BGP	……	Border Gateway Protocol
BGPsec	……	Border Gateway Protocol Security (extension)
BTC	……	Bitcoin
IP	……	Internet Protocol
P2P	……	Peer-to-Peer (Network)
PoW	……	Proof-of-Work
RC	……	Resource Certificate
RIR	……	Regional Internet Registries
ROA	……	Route Origination Authorization
RP	……	Relying Party
RPKI	……	Resource Public Key Infrastructure
USD	……	United States Dollar

Chapter 1

Introduction

Blockchains and other types of global scale distributed database protocols represent a valuable tool to computer system designers. However, their use can introduce new security and privacy challenges. How do you detect or prevent the misbehavior of parties and intermediaries that the protocol depends on? What is the security impact if a malicious party gains the ability to present inconsistent views of the database to different users? If any user can download and view the entirety of the distributed database how is privacy maintained?

In this dissertation I will explore these security and privacy issues within the context of two distributed database systems: the RPKI (Resource Public Key Infrastructure) (Lepinski and Kent, 2012) and Bitcoin (Nakamoto, 2008). The RPKI (covered in Chapter 2) is a distributed database for authenticating the right to allocate and originate IP (Internet Protocol) prefixes in internet routing. Bitcoin (covered in Chapters 3,4,5), is a virtual currency *aka,* a cryptocurrency, which tracks ownership of virtual coins using a type of distributed database known as a blockchain.

The overall plan of this dissertation is as follows. In Chapter 2 we start with RPKI and look at ways in which RPKI authorities can misbehave and deceptively manipulate the RPKI to present inconsistent views of the RPKI to hide this misbehavior. We follow this with a series of proposed improvements to the RPKI to detect and identify such misbehavior. The dissertation then turns its attention to Bitcoin. Chapter 3 continues the theme of misbehavior and deceit by exploiting and presenting inconsistent views of Bitcoin's distributed database. The final two chapters deal with the problem of designing privacy enhancing Bitcoin protocols which do not require a trusted party. Chapter 4 introduces *Blindly Signed Contracts*, a protocol to anonymize Bitcoin transactions by employing an intermediary called a tumbler. In Chapter 5 we develop *TumbleBit* which improves on the Blindly Signed Contracts protocol from the previous chapter.

1.1 RPKI and Bitcoin

Let us now compare the two protocols that this dissertation concerns itself with: RPKI and Bitcoin. These two protocols share many similarities since they both function to answer the question who controls what. The RPKI's main feature is the ability to provide an authentication mechanism to determine what parties have legitimate control over which IP (Internet Protocol) prefixes. To do this the RPKI uses cryptographically-signed statements to allocate and sub-allocate control of those IP prefixes. Comparatively, Bitcoin creates consensus on what parties own which bitcoins and uses cryptographically-signed statements to assign and re-assign the ownership of those bitcoins.

Beyond these similarities the RPKI and Bitcoin have significant differences. The RPKI propagates changes to its database using a tree of repositories maintained by the RPKI authorities. Bitcoin instead uses a P2P (Peer-to-Peer) network that anyone can join. The RPKI does not attempt to solve "the double-spending problem" (Osipkov et al., 2007), whereas solving this problem is Bitcoin's core innovation. The RPKI is not mainly concerned with privacy, Bitcoin is.

The RPKI propagates changes using a tree of repositories maintained by the RPKI authorities. Reads and writes are made in an asynchronous fashion. When a relying party wishes to download the RPKI, they start at the root of tree, recursively following links in each repository to get child repositories until they have downloaded the entire the RPKI. In Bitcoin any party can request the blockchain *i.e.,* Bitcoin's distributed database, from any other full member of the Peer-to-Peer network. Transactions, or updates to the database, are submitted and then propagated across Bitcoin's P2P network. These changes are added to Bitcoin's blockchain in batches via a process called mining.

The chief aim of Bitcoin is to model physical coinage by solving the "double-spending problem" *i.e.,* if Alice gives 0.1 bitcoins to Bob and that transaction is approved by Bitcoin's blockchain, then Alice should not also be able to give those same bitcoins to Carol. On the other hand, the RPKI does not aim for control of IP prefixes to be exclusive. An RPKI authority can allocate an IP prefix to multiple parties. To put it another way one party's right to control an IP Prefix does not require that no one else also have a right to control that prefix. In cryptocurrency parlance the RPKI does not seek to solve the "double-spending problem" (Osipkov et al., 2007).

Finally while the RPKI is not concerned with privacy, Bitcoin is. The RPKI is intended to be public record of IP prefix allocation and origination rights. Thus, anything in the RPKI is a matter of public record. Unlike the RPKI in Bitcoin privacy is an explicit goal of the protocol. In fact the first description of the Bitcoin protocol, the so-called Bitcoin whitepaper (Nakamoto, 2008), lists anonymity as one of the features of Bitcoin. Unfortunately achieving anonymity in Bitcoin is still an area of ongoing research.

1.2 Dissertation Overview

We will now look at each chapter of this dissertation in detail. In next section we will discuss the overarching themes that run across these chapters and the longer term impact of the research presented in this dissertation. Finally we provide a short summary of research contributions broken down by chapter.

1.2.1 Chatper 2: Improving the Transparency of the RPKI

The RPKI, which is the focus of Chapter 2, is a new infrastructure for securing interdomain routing over BGP (Border Gateway Protocol). BGP has traditionally operated as a "default-accept" architecture. That is, any autonomous system (AS) on the internet can claim to be the destination for any IP prefix, and by default other ASes will accept such a BGP announcement. This "default-accept" assumption at the heart of BGP has made BGP vulnerable to prefix- and subprefix hijacks (Rensys Blog, 2008; Misel, 1997; de Beaupre, 2013; Cowie, 2010; Pilosov and Kapela, 2009; Peterson, 2013) in which an AS announces IP prefixes for which they are not the legitimate destination and "hijacks" that IP prefix.

The RPKI seeks to prevent these attacks by providing a trusted mapping from allocated IP prefixes to ASes authorized to originate them in BGP. To do this, the RPKI establishes a top-down hierarchy of *authorities*, rooted at the Regional Internet Registries (RIRs), that allocate and suballocate IP address space, as well as authorize its use in BGP by individual ASes. Routers can then use the RPKI to distinguish between hijacked BGP routes and routes originated by a legitimate AS by authenticating these announcements with the RPKI.

However, the security benefits of the RPKI are accompanied by a drastic shift from BGP's traditional "default accept" policies, to a new "default deny" mode: to prevent (sub)prefix hijacks, routers should only accept routes authorized by the RPKI, and discard all other routes by default. This issue is further exacerbated by the fact that the RPKI's hierarchical architecture empowers centralized authorities to *unilaterally* revoke authorization for IP prefixes under their control.

This shift has lead to concerns (Amante, 2012; Mueller and Kuerbis., 2011; Communications Security, Reliability and Interoperability Council III (CSRIC), 2011; The President's National Security Telecommunications Advisory Committee, 2011; Mueller et al., 2013) that the RPKI creates powerful authorities with the technical means for taking down IP prefixes and could be exploited by abusive authorities or governments to settle disputes or block undesirable content. This is a stark departure from the status quo, where these authorities (RIRs, National/Local Internet Registries, *etc.*) had the power to allocate IP address space, but not to impact routing to IP prefixes that had already been allocated (Goldman, 2006; Project, 2011).

In Chapter 2 we examine various specific techniques in which the RPKI authorities may abuse and exploit their privileged position in the allocation hierarchy to misbehave. We note how the ability to perform targeted revocations and takedowns makes such misbehavior more likely. We introduce "mirror-world" attacks in which a malicious party presents inconsistent views of the RPKI in a targeted fashion to both hide misbehavior and cause other problems.

These "mirror world" attacks result from the fact that the RPKI gives authorities unilateral control over all records below them in the hierarchy. Thus misbehaving authorities have enormous latitude to create many sets of differing records. Compounding this issue is that the current RPKI does not guarantee consistency and does not maintain any sort of verifiable history of the past state of the RPKI. Even if two relying parties compare their respective views of the RPKI and notice that an authority has presented two different states, they may not be able to identify which authority misbehaved or detect that an authority misbehaved at all.

To remedy this, we propose modest modifications to the architecture of the RPKI to hold RPKI authorities accountable. In our design once an allocation is authorized by the RPKI, it switches from "default-deny" to "default-accept"; that is, authorities that revoke IP address space must first obtain cryptographic consent from all entities holding allocations and suballocations of that space. This enables the identification of authorities that fail to properly obtain consent and returns some power back to IP space holders.

Our proposed modification also enables parties viewing the RPKI from different vantage points to detect when the views they are presented with are not consistent. This is important because clever RPKI authorities may use “mirror world” attacks to evade identification and accountability. In our design authorities must commit to the past states of the RPKI by building hash-chains of previous states. This allows relying parties to detect inconsistencies in the local and global state of the RPKI and raise alarms if an RPKI authority is presenting different views to different parties.

1.2.2 Chapter 3: Eclipse Attacks on Bitcoin’s P2P Network

In our next chapter (Chapter 3) we switch our focus to the cryptocurrency Bitcoin. While cryptocurrencies have been studied since the 1980s (Chaum, 1983a; Brands, 1993; Camenisch et al., 2005), Bitcoin was the first to see widespread adoption. As of July 2021, Bitcoin is the most valuable cryptocurrency with a market cap of over 700 billion USD with a daily trading volume of 51 billion USD (Ledesma, 2021; CoinmarketCap, 2021) and is the national currency of El Salvador (Aleman, 2021).

A key reason for bitcoin’s success is its baked-in decentralization and open P2P (peer-to-peer) network. Instead of using a central bank as regulatory body, Bitcoin uses a decentralized and unpermissioned network of parties that reach consensus on a public distributed database *i.e.,* Bitcoin’s blockchain. Bitcoin’s consensus system is designed to approximately model an election in which parties in the Bitcoin network vote on consensus decisions by spending computational power such that each parties voting power in the election is roughly proportional to the share of the computational power each parties commands in the Bitcoin network. Satoshi Nakamoto (Nakamoto, 2008) argues that bitcoin is secure against attackers that seek to shift the blockchain to an inconsistent/incorrect state, as long as these attackers control less than half of the computational power in the network. But underlying this claim is the crucial assumption of *perfect information*; namely, that all members of the bitcoin ecosystem have an unfiltered view of the blockchain.

In this chapter we look at how to violate this core assumption by performing eclipse attacks against Bitcoin's P2P network and subverting Bitcoin's security. In an eclipse attack (Castro et al., 2002; Sit and Morris, 2002; Singh et al., 2006), the attacker monopolizes all of the victim's incoming and outgoing connections, thus isolating the victim from the rest of its peers in the network. Once isolated, a victim can be shown conflicting views of Bitcoin's blockchain. We investigate the security issues around this assumption in Bitcoin and propose countermeasures to such attacks.

The attacks we present in this chapter are *off-path*. We don't assume the attacker controls network infrastructure between the victim and the rest of the bitcoin network. The attacker merely controls hosts at different IP addresses which join the Bitcoin P2P network and isolate a victim node from the rest of the network by manipulating the peering mechanisms in Bitcoin.

The main challenge for the attacker is to obtain a sufficient number of IP addresses. We consider two attack types: (1) infrastructure attacks, modeling the threat of an ISP, company, or nation-state that holds several *contiguous* IP address blocks, and (2) botnet attacks, launched by bots with addresses in *diverse* IP address ranges. We use probabilistic analysis, measurements, and experiments on our own live bitcoin nodes to determine the number of IP address necessary to launch eclipse attacks for both infrastructure attacks and botnets. We find that while there are hundreds of organizations that have sufficient IP resources to launch eclipse attacks, botnet based attacks require far fewer IP addresses,

After modelling, measuring and experimentally validating this attack we propose a set of countermeasures that preserve Bitcoin's openness while increasing Bitcoin's resistance to eclipse attacks. Our countermeasures ensure that, with high probability, if a victim stores enough legitimate IP addresses that accept incoming connections, then the victim cannot be eclipsed *regardless of the number of IP addresses the attacker controls.* Eight of our ten countermeasures have been deployed and currently protect the Bitcoin network.

Chapter 3 has some strong parallels with the preceding Chapter 2 as both deal with the ramifications of an attacker who presents inconsistent views of a distributed database. Bitcoin has a built-in consensus mechanism and each update *i.e.,* block, commits to the full history of all updates it builds on. Thus, any two parties in the Bitcoin network, if they can communicate with each other, can compare their views of the Bitcoin blockchain and detect inconsistent views.

In the RPKI today, we don't have the ability to check consistency, so the countermeasure we propose in Chapter 2 is to modify the RPKI protocol to enable parties to detect inconsistent states. Thus, there is a escalating set of defenses for distributed databases. First, as Chapter 2 shows, being able to check consistency is critical to defending against and detecting misbehaving parties. Once we can check consistency using the countermeasures in Chapter 2 we must next concern ourselves with the attacks and countermeasures in Chapter 3. Thus to defend against an attacker presenting inconsistent states we propose consistency checks. However once consistency attacks are in place, an attacker who wishes to present inconsistent states may opt to bypass such checks by employing eclipse attacks. These eclipse attacks would prevent parties from communicating and thereby prevent these parties from detecting the inconsistent states. As a counter to this counter the defender must now deploy defenses against eclipse attacks to secure the ability of parties to run the consistency checks.

1.2.3 Chapters 4, 5: Blindly Signed Contracts and TumbleBit

In the final two chapters of this dissertation (Chapter 4 and Chapter 5), we shift to the pressing problem of anonymity in Bitcoin. Anonymity was initially one of the key selling points of Bitcoin. It is stated in the Bitcoin whitepaper that users should be able to spend bitcoins "without information linking the transaction to anyone" (Nakamoto, 2008). However later research has shown that Bitcoin offers much weaker anonymity than initially supposed (Meiklejohn et al., 2013; Ron and Shamir, 2013).

The solutions put forward by the cryptocurrency community to address the Bitcoin privacy problem fall into two categories. The first category are schemes that abandon Bitcoin altogether and propose new anonymous cryptocurrencies (Miers et al., 2013; Ben Sasson et al., 2014; Monero, 2016). The second category attempts to maintain compatibility with Bitcoin by building privacy protocols on top of Bitcoin (Barber et al., 2012; Bonneau et al., 2014; Ruffing et al., 2014; Valenta and Rowan, 2015; Bissias et al., 2014; Maxwell, 2013b; Ben Sasson et al., 2014; Saxena et al., 2014; Ziegeldorf et al., 2015).

The privacy work in this dissertation follows this second approach. We design two Bitcoin privacy protocols: Blindly Signed Contacts and TumbleBit. In Chapter 4 we cover our Bitcoin privacy protocol Blindly Signed Contracts. Blindly Signed Contracts is not fully compatible with Bitcoin as it requires a minor change to Bitcoin consensus. Chapter 5 introduces our more advanced protocol TumbleBit which improves on Blindly Signed Contracts and is fully compatible with Bitcoin; no consensus changes required.

Both of our approaches are built around the concept of a *tumbler*. Tumblers, *aka,* mixers, are a privacy enhancing intermediary used in Bitcoin and other cryptocurrencies. Users send bitcoins to a tumbler to be mixed with the bitcoins of other users. In theory tumblers obscure which user owns which bitcoins by providing set-anonymity between all users who submitted their coins in a mix. As discussed in both chapters, many deployed bitcoin tumblers are able to violate the anonymity of their users or to steal bitcoins from users as the coins are being mixed. The goal of our research with both Blindly Signed Contracts and TumbleBit is to design untrusted tumblers. By untrusted tumblers we mean tumblers which even if they are malicious can not violate the anonymity of their users and can not steal bitcoins from their users.

To build an untrusted tumbler we must overcome two problems. Problem one is how to achieve anonymity, that is, how to ensure that neither the Tumbler nor any party watching Bitcoin's blockchain can link or trace bitcoins through a mix performed by the tumbler. Problem two is how to guarantee *fair-exchange*, by which we mean that no participant in the protocol including the tumbler can steal bitcoins from another participant in the protocol.

Our anonymity property is based on *RSA Blind Signatures.* Blind Signatures were first introduced in eCash (Chaum, 1983a) for a similar purpose. The idea in eCash is that a bank can issue a coin to a user, say Alice, by signing a secret number that has been cryptographically obscured with a "blinding factor". Alice can then remove the blinding factor to get an unblinded signature which we call a "coin". Alice can then pay another user such as Bob by transmitting the coin to Bob. Later Bob can redeem the coin to the bank. The bank is able to check the signature is valid to ensure it actually issued the coin but because of the blinding factor the bank is unable to link the coin Bob redeemed with the bank, to the coin the bank issued to Alice. This enables Alice to pay Bob without revealing to the bank that the coin came from Alice.

A drawback of eCash is that a bank can cheat users by refusing to redeem a user's coin *i.e.,* fair-exchange is not a goal or feature of the eCash protocol. Our work improves on this by using Bitcoin's transaction scripting language *aka,* smart contracts (Szabo, 1997), to design protocols whereby fair exchange is achieved.

Our first protocol, Blindly Signed Contracts (Chapter 4) designs a protocol for a Bitcoin tumbler which allows parties to anonymously send Bitcoin payments using RSA Blind Signatures for anonymity and Bitcoin's scripting language to ensure the fair-exchange. This chapter includes two separate protocols. The first protocol is an on-chain scheme providing anonymity at reasonable speed. This protocol requires four transactions to be confirmed in three blocks which takes roughly 30 minutes. The second protocol is an off-chain *aka,* a layer-two scheme that uses*payment channel networks* (Poon and Dryja, 2015; Decker and Wattenhofer, 2015) to process payments in seconds rather than minutes. We show that our on-chain protocol maintains fair-exchange, preventing both the tumbler and the users from cheating each other or violating each others anonymity. The anonymity provided is set-anonymity for the set of participating users. Our off-chain scheme, while boasting faster payment confirmation times, only provides anonymity against an *honest-but-curious* tumbler. By *honest-but-curious* we mean a party that correctly follows the protocol but records everything and is not trusted to delete information.

While Blindly Signed Contracts is designed to work with Bitcoin, it requires that bitcoin transactions support the validation of RSA Blind Signatures. For this reason Blindly Signed Contracts is not fully compatible with Bitcoin and would require a soft-fork of Bitcoin to add this functionality.

In Chapter 5 we introduce TumbleBit. This chapter follows the pattern of the previous chapter (Chapter 4) and introduces a design for an untrusted Bitcoin tumbler protocol called TumbleBit. Like Blindly Signed Contracts, TumbleBit includes both an on-chain scheme and an off-chain scheme. However, Tumblebit is significantly more advanced and improves on Blindly Signed Contracts in every way.

Unlike Blindly Signed Contracts, TumbleBit does not require any protocol changes to Bitcoin. To accomplish this we invent a specialized contingent payment protocol that allows the atomic exchange of a bitcoin for an RSA signature or RSA decryption (essentially an RSA secret key exponentiation). To achieve this property we make use of the *cut-and-choose* paradigm along with ephemeral signing keys and time-locked Bitcoin transactions. Our proposed scheme, TumbleBit, achieves the optimal set of the desired properties while being relatively fast and fully compatible with Bitcoin. We formally analyze and prove the security and anonymity properties of our scheme even against a malicious Tumbler (aka intermediary). While Blindly Signed Contracts' off-chain scheme was only secure against an honest-but-curious tumbler both TumbleBit on-chain and off-chain schemes are secure against a fully-malicious tumbler.

At the time at which this research was first published, previous work either provided schemes that were efficient but achieved limited security or anonymity (Bonneau et al., 2014; Ruffing et al., 2014; Valenta and Rowan, 2015; Ziegeldorf et al., 2015; Saxena et al., 2014) or schemes that provided strong anonymity but were slow and required large numbers of transactions to achieve anonymity (Maxwell, 2013b; Bissias et al., 2014; Barber et al., 2012). Our schemes offered a new trade-off between transaction speed, security (*i.e.,* resistance to double-spending, denial of service (DoS) and Sybil attacks) and anonymity (*i.e.,* unlinkable transactions).

1.3 Protocol Design Philosophy and Impacts

There are two themes running through the research that we present in this work. The first theme is an approach that favors carefully improving existing protocols over complete reinvention. This is often called dirty slate or brown field design in contrast to clean slate or green field design. The second theme is that we approach these protocols with a healthy level of skepticism towards trusted parties. All of our designs work to reduce the necessary level of trust that must be placed in intermediaries, authorities or counter parties.

Our embrace of dirty slate design has born fruit. While the changes and countermeasures to the RPKI we proposed in Chapter 2 were not adopted, the risks of RPKI authority abuse that we investigated are viewed with serious concern by the RPKI community (Kuerbis, 2013). Research on how to counter these threats continues to be published and developed (Xing et al., 2018; Li et al., 2018; Kent and Ma, 2017; Shrishak and Shulman, 2020; Hlavacek et al., 2020). Our TumbleBit protocol became an open source project and was deployed in the privacy focused Breeze Wallet. More importantly TumbleBit helped introduce ideas of blind signatures to Bitcoin privacy wallets inspiring the current generation of Bitcoin privacy protocols. Finally many of our eclipse attack countermeasures were adopted by Bitcoin and several other cryptocurrencies including DogeCoin, Monero and ZCash. A long line of research on cryptocurrency eclipse attacks followed our original paper and this continues to be an active area of research.

In the next few paragraphs will give a brief summary of the impact of the TumbleBit protocol and its successors. We start with TumbleBit becoming an open source project, the deployment and use of this open source project and finally the next generation of protocols that it inspired. A timeline of these events is provided in Figure 1·1. Afterwards we will move on to examine the influence and impact of our Bitcoin eclipse attack research.

Following the publication of the TumbleBit protocol discussed in Chapter 5, Nicolas Dorier launched the open source project NTumbleBit (Dorier, 2016) a full implementation of the TumbleBit protocol. NTumbleBit was then used by Breeze Wallet to support the TumbleBit protocol and to provide privacy to their users. In parallel, Ádám Ficsór, one of the contributors to NTumbleBit, began developing a TumbleBit-

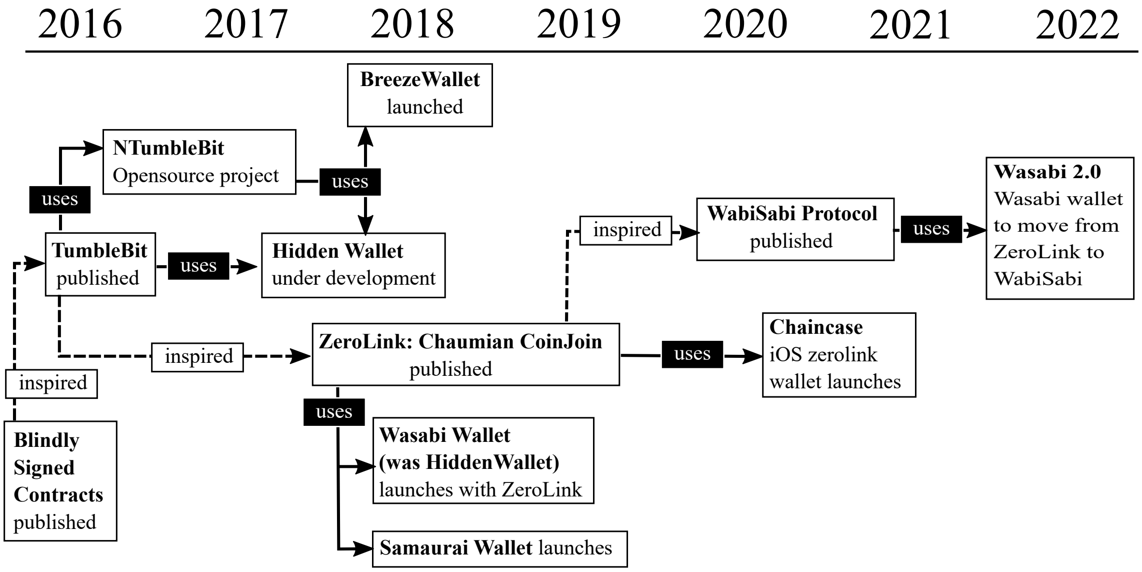

2016
2017
2018
2019
2020
2021
2022
BreezeWallet launched
NTumbleBit Opensource project
uses
uses
TumbleBit published
uses
Hidden Wallet under development
inspired
inspired
ZeroLink: Chaumian CoinJoin published
inspired
Blindly Signed Contracts published
uses
Wasabi Wallet (was HiddenWallet) launches with ZeroLink
Samaurai Wallet launches
inspired
WabiSabi Protocol published
uses
Wasabi 2.0 Wasabi wallet to move from ZeroLink to WabiSabi
uses
Chaincase iOS zerolink wallet launches

Figure 1·1: Timeline of TumbleBit and successors protocols. A solid line means that a project uses a protocol or library. A dotted line means that a protocol is inspired by another protocol.

based privacy wallet which at the time was called Hidden Wallet (nopara73, 2017) but would be later renamed to Wasabi Wallet. However, prior to its public release, Wasabi Wallet (formerly known as Hidden Wallet) opted to replace TumbleBit with a newly created privacy protocol called ZeroLink. In the next paragraph we will look at ZeroLink, its connections to TumbleBit and it's use in the Bitcoin privacy ecosystem.

The ZeroLink protocol (Ficsór and TDevD, 2017) was jointly created by Ádám Ficsór of Wasabi Wallet and TDev, the developer of another privacy wallet named Samourai Wallet. ZeroLink uses the same principles of blind signatures used in Blindly Signed Contracts and TumbleBit. According to Ádám Ficsór ZeroLink was directly inspired by TumbleBit (Ficsór, 2021b). This is not surprising since Ádám Ficsór was an active contributor to NTumbleBit and was using TumbleBit in early versions of Wasabi Wallet *aka,* Hidden Wallet. The connections between the NTumbleBit community and Wasabi Wallet are strong and to this day Wasabi Wallet development and research is organized on the TumbleBit slack server. The ZeroLink protocol is now widely used to protect privacy in Bitcoin since it is the privacy protocol behind both major Bitcoin privacy wallets: Wasabi Wallet and Samourai Wallet. One study (Stockinger et al., 2021) estimates that from Jan 2021 to Jun 2021 the average amount mixed was 4,563 BTC (204 million USD) per month for Wasabi and 847 BTC (35 million USD) per month for Samourai. This largely agrees with two similar investigations (Wu et al., 2021; Ficsór, 2021a) showing massive use of ZeroLink in the Bitcoin privacy ecosystem.

Work on deploying and improving these protocols continues to the present day. The ZeroLink protocol is now available on mobile phones with the creation of the ChainCase wallet (Gould, 2021). The Wasabi Wallet developers have created a ZeroLink successor called WabiSabi (Ficsór et al., 2021) which they are planning to deploy sometime in late 2021 or early 2022 as a replacement for ZeroLink. Even though TumbleBit is no longer widely used, it introduced the ideas of Chaumian eCash (Chaum, 1983b) and blind signatures to the privacy wallet space and thus helped inspire the design of one of the most widely used Bitcoin privacy protocols ZeroLink.

We now switch gears to look at the impact of the research presented in Chapter 3, eclipse attacks on Bitcoin. We will briefly examine how this research impacted the design of Bitcoin as well as several other cryptocurrencies and the line of research that followed it.

Prior to publishing the research in Chapter 3, we privately disclosed our attacks and countermeasures to the bitcoin-core developers. Their initial remediation was to implement and deploy three of our countermeasures in bitcoin-core version 0.10.1. As time went on, more of the countermeasures we proposed were deployed into bitcoin-core. Now, as of 2021, eight of the eclipse attack countermeasures proposed in Chapter 3 have been merged and deployed into the bitcoin-core project (see Table 1.1).

Countermeasure	Version	Date	Author	PR
1	v0.10.1	04-2015	Pieter Wuille	5941
2	v0.10.1	04-2015	Pieter Wuille	5941
6	v0.10.1	04-2015	Pieter Wuille	5941
9	v0.12	02-2016	Patrick Strateman	6374
4	v0.13.1	10-2016	Ethan Heilman	8282
3	v0.17.0	10-2018	Ethan Heilman	9037
7	v0.19.0.1	11-2019	Suhas Daftuar	15759
5	v0.21.0	08-2020	Hennadii Stepanov	17428

Table 1.1: A timeline of bitcoin-core deploying our proposed countermeasures

We were actively involved in this process and wrote the patches behind two of the countermeasures. Because bitcoin-core's source code is used by many other cryptocurrencies, these security improvements have been pulled into a large number of other bitcoin-derived cryptocurrency projects. For instance the patch that implements our countermeasure two has been used in approximately $3,000$ other git repositories including DogeCoin (Sipa, 2015). Our research has also influenced the design of the peer-to-peer networking architecture in ZCash (Liu and Hopwood, 2018) and one of our countermeasures has also been independently implemented and deployed in Monero (Monero, 2016; IPGlider, 2017).

In addition to this our research helped launch a line of research on cryptocurrency eclipse attacks and defenses. A systematic overview of this body of work is beyond the scope of this dissertation and would by itself represent a significant research project. Some highlights of this research are (Nayak et al., 2016; Natoli and Gramoli, 2017; Zhang and Lee, 2019), eclipse attacks on Ethereum (Marcus et al., 2018; Ekparinya et al., 2018), eclipse attacks on Monero (Cao et al., 2020), and eclipse attacks on Bitcoin via inter-domain routing (Apostolaki et al., 2017; Tran et al., 2020; Tran et al., 2021).

We believe that TumbleBit and our research on eclipse attacks had the impact that it did because of our focus on dirty slate design and our engagement with the developer community. While TumbleBit influenced the creation of ZeroLink, it was not the first paper to suggest the use of blind signatures for Bitcoin privacy. The original 2013 post on CoinJoin (Maxwell, 2013a) proposed using blind signatures and provided a brief sketch of how this might be done. In fact there are significant similarities between the sketch given in the CoinJoin post and ZeroLink. We posit that TumbleBit succeeded in transferring these ideas to the Bitcoin privacy community not only because our protocol was fully compatible with Bitcoin, but also because we published running code (Hughes, 1993) and because we engaged with and supported the Bitcoin privacy wallet developer community. In a similar fashion as TumbleBit, we believe our research on eclipse attacks benefited from the fact that we were studying a widely deployed protocol, namely Bitcoin, and that the countermeasures we proposed only required minor changes to bitcoin-core's peer-to-peer networking software. Had we proposed green field protocols and countermeasure our research may have had diminished value and interest to these communities. All that said, this impact would not have been possible without the bitcoin-core community or the Bitcoin privacy wallet community investing the time and effort to engage with us, give us feedback and share their knowledge with us.

1.4 Contributions

We will now briefly summarize the contributions of each of chapters of this dissertation.

1.4.1 Improving the Transparency of the RPKI

1. An exploration of the interactions between the RPKI and BGP, showing the circumstances under which an RPKI revocation of an IP prefix by an RPKI authority may or may *not* impact routing. We untangle these often-unintuitive interactions (Section 2.2) and discuss the ways in which the RPKI's architecture

makes accountability difficult (Section 2.4).

2. The design and implementation of analysis tools that can increase RPKI transparency by allowing an RPKI observer to detect and understand takedowns and other issues. The first tool is an RPKI *detector*. It identifies and visualizes changes to the RPKI that can impact IP prefixes (Sections 2.3). The second tool is a *repair localizer*, that identifies RPKI authorities that are able to fix RPKI problems (Section 2.5).

3. An RPKI measurement study showing the evolution of the state of the RPKI over time. We test our analysis tools on the production RPKI and identify real-life errors and revocations (Section 2.3.2). Since RPKI deployment is still in its infancy, we also develop and work with models of a future full-deployment of the RPKI, based on routing data and RIR information (Section 2.6.8).

4. Mirror world attacks, a new type of attack on RPKI's distributed database in which a malicious RPKI authority presents different views of RPKI to different relying parties to manipulate their views of RPKI state and to hide misbehavior.

5. Proposed improvements to RPKI to increase its transparency and protect its consistency even against mirror world attacks. The current RPKI specifications place power squarely in the hands of authorities and lack robust accountability mechanisms (Section 2.4.4). To remedy this, we propose modest modifications to the architecture of RPKI (Section 2.6). We prove the security properties of our design, and use data-driven analysis to estimate its overhead, including the number of entities that must participate in our consent mechanism.

1.4.2 Eclipse Attacks on Bitcoin's P2P Network

1. Analysis of how Bitcoin consensus and other security properties can be subverted by an eclipse attack on Bitcoin's P2P network. These attacks include the ability to cause adversarial forks in the blockchain which can allow an attacker the ability to double spend a transaction even after that transaction has had many confirmations on the blockchain.

2. Probabilistic modeling of Bitcoin's peer-to-peer network and peer discovery mechanism. These models allow us to quantify how much security the peer-to-peer network provides against an eclipse attacker exploiting Bitcoin's peer-discovery mechanisms.

3. Experiments validating our attacks under real world conditions. These experiments recorded the internal state of Bitcoin nodes allowing us to measure the degree to which our models correctly simulated the behavior of a Bitcoin node. We demonstrated the practically of our attack by performing it on our own live bitcoin nodes, we found that an attacker with 32 distinct /24 IP address blocks, or a 4600-node botnet, can eclipse a victim with over 85% probability in the attacker's *worst case*. Moreover, even a 400-node botnet sufficed to attack our own live bitcoin nodes. This contribution includes the release of open source software for conducting such experiments (Heilman, 2016a).

4. Measurement study of the peering behavior of the Bitcoin peer-to-peer network.

5. Proposed countermeasures that make eclipse attacks more difficult while still preserving bitcoin's openness and decentralization. Eight of these countermeasures were merged into Bitcoin including (Heilman, 2016c; Heilman, 2016b; Heilman, 2018).

1.4.3 Blindly Signed Contracts: Anonymous Bitcoin Transactions

1. A tumbler protocol (Heilman et al., 2016b), Blindly Signed Contracts, based on modifying the Bitcoin protocol so that it can evaluate RSA blind signatures in Bitcoin transactions. With this one change we build a Bitcoin tumbler which users do not need to trust.

2. A privacy preserving fee system, allowing the Tumbler to resist DoS (Denial of Service) attacks where attackers seek to overload the Tumbler by joining and then aborting the protocol.

3. We perform an analysis of our protocol showing that it achieves: Fair-exchange (neither the tumbler nor the users can cheat each other) and Sybil Resistance.

4. We extend this to develop a private micropayment network, a type of layer-two blockchain protocol, allowing parties to establish micropayment channels with the hub and then route payments off-chain through the hub while maintaining privacy using RSA blind signatures. This provides scalability benefits over the tumbler and fits the payment usecase.

1.4.4 Tumblebit: An Untrusted Bitcoin-compatible Anonymous Payment Hub

1. In Tumblebit we extend our work from Blindly Signed Contracts showing that an untrusted tumbler is possible without any changes to Bitcoin. This result in TumbleBit involves the invention of two novel fair-exchange protocols that are written in Bitcoin's smart contracts language. We developed (off-chain) cryptographic protocols that work with the very limited set of 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. To achieve this we introduce novel puzzle-prover and puzzle-solve cryptographic protocols that allow mutually untrusting parties to convince each other that various encrypted values decrypt to the output of blinded RSA operations.

2. We show how this protocol can be performed off-chain via uni-directional micropayment channels. This allows TumbleBit to both provide privacy and help scale the velocity (time to finalization for a payment) and volume of payments Bitcoin can support.

3. As with Blindly Signed Contracts we perform an analysis of the TumbleBit protocol showing that it provides set-anonymity, fair-exchange, DoS Resistance and Sybil Resistance.

4. We implemented TumbleBit[1] in C++ and python using LibreSSL and tumbled payments from 800 payers to 800 payees; the relevant transactions are visible on the blockchain. 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 $\approx$ 20 minutes. Meanwhile, off-blockchain payments can complete in seconds.

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

Chapter 2

Improving the Transparency of the RPKI

This chapter is based on work from (Heilman et al., 2014) which was written in collaboration with Danny Cooper, Leonid Reyzin and Sharon Goldberg.

2.1 Introduction

The RPKI (Lepinski and Kent, 2012) is a new infrastructure for securing interdomain routing with BGP. BGP has traditionally operated as a "default-accept" architecture: any autonomous system (AS) can originate a BGP routing announcement (*i.e.,* claim to be the destination for) for any IP prefix, and other ASes will accept the BGP announcement by default. This has made BGP vulnerable to a number of routing attacks, the most common (Rensys Blog, 2008; Misel, 1997; de Beaupre, 2013; Cowie, 2010; Pilosov and Kapela, 2009; Peterson, 2013) and devastating (Butler et al., 2010; Huston et al., 2011; Goldberg et al., 2010; Ballani et al., 2007) of which are prefix- and subprefix hijacks. In a prefix hijack, a hijacking AS originates BGP routes for IP prefixes that were not allocated to it, causing the traffic for those prefixes to be intercepted by the hijacker's AS.

The RPKI prevents these attacks by providing a trusted mapping from allocated IP prefixes to ASes authorized to originate them in BGP. To do this, the RPKI establishes a top-down hierarchy of *authorities*, rooted at the Regional Internet Registries (RIRs), that allocate and suballocate IP address space, as well as authorize its use in BGP by individual origin ASes; routers use the RPKI to distinguish between hijacked

BGP routes and routes originated by a legitimate AS. The RPKI also turns out to be surprisingly effective against attacks it was not designed to prevent (Goldberg et al., 2010); (Lychev et al., 2013) shows that more advanced secure routing solutions (Lepinski, 2012; Kent et al., 2000) provide limited benefits over what is already provided by the RPKI. The RPKI requires neither changes to BGP nor online cryptographic computations during routing. It is currently being rolled out by RIRs and adopted by individual network operators, and authorizes about 20,000 BGP routes as of January 2014 (Section 2.3.2).

From default-accept to default-deny. However, the security benefits of the RPKI are accompanied by a drastic shift from BGP's traditional "default accept" policies, to a new "default deny" mode: to prevent (sub)prefix hijacks, routers should only accept routes authorized by the RPKI, and discard all other routes by default (Section 2.2.3). This is further complicated by the fact that the RPKI's hierarchical architecture empowers centralized authorities to *unilaterally* revoke authorization for IP prefixes under their control. This shift has lead to concerns (Amante, 2012; Mueller and Kuerbis., 2011; Communications Security, Reliability and Interoperability Council III (CSRIC), 2011; The President's National Security Telecommunications Advisory Committee, 2011; Mueller et al., 2013) that the RPKI creates powerful authorities with the technical means for taking down IP prefixes, and could be exploited by abusive authorities or governments to settle disputes or block undesirable content. This is a stark departure from the status quo, where these authorities (RIRs, National/Local Internet Registries, *etc.*) had the power to allocate IP address space, but not to impact routing to space that has been allocated (Goldman, 2006; Project, 2011).

Transparency. In light of the risk of takedowns, it would be useful to have mechanisms that can hold misbehaving RPKI authorities accountable; this could create social (and possibly legal) pressure to motivate misbehaving authorities to fall in line. However, the architecture of the RPKI also makes it difficult to distinguish between revocations due to disputes and those reflecting legitimate business agreements, or even to *identify* the authority that performed a revocation (Section 2.4).

Our contributions. As RPKI deployment continues to gain traction, we present an investigation of the risk of RPKI takedowns, and propose technical solutions that mitigate this risk by improving the transparency of the RPKI. Our contributions are:

1. Security audit. Complex interactions between the RPKI and BGP mean that it is *not* always the case that a revocation in the RPKI can takedown an IP prefix in BGP. We untangle these often-unintuitive interactions (Section 2.2) and discuss how the RPKI's architecture makes accountability difficult (Section 2.4).

2. Tools, measurement & modeling. We start by working within the current RPKI specifications, and build two tools that can increase transparency by detecting and reacting to RPKI problems: a *detector*, that identifies and visualizes changes to the RPKI that can takedown IP prefixes (Sections 2.3), and a *repair localizer*, that identifies RPKI authorities that can fix the problems that result (Section 2.5). We test our tools on the production RPKI and identify real-life errors and revocations (Section 2.3.2). Since RPKI deployment is still in its infancy, we also develop and work with models of a future full-deployment of the RPKI, based on routing data and RIR information (Section 2.6.8).

3. Changes to the specifications. The current RPKI specifications still place power squarely in the hands of authorities, and lack robust accountability mechanisms (Section 2.4.4). To remedy this, we propose modest modifications to the architecture of the RPKI (Section 2.6). Our design ensures that once a route is authorized by the RPKI, it switches from “default-deny” to “default-accept”; that is, authorities that revoke IP address space must first obtain *consent* from all entities holding allocations (and suballocations) of that space. Our design also (1) provides *accountability* by guaranteeing identification of authorities that fail to properly obtain consent, and (2) allow parties viewing RPKI information from different vantage points to detect when their views are not *consistent*. We prove the security properties of our design, and use data-driven analysis to estimate its overhead, including the number of entities that must participate in our consent mechanism.

2.2 The risk of RPKI takedowns

We discuss the conditions under which manipulations of the RPKI can take IP prefixes offline. We overview the RPKI's certificate hierarchy, explain how RPKI information determines route validity, and show how route validity affects the availability of routes in BGP.

2.2.1 The hierarchical structure of the RPKI.

The RPKI follows the "principle of least privilege", arranging authorities in a strict hierarchy that mirrors the IP address allocation hierarchy. An authority may issue cryptographic objects for IP addresses that are *covered* by its own IP addresses.[1] Today, IANA sits at the root of this hierarchy, allocating IP addresses to the Regional Internet Registries (RIRs), who allocates subsets of their address space to national internet registries or ISPs, who further allocate subsets to others.[2]

RPKI Objects. In the RPKI, each authority has a *resource certificate (RC)*, a certificate that contains its cryptographic public key and its set of allocated IP addresses (Manderson et al., 1973). An authority may issue signed cryptographic objects for IP addresses covered by its allocation, specifically: (1) an RC that suballocates

[1] An IP prefix P *covers* prefix π if π is a subset of the address space in P (*e.g.,* 63.160.0.0/12 covers 63.160.1.0/24) or if $P = \pi$. Also, a prefix 63.160.0.0/12 has *length* 12.

[2] The roots of the RPKI are the five RIRs (Table 2.1); in the future, IANA could be a single root (Lepinski and Kent, 2012, Section 2.4).

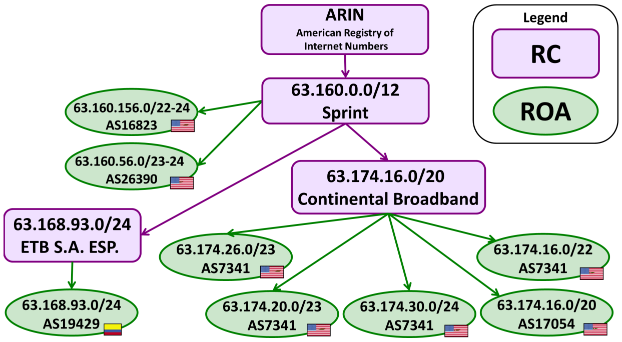


Figure 2·1: Excerpt of a model RPKI

a subset of its addresses to another authority, or (2) a *route origin authorization (ROA)*[3], that authorizes a specified AS to originate a set of prefixes, and its subprefixes up to a specified length, in BGP (Lepinski and Kent, 2012).

Model (Figure 2·1). We show how an RIR (ARIN) uses its RC to suballocate a prefix to another authority (Sprint), which then issues RCs suballocating this prefix to other authorities (ETB S.A. ESP., Continental Broadband). (This is a excerpt of one of our models of the fully-deployed RPKI; see Section 2.6.8.) We say Sprint is the *parent* of Continental Broadband, and extend this to child, grandparent, *etc.*in the obvious way. Sprint issues two ROAs that authorize specified prefix and its subprefixes of length up to 24; the remaining ROAs shown authorize only a single prefix.

Status of the RPKI (Table 2.1). As of January 13, 2014, the production RPKI contains ROAs for about 20,000 prefix-to-origin AS pairs. (Note that about 488K prefixes were announced in BGP on the same day). The RPKI's structure on January 13, 2014, (Table 2.1) is simpler than some of our models. At depth 0, there are trust anchors for each RIR (*e.g.,* ARIN); the trust anchors issue a handful of intermediate RCs (at depth 1); intermediate RCs issue leaf RCs (*e.g.,* Sprint) who then issue ROAs. ARIN has an extra layer of intermediate RCs. Currently, RCs for suballocations like Continental Broadband and ETB in Figure 2·1 are absent; their ROAs could just be issued by Sprint. ROAs usually contain one AS and many prefixes

Depth	1	2	3	4
RIPE	4 RC	1909 RC	1512 ROA	151 ROA
LACNIC	4 RC	282 RC	282 ROA	
ARIN	1 RC	1 RC	99 RC	
APNIC from IANA	1 RC	450 RC	58 ROA	
AfriNIC	1 RC	27 RC	48 ROA	

Table 2.1: Valid ROAs and RCs at each depth of the production RPKI on January 13, 2014.

(*e.g.,* all prefixes for AS 7341 in could be issued in one ROA).

Repositories. The RPKI was designed to require minimal changes to BGP, and therefore operates entirely out-of-band. RPKI objects are stored in publicly-available repositories. Every RC has its own *publication point* (*i.e.,* folder in a file system) where it publishes every object it issued. *Relying parties* download RPKI objects from publication points to their *local caches*, validate the objects, push information to their routers, and use it to inform routing decisions they make in BGP.

2.2.2 How the RPKI limits threats to BGP.

To see how threats to the RPKI can impact routing with BGP, we need to understand how threats to BGP influenced the design of the RPKI.

Threats to BGP. The RPKI is designed to prevent the most devastating attacks on interdomain routing with BGP: prefix and subprefix hijacks (Butler et al., 2010), where a hijacking AS originates BGP routes for IP prefixes that it is not authorized to originate, causing the traffic intended for those prefixes to be intercepted by the hijacker's AS (Rensys Blog, 2008; Misel, 1997; de Beaupre, 2013; Cowie, 2010; Pilosov and Kapela, 2009; Peterson, 2013). Prefix/subprefix hijacks are off-path attacks; any router connected to the Internet can launch one of these this attacks, with varying degrees of success at attracting traffic (Ballani et al., 2007; Goldberg et al., 2010). BGP is especially vulnerable to subprefix hijacks because of longest-prefix-match routing: when a router learns BGP routes for a prefix and its subprefix, it always prefers the subprefix route. Subprefix hijackers, which exploit this by originating routes for subprefixes of a victim prefix. This leads to a natural desideratum for the RPKI: a subprefix hijacker's route should be always be invalid when victim's route has a matching valid ROA.

Origin authentication. To achieve this desideratum, a relying party uses the RPKI for *origin authentication* as follows. For our purposes, a BGP *route* is an IP prefix π and an origin AS a. Once a relying party has "access to a local cache of the complete set of valid ROAs" (Huston and Michaelson, 2012, Sec. 2), these valid ROAs are used to classify each route learned in BGP into one of three *route validation states* (Mohapatra et al., 2013; Huston and Michaelson, 2012):

- *Valid:* There is a valid *matching ROA*. A matching ROA has (1) a matching origin AS a, and (2) a prefix P that covers prefix π, and (3) a specified maximum length no shorter than the length of π.
- *Unknown:* There is no valid *covering ROA*. A covering ROA is any ROA for a prefix that covers π.
- *Invalid:* The route is neither unknown or valid.

The rules above elegantly achieve the desiderata; the ROA for (63.174.16.0/20, AS 17054) protects the corresponding route from subprefix hijacks, because all routes for its subprefixes are invalid (except routes with matching ROAs of their own). Thus, to realize the RPKI's potential to stop BGP attacks, relying parties should *drop invalid routes*, *i.e.,* not select BGP routes that are classified as invalid by the RPKI.

2.2.3 A default-deny architecture.

The RPKI is, in many ways, a "default-deny" architecture; the presence of a valid ROA means that all routes for its subprefixes are invalid, unless they have their own matching valid ROAs. This has serious implications: a relying party that *drops invalid routes* can lose connectivity to corresponding IP prefix in BGP if problems

policy	routing attack	RPKI manipulat'n
drop invalid	stops (sub)prefix hijacks	prefix goes offline
depref invalid	subprefix hijacks possible	prefix may stay online

Table 2.2: Impact of different local policies.

with the RPKI cause a route to be wrongly classified as invalid. A misbehaving RPKI authority (or even a denial-of-service attack on the RPKI) can exploit the default-deny nature of the RPKI to block connectivity to IP prefixes in BGP. We discuss the mechanics of this in Sections 2.4.2,2.4.1, and 2.4.2.

Granularity. While the RPKI allows ROAs to have arbitrary prefix lengths, the longest IPv4 prefix length that is universally accepted by BGP routers is a /24. (Routers usually ignore longer prefixes to avoid bloating their routing tables.) Thus, our discussion on the RPKI's impact on IP prefix reachability should be thought of as having the granularity of a /24 (or shorter) IPv4 prefix, *i.e.,* no fewer than 256 IPv4 addresses; the RPKI, therefore, can be used for significantly coarser level of blocking than *e.g.,* the DNS (Piscitello, 2012; Piscitello, 2013).

Local policies. Because moving to a default-deny regime is a drastic change for the routing infrastructure, the RPKI specifications explicitly state that relying parties may use their own "local policies" to decide what to do with invalid routes (Huston and Michaelson, 2012). *Drop invalid routes* is one possible local policy.

An alternative, more lenient policy suggested by (Huston and Michaelson, 2012) is to *depref invalid routes*: for a given prefix, a router should prefers valid routes over invalid routes. This policy implies that a router still selects an invalid route when there is no valid route for the *exact same* IP prefix. Thus, the router may still be able to reach routes that are wrongly classified as invalid as a result of problems with the RPKI.[4] However, this policy does *not* prevent subprefix hijacks; see (Bush, 2012b,

[4] However, availability of a route at one router can depend strongly on local policy used at other routers. For example, a router that uses the lenient *depref invalid* policy can lose connectivity to an "invalid" route if all its neighboring routers use the strict *drop invalid* policy.

Section 5).

A difficult tradeoff. This highlights an inherent tradeoff that is implicit in the RPKI RFCs; namely, that the local policy that is best at protecting against attacks on BGP is worst at protecting against problems with RPKI. See Table 2.2. As of January 2014, there are ROAs authorizing ≈20,000 routes in the RPKI. However, the vast majority of routers *ignore* the information in the RPKI. While it is too early to tell what local policies will be adopted in the long run, it is clear that better assurances about the trustworthiness of RPKI information are need before relying parties can start using policies that reap the security benefits provided by the RPKI. Our focus is on improving these assurances.

2.3 Detecting downgraded routes

One way to improve the trustworthiness of RPKI data is to detect when changes to the RPKI can negatively impact BGP. We now present a tool that detects when any *change* to the RPKI causes a *downgrade* to the route validity state of any prefix-to-origin-AS pair (*i.e.,* causes a transition from "valid" to "invalid" or "unknown", or from "unknown" to "invalid").

2.3.1 A tool for detecting downgrades.

Our tool detects downgrades in the validity state of *all* possible routes, regardless of whether or not they are announced in BGP. It can therefore act as an alert system for potentially-harmful changes to the RPKI, independent of information available from a specific BGP vantage point. Our visualizer, described later in this section, incorporates information from a BGP feed to show the validity of specific routes announced in BGP.

Challenges. The main challenge in detecting downgrades due to changes in the

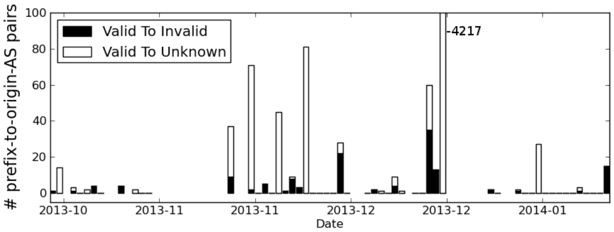


Figure 2·2: Downgrades due to whacked ROAs.

RPKI is that the relationship between a single prefix-to-origin-AS pair and the validity of potential routes in BGP is complex, and depends on the presence of other ROAs in the system. For example, a ROA giving an AS a prefix π of length 17 up to maxlength 22 actually makes $2^{(23-17)} - 1 = 63$ possible announcements by that AS "valid" in BGP. If the ROA gets whacked, those announcements do not necessarily become "invalid": some may become "unknown" (if there is no covering ROA for them), while others may remain valid (if there is another ROA for the same AS and a super or subprefix of π). Moreover, when such a ROA appears, it may downgrade "unknown" routes to "invalid" for any subprefix of π of any length, depending on what super or subprefixes of π exist in the RPKI.

Data structures. Thus, the naive approach to determining the impact of an added or whacked ROA is to search the RPKI for all super and subprefixes of the ROA, and to apply the route-validity rules (Section 2.2.2) to every subprefix of the prefixes in the ROA. Instead, we take a more efficient approach. We focus on the complete binary tree of all IP prefixes (with one IP address per leaf, and all possible prefixes as internal nodes) and observe that a prefix-to-origin-AS pair for prefix π, maxLength m, and origin AS α makes a subtree of this tree "valid" for α: the subtree is rooted at π and goes down to depth m. It also makes the subtree rooted at π and going all the way to the bottom "known" (*i.e.,* the complement of "unknown").

We call these subtrees *triangles* and build a data structure that uses *interval trees*

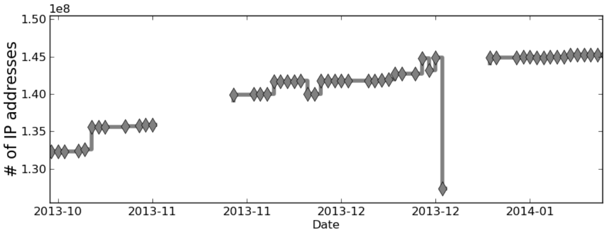


Figure 2·3: # of invalid IP addresses over time.

to efficiently perform unions, intersections, and complements of sets of triangles. From this, we get a "prefix-validity" data structure that stores the validity states for *all* possible routes, based on a set of ROAs in the RPKI; it can be created in time $O(n \log n)$ given ROAs for n (prefix, AS, maxlength)-tuples.

Our tool compares a current RPKI state S_{cur} and a previous state S_{prev}. It processes each state once and creates its prefix-validity data structure. It then evaluates the impact of every single change from S_{prev} to S_{cur} in the context of other changes that occurred, by going through each new or removed (prefix, AS, maxlength)-tuple and extracting the relevant information from the prefix validity data structures.

Visualizing downgrades. We present a new visualization of the effects of a single ROA on the validity states of multiple BGP routes, based on a modification of the Sierpiński triangle. Our visualization takes in information from our detector and from a BGP feed *e.g.,* (Views, 2015; RIPE, 2015), and presents a binary tree that visualizes the triangles described above, as well as the validity state of specific routes announced in BGP.

Figure 2·4(r). We visualize the prefixes covered by Continental Broadband in Figure 2·1. Suppose S_{prev} has the all ROAs issued by Continental Broadband *except* the covering ROA for 63.174.16.0/20, and that the covering ROA is added in S_{cur}. We show downgrades from "unknown" to "invalid" in the transition from S_{prev} to S_{cur}. Routes uncovered by the four ROAs in S_{prev} are impacted during the transition;

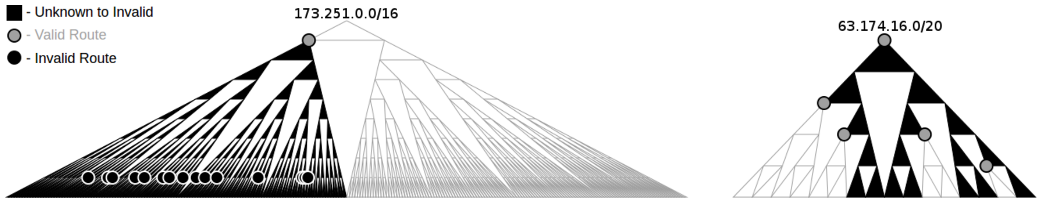


Figure 2·4: (l) Visualization of downgrades in Case Study 2. (r) Downgrades when the ROA (63.174.16.0/20, AS 17054) is added to the RPKI in Figure 2·1.

routes covered by the four ROAs in S_{prev} were *already* invalid in S_{prev}, and do not appear as downgrades.

Tool release. We plan to release this and all our other tools as libraries that can be incorporated into professionally-maintained systems like (NIST, 2013a; RIPE, 2013; LACNIC, 2013).

2.3.2 Tool evaluation & case studies.

A trace of the production RPKI. We collected production RPKI data for 2013/10/23–2014/01/13. Each day, we used rcynic (Lab, 2015) to pull the state of the RPKI to an empty local folder, and to cryptographically validate the result. We excluded from the trace a few days where our collector went down, and a few days where rcynic failed to properly sync to the repository.

We study two incidents from our trace to gain insight on how routes can downgrade in the RPKI.

Case Study 1: Valid to Invalid downgrade. On December 19, 2013, a ROA for (79.139.96.0/24, AS 51813), for a network in Russia, was deleted from the RPKI. Meanwhile, since at least November 21, RPKI also held a covering ROA mapping 79.139.96.0/19-20 to another Russian ISP, AS 43782. The presence of the covering ROA caused the route corresponding to the whacked ROA to downgrade from valid to invalid.

Case Study 2: ROA misconfiguration (Figure 2·4(l)). On December 13, a new ROA was added to the RPKI rooted at ARIN, authorizing prefix 173.251.0.0/17 with maxlength 24 to AS 6128. This caused a large portion of the address space to downgrade from unknown to invalid, including several legitimate /24 routes announced in BGP (*e.g.,* AS 53725 with prefix 173.251.91.0/24, AS 13599 with prefix 173.251.54.0/24, and others) that did not have matching ROAs. Figure 2·4(l) visualizes the incident. The tool outputs the prefix tree rooted at 173.251.0.0/16, and shows how the triangle rooted at 173.251.0.0/17 transitions from unknown to invalid; the valid route for prefix 173.251.0.0/17 is shown with a grey circle, while the legitimate /24 BGP routes that transitioned to invalid are shown with black circles.

We used our detector to check for downgrades between consecutive entries in our trace.

Downgrades due to added ROAs (Figure 2·3). Each time a ROA is added, more of the IP address space transitions to the default-deny state, *i.e.,* all the "unknown" address space it covers downgrades to "invalid" as in Case Study 2. Figure 2·3 shows this transition over the course of our trace. We show the number of IP addresses that are "invalid" for at least one AS over time, marking each entry in our trace with a diamond. The drop in the number of "invalid" addresses on December 20 was due to a single event; see Case Study 3.

Downgrades due to whacked ROAs (Figure 2·2). We show the number of prefix-to-origin AS pairs that downgraded from valid to invalid, and from valid to unknown, for each consecutive entry in our trace (that contained a successful pull of the RPKI). Figure 2·2 shows a value of zero if no downgrades occurred at some date, and a gap if our trace was missing an entry. To put these results in perspective, in January 2014, there were about 20,000 prefix-origin AS pairs authorized by ROAs in the RPKI. Most of the incidents in Figure 2·2 correspond to the whacking of a single

ROA containing multiple prefixes; in many incidents, a new ROA appeared in the RPKI authorizing the prefix(es) in the whacked ROA to some other AS. The most dramatic event (December 20) is discussed in Case Study 3.

2.4 Why accountability is hard

Accountability means that responsibility for a problem with the RPKI can be attributed to an entity (*e.g.,* the issuer of an RC, the subject of a ROA, a disruption on the communication path to an RPKI repository, *etc.*). Accountability means misbehaving parties can be identified (and disciplined by the RPKI community). While accountability is easy for wrongly added ROAs (just blame the RC that issued the ROA in question), attributing blame for whacked ROAs is much complex. We now discuss how ROAs can be whacked, and how adversarial authorities can manipulate the RPKI to avoid being held accountable.

2.4.1 Attacks that disrupt delivery of objects.

A ROA can be whacked by a third party (that does not have access to the keys of an RPKI authority) that disrupts the delivery of information from an RPKI repository to a relying party. In fact, it suffices to corrupt just a single bit in a ROA, causing the ROA to fail cryptographic validation. The RPKI has a mechanism for detecting lost/corrupted information:

Manifests. To provide assurance that no items have been deleted from a publication point, a collision-resistant hash of the contents of every file issued by an RC is listed in a single file called a *manifest* (Huston et al., 2012a, Section 2.1), which is digitally signed by the RC and stored at its publication point. The manifest must be updated whenever an RC issues, modifies, or revokes an object it issued. To prevent replay attacks and the propagation of stale information, manifests are short-lived objects;

they typically expire and are renewed every 24 hours.

Of course, a third party can always corrupt or disrupt the delivery of the manifest itself.

Accountability? Fortunately, these attacks are quite transparent: a relying party can always check that it received all the objects in a manifest, and that the manifest was valid and current. If not the relying party should raise a *missing-information alarm* that attributes blame to the communication path, or to the RPKI authority who issued the manifest (since the authority could have sneakily deleted an object from its publication point but not from its manifest).

Reacting to alarms? A relying party should search for missing objects in its local cache or at other relying parties, and search for unusual routing activity in the IP address space covered in the RC that issued the manifest that triggered the alarm (Austein et al., 2012, Sect 6.5).

2.4.2 How can I whack thee? We count the ways.

Because relying parties download RPKI objects from publication points that are *controlled by their issuer* (Lepinski and Kent, 2012, Section 8), the issuer can manipulate the contents of its publication points any way it likes. An RPKI authority can whack descendant ROAs in various ways, including:

Deleting or corrupting. An authority can delete or corrupt any object in its publication point. If the authority logs appropriate change in its manifest, relying parties will accept the change without complaint (as in Case Study 1). This is in contrast to attacks on object delivery (Section 2.4.1), where relying parties would alarm because of a mismatch between the manifest and the publication point's state.

Overwriting. Each RPKI object is identified by a uniform resource identifier (URI), and an authority may overwrite any RC it issued, so that modified objects can have persistent URIs (thus simplifying operations like certificate renewal and key rollover (Huston et al., 2012b)). An authority can always overwrite an RPKI object with one for a different set of IP addresses (or a different key, *etc.*) and log the appropriate change in its manifest; children of the overwritten RC can be whacked as a result, because they are no longer covered by the RC (or they are signed by the wrong key, *etc.*), as in Case Study 4.

Revoking. Finally, an issuer can revoke any object it issues, because the RPKI also inherits certificate revocation lists (CRL) from the X.509 standard. A CRL is a list, signed by an RPKI authority A of the objects issued by A that A has revoked (Cooper et al., 2008).

Moreover, if an object's ancestor RC gets whacked, the object is whacked by association.

2.4.3 Who's to blame? A few case studies.

Who should we blame when a ROA gets whacked? To gain some insight, we present two incidents from our our trace of the production RPKI that were identified by our detector (Section 2.3.1). These incidents likely resulted from benign errors or normal churn, but we will use them to explain how an authority can manipulate RPKI information in order to avoid be held accountable.

Case Study 3: Stale objects at LACNIC. On December 20, 2013, an error at LACNIC whacked 4217 prefix-to-origin-AS pairs. The day before the incident the RPKI rooted at LACNIC looked very similar to what is shown in Table 2.1. On December 20, the manifests and CRLs issued by all four of LACNIC's intermediate RCs (at depth 2) in our local cache all expired. As a result, all four of the intermediate RC became invalid, thus whacking all of their descendant RCs and ROAs. All objects in subtree rooted at LACNIC became invalid; thus, there were no valid ROAs for any address space allocated by LACNIC, and routes in the impacted LACNIC ROAs downgraded from valid to unknown (Figure 2·2).

Who should be held accountable? This is a classic disruption-of-delivery incident, in which a relying party (*i.e.,* our collector) saw a stale manifest. Thus, we can blame the four intermediate RCs, the communication path to LACNIC's publication point, or our collector.

Case Study 4: Overwritten parent RC. On January 5, 2014, a ROA for (196.6.174.0/23, AS 37688) for a backbone connectivity network in Nigeria was whacked because its parent RC was overwritten. The incident occurred as follows: On January 4, the ROA's parent RC was allocated prefix 196.6.174.0/23. On January 5, the RC was overwritten with an RC with the same key but for an IPv6 prefix 2c0f:f668::/32. The ROA in question (which remained in the publication point) became invalid, because it was no longer covered by its parent. Interestingly, the RC had no valid descendants until January 6, when it issued ROAs covered by the IPv6 prefix to a different AS (AS 37600, in Mauritius).

It is worth considering how an RPKI authority might behave similarly to avoid being blamed for an RPKI manipulation. Specifically, on January 5, it would have been impossible to determine if this action was legitimately requested by a subject of the overwritten RC, or if *issuer* of the RC (AfriNIC) took this action unilaterally, without the agreement or knowledge of the subject. (For this particular incident, we speculate that the latter is true, since the overwritten RC did manage to issue a new valid ROA on January 6.)

2.4.4 Holding an adversary accountable.

In light of Case Study 4, one might be satisfied with a tool that attributed blame to a *pair* of entities in the RPKI, *i.e.,* the subject of an RC and its issuer. While this might suffice for localizing the root causes of common errors and misconfigurations,

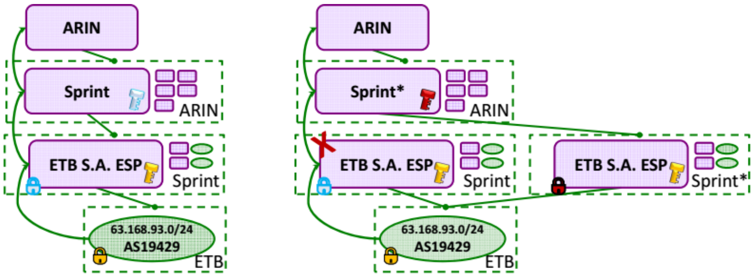


Figure 2·5: Devious overwritings. (l) Before (r) After.

we now show how an adversarial RPKI authority could circumvent such a tool. We first need some background:

RPKI repository structure (Figure 2·5 (l)). This is a simplified schematic view of part of the RPKI in Figure 2·1. Every object issued by an RC is in the issuer's publication point. Sprint's RC has a *child pointer* (the RFCs call this the "Subject Information Access (SIA)" field) that points to the URI of Sprint's publication point. Meanwhile, every RPKI object has a *parent pointer*, which is the URI for its issuer's RC (the "Authority Information Access (AIA)" field). The key belonging to an RC's subject is shown with a key graphic, and the signature of an RC's issuer with a lock graphic.

Case Study 5: Devious overwritings. ARIN whacks the ROA for AS 19429 in the **hypothetical** manner shown in **Figure 2·5(r)**. ARIN overwrites Sprint's RC with a new RC for Sprint*, that has the same IP prefix, but for a different public key (whose secret key is known to ARIN, but not to Sprint) and publication point (controlled by ARIN). As a result, the children of Sprint become invalid, because their parent pointer points to the new Sprint* RC, but they are signed by Sprint's old key. Next, because ARIN knows the secret key in the new Sprint* RC, ARIN covers its tracks by issuing new objects identical to those issued by the original Sprint RC, except that they live in the new publication point (controlled by ARIN). What happens, then, to the ROA for AS 19429 that was originally issued by ETB? The

ROA remains invalid, because its parent pointer points to the invalid RC for ETB, in Sprint's original publication point, rather than new valid RC for ETB, in the publication point controlled by ARIN.

The story above attributes blame to ARIN. However, ETB could just as well be responsible: ETB has access to the key and publication point that houses the whacked ROA, so why does ETB not renew it? One could even hold Sprint accountable: perhaps Sprint wanted to refresh its key and publication point, and asked ARIN to modify its RC accordingly. Thus, we could blame any one of the three parties.

One can construct ever more complicated situations that adversarial RPKI authorities can use to confound accountability mechanisms. We will instead move on to designing a tool that can be used to help repair problems caused by whacked ROAs (Section 2.5), and then propose modifications to the RPKI's architecture to allow for robust accountability (Section 2.6).

2.5 Repairing whacked objects

We now present a tool that allows relying parties to identify RCs that can "repair" whacked ROAs.

Tool design. Our *repair localizer* tool takes in a current RPKI state S_{cur} and previous state S_{prev}. Given a ROA R for a prefix π that was whacked in the transition from S_{prev} to S_{cur} (as identified by our detector, Section 2.3.1), the tool identifies RCs that can issue ROAs for π in S_{cur}. However, since any RC whose resources cover π can do so, and RC allocations are not exclusive, there may be many such RCs. We search for someone who was "responsible" for R in S_{prev}. We therefore look for the lowest surviving ancestor of R covering π.

Looking for this ancestor in S_{cur} is challenging, because S_{prev} and S_{cur} may look completely different due to multiple changes—in particular, because the RPKI has no persistent names for its authorities. For example, it would be nice if any RC for Sprint had the string “Sprint” as its subject name, but RCs “must not attempt to convey the identity of the subject in a descriptive fashion” (Lepinski and Kent, 2012) to avoid “legal and liability concerns” (FCC, 2013). Thus, for reasons discussed in our technical report, we try to find the ancestor RCs by matching on (a) URI, (b) public key, and/or (c) child pointer.

Testing our tool. We tested our tool on whacked-ROA incidents in our trace of the production RPKI (Section 2.3.2), as well as Case Study 5 and other situations that we injected into our models of a fully-deployed RPKI (Section 2.6.8). For brevity, we just mention what the repair localizer did in a few cases:

Case Study 1. The parent of the whacked ROA (a leaf RC in the RPKI rooted at RIPE) was identified.

Case Study 4. The parent of the overwritten RC (an intermediate RC held by AfriNIC) was identified.

Case Study 5. The RC belonging to ETB SA ESP was identified; as discussed in Section 2.4.4, this RC can repair the problem, but may not be responsible for it.

2.6 Fixing the balance of power

We have seen that the RPKI’s design empowers authorities to unilaterally whack their descendants (Section 2.4.2), and that adversarial authorities can evade accountability mechanisms (Section 2.4). We therefore propose restoring the balance of power via modest modifications to the RPKI specifications. We start with our design goals, and then discuss our modifications to the objects issued by the authorities, and how they allow relying parties to verify not only the validity of RPKI objects, but also the validity of authorities’ behavior. We then justify our design by presenting its exact security guarantees, explain why alternative designs would not suffice to provide these guarantees, and estimate its overhead via data-driven analysis.

This section provides an overview of our design; full details and security proofs are in our technical report.

2.6.1 Design goals.

Consent. To correct the power imbalance created by the RPKI, one might consider distributing the role of RPKI authorities; namely, having multiple entities jointly issue objects, as in (Zhou et al., 2002; Cachin and Samar, 2004). However, in a default-deny architecture like the RPKI, there is little sense in distributing the issuance of RPKI objects, because this rarely causes IP prefixes to go offline. Instead, we should distribute the *revocation* of RPKI objects, since revocation can downgrade BGP routes from "valid" (authorized) to "invalid" (denied). We will therefore require that any action that can *remove resources* (IP prefixes) from or *invalidate* RCs must obtain the *consent* of all impacted RCs.[5] This approach limits bureaucratic overhead, since it only involves entities that already have a stake in the action. To simplify the exposition, we will not discuss consent from ROAs; instead we just suppose than any ROA that wishes to be entitled to consent is issued its own covering RC.[6]

Consistency. A relying party must be able to verify that it has the same view of the RPKI as other relying parties, and raise an alarm if it does not. This prevents adversarial RPKI authorities from launching **mirror world attacks**, where they present one view of the RPKI to some relying parties, and a different view to others. (For example, the subject of an RC could shown a view containing the RC, but other

[5] The addition of a ROA can also cause routes to downgrade from "unknown" to "invalid". However, there is no way to obtain consent from the entities originating the "unknown" routes, since they do not participate in the RPKI. (If they did, they would already have ROAs, and their routes would not be "unknown".)

[6] Actually, each ROA's parent issues an X.509 end-entity (EE) certificate for an ephemeral one-time-use key, which is used to sign the ROA message (Lepinski and Kent, 2012, Section 2.3). The ROA and EE cert are stored in a single `.roa` file, so we have treated them as one object. But a ROA could instead consent via its EE cert, instead of asking for its own RC.

relying parties can be shown a view that omits the RC.)

Accountability. We also require that relying parties can verify that consent has been properly obtained, and to raise an alarm if it has not. Alarms hold accountable any authority that failed to properly obtain consent, or that launched a mirror-world attack. Alarms are resolved via out-of-band mechanisms, just like other routing anomalies, and indicate that relying parties should monitor the relevant portion of the IP address space for unusual routing activity (as in Section 2.4.1).

Formal statements of these properties are in Section 2.6.5.

2.6.2 Overview of our design.

Our design introduces `.dead` objects to enable consent (Section 2.6.3). When an authority wants to revoke (or remove resources from) a child RC, the authority must first obtain a signed `.dead` object from the child RC and all its impacted descendant RCs; the authority then revokes the child by publishing the `.dead` object in its publication point, and deleting the child RC.

We provide accountability and consistency using manifests signed by RPKI authorities (Austein et al., 2012) (Section 2.6.3), and alarms raised by relying parties (Section 2.6.4, Appendix A.1). An RPKI authority's manifest *positively attests* to the set of objects the authority issues (Austein et al., 2012). We additionally require consecutive manifests to be hash chained. Relying parties will raise alarms if manifests indicate that an object is revoked without a `.dead`, or if consecutive manifests are inconsistent. This allows us to hold authorities accountable when RCs are improperly revoked/modified, or during mirror world attacks.

The purpose of alarms is to increase transparency by detecting misbehavior, and also to inform relying parties' local routing decisions (Section 2.6.5). As in the current RPKI specifications, we suppose that relying parties use their own local policies to decide how to resolve alarms (Austein et al., 2012, Section 6); alarms could indicate that certain IP prefixes should be monitored for routing anomalies, or that a relying party should revert to an older ("stale") set of RPKI objects that did not raise alarms. Alarms also allow relying parties to distinguish between consensual revocations and disputes between a subject and issuer; for example, if an issuer unilaterally revokes an RC during a dispute, relying parties raise alarms that inform their local policies and could also launch investigations into the nature of the dispute.

2.6.3 Procedures for RPKI authorities.

We now discuss our new `.dead` objects, and our modifications to the RPKI's specification of manifests (Austein et al., 2012).

Consent via `.dead` objects

Consent for revocation. For RC A to revoke a valid child RC B, it needs the *consent* of B and all the RCs that descend from B. Consent is provided to A via a `.dead` object signed by each consenting RC .

`.dead` objects are constructed recursively as follows: Let D be a descendant of B that consents to its own revocation. Before D can sign its own `.dead` object, D first collects `.dead` objects from each of its descendants. D then signs its own `.dead` object, that includes (1) the hash of the `.dead` object issued by each *child* of D, (2) the hash of m_D, the manifest issued by D at the time D signs its own `.dead` object, and (3) the hash of the RC of D. D then provides its own `.dead` object and the `.dead` objects for all its descendants to its issuer C.

At the end of this process, A has received the `.dead` objects from B and all descendants of B. (This recursive collection of `.dead` objects protects A, and any of its descendants, from being falsely accused of revoking a descendant without consent.)RC A then simultaneously (a) deletes RC B, (b) puts the `.dead` object for B and all of descendants of B in the publication point of A, and (c) logs all the `.dead` objects in the updated manifest of A.

Consent for modification. An RC B is "modified" when it is overwritten by a new RC B' (with the same URI). Because many modifications to an RC can whack its descendants, our design only permits modifications to resources or parent pointers (to accommodate key rollover).[7] Consent (via a `.dead` object) is required from the original RC B when the modified RC B' lacks resources in B; A must recursively collect and publish `.dead` objects from B and every valid descendant of B that was whacked due to the modification. Other modifications to B must be accomplished by issuing a fresh RC at a different URI.

Complexity? While this consent mechanism adds complexity, we emphasize that, besides B, only the RCs that become invalid as a result of the modification need to sign a `.dead` object. Some modifications, such as revoking B, will require the consent of all its descendants. Other modifications, such as removing IP prefixes from B, may invalidate only the subset of descendants that overlap with the removed resources. Finally, many modifications have no adverse impact on anyone at all (not even on B) and do not require `.dead` objects (*e.g.,* modifying the parent pointer of an RC, or increasing the set of IP prefixes or AS numbers it certifies). We estimate the number of entities that must be involved in issuing `.dead` objects in Section 2.6.8.

Make before break. Consent should be given and obtained in a "make-before-break" fashion. For B, this means (a) consenting to the modification only *after* it knows that a validly-issued replacement object (with a different URI) exists in the RPKI or (b) knowingly consenting to have its removed resources disappear completely from the RPKI. For A, it means obtaining the needed `.dead` objects ahead of time, so they can be published at the same time that A revokes/modifies B.

Key rollover. Any PKI needs a mechanism for refreshing cryptographic keys.

[7] Other modifications can be accommodated at the cost of complicating the design; we avoid this here.

Adapting the RPKI's key rollover mechanism to our design, while still preserving accountability and consistency, requires some care. Details are in our tech report. Importantly, key rollover only requires consent from the RC whose key was rolled.

Emergencies. In some situations (*e.g.,* key-rollover due to stolen keys, disputes *etc.*), it will be impossible to obtain consent in a timely manner. In this cases, the issuer can just unilaterally revoke (or otherwise modify) the RC; these actions will be visible to relying parties, who will raise alarms per Section 2.6.4.

Manifests as positive attestations

Manifests (Austein et al., 2012) lie at the core of our design. We (a) add information to manifests, and (b) modify the semantics by which they are interpreted. The details follow; readers more interested in our security analysis can jump ahead to Section 2.6.5.

Normative manifests. Manifests *positively attest* to the set of objects issued by an RPKI authority. Because the RPKI is a default-deny architecture, a relying party must know that it has *all* the objects issued by an authority. We therefore make manifests normative; namely, any object not logged in the issuer's current manifest is treated as nonexistent by relying parties. Once manifests become normative, we can simplify other aspects of the RPKI:

1. Only manifests may expire. In the current RPKI, manifest are "updated" when their issuer overwrites them with a fresh manifest with a higher 'manifestNumber' (Austein et al., 2012); manifests therefore short lived objects — most expire and updated with fresh versions within 24 hours. In our design, if a manifest expires before it is updated, then all objects logged in the manifest become "stale", rather than "invalid"; "stale" indicates that up-to-date information is unavailable, but does *not* indicate that the objects logged in the expired manifest have explicitly been revoked.

A stale manifest (or any object logged in a current manifest but not obtained by a relying party) raises a *missing-information alarm* at the relying party (Section 2.6.4).

Thus our design no longer requires expiration dates on ROAs and RCs: any RC/ROA not logged in the current manifest is automatically invalid. We do this to prevent an issuer from issuing short-lived ROAs/RCs in order to circumvent the need to obtain consent.

2. Manifests must log only valid objects. Any issuer that logs an invalid object in its manifest (*e.g.,* an object pointing to the wrong parent, has a prefix that not covered by the issuer's RC, *etc.*) risks the ire of relying parties, who raise alarms per Section 2.6.4. (This is necessary for our consistency mechanisms to work properly; see Counterexample 2 in Section 2.6.7.)

By ensuring that manifests only attest to what is valid, we no longer need CRLs to attest to what is invalid. In fact, RCs/ROAs need not even be signed: signature on a manifest suffices, because the manifest contains the collision-resistant hash of every valid object (as in (Gassko et al., 2000)'s approach).

These changes may seem unorthodox, but we note that the RPKI is different from the usual PKI where relying parties obtain and validate objects one-by-one; instead, all objects in a publication point are downloaded *en masse*, so it suffices just to validate the manifest. All these changes can be implemented without any modification to object formats by having relying parties ignore CRLs, expiration times/signatures on RCs and ROAs.

Reconstructing intermediate states. To prevent mirror world and other attacks (*e.g.,* Counterexample 1 in Section 2.6.7), relying parties must be able to reconstruct states that could have been seen by other relying parties. Thus, issuers must also provide relying parties with "hints" to enable reconstruction of any intermediate manifest/publication-point state between two syncs to a publication point;

alarm	description
missing-information	manifest is stale/missing OR object logged in manifest is missing
bad rollover	RC issued “post-rollover manifest” but performed incorrect key rollover procedure
invalid syntax	RC issued malformed object
child too broad	RC issued object it does not cover
unilateral revocation	RC deleted/modified object without `.dead`
global inconsistency	manifest failed global consistency check

Table 2.3: Alarms

our technical report has details on the small amount of additional information that needs to be maintained at a publication point.

Hash chaining. Manifests are hash-chained: each new manifest includes the hash of the contents (excluding the signature) of the manifest it supersedes. A *horizontal chain* is a sequence of consecutively issued manifests, each superseding the next. For each manifest m, hash chaining defines successor and predecessor manifests of manifest m in the obvious way.

Disambiguating the issuer. If an RC is overwritten, this can introduce some ambiguity as to who issued its manifest; was it issued by the original RC, or its overwritten version? (Counterexample 2 explains how this can complicates consistency checking.) We therefore require every manifest to include the hash of the manifest containing its issuer's RC (“parent manifest”), so that we can definitively determine the resources allocated to an RC at the time it signed its manifest. A *vertical chain* is a sequence of manifests, each containing the hash of the manifest above it and the hash of an RC that signed the manifest below it.

2.6.4 Validation procedures for relying parties.

The validation procedures for relying parties have two purposes: (1) to determine the set of valid RPKI objects, and (2) to raise alarms that hold authorities accountable when they violate procedures in Section 2.6.3.

Local consistency check. A relying party performs this check locally, one pair of consecutive manifests at a time, to validate objects and raise alarms when procedures in Section 2.6.3 are violated; see Appendix A.1.

Global consistency check. To defeat mirror-world attacks, relying parties can confirm that others see the same objects as they do. A trivial, but unwieldy, solution has relying parties check consistency by synchronizing and exchanging their entire local caches.

Our solution dispenses with synchronization and requires the exchange of much less information. We ask only that one party (Alice) is no more than time t_g ("global consistency window") ahead of the other (Bob).

Bob sends Alice the hash h of the contents (excluding the signature) of latest manifest that he *obtained* (that passed cryptographic signature and hash verification) for each publication point in his local cache. For every hash value h received from Bob, Alice checks that $h \in H_A$, where H_A is the set that contains the hash of each manifest that Alice obtained (again, as defined in Appendix A.1), going back for time t_g. The check fails if $h \notin H_A$, and Alice raises a *global-inconsistency alarm* implicating the manifest corresponding to h.

A pairwise interactive protocol is not actually required here; Bob can just post his hash values in a public location for any other relying party to use.

2.6.5 Security analysis.

We now discuss the security properties of our design. We only state theorems here; proofs are in our tech report.

Threat model. We suppose that the relying parties named in Theorems 2.6.1-2.6.3 are honest; everyone else can be arbitrarily malicious, but cannot break the cryptographic primitives (*e.g.,* forge digital signatures, find hash collisions, *etc.*).

RC Successors. The theorem uses the following definition: for an RC R and relying party Alice, let the *immediate successor* of R be (a) an RC that overwrites R, whichever happens first from Alice's point of view. Define the set of *successors* of R inductively as the set containing R and immediate successors for each of its elements.

Valid remains valid. The following theorem tells us that once an honest relying party Alice sees a valid object, the object will remain valid for Alice until the object consents to revocation (or Alice raises an alarm):

Theorem 2.6.1. *Suppose an RC R was valid for a relying party Alice at time t_1. Consider some time $t_2 > t_1$. Then at time t_2, at least one of the following is true:*

1. *a successor of R is valid in the local cache of Alice and has all the resources of R;*
2. *a successor of R is valid in the local cache of Alice, is missing some resources that R had, and Alice observed `.dead` object(s) signed successors of R consenting to the revocation(s) of those resources;*
3. *at or before time t_2, Alice saw a `.dead` object signed by a successor of R, consenting to its revocation;*
4. *at or before time t_2, Alice raised a unilateral revocation alarm in response to a deleted or overwritten certificate; the alarm included a successor of R as a victim, and blamed an ancestor (or a successor of the ancestor) of the deleted or overwritten certificate.*

No mirror worlds. We prove robustness to mirror world attacks; relying parties can be sure that if they see a valid RC in a manifest, and use this manifest in the global consistency check, others who successfully check against them will also see the same valid object:

Theorem 2.6.2. *Suppose the local cache of a relying party Bob contains a valid RC R and a manifest m that was issued by the parent of R. Suppose that R is not marked as stale; thus, the version of R in Bob's local cache is same as the one logged in m. Suppose another relying party Alice performs a global consistency check against Bob; Bob sends the hash of m to Alice, and Alice looks for it in her set H_A of hashed manifests. Suppose the global consistency check does not raise the global-inconsistency alarm for m. Then either*

- *Alice raised an alarm with R as the victim when performing the local consistency check on one of the manifests in H_A, or*
- *at any time t_2 after the global consistency check, at least one of the conditions in Theorem 2.6.1 holds for R and Alice.*

No mirror worlds in the past. Because manifests are hash-chained, a successful global consistency check also implies that parties were consistent in the past. We state the theorem only informally here:

Theorem 2.6.3 (Informal)**.** *Suppose neither Alice nor Bob raised alarms for k manifest updates in a row. Then if Alice passes a global consistency check against Bob, then for every RC R seen as valid by Bob in any of the past k manifests, at least one the conditions specified in Theorem 2.6.1 holds for R and Alice.*

Alarms & accountability. Our theorems apply to situations in which honest relying parties Alice and/or Bob have valid objects and manifests. Sometimes, however, Alice may be unable to find a valid object (because communication is disrupted) or to run a global consistency check because of an invalid manifest. Our design ensures that Alice will also raise alarms in response to these problems. Table 2.3 summarizes these alarms; missing information alarms are raised when a valid object cannot be found, and global inconsistency alarms are raised when the global consistency check fails.

The remaining alarms in Table 2.3 can be raised by Alice during the local consistency check, described in Appendix A.1. These local-inconsistency alarms are for a particular victim object O (*e.g.,* a `.dead` or manifest) and blame the RC R that signed O. The validity of O is defined in the context of other objects (*e.g.,* the validity of a `.dead` depends on its signer's manifest, while the validity of manifest m depends the validity of the manifest that logs the RC that issued m). In our design, O includes this context as hashes of other objects (*e.g.,* a `.dead` includes the hash of its signer's manifest (Section 2.6.3), while a manifest m includes the hash of its parent's manifest

(Section 2.6.3)). Thus, the blame attributed by Alice's local consistency alarm are *provable* if her local cache contains the necessary context for O (*i.e.,* objects whose hashes are consistent with those in O). Because hashes are collision resistant and objects are signed, a third party Bob can take O and its context as sufficient proof that the blamed RC R misbehaved, even if Bob does not trust Alice.

On the other hand, if Alice's local cache does not have the necessary context (*i.e.,* the objects whose hashes are present in O), then a *missing-information alarm* is raised, and the blame attributed by Alice is no longer provable. This is because Alice is missing some information that the blamed RC claims to have, perhaps because the blamed RC is lying, or because Alice and the blamed RC are out of sync due to a race condition, or because Alice and the blamed RC are living in mirror worlds. In these situations, determining which RC is guilty of misbehavior could require a forensic investigation that collects the missing information needed to reconstruct the local caches of the blamed RC (and potentially other RCs) over time. Horizontal and vertical manifest chaining can be used to resolve inconsistencies.

Resolving alarms. Alarms increase transparency. Because alarms have specific victims, relying parties can use their own local policies to make informed decisions on how to resolve them; they may revert to the last-known manifest that raised no alarms, require extra verification of BGP announcements for IP prefixes impacted by the alarm, use other out-of-band mechanisms, *etc.* RPKI authorities holding RCs that are regularly implicated in alarms can be disciplined by the RPKI community (*e.g.,* via mailing lists, blogs, SLAs).

2.6.6 What about all those bad examples?

We consider how some whacked-ROA incidents we discussed earlier would look in our design. In Case Study 5, we would be able to provably implicate ARIN by presenting its manifest from before and after it modified Sprint's RC; the same is true for the

AfriNIC RC that overwrote the RC in Case Study 4. In Case Study 4, AfriNIC could avoid being implicated in a unilateral-revocation alarm by obtaining `.dead` objects from the overwritten RC and its descendants. In Case Study 5, ARIN could avoid being implicated by placing the RC for Sprint* at a fresh URI, and obtaining `.dead` objects from Sprint and its descendants; alternatively, ARIN could use a proper key rollover procedure, which would only require consent from Sprint.

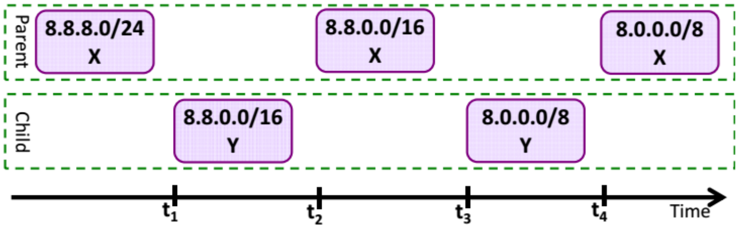
2.6.7 On the necessity of our modifications.

We give intuition for our design via two examples.

Counterexample 1: Not checking intermediate state. Suppose we did not require relying parties to verify the state at every consecutive manifest update. An authority X could exploit this in a mirror world attack that would violate Theorems 2.6.2 and 2.6.3.

At time t_1, X issues an RC Y; at time t_2, X replaces Y with a new valid RC Y' that contains an additional IP prefix; and continues swapping between Y (at odd-numbered times), and Y' (at even-number times). Alice syncs to X's publication point at odd-numbered times, and sees only Y; Bob syncs at even-numbered times and sees RC Y'. But, since Alice does not check full intermediate states (only manifest chains), she does not notice the transition from Y' to Y, and does not realize that X needs to get a `.dead` object from Y'. Thus, Alice and Bob live happily in their mirror worlds.

Counterexample 2: Race conditions. A major challenge we faced was that relying parties can sync to different publication points at different times. This creates race conditions that complicate global consistency:



Suppose an authority X is allocated a small address block. At time t_1, X properly issues a child RC Y that is invalid because Y contains more addresses than its parent X. A time t_2, parent X is overwritten by an RC for those addresses, and its child Y becomes valid. At time t_3, parent X overwrites Y with an RC that is invalid because it contains even more addresses. This process continues as above. Notice that from time t_1 to t_2, and from t_3 to t_4, the manifest of X logs an invalid object. Suppose Alice syncs just after t_1 and just after t_3; she decides that Y is invalid. Bob syncs just after t_2 and just after t_4; he decides that Y is valid. Hence, mirror worlds!

This problem cannot be caught by checking for `.dead` objects for Y, since in Alice's view Y is always invalid, and in Bob's view Y just keeps getting more resources. Nor can it be caught by the global consistency check, because Alice and Bob see the same manifests. Our design eliminates this problem by requiring relying parties to alarm if invalid objects are logged in the manifest; in this example, Alice would raise an alarm. Theorems 2.6.2 and 2.6.3 would be false without this requirement.

2.6.8 Data-driven analysis of our design.

We discuss the impact of our changes on the RPKI.

Less crypto. One immediate improvement is that a single digital signature on the manifest needs to be issued and verified, rather than individual signatures on each RC, ROA, CRL, and manifest (Section 2.6.3). To put this perspective, on January 13, 2014 there were $\approx 10,400$ validly-signed objects in the RPKI; our changes require the validation of only $\approx 2,800$ manifests.

No renewals. RCs/ROAs do not expire in our design (Section 2.6.3); hence recipients of resources are no longer dependent on their issuers for routine renewals.

Mandated interaction for obtaining consent. However, issuers are dependent on recipients to provide consent (via `.dead` objects) for revocations and certain modifications (Section 2.6.3). To find out how often those events happen, we use our trace of the production RPKI for 2013/10/23–2014/01/21 (Section 2.3.2). The largest event we observed was in mid-November 2013, when RIPE removed 3,336 RCs/ROAs and issued new ones with new parent/child pointers and public keys, as part of repository restructuring. Our design would require RIPE to obtain `.dead` objects from all of them, which seems to impose a large burden on RIPE. However, unless RIPE holds the secret keys of its descendants (in which case it can just issue `.dead` objects by itself), interaction is needed even if `.dead` objects were not required, because RIPE's descendant RCs would need to reissue their objects in new publication points.

Besides this event, we saw 4,443 instances of modified/revoked RCs/ROAs. Of these, 3,569 (80%) were renewals, that are not needed in our design. Meanwhile, at most 230 (5%) would need a `.dead` object.

How many parties need to consent? Next, we consider how many parties need to be involved in signing a `.dead` when an RC is revoked. Our estimates, made from the production RPKI and a model of the RPKI, suggest that we do not require many `.dead` objects *on average*. The details of our estimates follow:

1. Production RPKI. **Table 2.1** shows the structure of the production RPKI on January 13, 2014. Suppose that ROAs were able to sign off on `.dead` objects (*e.g.,* because they had requested their own covering RCs, or via the mechanism in footnote 6). How many entities would need to sign a `.dead` object if we wanted to revoke a leaf RC in Table 2.1? We use ASes as a proxy for entities, and estimate this by counting the number of ASes in ROAs issued by each leaf RC; the results for the

# ASes	1	2	3	4	5	6-10	10-30	98
RIPE	678	122	51	13	12	30	8	1
LACNIC	123	20	9	2	1	2	0	0
APNIC	26	8	2	0	2	0	0	0
ARIN	30	5	4	4	3	0	0	0
AfriNIC	9	2	1	1	0	0	0	0

Table 2.4: # of leaf RCs issuing ROAs for X ASes on January 13, 2014; X is in the top row.

production RPKI are in **Table 2.4**, which shows that on average, only 1.6 ASes need to consent to revoking a leaf RC, and that 93% of leaf RCs can be revoked with the consent of no more than 3 ASes.

2. Model. Because the RPKI is far from fully deployed, we also created a model for a future (full) deployment of the RPKI using routing data for the week starting 2012/05/06. In our model, the RIRs sit at the highest layer of the hierarchy; they issues RCs to "direct allocations," *i.e.,* IP prefixes directly allocated by the RIRs (*e.g.,* Sprint in Figure 2·1). We obtained these top two layers of the hierarchy by using files retrieved from the FTP site of each RIR. Our model omits intermediate RCs (since the are just held by the RIRs). To model the ROAs descended from each direct allocation, we extracted (prefix, origin AS)-tuples from BGP feeds (Views, 2015; RIPE, 2015) for the week starting 2012/05/06. (Any tuple not covered by a directly-allocated prefix was discarded as a bogon.)[8]We grouped tuples by AS and found that, on average, each direct allocation issues ROAs for only 1.5 ASes; the distribution is in **Table 2.5**.

With great power comes great responsibility. Tables 2.4, 2.5 indicate that there are a small percentage of outlier RCs that issue ROAs for many (even hundreds!) of ASes. (In our model, out of 116,357 total direct-allocation RCs, 26 (0.02%) have

[8] Figure 2·1 is derived from this model. We built a subtree of RCs below each direct allocation RC, with one RC for every prefix with (prefix, AS)-tuple in the BGP feeds. The ancestor relationship corresponds to the cover relationship for prefixes, and we collapsed parent-child pairs of RCs generated for the same AS, ASes in the same organization per (CAIDA, 2014a) or when if child was a "stub AS" per (Wang et al., 2013).

# ASes	1-10	11-30	31-100	100-200	200 − 1073
	115,605	594	132	15	11

Table 2.5: Similar to the distribution in Table 2.4, except for direct-allocation RCs in our model.

more than 100 ASes and 221 (0.18%) have more than 25 ASes.) Revoking these outliers requires a large number of `.dead` objects. However, we consider this to be a feature, not a bug: these RCs can impact routing to a large number of ASes, so revoking them should not be easy. Moveover, outright revocation may not be necessary if the goal is simply to change the resources given to them slightly, because we provide mechanisms for removing resources from RC that only require `.dead` objects from descendants that become invalid as a result (Section 2.6.3).

2.7 Related work

Routing security. The RPKI is a realization of a series of routing security proposals (Kent et al., 2000; White, ired; Aiello et al., 2003; Osterweil et al., 2011) for origin authentication (Section 2.2.2). A number of works (Ballani et al., 2007; Goldberg et al., 2010; Lychev et al., 2013) argue that origin authentication can significantly improve routing security. The RPKI is also the first step towards a comprehensive solution for securing the current routing system with BGPSEC (Lepinski, 2012) or other proposals surveyed in (Butler et al., 2010; Huston et al., 2011). See also (Zhang et al., 2011) for clean-slate architectures that are robust to routing problems.

Censorship. Censorship is known to occur at all layers of the Internet's architecture; see *e.g.,* (Murdoch and Anderson, 2008) for an overview. There is already evidence (Rensys Blog, 2008; Qiu et al., 2011; Anderson, 2012) of routing-based censorship with BGP; we explored the risk that the RPKI could also be used for this purpose.

PKI design. Our modified RPKI architecture (Section 2.6) is related to an long line of work on public key infrastructure (PKI) design, culminating in recent efforts to harden the web PKI (NIST, 2013b). The idea of validating the state of the RPKI over time is related to certificate pinning (Evans et al., 2013), and the idea of hash-chaining manifests is related to append-only-logs (Schneier and Kelsey, 1997) (or see (Crosby and Wallach, 2009) and references therein) and certificate transparency (Laurie et al., 2013). While these works suppose that a single logger tracks a stream of ordered events, we have to deal with the race conditions (Counterexample 2) that result from allowing individual RPKI authorities to maintain their own logs (*i.e.,* manifests). The idea of correcting the balance of power in a hierarchical system has also appeared in work on distributing certificate authorities (Zhou et al., 2002) and centralized systems like the DNS (Cachin and Samar, 2004; namecoin, 2015; Ramasubramanian and Sirer, 2004); these works distribute the *issuance* of objects, but we only distribute *revocation* (since revocation can harm IP prefix reachability).

The RPKI. Research on the RPKI covers measurement (Wählisch et al., 2012; Osterweil et al., 2012) and policy questions (Mueller and Kuerbis., 2011; Communications Security, Reliability and Interoperability Council III (CSRIC), 2011; The President's National Security Telecommunications Advisory Committee, 2011; Mueller et al., 2013). An earlier HotNets paper (Cooper et al., 2013) discusses the threat of misbehaving RPKI authorities, but does not provide solutions to any the threats it discusses; portions of our security audit (Sections 2.2, 2.4.1, 2.4.2) overlap with (Cooper et al., 2013), but the remainder of this chapter presents new results.

Concurrently to our work, there have been efforts within the IETF to harden the RPKI against authorities that abuse their power (Bush, 2013; Kent and Mandelberg, 2013; Bush, 2012a). Kent *et al.* (Kent and Mandelberg, 2013) considers the threat of whacked ROAs and of "competing ROAs". A new ROA "competes" with an existing ROA if it contains prefixes covered by the older ROA; a competing ROA is a threat if BGP is attacked, since the AS in the competing ROA can perform a (sub)prefix hijack on the AS in the older ROA. Our architecture (Section 2.6) ignores this threat because we focus on the risk that the RPKI can take IP prefixes offline in the *absence* of an attack on BGP. Moreover, any authority that issues a competing ROA and then attacks BGP can be held accountable; the competing ROA itself is non-repudiable evidence of the attack. Both (Kent and Mandelberg, 2013) and our design defend against whacked ROAs by comparing the state of the RPKI over time, but we also detect mirror world attacks and have exact security guarantees.

Systems have also been developed to monitor the RPKI (Spider, 2015; LACNIC, 2013; NIST, 2013a; RIPE, 2013; Surfnet, 2013); most use a snapshot of ROAs from the RPKI to determine the validity state of routes in publicly-available BGP route collectors (Views, 2015; RIPE, 2015). Our detector (Section 2.3.1) is complementary because we detect when any *change* to the RPKI alters the validity state of *all* possible routes, not just the ones visible from a particular BGP vantage point at a specific time; it can therefore be used as an alert system (especially when RPKI deployment reaches steady state), even if a particular vantage point does not obtain a complete view of all routes announced in BGP. Our tools also provide a new way of visualizing downgrade events (Section 2.3.1) and to repair the problems that result (Section 2.5).

2.8 Conclusion

We have explored a number of techniques to harden the RPKI against the risk of IP prefix takedowns. We have built tools for detecting and reacting to takedowns within the existing RPKI specifications. We have also proposed changes to the specifications that (1) entitle parties to *consent* to revocations of their IP address space, and guarantee that (2) a misbehaving RPKI authority can be held *accountable* for its actions and that (3) relying parties obtain a *consistent* view of information in the

RPKI. Given the security improvements promised by the RPKI (Lychev et al., 2013; Goldberg et al., 2010; Ballani et al., 2007), we hope our work will catalyze further efforts to harden the RPKI against abusive authorities.

Chapter 3

Eclipse Attacks on Bitcoin's P2P Network

This chapter uses research from (Heilman et al., 2015a) which was written in collaboration with Alison Kendler, Aviv Zohar and Sharon Goldberg.

3.1 Introduction

While cryptocurrency has been studied since the 1980s (Chaum, 1983a; Brands, 1993; Camenisch et al., 2005), bitcoin is the first to see widespread adoption.A key reason for bitcoin's success is its baked-in decentralization. Instead of using a central bank to regulate currency, bitcoin uses a decentralized network of nodes that use computational proofs-of-work to reach consensus on a distributed public ledger of transactions, *aka.,* the *blockchain*. Satoshi Nakamoto (Nakamoto, 2008) argues that bitcoin is secure against attackers that seek to shift the blockchain to an inconsistent/incorrect state, as long as these attackers control less than half of the computational power in the network. But underlying this security analysis is the crucial assumption of *perfect information*; namely, that all members of the bitcoin ecosystem can observe the proofs-of-work done by their peers.

While the last few years have seen extensive research into the security of bitcoin's computational proof-of-work protocol *e.g.,* (Nakamoto, 2008; Eyal and Sirer, 2014; Shomer, 2014; Bahack, 2013; Kroll et al., 2013; Johnson et al., 2014; Courtois and Bahack, 2014; Eyal, 2014; Laszka et al., 2015; Rosenfeld, 2014), less attention has been paid to the peer-to-peer network used to broadcast information between bitcoin nodes (see Section 3.8). The bitcoin peer-to-peer network, which is bundled into the core bitcoind implementation, *aka.,* the Satoshi client, is designed to be open, decentralized, and independent of a public-key infrastructure. As such, cryptographic authentication between peers is not used, and nodes are identified by their IP addresses (Section 3.2). Each node uses a randomized protocol to select eight peers with which it forms long-lived *outgoing connections*, and to propagate and store addresses of other potential peers in the network. Nodes with public IPs also accept up to 117 *unsolicited incoming connections* from any IP address. Nodes exchange views of the state of the blockchain with their incoming and outgoing peers.

Eclipse attacks. This openness, however, also makes it possible for adversarial nodes to join and attack the peer-to-peer network. In this chapter, we present and quantify the resources required for *eclipse attacks* on nodes with public IPs running bitcoind version 0.9.3. In an eclipse attack (Castro et al., 2002; Sit and Morris, 2002; Singh et al., 2006), the attacker monopolizes all of the victim's incoming and outgoing connections, thus isolating the victim from the rest of its peers in the network. The attacker can then filter the victim's view of the blockchain, force the victim to waste compute power on obsolete views of the blockchain, or coopt the victim's compute power for its own nefarious purposes (Section 3.1.1). We present *off-path* attacks, where the attacker controls endhosts, but not key network infrastructure between the victim and the rest of the bitcoin network. Our attack involves rapidly and repeatedly forming unsolicited incoming connections to the victim from a set of endhosts at attacker-controlled IP addresses, sending bogus network information, and waiting until the victim restarts (Section 3.3). With high probability, the victim then forms all eight of its outgoing connections to attacker-controlled addresses, and the attacker also monopolizes the victim's 117 incoming connections.

Our eclipse attack uses extremely low-rate TCP connections, so the main challenge for the attacker is to obtain a sufficient number of IP addresses (Section 3.4). We consider two attack types: (1) infrastructure attacks, modeling the threat of an ISP, company, or nation-state that holds several *contiguous* IP address blocks and seeks to subvert bitcoin by attacking its peer-to-peer network, and (2) botnet attacks, launched by bots with addresses in *diverse* IP address ranges. We use probabilistic analysis, (Section 3.4) measurements (Section 3.5), and experiments on our own live bitcoin nodes (Section 3.6) to find that while botnet attacks require far fewer IP addresses, there are hundreds of organizations that have sufficient IP resources to launch eclipse attacks (Section 3.4.2). For example, we show how an infrastructure attacker with 32 distinct /24 IP address blocks (8192 address total), or a botnet of 4600 bots, can always eclipse a victim with at least 85% probability; this is independent of the number of nodes in the network. Moreover, 400 bots sufficed in tests on our live bitcoin nodes. To put this in context, if 8192 attack nodes joined today's network (containing $\approx$ 7200 public-IP nodes (Bitnode, 2014)) and honestly followed the peer-to-peer protocol, they could eclipse a target with probability about $(\frac{8192}{7200+8192})^8 =$ 0.6%.

Our attack is only for nodes with public IPs; nodes with private IPs may be affected if all of their outgoing connections are to eclipsed public-IP nodes.

Countermeasures. Large miners, merchant clients and online wallets have been known to modify bitcoin's networking code to reduce the risk of network-based attacks. Two countermeasures are typically recommended (btcwiki, 2014b): (1) disabling incoming connections, and (2) choosing 'specific' outgoing connections to well-connected peers or known miners (*i.e.,* use whitelists). However, there are several problems with scaling this to the full bitcoin network. First, if incoming connections are banned, how do new nodes join the network? Second, how does one decide which 'specific' peers to connect to? Should bitcoin nodes form a private network? If so, how do they ensure compute power is sufficiently decentralized to prevent mining attacks?

Indeed, if bitcoin is to live up to its promise as an open and decentralized cryptocurrency, we believe its peer-to-peer network should be open and decentralized as well. Thus, our next contribution is a set of countermeasures that preserve openness by allowing unsolicited incoming connections, while raising the bar for eclipse attacks (Section 3.7). Today, an attacker with enough addresses can eclipse *any* victim that accepts incoming connections and then restarts. Our countermeasures ensure that, with high probability, if a victim stores enough legitimate addresses that accept incoming connections, then the victim be cannot eclipsed *regardless of the number of IP addresses the attacker controls.* Eight of our countermeasures are deployed in bitcoin project as of the year 2020 in bitcoind v0.21. In Table 1.1 we provide a table of bitcoin's deployment of our countermeasures.

3.1.1 Implications of eclipse attacks

Apart from disrupting the bitcoin network or selectively filtering a victim's view of the blockchain, eclipse attacks are a useful building block for other attacks.

Engineering block races. A block race occurs when two miners discover blocks at the same time; one block will become part of the blockchain, while the other "orphan block" will be ignored, yielding no mining rewards for the miner that discovered it. An attacker that eclipses many miners can engineer block races by hording blocks discovered by eclipsed miners, and releasing blocks to both the eclipsed and non-eclipsed miners once a competing block has been found. Thus, the eclipsed miners waste effort on orphan blocks.

Splitting mining power. Eclipsing an x-fraction of miners eliminates their mining power from the rest of the network, making it easier to launch mining attacks (*e.g.,* the 51% attack (Nakamoto, 2008)). To hide the change in mining power under natural variations (Bitcoin Wisdom, 2015), miners could be eclipsed gradually or intermittently.

Selfish mining. With selfish mining (Eyal and Sirer, 2014; Shomer, 2014; Bahack, 2013; Courtois and Bahack, 2014), the attacker strategically withholds blocks to win more than its fair share of mining rewards. The attack's success is parameterized by two values: α, the ratio of mining power controlled by the attacker, and γ, the ratio of honest mining power that will mine on the attacker's blocks during a block race. If γ is large, then α can be small. By eclipsing miners, the attacker increases γ, and thus decreases α so that selfish mining is easier. To do this, the attacker drops any blocks discovered by eclipsed miners that compete with the blocks discovered by the selfish miners. Next, the attacker increases γ by feeding only the selfish miner's view of the blockchain to the eclipsed miner; this coopts the eclipsed miner's compute power, using it to mine on the selfish-miner's blockchain.

Attacks on miners can harm the entire bitcoin ecosystem; mining pools are also vulnerable if their gateways to the public bitcoin network can be eclipsed. Eclipsing can also be used for double-spend attacks on non-miners, where the attacker spends some bitcoins multiple times:

0-confirmation double spend. In a 0-confirmation transaction, a customer pays a transaction to a merchant who releases goods to the customer *before* seeing a block confirmation *i.e.,* seeing the transaction in the blockchain (Bitcoin Wiki, 2015). These transactions are used when it is inappropriate to wait the 5-10 minutes typically needed to for a block confirmation (blockchain.io, 2015), *e.g.,* in retail point-of-sale systems like BitPay (bitpay, 2014), or online gambling sites like Betcoin (RoadTrain, 2013). To launch a double-spend attack against the merchant (Karame et al., 2012), the attacker eclipses the merchant's bitcoin node, sends the merchant a transaction T for goods, and sends transaction T' double-spending those bitcoins to the rest of the network. The merchant releases the goods to the attacker, but since the attacker controls all of the merchant's connections, the merchant cannot tell the rest of the network about T, which meanwhile confirms T'. The attacker thus obtains the goods without paying. 0-confirmation double-spends have occurred in the wild (RoadTrain, 2013). This attack is as effective as a Finney attack (Finney, 2011), but uses eclipsing instead of mining power.

N-confirmation double spend. If the attacker has eclipsed an x-fraction of miners, it can also launch N-confirmation double-spending attacks on an eclipsed merchant. In an N-confirmation transaction, a merchant releases goods only after the transaction is confirmed in a block of depth $N-1$ in the blockchain (Bitcoin Wiki, 2015). The attacker sends its transaction to the eclipsed miners, who incorporate it into their (obsolete) view of the blockchain. The attacker then shows this view of blockchain to the eclipsed merchant, receives the goods, and sends both the merchant and eclipsed miners the (non-obsolete) view of blockchain from the non-eclipsed miners. The eclipsed miners' blockchain is orphaned, and the attacker obtains goods without paying. This is similar to an attack launched by a mining pool (mmitech, 2013), but our attacker eclipses miners instead of using his own mining power.

Other attacks exist, *e.g.,* a transaction hiding attack on nodes running in SPV mode (Biryukov et al., 2014a).

3.2 Bitcoin's Peer-to-Peer Network

We now describe bitcoin's peer-to-peer network, based on bitcoind version 0.9.3, the most current release from 9/27/2014 to 2/16/2015, whose networking code was largely unchanged since 2013. This client was originally written by Satoshi Nakamoto, and has near universal market share for public-IP nodes (97% of public-IP nodes according to Bitnode.io on 2/11/2015 (Bitnode, 2014)).

Peers in the bitcoin network are identified by their IP addresses. A node with a public IP can initiate up to *eight outgoing connections* with other bitcoin nodes, and accept up to 117 *incoming connections*.[1] A node with a private IP only initiates eight outgoing connections. Connections are over TCP. Nodes only propagate and store public IPs; a node can determine if its peer has a public IP by comparing the IP packet header with the bitcoin `VERSION` message. A node can also connect via Tor; we do not study this, see (Biryukov and Pustogarov, 2014; Biryukov et al., 2014a) instead. We now describe how nodes propagate and store network information, and how they select outgoing connections.

3.2.1 Propagating network information

Network information propagates through the bitcoin network via DNS seeders and `ADDR` messages.

DNS seeders. A DNS seeder is a server that responds to DNS queries from bitcoin nodes with a (not cryptographically-authenticated) list of IP addresses for bitcoin nodes. The seeder obtains these addresses by periodically crawling the bitcoin network. The bitcoin network has six seeders which are queried in two cases only. The first when a new node joins the network for the first time; it tries to connect to the seeders to get a list of active IPs, and otherwise fails over to a hardcoded list of about 600 IP addresses. The second is when an existing node restarts and reconnects to new peers; here, the seeder is queried only if 11 seconds have elapsed since the node began attempting to establish connections and the node has less than two outgoing connections.

`ADDR` messages. `ADDR` messages, containing up to 1000 IP address and their timestamps, are used to obtain network information from peers. Nodes accept unsolicited

[1] This is a configurable. Our analysis only assumes that nodes have 8 outgoing connections, which was confirmed by (Miller et al., 2015)'s measurements.

ADDR messages. An ADDR message is solicited *only* upon establishing a outgoing connection with a peer; the peer responds with up to three ADDR message each containing up to 1000 addresses randomly selected from its tables. Nodes push ADDR messages to peers in two cases. Each day, a node sends its own IP address in a ADDR message to each peer. Also, when a node receives an ADDR message with no more than 10 addresses, it forwards the ADDR message to two randomly-selected connected peers.

3.2.2 Storing network information

Public IPs are stored in a node's tried and new tables. Tables are stored on disk and persist when a node restarts.

The tried table. The tried table consists of 64 *buckets*, each of which can store up to 64 unique addresses for peers to whom the node has successfully established an incoming or outgoing connection. Along with each stored peer's address, the node keeps the timestamp for the most recent successful connection to this peer.

Each peer's address is mapped to a bucket in tried by taking the hash of the peer's (a) IP address and (b) *group*, where the group defined is the /16 IPv4 prefix containing the peer's IP address.A bucket is selected as follows:

```
SK = random value chosen when node is born.
IP = the peer's IP address and port number.
Group = the peer's group

i = Hash( SK, IP ) % 4
Bucket = Hash( SK, Group, i ) % 64
return Bucket
```

Thus, every IP address maps to a single bucket in tried, and each group maps to up to four buckets.

When a node successfully connects to a peer, the peer's address is inserted into the appropriate tried bucket. If the bucket is full (*i.e.*, contains 64 addresses), then *bitcoin eviction* is used: four addresses are randomly selected from the bucket, and the oldest is (1) replaced by the new peer's address in `tried`, and then (2) inserted into the `new` table. If the peer's address is already present in the bucket, the timestamp associated with the peer's address is updated. The timestamp is also updated when an actively connected peer sends a `VERSION`, `ADDR`, `INVENTORY`, `GETDATA` or `PING` message and more than 20 minutes elapsed since the last update.

The `new` table. The `new` table consists of 256 buckets, each of which can hold up 64 addresses for peers to whom the node has not yet initiated a successful connection. A node populates the `new` table with information learned from the DNS seeders, or from `ADDR` messages.

Every address a inserted in `new` belongs to (1) a *group*, defined in our description of the `tried` table, and (2) a *source group*, the group the contains the IP address of the connected peer or DNS seeder from which the node learned address a. The bucket is selected as follows:

```
SK = random value chosen when node is born.
Group     = /16 containing IP to be inserted.
Src_Group = /16 containing IP of peer sending IP.

i = Hash( SK, Src_Group, Group ) % 32
Bucket = Hash( SK, Src_Group, i) % 256
return Bucket
```

Each (*group*, *source group*) pair hashes to a single `new` bucket, while each *group* selects up to 32 buckets in `new`. Each bucket holds unique addresses. If a bucket is full, then a function called `isTerrible` is run over all 64 addresses in the bucket; if any one of the addresses is terrible, in that it is (a) more than 30 days old, or (b) has had too many failed connection attempts, then the terrible address is evicted in favor of the new address; otherwise, *bitcoin eviction* is used with the small change that the evicted address is discarded.

3.2.3 Selecting peers

New outgoing connections are selected if a node restarts or if an outgoing connection is dropped by the network. A bitcoin node never deliberately drops a connection, except when a blacklisting condition is met (*e.g.,* the peer sends `ADDR` messages that are too large).

A node with $\omega \in [0,7]$ outgoing connections selects the $\omega + 1^{th}$ connection as follows:

(1) Decide whether to select from `tried` or `new`, where

$$\Pr[\text{Select from } \texttt{tried}] = \frac{\sqrt{\rho}(9-\omega)}{(\omega+1)+\sqrt{\rho}(9-\omega)} \tag{3.1}$$

and ρ is the ratio between the number of addresses stored in `tried` and the number of addresses stored in `new`.

(2) Select a random address from the table, with a bias towards addresses with fresher timestamps: (i) Choose a random non-empty bucket in the table. (ii) Choose a random position in that bucket. (ii) If there is an address at that position, return the address with probability

$$p(r,\tau) = \min(1, \tfrac{1.2^r}{1+\tau}) \tag{3.2}$$

else, reject the address and return to (i). The acceptance probability $p(r,\tau)$ is a function of r, the number of addresses that have been rejected so far, and τ, the difference between the address's timestamp and the current time in measured in ten minute increments.[2]

(3) Connect to the address. If connection fails, go to (1).

[2] The algorithm also considers the number of failed connections to this address; we omit this because it does not affect our analysis.

3.3 The Eclipse Attack

Our attack is for a victim with a public IP. Our attacker (1) populates the `tried` table with addresses for its attack nodes, and (2) overwrites addresses in the `new` table with "trash" IP addresses that are not part of the bitcoin network. The "trash" addresses are unallocated (*e.g.,* listed as "available" by (RIPE, 2015)) or as "reserved for future use" by (IANA, 2015) (*e.g.,* 252.0.0.0/8). We fill `new` with "trash" because, unlike attacker addresses, "trash" is not a scarce resource. The attack continues until (3) the victim node restarts and chooses new outgoing connections from the `tried` and `new` tables in its persistant storage (Section 3.2.3). With high probability, the victim establishes all eight outgoing connections to attacker addresses; all eight addresses will be from `tried`, since the victim cannot connect to the "trash" in `new`. Finally, the attacker (5) occupies the victim's remaining 117 incoming connections. We now detail each step of our attack.

3.3.1 Populating `tried` and `new`

The attacker exploits the following to fill `tried` and `new`:

1. Addresses from unsolicited incoming connections are stored in the `tried` table; thus, the attacker can insert an address into the victim's `tried` table simply by connecting to the victim from that address. Moreover, the *bitcoin eviction* discipline means that the attacker's fresher addresses are likely to evict any older legitimate addresses stored in the `tried` table (Section 3.2.2).

2. A node accepts unsolicited `ADDR` messages; these addresses are inserted directly into the `new` table without testing their connectivity (Section 3.2.2). Thus, when our attacker connects to the victim from an adversarial address, it can also send `ADDR` messages with 1000 "trash" addresses. Eventually, the trash overwrites all legitimate addresses in `new`. We use "trash" because we do not want to waste our IP address

resources on overwriting `new`.

3. Nodes only rarely solicit network information from peers and DNS seeders (Section 3.2.1). Thus, while the attacker overwrites the victim's `tried` and `new` tables, the victim almost never counteracts the flood of adversarial information by querying legitimate peers or seeders.

3.3.2 Restarting the victim

Our attack requires the victim to restart so it can connect to adversarial addresses. There are several reasons why a bitcoin node could restart, including ISP outages, power failures, and upgrades, failures or attacks on the host OS; indeed, (Biryukov et al., 2014a) found that a node with a public IP has a 25% chance of going offline after 10 hours. Another predictable reason to restart is a software update; on 1/10/2014, for example, bitnodes.io saw 942 nodes running Satoshi client version 0.9.3, and by 29/12/2014, that number had risen to 3018 nodes, corresponding to over 2000 restarts. Since updating is often *not* optional, especially when it corresponds to critical security issues; 2013 saw three such bitcoin upgrades, and the heartbleed bug (OpenSSL, 2014) caused one in 2014. Also, since the community needs to be notified about an upgrade in advance, the attacker could watch for notifications and then commence its attack (btcwiki, 2014a). Restarts can also be deliberately elicited via DDoS (King, 2014; Vasek et al., 2014), memory exhaustion (Biryukov et al., 2014a), or packets-of-death (which have been found for bitcoind (Andresen, 2014a; EvilKnievel, 2015)). The bottom line is that the security of the peer-to-peer network should not rely on 100% node uptime.

3.3.3 Selecting outgoing connections

Our attack succeeds if, upon restart, the victim makes all its outgoing connections to attacker addresses. To do this, we exploit the bias towards selecting addresses with fresh timestamps from `tried`; by investing extra time into the attack, our attacker ensures its addresses are fresh, while all legitimate addresses become increasingly stale. We analyze this with few simple assumptions:

1. An f-fraction of the addresses in the victim's `tried` table are controlled by the adversary and the remaining $1 - f$-fraction are legitimate. (Section 3.4 analyzes how many addresses the adversary therefore must control.)

2. *All* addresses in `new` are "trash"; all connections to addresses in `new` fail, and the victim is forced to connect to addresses from `tried` (Section 3.2.3).

3. The attack proceeds in *rounds*, and repeats each round until the moment that the victim restarts. During a single round, the attacker connects to the victim from each of its adversarial IP addresses. A round takes time τ_a, so all adversarial addresses in `tried` are younger than τ_a.

4. An f'-fraction addresses in `tried` are actively connected to the victim before the victim restarts. The timestamps on these legitimate addresses are updated every 20 minute or more (Section 3.2.2). We assume these timestamps are fresh (*i.e.,* $\tau = 0$) when the victim restarts; this is the worst case for the attacker.

5. The *time invested in the attack* τ_ℓ is the time elapsed from the moment the adversary starts the attack, until the victim restarts. If the victim did not obtain new legitimate network information during of the attack, then, excluding the f'-fraction described above, the legitimate addresses in `tried` are older than τ_ℓ.

Success probability. If the adversary owns an f-fraction of the addresses in `tried`, the probability that an adversarial address is accepted on the first try is $p(1, \tau_a) \cdot f$ where $p(1, \tau_a)$ is as in equation (3.2); here we use the fact that the adversary's addresses are no older than τ_a, the length of the round. If $r - 1$ addresses were rejected during this attempt to select an address from `tried`, then the probability that an adversarial address is accepted on the r^{th} try is bounded by

$$p(r,\tau_a)\cdot f\prod_{i=1}^{r-1} g(i,f,f',\tau_a,\tau_\ell)$$

where

$$\begin{aligned} g(i,f,f',\tau_a,\tau_\ell) = {} & (1-p(i,\tau_a))\cdot f + (1-p(i,0))\cdot f' \\ & + (1-p(i,\tau_\ell))\cdot(1-f-f') \end{aligned}$$

is the probability that an address was rejected on the i^{th} try given that it was also rejected on the $i-1^{th}$ try. An adversarial address is thus accepted with probability

$$q(f,f',\tau_a,\tau_\ell) = \quad \sum_{r=1}^{\infty} p(r,\tau_a)\cdot f\prod_{i=1}^{r-1} g(i,f,f',\tau_a,\tau_\ell) \tag{3.3}$$

and the victim is eclipsed if all eight outgoing connections are to adversarial addresses, which happens with probability $q(f,f',\tau_a,\tau_\ell)^8$. Figure 3·1 plots $q(f,f',\tau_a,\tau_\ell)^8$ vs f for $\tau_a = 27$ minutes and different choices of τ_ℓ; we assume that $f' = \frac{8}{64\times 64}$, which corresponds to a full `tried` table containing eight addresses that are actively connected before the victim restarts.

Random selection. Figure 3·1 also shows success probability if addresses were just selected uniformly at random from each table. We do this by plotting f^8 vs f. Without random selection, the adversary has a 90% success probability even if it only fills $f = 72\%$ of `tried`, as long as it attacks for $\tau_\ell = 48$ hours with $\tau_a = 27$ minute rounds. With random selection, 90% success probability requires $f = 98.7\%$ of `tried` to be attacker addresses.

3.3.4 Monopolizing the eclipsed victim

Figure 3·1 assumes that the victim has exactly eight *outgoing connections*; all we require in terms of *incoming connections* is that the victim has a few open slots to

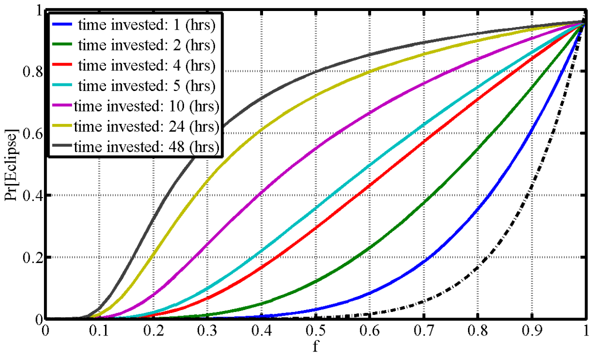


Figure 3·1: Probability of eclipsing a node $q(f, f', \tau_a, \tau_\ell)^8$ (equation (3.3)) vs f the fraction of adversarial addresses in `tried`, for different values of time invested in the attack τ_ℓ. Round length is $\tau_a = 27$ minutes, and $f' = \frac{8}{64\times64}$. The dotted line shows the probability of eclipsing a node if random selection is used instead.

accept incoming TCP connections from the attacker.

While it is often assumed that the number of TCP connections a computer can make is limited by the OS or the number of source ports, this applies only when OS-provided TCP sockets are used; a dedicated attacker can open an arbitrary number of TCP connections using a custom TCP stack. A custom TCP stack (see *e.g.,* zmap (Durumeric et al., 2013)) requires minimal CPU and memory, and is typically bottlenecked only by bandwidth, and the bandwidth cost of our attack is minimal:

Attack connections. To fill the `tried` table, our attacker repeatedly connects to the victim from each of its addresses. Each connection consists of a TCP handshake, bitcoin `VERSION` message, and then disconnection via TCP RST; this costs 371 bytes upstream and 377 bytes downstream. Some attack connections also send one `ADDR` message containing 1000 addresses; these `ADDR` messages cost 120087 bytes upstream and 437 bytes downstream including TCP ACKs.

Monopolizing connections. If that attack succeeds, the victim has eight outgoing connections to the attack nodes, and the attacker must occupy the victim's remaining incoming connections. To prevent others from connecting to the victim, these TCP connections could be maintained for 30 days, at which point the victim's address is `terrible` and forgotten by the network. While bitcoin supports block inventory requests and the sending of blocks and transactions, this consumes significant bandwidth; our attacker thus does not to respond to inventory requests. As such, setting up each TCP connection costs 377 bytes upstream and 377 bytes downstream, and is maintained by ping-pong packets and TCP ACKs consuming 164 bytes every 80 minutes.

We experimentally confirmed that a bitcoin node will accept all incoming connections from the same IP address. (We presume this is done to allow multiple nodes behind a NAT to connect to the same node.) Maintaining the default 117 incoming TCP connections costs $\frac{164\times117}{80\times60} \approx 4$ bytes per second, easily allowing one computer to monopolize multiple victims at the same time. As an aside, this also allows for *connection starvation attacks* (Dillon, 2013), where an attacker monopolizes all the incoming connections in the peer-to-peer network, making it impossible for new nodes to connect to new peers.

3.4 How Many Attack Addresses?

Section 3.3.3 showed that the success of our attack depends heavily on τ_ℓ, the time invested in the attack, and f, the fraction of attacker addresses in the victim's `tried` table. We now use probabilistic analysis to determine how many addresses the attacker must control for a given value of f; it's important to remember, however, that even if f is small, our attacker can still succeed by increasing τ_ℓ. Recall from Section 3.2.2 that bitcoin is careful to ensure that a node does not store too many IP addresses from the same *group* (*i.e.,* /16 IPv4 address block). We therefore consider two attack variants:

Botnet attack (Section 3.4.1). The attacker holds several IP addresses, each

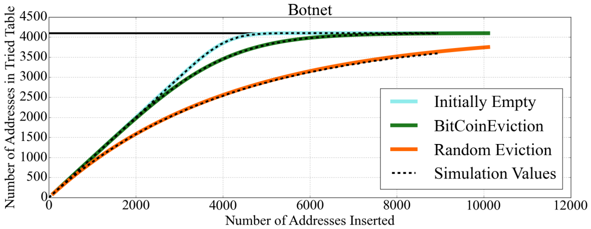


Figure 3·2: Botnet attack: the expected number of addresses stored in `tried` for different scenarios vs the number of addresses (bots) t. Values were computed from equations (3.4), (3.7) and (3.8), and confirmed by Monte Carlo simulations (with 100 trials/data point).

in a *distinct* group. This models attacks by a botnet of hosts scattered in diverse IP address blocks. Section 3.4.1 explains why many botnets have enough IP address diversity for this attack.

Infrastructure attack (Section 3.4.2). The attacker controls several IP address blocks, and can intercept bitcoin traffic sent to any IP address in the block, *i.e.*, the attacker holds multiple sets of addresses in the same *group*. This models a company or nation-state that seeks to undermine bitcoin by attacking its network. Section 3.4.2 discusses organizations that can launch this attack.

We focus here on `tried`; Appendix B.2 considers how to send “trash”-filled `ADDR` messages that overwrite `new`.

3.4.1 Botnet attack

The botnet attacker holds t addresses in distinct groups. We model each address as hashing to a uniformly-random bucket in `tried`, so the number of addresses hashing to each bucket is binomally distributed[3] as $B(t, \frac{1}{64})$. How many of the 64×64 entries in `tried` can the attacker occupy? We model various scenarios, and plot results in Figure 3·2.

[3] $B(n, p)$ is a binomial distribution counting successes in a sequence of n independent yes/no trials, each yielding ‘yes’ with probability p.

1. Initially empty. In the best case for the attacker, all 64 buckets are initially empty and the expected number of adversarial addresses stored in the `tried` table is

$$64E[\min(64, B(t, \tfrac{1}{64}))] \tag{3.4}$$

2. Bitcoin eviction. Now consider the worst case for the attacker, where each bucket i is full of 64 legitimate addresses. These addresses, however, will be *older* than all A_i distinct adversarial addresses that the adversary attempts to insert into to bucket i. Since the bitcoin eviction discipline requires each newly inserted address to select four random addresses stored in the bucket and to evict the oldest, if one of the four selected addresses is a legitimate address (which will be older than all of the adversary's addresses), the legitimate address will be overwritten by the adversarial addresses.

For $a = 0....A_i$, let Y_a be the number of adversarial addresses actually stored in bucket i, given that the adversary inserted a unique addresses into bucket i. Let $X_a = 1$ if the a^{th} inserted address successfully overwrites a legitimate address, and $X_a = 0$ otherwise. Then,

$$E[X_a|Y_{a-1}] = 1 - (\tfrac{Y_{a-1}}{64})^4$$

and it follows that

$$E[Y_a|Y_{a-1}] = Y_{a-1} + 1 - (\tfrac{Y_{a-1}}{64})^4 \tag{3.5}$$

$$E[Y_1] = 1 \tag{3.6}$$

where (3.6) follows because the bucket is initially full of legitimate addresses. We now have a recurrence relation for $E[Y_a]$, which we can solve numerically. The expected number of adversarial addresses in all buckets is thus

$$64 \sum_{a=1}^{t} E[Y_a] \Pr[B(t, \tfrac{1}{64}) = a] \tag{3.7}$$

3. Random eviction. We again consider the attacker's worst case, where each bucket is full of legitimate addresses, but now we assume that each inserted address evicts a randomly-selected address. (This is not what bitcoin does, but we analyze it for comparison.) Applying Lemma B.1.1 (Appendix B.1) we find the expected number of adversarial addresses in all buckets is

$$4096(1 - (\tfrac{4095}{4096})^t) \tag{3.8}$$

4. Exploiting multiple rounds. Our eclipse attack proceeds in *rounds*; in each round the attacker repeatedly inserts each of his t addresses into the `tried` table. While each address always maps to the same bucket in `tried` in each round, bitcoin eviction maps each address to a *different slot* in that bucket in every round. Thus, an adversarial address that is not stored into its `tried` bucket at the end of one round, might still be successfully stored into that bucket in a future round. Thus far, this section has only considered a single round. But, more addresses can be stored in `tried` by repeating the attack for multiple rounds. After sufficient rounds, the expected number of addresses is given by equation (3.4), *i.e.,* the attack performs as in the best-case for the attacker!

Who can launch a botnet attack?

The 'initially empty' line in Figure 3·2 indicates that a botnet exploiting multiple rounds can completely fill `tried` with ≈ 6000 addresses. While such an attack cannot easily be launched from a legitimate cloud service (which typically allocates < 20 addresses per tenant (AWS, 2014; azure, 2014; Rackspace, 2014)), botnets of this size and larger than this have attacked bitcoin (Vasek et al., 2014; Johnson et al., 2014; King, 2014); the Miner botnet, for example, had 29,000 hosts with public IPs (Plohmann and Gerhards-Padilla, 2012). While some botnet infestations concen-

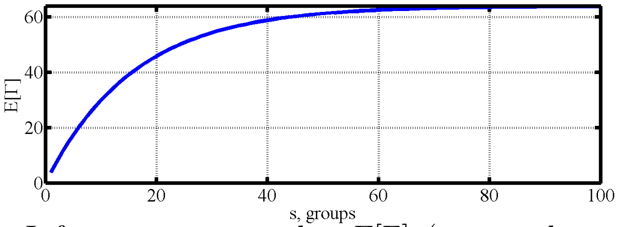


Figure 3·3: Infrastructure attack. $E[\Gamma]$ (expected number of non-empty buckets) in `tried` vs s (number of groups).

trate in a few IP address ranges (Stock et al., 2009), it is important to remember that our botnet attack requires no more than ≈ 6000 groups; many botnets are orders of magnitude larger (Rossow et al., 2013). For example, the Walowdac botnet was mostly in ranges 58.x-100.x and 188.x-233.x (Stock et al., 2009), which creates $42 \times 2^8 + 55 \times 2^8 = 24832$ groups. Randomly sampling from the list of hosts in the Carna botnet (CarnaBotnet, 2012) 5000 times, we find that 1250 bots gives on average 402 distinct groups, enough to attack our live bitcoin nodes (Section 3.6). Furthermore, we soon show in Figure 3·3 that an infrastructure attack with $s > 200$ groups easily fills every bucket in `tried`; thus, with $s > 400$ groups, the attack performs as in Figure 3·2, even if many bots are in the same group. .

3.4.2 Infrastructure attack

The attacker holds addresses in s distinct *groups*. We determine how much of `tried` can be filled by an attacker controlling s groups s containing t IP addresses/group.

How many groups? We model the process of populating `tried` (per Section 3.2.2) by supposing that four independent hash functions map each of the s groups to one of 64 buckets in `tried`. Thus, if $\Gamma \in [0, 64]$ counts the number of non-empty buckets in `tried`, we use Lemma B.1.1 to find that

$$E[\Gamma] = 64\left(1 - (\tfrac{63}{64})^{4s}\right) \approx (1 - e^{-\frac{4s}{64}}) \tag{3.9}$$

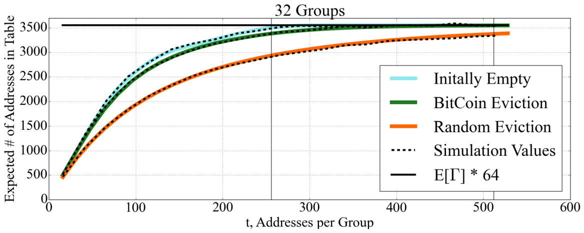


Figure 3·4: Infrastructure attack with $s = 32$ groups: the expected number of addresses stored in `tried` for different scenarios vs the number of addresses per group t. Results obtained by taking the product of equation (3.9) and equations from the full version (Heilman et al., 2015b), and confirmed by Monte Carlo simulations (100 trials/data point). The horizontal line assumes all $E[\Gamma]$ buckets per (3.9) are full.

Figure 3·3 plots $E[\Gamma]$; we expect to fill 55.5 of 64 buckets with $s = 32$, and all but one bucket with $s > 67$ groups.

How full is the `tried` table? The full version (Heilman et al., 2015b) determines the expected number of addresses stored per bucket for the first three scenarios described in Section 3.4.1; the expected fraction $E[f]$ of `tried` filled by adversarial addresses is plotted in in Figure 3·4. The horizontal line in Figure 3·4 show what happens if each of $E[\Gamma]$ buckets per equation (3.9) is full of attack addresses.

The adversary's task is easiest when all buckets are initially empty, or when a sufficient number of rounds are used; a single /24 address block of 256 addresses suffices to fill each bucket when $s = 32$ grouips is used. Moreover, as in Section 3.4.1, an attack that exploits multiple rounds performs as in the 'initially empty' scenario. Concretely, with 32 groups of 256 addresses each (8192 addresses in total) an adversary can expect to fill about $f = 86\%$ of the `tried` table after a sufficient number of rounds. The attacker is almost as effective in the bitcoin-eviction scenario with only one round; meanwhile, one round is much less effective with random eviction.

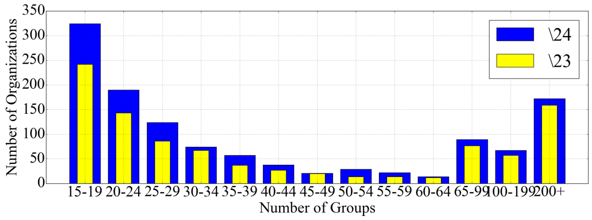


Figure 3·5: Histogram of the number of organizations with s groups. For the /24 data, we require $t = 256$ addresses per group; for /23, we require $t = 512$.

Who can launch an infrastructure attack?

Which organizations have enough IP address resources to launch infrastructure attacks? We compiled data mapping IPv4 address allocation to organizations, using CAIDA's AS to organization dataset (CAIDA, 2014a) and AS to prefix dataset (CAIDA, 2014b) from July 2014, supplementing our data with information from the RIPE database (RIPE, 2014). We determined how many groups (*i.e.,* addresses in the same /16 IPv4 address block) and addresses per group are allocated to each organization; see Figure 3·5. There are 448 organizations with over $s = 32$ groups and at least $t = 256$ addresses per group; if these organizations invest $\tau_\ell = 5$ hours into an attack with a $\tau_a = 27$-minute round, then they eclipse the victim with probability greater than 80%.

National ISPs in various countries hold a sufficient number of groups ($s \geq 32$) for this purpose; for example, in Sudan (Sudanese Mobile), Columbia (ETB), UAE (Etisalat), Guatemala (Telgua), Tunisia (Tunisia Telecom), Saudi Arabia (Saudi Telecom Company) and Dominica (Cable and Wireless). The United States Department of the Interior has enough groups ($s = 35$), as does the S. Korean Ministry of Information and Communication ($s = 41$), as do hundreds of others.

3.4.3 Summary: infrastructure or botnet?

Figures 3·4, 3·2 show that the botnet attack is far superior to the infrastructure attack. Filling $f = 98\%$ of the victim's `tried` table requires a 4600 node botnet (attacking for a sufficient number of rounds, per equation (3.4)). By contrast, an infrastructure attacker needs $16,000$ addresses, consisting of $s = 63$ groups (equation (3.9)) with $t = 256$ addresses per group. However, per Section 3.3.3, if our attacker increases the time invested in the attack τ_ℓ, it can be far less aggressive about filling `tried`. For example, per Figure 3·1, attacking for $\tau_\ell = 24$ hours with $\tau_a = 27$ minute rounds, our success probability exceeds 85% with just $f = 72\%$; in the worst case for the attacker, this requires only 3000 bots, or an infrastructure attack of $s = 20$ groups and $t = 256$ addresses per group (5120 addresses). The same attack ($f = 72\%$, $\tau_a = 27$ minutes) running for just 4 hours still has $> 55\%$ success probability. To put this in context, if 3000 bots joined today's network (with < 7200 public-IP nodes (Bitnode, 2014)) and honestly followed the peer-to-peer protocol, they could eclipse a victim with probability $\approx (\frac{3000}{7200+3000})^8 = 0.006\%$.

3.5 Measuring Live Bitcoin Nodes

We briefly consider how parameters affecting the success of our eclipse attacks look on "typical" bitcoin nodes. We thus instrumented five bitcoin nodes with public IPs that we ran (continuously, without restarting) for 43 days from 12/23/2014 to 2/4/2015. We also analyze several peers files that others donated to us on 2/15/2015. Note that there is evidence of wide variations in metrics for nodes of different ages and in different regions (Karame et al., 2012); as such, our analysis (Section 3.3-3.4) and some of our experiments (Section 3.6) focus on the attacker's worst-case scenario, where tables are initially full of fresh addresses.

Number of connections. Our attack requires the victim to have available slots

oldest addr	# addr	% live	Age of addresses (in days) < 1	 $1 - 5$	 $5 - 10$	 $10 - 30$	 > 30
38 d*	243	28%	36	71	28	79	29
41 d*	162	28%	23	29	27	44	39
42 d*	244	19%	25	45	29	95	50
42 d*	195	23%	23	40	23	64	45
43 d*	219	20%	66	57	23	50	23
103 d	4096	8%	722	645	236	819	1674
127 d	4096	8%	90	290	328	897	2491
271 d	4096	8%	750	693	356	809	1488
240 d	4096	6%	419	445	32	79	3121
373 d	4096	5%	9	14	1	216	3856

Table 3.1: Age and churn of addresses in `tried` for our nodes (marked with *) and donated peers files.

for incoming connections. Figure 3·6 shows the number of connections over time for one of our bitcoin nodes, broken out by connections to public or private IPs. There are plenty of available slots; while our node can accommodate 125 connections, we never see more than 60 at a time. Similar measurements in (Biryukov and Pustogarov, 2014) indicate that 80% of bitcoin peers allow at least 40 incoming connections. Our node saw, on average, 9.9 connections to public IPs over the course of its lifetime; of these, 8 correspond to *outgoing* connections, which means we rarely see incoming connections from public IPs. Results for our other nodes are similar.

Connection length. Because public bitcoin nodes rarely drop outgoing connections to their peers (except upon restart, network failure, or due to blacklisting, see Section 3.2.3), many connections are fairly long lived. When we sampled our nodes on 2/4/2015, across all of our nodes, 17% of connections had lasted more than 15 days, and of these, 65.6% were to public IPs. On the other hand, many bitcoin nodes restart frequently; we saw that 43% of connections lasted less than two days and of these, 97% were to nodes with private IPs. This may explain why we see so few incoming connections from public IPs; many public-IP nodes stick to their mature long-term peers, rather than our young-ish nodes.

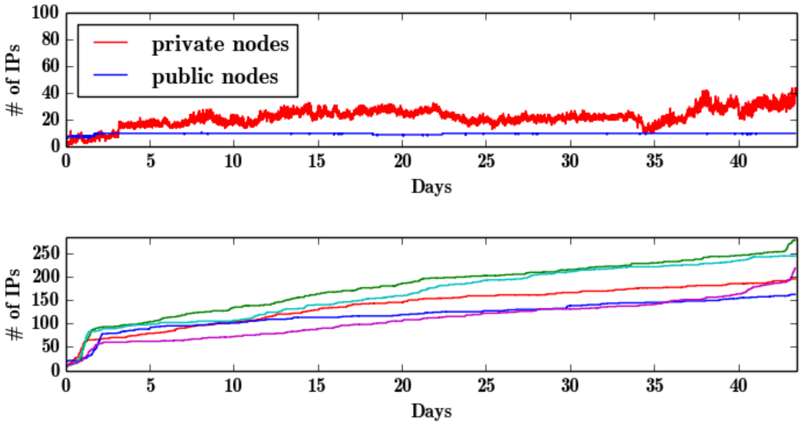


Figure 3·6: (Top) Incoming + outgoing connections vs time for one of our nodes. (Bottom) Number of addresses in `tried` vs time for all our nodes.

Size of `tried` and `new` tables. In our worst case attack, we supposed that the `tried` and `new` tables were completely full of fresh addresses. While our Bitcoin nodes' `new` tables filled up quite quickly (99% within 48 hours), Table 3.1 reveals that their `tried` tables were far from full of fresh addresses. Even after 43 days, the `tried` tables for our nodes were no more than 300/4096 $\approx$ 8% full. This likely follows because our nodes had very few incoming connections from public IPs; thus, most addresses in `tried` result from successful outgoing connections to public IPs (infrequently) drawn from `new`.

Freshness of `tried`. Even those few addresses in `tried` are not especially fresh. Table 3.1 shows the age distribution of the addresses in `tried` for our nodes and from donated peers files. For our nodes, 17% of addresses were more than 30 days old, and 48% were more than 10 days old; these addresses will therefore be less preferred than the adversarial ones inserted during an eclipse attack, even if the adversary does not invest much time τ_ℓ in attacking the victim.

Churn. Table 3.1 also shows that a small fraction of addresses in `tried` were online

	Attacker resources					Experiment							Predicted		
	grps	addrs	total	τ_ℓ	τ_a,	pre-attack		post-attack		Attack addrs			Attack addrs		
Attack	s	/grp t	addrs		rnd	`new`	`tried`	`new`	`tried`	`new`	`tried`	**Wins**	`new`	`tried`	**Wins**
Infra (WC)	32	256	8192	10 h	43 m	16384	4090	16384	4096	15871	3404	**98%**	16064	3501	**87%**
Infra (TP)	20	256	5120	1 hr	27 m	16380	278	16383	3087	14974	2947	**82%**	15040	2868	**77%**
Infra (TP)	20	256	5120	2 hr	27 m	16380	278	16383	3088	14920	2966	**78%**	15040	2868	**87%**
Infra (TP)	20	256	5120	4 hr	27 m	16380	278	16384	3088	14819	2972	**86%**	15040	2868	**91%**
Infra (L)	20	256	5120	1 hr	27 m	16381	346	16384	3116	14341	2942	**84%**	15040	2868	**75%**
Bots (WC)	2300	2	4600	5 h	26 m	16080	4093	16384	4096	16383	4015	**100%**	16384	4048	**96%**
Bots (TP)	200	1	200	1 hr	74 s	16380	278	16384	448	16375	200	**60%**	16384	200	**11%**
Bots (TP)	400	1	400	1 hr	90 s	16380	278	16384	648	16384	400	**88%**	16384	400	**34%**
Bots (TP)	400	1	400	4 hr	90 s	16380	278	16384	650	16383	400	**84%**	16384	400	**61%**
Bots (TP)	600	1	600	1 hr	209 s	16380	278	16384	848	16384	600	**96%**	16384	600	**47%**
Bots (L)	400	1	400	1 hr	90 s	16380	298	16384	698	16384	400	**84%**	16384	400	**28%**

Table 3.2: Summary of our experiments.
WC=Worstcase, TP=Transplant, L=Live

when we tried connecting to them on 2/17/2015.[4] This suggests further vulnerability to eclipse attacks, because if most legitimate addresses in `tried` are offline when a victim resets, the victim is likely to connect to an adversarial address.

3.6 Experiments

We now validate our analysis with experiments.

Methodology. In each of our experiments, the victim (bitcoind) node is on a virtual machine on the attacking machine; we also instrument the victim's code. The victim node runs on the public bitcoin network (*aka,* mainnet). The attacking machine can read all the victim's packets to/from the public bitcoin network, and can therefore forge TCP connections from arbitrary IP addresses. To launch the attack, the attacking machine forges TCP connections from each of its attacker addresses, making an incoming connection to the victim, sending a `VERSION` message and sometimes also an `ADDR` message (per Appendix B.2) and then disconnecting; the attack connections, which are launched at regular intervals, rarely occupy all of the victim's available slots for incoming connections. To avoid harming the public bitcoin network, (1) we use "reserved for future use" (IANA, 2015) IPs in 240.0.0.0/8-249.0.0.0/8 as

[4] For consistency with the rest of this section, we tested our nodes tables from 2/4/2015. We also repeated this test for tables taken from our nodes on 2/17/2015, and the results did not deviate more than 6% from those of Table 3.1.

attack addresses, and 252.0.0.0/8 as "trash" sent in `ADDR` messages, and (2) we drop any `ADDR` messages the (polluted) victim attempts to send to the public network.

At the end of the attack, we repeatedly restart the victim and see what outgoing connections it makes, dropping connections to the "trash" addresses and forging connections for the attacker addresses. If all 8 outgoing connections are to attacker addresses, the attack succeeds, and otherwise it fails. Each experiment restarts the victim 50 times, and reports the fraction of successes. At each restart, we revert the victim's tables to their state at the end of the attack, and rewind the victim's system time to the moment the attack ended (to avoid dating timestamps in `tried` and `new`). We restart the victim 50 times to measure the success rate of our (probabilistic) attack; in a real attack, the victim would only restart once.

Initial conditions. We try various initial conditions:

1. Worst case. In the attacker's worst-case scenario, the victim initially has `tried` and `new` tables that are completely full of legitimate addresses with fresh timestamps. To set up the initial condition, we run our attack for no longer than one hour on a freshly-born victim node, filling `tried` and `new` with IP addresses from 251.0.0.0/8, 253.0.0.0/8 and 254.0.0.0/8, which we designate as "legitimate addresses"; these addresses are no older than one hour when the attack starts. We then restart the victim and commence attacking it.

2. Transplant case. In our transplant experiments, we copied the `tried` and `new` tables from one of our five live bitcoin nodes on 8/2/2015, installed them in a fresh victim with a different public IP address, restarted the victim, waited for it to establish eight outgoing connections, and then commenced attacking. This allowed us to try various attacks with a consistent initial condition.

3. Live case. Finally, on 2/17/2015 and 2/18/2015 we attacked our live bitcoin nodes while they were connected to the public bitcoin network; at this point our nodes had been online for 52 or 53 days.

Results (Table 3.2). Results are in Table 3.2. The first five columns summarize attacker resources (the number of groups s, addresses per group t, time invested in the attack τ_ℓ, and length of a round τ_a per Sections 3.3-3.4). The next two columns present the initial condition: the number of addresses in `tried` and `new` prior to the attack. The following four columns give the size of `tried` and `new`, and the number of attacker addresses they store, at the end of the attack (when the victim first restarts). The *wins* columns counts the fraction of times our attack succeeds after restarting the victim 50 times.

The final three columns give predictions from Sections 3.3.3, 3.4. The *attack addrs* columns give the expected number of addresses in `new` (Appendix B.2) and `tried`. For `tried`, we assume that the attacker runs his attack for enough rounds so that the expected number of addresses in `tried` is governed by equation (3.4) for the botnet, and the 'initially empty' curve of Figure 3·4 for the infrastructure attack. The final column predicts success per Section 3.3.3 using *experimental values* of τ_a, τ_ℓ, f, f'.

Observations. Our results indicate the following:

1. Success in worst case. Our experiments confirm that an infrastructure attack with 32 groups of size /24 (8192 attack addresses total) succeeds in the worst case with very high probability. We also confirm that botnets are superior to infrastructure attacks; 4600 bots had 100% success even with a worst-case initial condition.

2. Accuracy of predictions. Almost all of our attacks had an experimental success rate that was *higher* than the predicted success rate. To explain this, recall that our predictions from Section 3.3.3 assume that legitimate addresses are exactly τ_ℓ old (where τ_ℓ is the time invested in the attack); in practice, legitimate addresses are likely to be even older, especially when we work with `tried` tables of real nodes (Table 3.1). Thus, Section 3.3.3's predictions are a lower bound on the success rate.

Our experimental botnet attacks were dramatically more successful than their predictions (*e.g.,* 88% actual *vs.* 34% predicted), most likely because the addresses initially in tried were already very stale prior to the attack (Table 3.1). Our infrastructure attacks were also more successful then their predictions, but here the difference was much less dramatic. To explain this, we look to the new table. While our success-rate predictions assume that new is completely overwritten, our infrastructure attacks failed to completely overwrite the new table;[5] thus, we have some extra failures because the victim made outgoing connections to addresses in new.

3. Success in a 'typical' case. Our attacks are successful with even fewer addresses when we test them on our live nodes, or on tables taken from those live nodes. Most strikingly, a small botnet of 400 bots succeeds with very high probability; while this botnet completely overwrites new, it fills only 400/650 = 62% of tried, and still manages to win with more than 80% probability.

3.7 Countermeasures

We have shown how an attacker with enough IP addresses and time can eclipse any target victim, regardless of the state of the victim's tried and new tables. We now present countermeasures that make eclipse attacks more difficult. Our countermeasures are inspired by botnet architectures (Section 3.8), and designed to be faithful to bitcoin's network architecture.

The following five countermeasures ensure that: (1) If the victim has h legitimate addresses in tried before the attack, and a p-fraction of them accept incoming connections during the attack when the victim restarts, then even an attacker *with an unbounded number of addresses* cannot eclipse the victim with probability exceeding equation (3.10). (2) If the victim's oldest outgoing connection is to a legitimate

[5] The new table holds 16384 addresses and from 6^{th} last column of Table 3.2 we see the new is not full for our infrastructure attacks. Indeed, we predict this in Appendix B.2.

peer before the attack, then the eclipse attack *fails* if that peer accepts incoming connections when the victim restarts.

1. Deterministic random eviction. Replace bitcoin eviction as follows: just as each address deterministically hashes to a single bucket in `tried` and `new` (Section 3.2.2), an address also deterministically hashes to a single slot in that bucket. This way, an attacker cannot increase the number of addresses stored by repeatedly inserting the same address in multiple rounds (Section 3.4.1). Instead, addresses stored in `tried` are given by the 'random eviction' curves in Figures 3·2, 3·4, reducing the attack addresses stored in `tried`.

2. Random selection. Our attacks also exploit the heavy bias towards forming outgoing connections to addresses with fresh timestamps, so that an attacker that owns only a small fraction $f = 30\%$ of the victim's `tried` table can increase its success probability (to say 50%) by increasing τ_ℓ, the time it invests in the attack (Section 3.3.3). We can eliminate this advantage for the attacker if addresses are selected at random from `tried` and `new`; this way, a success rate of 50% always requires the adversary to fill $\sqrt[8]{0.5} = 91.7\%$ of `tried`, which requires 40 groups in an infrastructure attack, or about 3680 peers in a botnet attack. Combining this with deterministic random eviction, the figure jumps to 10194 bots for 50% success probability.

These countermeasures harden the network, but still allow an attacker with enough addresses to overwrite all of `tried`. The next countermeasure remedies this:

3. Test before evict. Before storing an address in its (deterministically-chosen) slot in a bucket in `tried`, first check if there is an older address stored in that slot. If so, briefly attempt to connect to the older address, and if connection is successful, then the older address is *not* evicted from the `tried` table; the new address is stored in `tried` only if the connection fails.

We analyze these three countermeasures. Suppose that there are h legitimate addresses in the tried table prior to the attack, and model network churn by supposing that each of the h legitimate addresses in tried is live (*i.e.,* accepts incoming connections) independently with probability p. With test-before-evict, the adversary cannot evict $p \times h$ legitimate addresses (in expectation) from tried, regardless of the number of distinct addresses it controls. Thus, even if the rest of tried is full of adversarial addresses, the probability of eclipsing the victim is bounded to about

$$\Pr[\text{eclipse}] = f^8 < \left(1 - \tfrac{p \times h}{64 \times 64}\right)^8 \tag{3.10}$$

This is in stark contrast to today's protocol, where attackers with enough addresses have *unbounded* success probability even if tried is *full* of legitimate addresses.

We perform Monte-Carlo simulations assuming churn p, h legitimate addresses initially stored in tried, and a botnet inserting a addresses into tried via unsolicited incoming connections. The area below each curve in Figure 3·7 is the number of bots a that can eclipse a victim with probability at least 50%, given that there are initially h legitimate addresses in tried. With test-before-evict, the curves plateau horizontally at $h = 4096(1 - \sqrt[8]{0.5})/p$; as long as h is greater than this quantity, even a botnet *with an infinite number of addresses* has success probability bounded by 50%. Importantly, the plateau is absent without test-before-evict; a botnet with enough addresses can eclipse a victim *regardless* of the number of legitimate addresses h initially in tried.

There is one problem, however. Our bitcoin nodes saw high churn rates (Table 3.1). With a $p = 28\%$ churn rate, for example, bounding the adversary's success probability to 10% requires about $h = 3700$ addresses in tried; our nodes had $h < 400$. Our next countermeasure thus adds more legitimate addresses to tried:

4. Feeler Connections. Add an outgoing connection that establish short-lived test connections to randomly-selected addresses in new. If connection succeeds, the

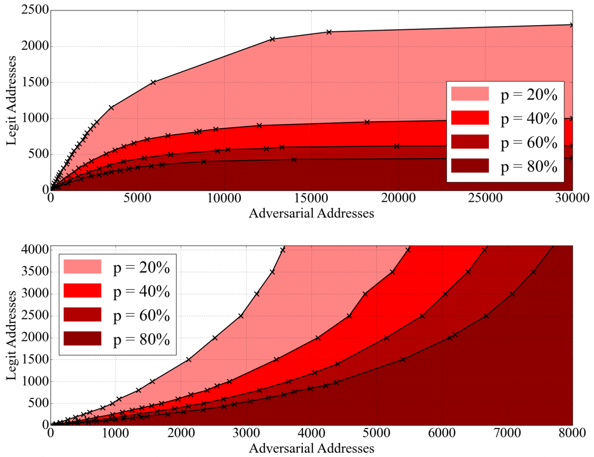


Figure 3·7: The area below each curve corresponds to a number of bots a that can eclipse a victim with probability at least 50%, given that the victim initially has h legitimate addresses in `tried`. We show one curve per churn rate p. (Top) With test before evict. (Bottom) Without.

address is evicted from `new` and inserted into `tried`; otherwise, the address is evicted from `new`.

Feeler connections clean trash out of `new` while increasing the number of fresh address in `tried` that are likely to be online when a node restarts. Our fifth countermeasure is orthogonal to those above:

5. Anchor connections. Inspired by Tor entry guard rotation rates (Dingledine et al., 2014), we add two connections that persist between restarts. Thus, we add an `anchor` table, recording addresses of current outgoing connections and the time of first connection to each address. Upon restart, the node dedicates two extra outgoing connections to the oldest `anchor` addresses that accept incoming connections. Now, in addition to defeating our other countermeasures, a successful attacker must also disrupt `anchor` connections; eclipse attacks fail if the victim connects to an `anchor` address not controlled by the attacker.

Apart from these five countermeasures, a few other ideas can raise the bar for eclipse attacks:

6. More buckets. Among the most obvious countermeasure is to increase the size of the tried and new tables. Suppose we doubled the number of buckets in the tried table. If we consider the infrastructure attack, the buckets filled by s groups jumps from $(1 - e^{-\frac{4s}{64}})$ (per equation (3.9) to $(1 - e^{-\frac{4s}{128}})$. Thus, an infrastructure attacker needs double the number of groups in order to expect to fill the same fraction of tried. Similarly, a botnet needs to double the number of bots. Importantly, however, this countermeasure is helpful only when tried already contains many legitimate addresses, so that attacker owns a smaller fraction of the addresses in tried. However, if tried is mostly empty (or contains mostly stale addresses for nodes that are no longer online), the attacker will still own a large fraction of the addresses in tried, even though the number of tried buckets has increased. Thus, this countermeasure should also be accompanied by another countermeasure (*e.g.,* feeler connections) that increases the number of legitimate addresses stored in tried.

7. More outgoing connections. Figure 3·6 indicates our test bitcoin nodes had at least 65 connections slots available, and (Biryukov and Pustogarov, 2014) indicates that 80% of bitcoin peers allow at least 40 incoming connections. Thus, we can require nodes to make a few additional outgoing connections without risking that the network will run out of connection capacity. Indeed, recent measurements (Miller et al., 2015) indicate that certain nodes (*e.g.,* mining-pool gateways) do this already. For example, using twelve outgoing connections instead of eight (in addition to the feeler connection and two anchor connections), decreases the attack's success probability from f^8 to f^{12}; to achieve 50% success probability the infrastructure attacker now needs 46 groups, and the botnet needs 11796 bots.

8. Ban unsolicited ADDR messages. A node could choose not to accept large unsolicited ADDR messages (with > 10 addresses) from incoming peers, and only solicit ADDR messages from outgoing connections when its new table is too empty. This prevents adversarial incoming connections from flooding a victim's `new` table with trash addresses. We argue that this change is not harmful, since even in the current network, there is no shortage of address in the `new` table (Section 3.5). To make this more concrete, note that a node request `ADDR` messages upon establishing an outgoing connection. The peer responds with n randomly selected addresses from its `tried` and `new` tables, where n is a random number between x and 2500 and x is 23% of the addresses the peer has stored. If each peer sends, say, about $n = 1700$ addresses, then `new` is already $8n/16384 = 83\%$ full the moment that the bitcoin node finishing establishing outgoing connections.

9. Diversify incoming connections. Today, a bitcoin node can have all of its incoming connections come from the same IP address, making it far too easy for a single computer to monopolize a victim's incoming connections during an eclipse attack or connection-starvation attack (Dillon, 2013). We suggest a node accept only a limited number of connections from the same IP address.

10. Anomaly detection. Our attack has several specific "signatures" that make it detectable including: (1) a flurry of short-lived incoming TCP connections from diverse IP addresses, that send (2) large `ADDR` messages (3) containing "trash" IP addresses. An attacker that suddenly connects a large number of nodes to the bitcoin network could also be detected, as could one that uses eclipsing per Section 3.1.1 to dramatically decrease the network's mining power. Thus, monitoring and anomaly detection systems that look for this behavior are also be useful; at the very least, they would force an eclipse attacker to attack at low rate, or to waste resources on overwriting `new` (instead of using "trash" IP addresses).

Status of our countermeasures. In Table 1.1 we provide a timeline of current status of our countermeasures in bitcoind. We disclosed our results to the bitcoin core developers on 02/2015. They deployed Countermeasures 1, 2, and 6 in the bitcoind v0.10.1 release, which now uses deterministic random eviction, random selection, and scales up the number of buckets in `tried` and `new` by a factor of four. To illustrate the efficacy of this, consider the worst-case scenario for the attacker where `tried` is completely full of legitimate addresses. We use Lemma B.1.1 to estimate the success rate of a botnet with t IP addresses as

$$\Pr[\text{Eclipse}] \approx \left(1 - \left(\tfrac{16383}{16384}\right)^t\right)^8 \tag{3.11}$$

Plotting (3.11) in Figure 3·8, we see that this botnet requires 163K addresses for a 50% success rate, and 284K address for a 90% success rate. This is good news, but we caution that ensuring that `tried` is full of legitimate address is still a challenge (Section 3.5), especially since there may be fewer than 16384 public-IP nodes in the bitcoin network at a given time. Countermeasures 3 and 4 are designed to deal with this, and so we have also developed two patches for each of these two countermeasures; our patch for countermeasure 4 was adopted and deployed in bitcoind v0.13.1, our patch for countermeasure 3 was merged and deployed in bitcoind v0.17.0. Countermeasure 5 was implemented and merged into bitcoind v0.21.0. Only 2 of our 10 proposed countermeasures remain undeployed.

While countermeasures 7 and 9 were also now deployed in bitcoind, it is unclear if our research motivated these changes. The pull-request which implemented countermeasure 7 lists its purpose as defend against the attacks this other research paper (Delgado-Segura et al., 2019). Countermeasure 9 was adopted as a general DoS protection mechanism and does not discuss eclipse attacks.

3.8 Related Work

The bitcoin peer-to-peer (p2p) network. Recent work considers how bitcoin's network can delay or prevent block propagation (Decker and Wattenhofer, 2013) or

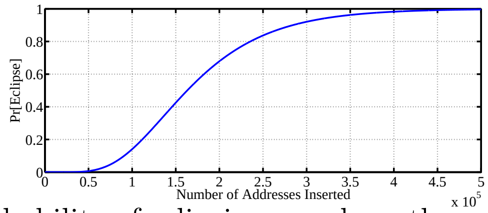


Figure 3·8: Probability of eclipsing a node vs the number of addresses (bots) t for bitcoind v0.10.1 (with Countermeasures 1,2 and 6) when `tried` is initially full of legitimate addresses per equation (3.11).

be used to deanonymize bitcoin users (Koshy et al., 2014; Biryukov et al., 2014a; Biryukov and Pustogarov, 2014). These works discuss aspects of bitcoin's networking protocol, with (Biryukov et al., 2014a) providing an excellent description of `ADDR` message propagation; we focus instead on the structure of the `tried` and `new` tables, timestamps and their impact on address selection (Section 3.2). (Biryukov and Pustogarov, 2014) shows that nodes connecting over Tor can be eclipsed by a Tor exit node that manipulates both bitcoin and Tor. Other work has mapped bitcoin peers to autonomous systems (Feld et al., 2014), geolocated peers and measured churn (Donet et al., 2014), and used side channels to learn the bitcoin network topology (Biryukov et al., 2014a; Miller et al., 2015).

p2p and botnet architectures. There has been extensive research on eclipse attacks (Castro et al., 2002; Sit and Morris, 2002; Singh et al., 2006) in structured p2p networks built upon distributed hash tables (DHTs); see (Urdaneta et al., 2011) for a survey. Many proposals defend against eclipse attacks by adding more structure; (Singh et al., 2006) constrains peer degree, while others use constraints based on distance metrics like latency (Hildrum and Kubiatowicz, 2003) or DHT identifiers (Awerbuch and Scheideler, 2006). Bitcoin, by contrast, uses an unstructured network. While we have focused on exploiting specific quirks in bitcoin's existing network, other works *e.g.,* (Bortnikov et al., 2009; Bakker and Van Steen, 2008; Anceaume et al., 2013; Jesi et al., 2010) design new unstructured networks that are robust to Byzantine attacks. (Jesi et al., 2010) blacklists misbehaving peers. Puppetcast's (Bakker and Van Steen, 2008) centralized solution is based on public-key infrastructure (Bakker and Van Steen, 2008), which is not appropriate for bitcoin. Brahms (Bortnikov et al., 2009) is fully decentralized, and instead constrains the rate at which peers exchange network information—a useful idea that is a significant departure from bitcoin's current approach. Meanwhile, our goals are also more modest than those in these works; rather than requiring that each node is *equally likely* to be sampled by an honest node, we just want to limit eclipse attacks on initially well-connected nodes. Thus, our countermeasures are inspired by botnet architectures, which share this same goal. Rossow *et al.* (Rossow et al., 2013) finds that many botnets, like bitcoin, use unstructured peer-to-peer networks and gossip (*i.e.,* `ADDR` messages), and describes how botnets defend against attacks that flood local address tables with bogus information. The Sality botnet refuses to evict "high-reputation" addresses; our `anchor` countermeasure is similar (Section 3.7). Storm uses test-before-evict (Davis et al., 2008), which we have also recommended for bitcoin. Zeus (Andriesse and Bos, 2014) disallows connections from multiple IP in the same /20, and regularly clean tables by testing if peers are online; our feeler connections are similar.

3.9 Conclusion

In this chapter we presented an eclipse attack on bitcoin's peer-to-peer network that undermines bitcoin's core security guarantees, allowing attacks on the mining and consensus system, including N-confirmation double spending and adversarial forks in the blockchain. Our attack is for nodes with public IPs. We developed mathematical models of our attack, and validated them with Monte Carlo simulations, measurements and experiments. We demonstrated the practically of our attack by performing it on our own live bitcoin nodes, finding that an attacker with 32 distinct /24 IP address blocks, or a 4600-node botnet, can eclipse a victim with over 85% probability in the attacker's *worst case.* Moreover, even a 400-node botnet sufficed to attack our own live bitcoin nodes. Finally, we proposed countermeasures that make eclipse attacks more difficult while still preserving bitcoin's openness and decentralization; several of these were incorporated in a recent bitcoin software upgrade.

Chapter 4

Blindly Signed Contracts: Anonymous Bitcoin Transactions

This chapter uses work from (Heilman et al., 2016b) which was written in collaboration with Foteini Baldimtsi and Sharon Goldberg.

4.1 Introduction

When Bitcoin was first introduced in 2008, one of its key selling points was anonymity—users should be able to spend bitcoins "without information linking the transaction to anyone" (Nakamoto, 2008). In the last few years, however, researchers have shown that Bitcoin offers much weaker anonymity than was initially expected (Meiklejohn et al., 2013; Ron and Shamir, 2013), by demonstrating that they could follow the movement of funds on the Bitcoin blockchain. The community has reacted to this by proposing two key approaches to improve the anonymity of Bitcoin: (1) new anonymity schemes that are compatible with Bitcoin (Barber et al., 2012; Bonneau et al., 2014; Ruffing et al., 2014; Valenta and Rowan, 2015; Bissias et al., 2014; Maxwell, 2013b; Ben Sasson et al., 2014; Saxena et al., 2014; Ziegeldorf et al., 2015), and (2) new anonymous cryptocurrencies that are independent of Bitcoin (Miers et al., 2013; Ben Sasson et al., 2014).

In this chapter we take the former approach by developing new anonymity schemes that are compatible with Bitcoin via a soft fork. Our schemes offer a new trade-off between practicality (*i.e.,* transaction speed), security (*i.e.,* resistance to double-spending, denial of service (DoS) and Sybil attacks) and anonymity (*i.e.,* unlinkable transactions). As we will see below, previous work either provided schemes that are efficient but achieve limited security or anonymity (Bonneau et al., 2014; Ruffing et al., 2014; Valenta and Rowan, 2015; Ziegeldorf et al., 2015; Saxena et al., 2014) or schemes that provide strong anonymity but are slow and require large numbers of transactions (Maxwell, 2013b; Bissias et al., 2014; Barber et al., 2012).

Our first scheme is an "on-blockchain" scheme providing anonymity at reasonable speed, *i.e.,* requiring four transactions to be confirmed in three blocks ($\approx$ 30 mins). Our protocol runs in epochs, and provides *set-anonymity within each epoch.* That is, while the blockchain publicly displays the *set* of payers and payees during an epoch, no one can tell which payer paid which payee. To do this, we introduce an untrusted (possibly malicious) intermediary $\mathcal{I}$ between all payers and payees.

Our second "off-blockchain" scheme uses a new payment technology called *micropayment channel networks* (Poon and Dryja, 2015; Decker and Wattenhofer, 2015). Micropayment channel networks use Bitcoin as a platform to confirm transactions within seconds, rather than minutes, and already provide a degree of anonymity—most of the transactions are made outside of the blockchain, and thus not shown to the public—but this anonymity is incomplete. Critically, because micropayment channel networks chain payments through pre-established paths of connected users (explained in Section 4.5.1), these users that participate in the path learn transaction details, including the cryptographic identities of the sending and receiving party. We provide anonymity against malicious users by using an *honest-but-curious* intermediary $\mathcal{I}$ (Section 4.5.3); set-anonymity within an epoch is preserved as long as $\mathcal{I}$ does not abort or deny service to payers or payees.

Our technique, inspired by eCash (Chaum, 1983a), works as follows. For a user $\mathcal{A}$ to anonymously pay another user $\mathcal{B}$, she would first exchange a bitcoin for an *anony-*

mous voucher through intermediary $\mathcal{I}$. $\mathcal{B}$ could then redeem the anonymous voucher with $\mathcal{I}$ to receive a bitcoin back. Our scheme overcomes two main challenges: (i) Ensuring that the vouchers are unlinkable (*i.e.,* hiding the link between the issuance and the redemption of a voucher), and (ii) enforcing fair exchange between participants (*i.e.,* users can redeem issued vouchers even against an uncooperative or malicious $\mathcal{I}$, and no party can steal or double-spend vouchers and bitcoins). We use *blind signatures* to achieve unlinkability, and the *scripting* functionality of Bitcoin transactions to achieve fair exchange via transaction contracts (aka smart contracts (Szabo, 1997)).

We provide an overview of our scheme in Section 4.2 and define the required properties. We discuss our use of transaction contracts in Section 4.3. Our scheme for on-blockchain anonymous transactions is in Section 4.4. Our off-blockchain scheme which uses micropayment channel networks is in Section 4.5. Finally, we analyze the anonymity of our schemes in Sections 4.4.2 and 4.5.3 and their security in Section 4.6.

4.1.1 Related Work

We now review some of the most representative related works in the literature.

Anonymous Payment Schemes. Zerocash (Ben Sasson et al., 2014) and Zerocoin (Miers et al., 2013) provide anonymous payments through the use of a novel type of cryptographic proofs (ZK-SNARKs). Unlike our schemes, they are "stand-alone" cryptocurrencies and can not be integrated with Bitcoin. Meanwhile, (Saxena et al., 2014) is an anonymous payment scheme that can offer anonymity protections to Bitcoin that provides excellent blockchain privacy and is very fast. However, the parties entrusted to anonymize transactions in (Saxena et al., 2014) can still violate users' anonymity, even if they are honest-but-curious.

Mixing Services. A bitcoin mixing service provides anonymity by transferring payments from an input set of bitcoin addresses to an output set of bitcoin addresses, such that is it hard to trace which input address paid which output address. Mixcoin (Bonneau et al., 2014) uses a trusted third party to mix Bitcoin addresses, but this third party can violate users privacy and steal users' bitcoins; theft is detected but not prevented. Blindcoin (Valenta and Rowan, 2015) improves on Mixcoin by preserving users privacy against the mixing service, as with Mixcoin, theft is still not prevented. CoinParty (Ziegeldorf et al., 2015) is secure if 2/3 of the mixing parties are honest. CoinJoin (Maxwell, 2013a) and CoinShuffle (Ruffing et al., 2014) improve on prior work by preventing theft. (Meiklejohn and Orlandi, 2015) shows a rigorous proof of anonymity for a scheme "almost identical" to CoinShuffle.

CoinShuffle's anonymity set is thought to be small due to coordination costs (Bissias et al., 2014; Bonneau et al., 2015); meanwhile, our schemes are not limited to small anonymity sets. Moreover, both CoinShuffle and CoinJoin run an entire mix in a single bitcoin transaction. Thus, a single aborting user disrupts the mix for all other users. Moreover, mix users cannot be forced to pay fees upfront, so that these schemes are vulnerable to DoS attacks (Bonneau et al., 2015; Tschorsch and Scheuermann, 2016) (where users join the mix and then abort) and Sybil attacks (where an adversary deanonymizes a user by forcing it to mix with Sybil identities secretly under her control) (Bissias et al., 2014).

XIM (Bissias et al., 2014) is a decentralized protocol which builds on the fair-exchange mixer in (Barber et al., 2012) and prevents bitcoin theft and resists DoS and Sybil attacks via fees. We also prevent bitcoin theft resist DoS and Sybil attacks with fees (Section 4.4.1). One of XIM's key innovations is a secure method for partnering mix users. Unfortunately, this partnering method adds several hours to the protocol execution because users have to advertise themselves as mix partners on the blockchain. Our schemes are faster because they do not require a partnering service.

CoinSwap (Maxwell, 2013b) is a fair-exchange mixer that allows two parties to anonymously send Bitcoins through an intermediary. Like our schemes, the CoinSwap intermediary is prevented from stealing funds by the use of fair exchange. Unlike our schemes, however, CoinSwap does not provide anonymity against even a honest but curious intermediary. Our on-blockchain scheme takes ≈ 30 mins, slower than Coinshuffle's ≈ 10 mins. Off-blockchain however, our scheme is faster than CoinShuffle, since it only runs in seconds (Poon and Dryja, 2015); however, our off-blockchain only supports anonymity against a honest-but-curious intermediary[1].

4.2 Overview and Security Properties

We introduce two schemes: (a) on-blockchain anonymous payments and coin mixing, and (b) off-blockchain anonymous payments. By on-blockchain we denote the standard method of transferring bitcoins *i.e.,* using the Bitcoin blockchain, as opposed to the newly proposed "off-blockchain" methods that utilize micropayment channel networks.

On-blockchain Anonymous Payments. We first consider the scenario where a user $\mathcal{A}$, the *payer* wants to anonymously send 1 bitcoin, *btc*, to another user $\mathcal{B}$, the *payee*[2]

If $\mathcal{A}$ were to perform a standard Bitcoin transaction, sending 1 bitcoin from an address $Addr_A$ (owned by $\mathcal{A}$) to a fresh ephemeral address $Addr_B$ (owned by $\mathcal{B}$) there would be a record of this transaction on Bitcoin's blockchain linking $Addr_A$ to $Addr_B$. Even if $\mathcal{A}$ and $\mathcal{B}$ always create a fresh address for each payment they receive, the links between addresses can be used to de-anonymize users if, at some point, they "non-anonymously" spend a payment (*e.g.,* buying goods from third party that learns their mailing address) or receive a payment (*e.g.,* a Bitcoin payment processor like

[1] Our off-blockchain scheme is fast because it uses micropayment channel networks. It's unclear how to retrofit prior work onto these networks, *e.g.,* mapping Coinshuffle's single atomic transaction onto the arbitrary graph topology of a micropayment channel network.

[2] We assume that all transactions in our schemes are of 1 bitcoin value.

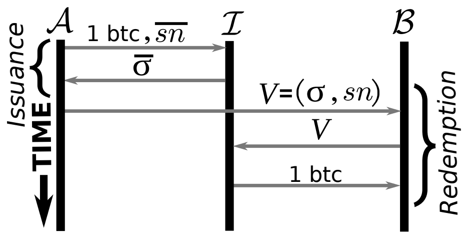


Figure 4·1: Strawman eCash protocol.

BitPay) (Meiklejohn et al., 2013).

One idea $\mathcal{A}$ and $\mathcal{B}$ could use to protect their privacy is to employ an intermediary party $\mathcal{I}$ that breaks the link between them. $\mathcal{A}$ would first send one bitcoin to $\mathcal{I}$, and then $\mathcal{I}$ would send a different bitcoin to $\mathcal{B}$. Assuming that a sufficient number of users make payments through $\mathcal{I}$, it becomes more difficult for an outsider to link $\mathcal{A}$ to $\mathcal{B}$ by looking at the blockchain (more on this below). The downside of this idea, however, is that the intermediary $\mathcal{I}$ knows everything about all users' payments, violating their anonymity.

We could apply techniques used in online anonymous eCash schemes (Chaum, 1983a) to prevent $\mathcal{I}$ from learning who $\mathcal{A}$ wants to pay. The protocol is in Figure 4·1. $\mathcal{A}$ pays one bitcoin to $\mathcal{I}$, and obtains an *anonymous voucher* $V = (sn, \sigma)$ in return. ($\mathcal{A}$ chooses a random serial number sn, blinds it to $\overline{sn}$ and asks $\mathcal{I}$ to compute a blind signature $\overline{\sigma}$ on $\overline{sn}$. $\mathcal{A}$ unblinds these values to obtain $V = (sn, \sigma)$. The blind signature requires only a minor change to Bitcoin and can be implemented using a soft fork (Section 4.3). Then $\mathcal{A}$ pays $\mathcal{B}$ using V, and finally $\mathcal{B}$ redeems V with $\mathcal{I}$ to obtain one bitcoin.

How do we ensure that $\mathcal{I}$ does not know who $\mathcal{A}$ wants to pay? This follows from the *blindness* of blind signatures—namely, that the signer ($\mathcal{I}$) cannot read the blinded serial number $\overline{sn}$ that it signs, and also cannot link a message/signature (sn, σ) pair to its blinded value $(\overline{sn}, \overline{\sigma})$. Blindness therefore ensures that even a malicious $\mathcal{I}$

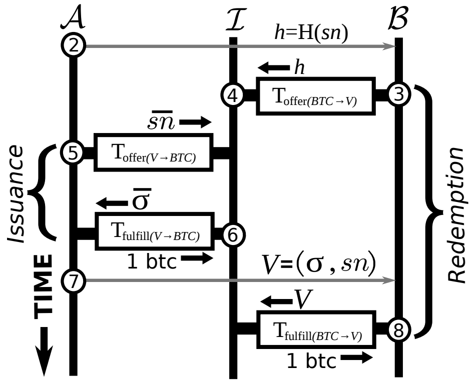


Figure 4·2: Our protocol: Circles (step numbers from Section 4.4), black arrows (objects transfered via transaction), grey arrows (messages).

cannot link a voucher it redeems with a voucher it issues. Blind signatures are also *unforgeable*, which ensures that a malicious user cannot issue a valid voucher to itself.

While this eCash-based approach solves our anonymity problem, it fails when $\mathcal{I}$ is malicious since it could just refuse to issue a voucher to $\mathcal{A}$ after receiving her bitcoin. To solve this, we use Bitcoin transaction contracts to achieve blockchain-enforced *fair exchange* (as in prior work, fair-exchange denotes an atomic swap). The key idea is that $\mathcal{A}$ transfers a bitcoin to $\mathcal{I}$ *if and only if* it receives a valid voucher V in return. Figure 4·2 presents the high-level idea, and full description is in Section 4.4.

At a high-level, our scheme consists of four blockchain transactions that are confirmed in three blocks on the blockchain, as shown in Figures 4·2-4·3. The protocol involves two blockchain-enforced fair exchanges. The first is $V\!\rightarrow\! btc$, which exchanges a voucher from $\mathcal{B}$ for a bitcoin from $\mathcal{I}$, and is realized using the following two transaction contracts: (1) $T_{offer(V\rightarrow btc)}$, which is created by $\mathcal{I}$, confirmed in the first block on the blockchain and offers a fair exchange of one bitcoin (from $\mathcal{I}$) for one voucher (from $\mathcal{B}$), and (2) $T_{fulfill(V\rightarrow btc)}$, which is created by $\mathcal{B}$ to fulfill the offer by $\mathcal{I}$ and is confirmed in the third block on the blockchain. These transaction contracts ensure that a malicious $\mathcal{I}$ cannot redeem $\mathcal{B}$'s voucher without providing $\mathcal{B}$ with a bitcoin

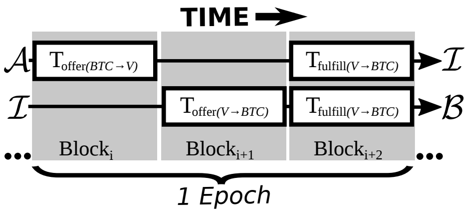


Figure 4·3: Payment Epoch

in return (see Sections 4.3,4.4). The second fair exchange is $btc \to V$ and works in a similar fashion, fairly exchanging a bitcoin from $\mathcal{A}$ for a voucher from $\mathcal{I}$ via two transaction contracts: (1) $T_{\mathit{offer}(btc \to V)}$, created by $\mathcal{A}$ and confirmed on the second block, and (2) $T_{\mathit{fulfill}(btc \to V)}$, created by $\mathcal{I}$ and confirmed in the third block. These two fair exchanges are arranged to realize the anonymity protocol shown on the previous page; the fair exchange $btc \to V$ stands in for the interaction between $\mathcal{A}$ and $\mathcal{I}$, while the fair exchange $V \to btc$ stands in for the interaction between $\mathcal{B}$ and $\mathcal{I}$.

Mixing Service. A mixing service allows a user to move bitcoins from one address it controls to a fresh ephemeral (thus anonymous) address, without directly linking the two addresses on the blockchain. To use our on-blockchain anonymous payments as a mixing service, users can just anonymously pay themselves from one address to another fresh ephemeral address, thus playing the role of both $\mathcal{A}$ and $\mathcal{B}$ in the protocol above.

Off-blockchain Payments. We also adapt our scheme to the recently proposed off-blockchain *micropayment channel networks.* Our off-blockchain scheme uses the same four transactions described above, but confirms them on a micropayment channel network. See Section 4.5 for details.

4.2.1 Anonymity Properties

In the strawman eCash protocol of Figure 4·1, the anonymity level of users depends on the total number of payments using $\mathcal{I}$ as users can obtain or redeem vouchers at arbitrary times. However, our anonymous fair-exchange protocol of Figure 4·2 provides anonymity only for payments starting and completing within an *epoch* (Figure 4·3) *i.e.,* a three block window.

Assumptions. We make the following assumptions for our schemes:

1. We assume that all users coordinate on epochs (by *e.g.,* choosing the starting block to have a block height that is divisible by three).

2. As with traditional eCash schemes, we assume that if $\mathcal{A}$ pays $\mathcal{B}$, then $\mathcal{A}$ and $\mathcal{B}$ trust each other. (A malicious $\mathcal{A}$ or $\mathcal{B}$ could easily conspire with $\mathcal{I}$ to reveal the other party of the transaction; for instance, $\mathcal{A}$ could just tell $\mathcal{I}$ the serial number the voucher she was issued, and then $\mathcal{I}$ can identity $\mathcal{B}$ when he redeems that voucher.) This is a reasonable assumption in cases where $\mathcal{A}$ is purchasing goods from $\mathcal{B}$, since $\mathcal{A}$ is likely already trusting $\mathcal{B}$ with far more personal and identifying information including *e.g.,* her shipping address or IP address.

3. For our on-blockchain scheme only, payees $\mathcal{B}$ always receive payments in a fresh ephemeral Bitcoin address $Addr_B$ controlled by them. Any communication between $Addr_B$ and $\mathcal{I}$ is done anonymously (*e.g.,* using Tor). The payee can transfer the payment from $Addr_B$ to his long-lived Bitcoin address if the protocol successfully completes.

4. Payers only make one anonymous payment per epoch. Similarly, payees only accept one payment per epoch (*i.e.,* we assume they do not create multiple ephemeral addresses to receive multiple payments in one epoch). [3]

Given these assumptions, the anonymity properties of our on-blockchain scheme are:

[3] We could allow users to perform multiple payments (by using multiple Bitcoin addresses that belong to them) but this would reduce their anonymity and make our analysis more complex.

Set-Anonymity within an Epoch. Our assumptions imply that in every epoch there are exactly n addresses making payments (playing the role of payer $\mathcal{A}$) and n receiving addresses (playing the role of $\mathcal{B}$). All these Bitcoin addresses should belong to different users. Anyone looking at the blockchain can see the participating addresses of payers and payees, but should not be able to distinguish which payer paid which payee within a specific epoch. Thus, for all successfully completed payments within an epoch, the offered anonymity set has size n. In other words, the probability of successfully linking any chosen payer $\mathcal{A}$ to a payee should not be more than $1/n$ plus some negligible function. This means that an adversary (or a potentially malicious $\mathcal{I}$) can do no better than randomly guessing who paid whom during an epoch.

Resilient Anonymity.

All payments should be totally anonymous until the recipient, $\mathcal{B}$, chooses to transfer them to an address linkable to $\mathcal{B}$. Even if a party *aborts* our protocol before it completes in an epoch, the *intended* recipient of a payment should remain totally anonymous.

Transparency of Anonymity Set. Users in our on-blockchain scheme learn the membership of their anonymity set after a transaction completes, just like anyone else who might be looking at the blockchain. This property is unusual for eCash schemes, but quite common for bitcoin mixes. Thus, if a particular $\mathcal{B}$ feels his anonymity set is too small in one epoch, he can increase the size of his anonymity set by remixing in a subsequent epoch. For instance, if $Addr_B$ gets paid in an epoch with $n = 4$, he can create a fresh ephemeral address $Addr'_B$ and have $Addr_B$ pay $Addr'_B$ in a subsequent epoch. If the subsequent epoch has a $n = 100$, then $\mathcal{B}$ increases the size of his anonymity set.

Our on-blockchain protocol achieves all the above anonymity properties, which also generalize to our mixing service (Section 4.4.2). Our mixing service has the additional advantage that $\mathcal{A}$ does not need to trust $\mathcal{B}$ since they are the same user. Our off-blockchain scheme only offers set-anonymity against $\mathcal{I}$ when $\mathcal{I}$ is honest-but-curious, rather than malicious (Section 4.5.3). Additionally, our off-blockchain scheme does not achieve the anonymity-set transparency or the anonymity resilience property.

Remark: Intersection Attacks. Anyone observing the anonymity-set membership in each epoch can attempt *intersection attacks* that de-anonymize users across epochs (*e.g.,* frequency analysis). This follows because we are *composing* set-anonymity across multiple epochs, and is a downside of any mix-based service that composes across epochs. ((Bissias et al., 2014) has a detailed description of intersection attacks.) By anonymity transparency, anyone looking at the blockchain can attempt an intersection attack on our on-blockchain scheme. Our off-blockchain scheme (roughly) only allows $\mathcal{I}$ to do this (Section 4.5.3).

4.2.2 Security properties

Fair-exchange. There will always be a *fair-exchange* between $V \leftrightarrow btc$. Our property ensures that: (i) malicious intermediary $\mathcal{I}$ cannot obtain a bitcoin from $\mathcal{A}$ unless it honestly creates a voucher for her, and (ii) malicious intermediary $\mathcal{I}$ cannot obtain a voucher V from $\mathcal{B}$ and refuse to pay a bitcoin back. This property is also true against malicious users: (iii) malicious $\mathcal{A}$ cannot refuse to give a bitcoin to $\mathcal{I}$ when receiving V, and (iv) malicious $\mathcal{B}$ cannot receive a bitcoin from $\mathcal{I}$ without presenting a (valid) V.

Unforgeability. A user cannot create a valid V without interacting with $\mathcal{I}$.

Double-spending Security. A user can not redeem the same V more than once.

DoS Resistance. The intermediary $\mathcal{I}$ should be resistant to Denial of Service (DoS) attacks where a malicious user starts but never finishes many parallel fair exchanges (redemptions) of a V for a btc.

Sybil Resistance. The protocol should be resistant to a Sybils (*i.e.,* identities that are under the control of single user) that attempt to de-anonymize a target user.

4.3 Implementing fair exchange via scripts and blind signatures

We explain how the transaction contracts $T_{\mathit{offer}(btc \to V)}$ and $T_{\mathit{fulfill}(btc \to V)}$ implement the fair exchange $btc \to V$ used in our protocol ($V \to btc$ is analogous).

We start with some background on transaction contracts. Recall that Bitcoin has no inherent notion of an "account"; instead, users merely move bitcoins from old transactions to new transactions, with the blockchain providing a public record of all valid moves. To do this, each transaction contains a list of *outputs.* These outputs hold a portion of that transaction's bitcoins and a set of rules describing the conditions under which the portioned bitcoins in that output can be transferred to a new transaction. The rules for spending outputs are written in a non-Turing-complete language called *Script.* One transaction *spends* another transaction when it successfully satisfies the rules in a script. Transaction contracts (aka smart contracts (Szabo, 1997)) are written as scripts, *e.g.,* $\mathcal{A}$ will only pay $\mathcal{B}$ if some condition is met. Using the CHECKLOCKTIMEVERIFY feature (Todd, 2014) of scripts, we can *timelock* a transaction, so that funds can be reclaimed if a contract has not been spent within a given time window tw.

We use timelocking to implement the $btc \to V$ fair exchange. The fair exchange begins when a user $\mathcal{A}$ generates (and the blockchain confirms) a transaction contract $T_{\mathit{offer}(btc \to V)}$ which says that $\mathcal{A}$ offers one bitcoin to $\mathcal{I}$ under the condition "$\mathcal{I}$ must compute a valid blind signature on the blinded serial number $\overline{sn}$ within time window tw"; if the condition is not satisfied, the bitcoin reverts to $\mathcal{A}$. More precisely, $\mathcal{A}$ first chooses a random serial number sn, blinds it to $\overline{sn}$, and then uses $\overline{sn}$ to create a transaction contract $T_{\mathit{offer}(btc \rightarrow V)}$ with an output of one bitcoin that is spendable in a future transaction T_f if one of the following conditions is satisfied:

1. T_f is signed by $\mathcal{I}$ and contains a valid blind signature $\overline{\sigma}$ on $\overline{sn}$[4], or

2. T_f is signed by $\mathcal{A}$ and the time window tw has expired.

The contract $T_{\mathit{offer}(btc \rightarrow V)}$ is *fulfilled* if $\mathcal{I}$ posts a transaction $T_f = T_{\mathit{fulfill}(btc \rightarrow V)}$ that contains a valid blind signature $\overline{\sigma}$ on $\overline{sn}$. This would satisfy the first condition of $T_{\mathit{offer}(btc \rightarrow V)}$ and so the offered bitcoin is transferred from $\mathcal{A}$ to $\mathcal{I}$. If $\mathcal{I}$ does not fulfill the contract within the time window tw, then $\mathcal{A}$ signs and posts a transaction T_f that returns the offered bitcoin back to $\mathcal{A}$, thus satisfying the second condition of $T_{\mathit{offer}(btc \rightarrow V)}$.

Blind Signature Scheme. Our fair exchange requires blind signatures with exactly two rounds of interaction. We use Boldyreva's (Boldyreva, 2003) scheme, instantiated with elliptic curves for which the Weil or Tate pairing are efficiently computable and the computational Diffie-Hellman problem is sufficiently hard. While bitcoin supports elliptic curve operations, it uses a curve (Secp256k1) that does not support the required bilinear pairings. Thus, we need a soft fork to add an opcode that supports elliptic curves with efficient bilinear pairings (*i.e.,* supersingular curves of the type $y^2 = x^3 + 2x \pm 1$ over $\mathbb{F}_{e^\ell}$).

We use standard multiplicative notation and overlines to denote blinded values. Let $\mathbb{G}$ be a cyclic additive group of prime order p in which the gap Diffie-Hellman problem (Boneh et al., 2001) is hard and $\mathbb{G}'$ a cyclic multiplicative group of prime order q. By e we denote the bilinear pairing map: $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}'$. Let g be a generator of the group and H be a hash function mapping arbitrary strings to elements of $\mathbb{G}\backslash\{1\}$. Let (p, g, H) be public parameters and $(sk, pk = g^{sk})$ be the signer's secret/public key pair.

[4] $\mathcal{I}$ signs T_f to stop a malicious miner that learns $\overline{\sigma}$ from stealing the bitcoin $\mathcal{A}$ gives $\mathcal{I}$.

- To blind sn, user $\mathcal{A}$ picks random $r \in \mathbb{Z}_p^*$ and sets $\overline{sn} = H(sn)g^r$.
- To sign $\overline{sn}$, signer $\mathcal{I}$ computes $\overline{\sigma} = \overline{sn}^{sk}$.
- To unblind the blind signature $\overline{\sigma}$, user $\mathcal{A}$ computes $\sigma = \overline{\sigma} pk^{-r}$.
- To verify the signature σ on sn, anyone holding pk checks that the bilinear pairing $e(pk, H(sn))$ is equal to $e(g, \sigma)$.
- To verify that the *blinded signature* $\overline{\sigma}$ on the blinded $\overline{sn}$, anyone holding pk can verify that this is valid (intermediate) signature by checking if $e(pk, \overline{m}) = e(g, \overline{\sigma})$.

4.4 On-Blockchain Anonymous Protocols

We now discuss the details of on-blockchain protocol depicted in Figure 4·2-4·3.

As shown in Figure 4·2, our protocol interleaves two fair exchanges $btc \rightarrow V$ (implemented using $T_{\mathit{offer}(V \rightarrow btc)}$ and $T_{\mathit{fulfill}(V \rightarrow btc)}$) and $V \rightarrow btc$ (implemented using $T_{\mathit{offer}(btc \rightarrow V)}$ and $T_{\mathit{fulfill}(btc \rightarrow V)}$). The interleaving is designed to ensure that a malicious $\mathcal{I}$ cannot issue a voucher V to $\mathcal{A}$ and then subsequently refuse to redeem V from $\mathcal{B}$. The key idea is that it is in the interest of both $\mathcal{A}$ and $\mathcal{B}$ to force $\mathcal{I}$ to *commit* to redeeming the voucher $V = (sn, \sigma)$. To do this $\mathcal{A}$ starts by choosing the serial number sn for the voucher and sending its hash $h = H(sn)$ to $\mathcal{B}$; notice that h hides the value of sn and thus does not harm anonymity. Then, $\mathcal{B}$ uses h to force $\mathcal{I}$ to commit to redeeming a voucher with serial number sn. Specifically, $\mathcal{B}$ asks $\mathcal{I}$ to create the transaction contract $T_{\mathit{offer}(V \rightarrow btc)}$ that offers one bitcoin to $\mathcal{B}$ under the condition "$\mathcal{B}$ must provide a *valid* voucher V with serial number sn such that $h = H(sn)$ within time window tw". To prevent double-spending, $\mathcal{I}$ agrees to create $T_{\mathit{offer}(V \rightarrow btc)}$ iff the hash value h does *not* match the h of any prior transaction contract that $\mathcal{I}$ has signed. Once

$T_{\mathit{offer}(V \rightarrow btc)}$ is on the blockchain, committing that $\mathcal{I}$ will redeem the voucher with serial number sn, our two fair exchanges proceed as in Figure 4·2.

The details of the scheme are as follows. Let k be the security parameter. We assume that $\mathcal{I}$ performs a one-time setup by posting public parameters on the blockchain. These parameters include the public parameters for the blind signature scheme, the fee value f and reward value w, and the time windows (tw_1, tw_2). (We define f, w below.)

1. $\mathcal{B}$ creates a fresh ephemeral Bitcoin address to receive the payment.

2. $\mathcal{A}$ randomly chooses $sn \overset{r}{\leftarrow} \{0,1\}^k$, computes $h \leftarrow H(sn)$ and sends h to $\mathcal{B}$.

3. $\mathcal{B}$ sends h to $\mathcal{I}$ and asks $\mathcal{I}$ to create transaction contract $T_{\mathit{offer}(V \rightarrow btc)}$ offering one bitcoin to $\mathcal{B}$ under condition: "$\mathcal{B}$ must provide a *valid* voucher V with serial number whose hash is equal to h within time window tw_2".

4. If h does *not* match any h from prior transaction contracts signed by $\mathcal{I}$, then $\mathcal{I}$ creates the requested contract $T_{\mathit{offer}(V \rightarrow btc)}$ and posts it to the blockchain.

5. $\mathcal{A}$ blinds sn to obtain $\overline{sn}$ and waits for $T_{\mathit{offer}(V \rightarrow btc)}$ to be confirmed on the blockchain. Then $\mathcal{A}$ creates transaction $T_{\mathit{offer}(btc \rightarrow V)}$, offering a $1 + w$ bitcoins to $\mathcal{I}$ under the condition "$\mathcal{I}$ must provide a valid blind signature on the blinded serial number $\overline{sn}$ within time window tw_1" (where $tw_1 > tw_2$ so that $\mathcal{I}$ cannot cheat by waiting until $T_{\mathit{offer}(btc \rightarrow V)}$ expires but $T_{\mathit{offer}(V \rightarrow btc)}$ has not).

6. To prevent $\mathcal{A}$ from double-spending the bitcoin offered in $T_{\mathit{offer}(btc \rightarrow V)}$, $\mathcal{I}$ waits until the blockchain confirms $T_{\mathit{offer}(btc \rightarrow V)}$. $\mathcal{I}$ then fulfills the $btc \rightarrow V$ fair exchange by creating transaction $T_{\mathit{fulfill}(btc \rightarrow V)}$ which contains the blinded signature $\overline{\sigma}$ on $\overline{sn}$. $T_{\mathit{fulfill}(btc \rightarrow V)}$ is posted to the blockchain, and transfers $(1 + w)$ bitcoins from $T_{\mathit{offer}(btc \rightarrow V)}$ to $\mathcal{I}$.

7. $\mathcal{A}$ learns $\overline{\sigma}$ from $T_{\mathit{fulfill}(btc \to V)}$, unblinds $\overline{\sigma}$ to σ and sends $V = (sn, \sigma)$ to $\mathcal{B}$.

8. $\mathcal{B}$ creates a transaction $T_{\mathit{fulfill}(V \to btc)}$ which contains the voucher $V = (sn, \sigma)$, and thus transfers the bitcoin in $T_{\mathit{offer}(V \to btc)}$ to $\mathcal{B}$. $T_{\mathit{fulfill}(V \to btc)}$ is posted to the blockchain and confirmed in the same block as $T_{\mathit{fulfill}(btc \to V)}$.

Rewards. $\mathcal{A}$ offers $1 + w$ bitcoins to $\mathcal{I}$ in $T_{\mathit{offer}(btc \to V)}$, but $\mathcal{I}$ only offers $\mathcal{B}$ 1 bitcoin in $T_{\mathit{offer}(V \to btc)}$. The remaining w bitcoin is kept by $\mathcal{I}$ as a "reward" for completing its role in the protocol. $\mathcal{I}$ cannot steal w because w is paid via a fair exchange.

4.4.1 Anonymous Fee Vouchers

Bitcoin transactions include a transaction fee that is paid to the miner who confirms the transaction in the blockchain; if this transaction fee is not paid or is too low, it is extremely unlikely that this transaction will be confirmed. Since $\mathcal{I}$ can not trust $\mathcal{B}$ or $\mathcal{A}$, $\mathcal{I}$ should not be required to cover the cost of the transaction fee for first transaction contract $T_{\mathit{offer}(V \to btc)}$ that $\mathcal{I}$ posts to the blockchain.

Following ideas from (Bissias et al., 2014), we have $\mathcal{A}$ buy a special *anonymous fee voucher* V' of value f bitcoin from $\mathcal{I}$. The value $f \ll 1$ should be very small and is set as a public parameter. Since fee vouchers are anonymous and have low value, $\mathcal{A}$ should buy them out-of-band in bulk with cash, credit or bitcoin. Then, whenever $\mathcal{A}$ wishes to mix or make an anonymous payment, $\mathcal{A}$ sends an anonymous fee voucher V' to $\mathcal{B}$, who in turn sends it to $\mathcal{I}$ with a request that $\mathcal{I}$ initiate the protocol. All this happens out-of-band. Note though, that the fee voucher V' is *not* created with a fair exchange, and thus $\mathcal{I}$ could steal f bitcoin by accepting V' but refusing to initiate the protocol. However, we argue that $\mathcal{I}$ has very little incentive to do this if, upon completing the protocol, $\mathcal{I}$ obtains a reward w that is significantly larger than f.

DoS Resistance. Fees raise the cost of an DoS attack where $\mathcal{B}$ starts and aborts many parallel sessions, locking $\mathcal{I}$'s bitcoins in many $T_{\mathit{offer}(V \to btc)}$ transaction contracts. This is because $\mathcal{B}$ must forward an anonymous fee voucher V' from $\mathcal{A}$ to $\mathcal{I}$ every time $\mathcal{B}$ wishes to initiate our protocol. This method also works for our mixing service, where $\mathcal{A}$ and $\mathcal{B}$ are the same user. Moreover, if a party aborts a run of our protocol during an epoch, this has no affect on other runs in that epoch. This is in contrast to (Ruffing et al., 2014; Maxwell, 2013a) where a single aborting player terminates the protocol for all parties in that mix.

Sybil Resistance. In a sybil attack, the adversary creates many sybil identities secretly under her control, and deanonymizes a target user by forcing the target to mix only with sybils (Bissias et al., 2014; Tschorsch and Scheuermann, 2016). To launch this attack on our protocol, the attacker could create m runs of our protocol (*i.e.,* m payers and payees) that occupy most of the intermediary $\mathcal{I}$'s resources, leaving only a single slot available for the targeted payer and payee. Again, we use fees to raise the cost of this attack, by requiring each of the m Sybil runs to pay a fee voucher of value f. If $\mathcal{I}$ performs a sybil attack, $\mathcal{I}$ avoids paying f but must pay all four transaction fees.

4.4.2 Anonymity Analysis

Before discussing the anonymity properties of our scheme we start by noting that in the first step of our protocol, $\mathcal{B}$ is always required to create a fresh ephemeral Bitcoin address $Addr_0$. Upon creation, this address is *completely anonymous*, in the sense that there is no way to link it to $\mathcal{B}$'s identity; this is a much stronger notion than the set anonymity defined in Section 4.2.1. Now suppose that $\mathcal{A}$ uses our protocol to pay a bitcoin to $Addr_0$. Then, as we will argue below, $Addr_0$ is now linkable to $\mathcal{A}$ with probability $1/n$ (if n payments happened in that epoch). However $Addr_0$ is still completely anonymous with respect to $\mathcal{B}$'s "Bitcoin identity", *i.e.,* the long-lived Bitcoin address that $\mathcal{B}$ uses to send and receive payments. If the funds from $Addr_0$ were paid into another fresh ephemeral Bitcoin address $Addr_1$ controlled by $\mathcal{B}$, these funds would still be unlinkable to $\mathcal{B}$. Indeed, the funds in $Addr_0$ only become linkable to $\mathcal{B}$ if they are transferred to an address controlled by $\mathcal{B}$ that already contains some bitcoins.

Set-Anonymity within an Epoch. Our on-blockchain payment scheme achieves an anonymity set of size n within an epoch, as defined in Section 4.2.1. Suppose that n payments successfully complete during an epoch, and recall that each payer may only perform one payment per epoch and each payment is made to a fresh ephemeral address. It follows that there are n payers and n payees during the epoch. Any adversary (including $\mathcal{I}$) observing the blockchain can see the following: n payers' addresses, n payees' addresses, and n sets of transactions of the type $T_{\mathit{offer}(btc \to V)}$, $T_{\mathit{fulfill}(btc \to V)}$, $T_{\mathit{offer}(V \to btc)}$, $T_{\mathit{fulfill}(V \to btc)}$. For the adversary to link a payer to a payee, it would need to link a $T_{\mathit{offer}(btc \to V)}$, $T_{\mathit{fulfill}(btc \to V)}$ pair $(btc \to V)$ to a $T_{\mathit{offer}(V \to btc)}$, $T_{\mathit{fulfill}(V \to btc)}$ pair $(V \to btc)$. Let us first examine what do these pairs of transaction contracts reveal on the blockchain. The $btc \to V$ pair reveals a blinded serial number $\overline{sn}$ and the corresponding intermediate (blinded) blind signature $\overline{\sigma}$. Meanwhile, the $V \to btc$ pair reveals a serial number sn and the corresponding signature σ. As long as the blinding factor of sn is not revealed, the blind signature ensures that no one can link an sn to an $\overline{sn}$. The signatures $\overline{\sigma}$ and σ are similarly unlinkable (except with some negligible probability $\nu(k)$). Thus, the adversary's best strategy is to randomly link a payer to a payee, which succeeds with probability $1/n + \nu(k)$.

The same analysis applies to our mixing service. Moreover, mix users can repeatedly rerun the mix over several epochs, thus boosting the size of their anonymity set beyond what could be provided during a single epoch.

Resilient Anonymity and Transparency of Anonymity Set. Ephemeral addresses prevent $\mathcal{I}$ from de-anonymizing a payment from $\mathcal{A}$ to $\mathcal{B}$ by aborting or denying service. Suppose $\mathcal{I}$ aborts by refusing to issue $T_{\mathit{fulfill}(btc \to V)}$ to $\mathcal{A}$ (Figure 4·2).

If this happens, $\mathcal{A}$ does not obtain voucher $V = (sn, \sigma)$ and cannot pass V on to $\mathcal{B}$. By the unforgeability of vouchers, it follows that $\mathcal{B}$ will not be able to issue a valid $T_{\textit{fulfill}(V \to btc)}$ that fulfills $T_{\textit{offer}(V \to btc)}$. Thus, $\mathcal{I}$ can de-anonymize the payment between $\mathcal{A}$ and $\mathcal{B}$ by matching the aborted exchange with $\mathcal{A}$ with the incomplete exchange with $\mathcal{B}$. As another possible attack, malicious $\mathcal{I}$ could instead refuse service to all payers apart from a target $\mathcal{A}$, and then identify $\mathcal{B}$ by finding the single $V \to btc$ exchange that completes during the epoch. Fortunately, however, anonymity-set transparency allows $\mathcal{B}$ to detect these attacks. $\mathcal{B}$ can recover by discarding the ephemeral address it used in the attacked epoch, and chose a fresh ephemeral address in a subsequent epoch.

Note that for both our payment and mixing service one could attempt an intersection attack as discussed in Section 4.2.1.

4.5 Off-Blockchain Anonymous Payments over Micropayment Channel Networks

We start by reviewing off-blockchain transactions via micropayment channel networks and then describe how to make our protocol faster by adapting it to work with them.

4.5.1 Micropayment Channel Networks

Micropayment Channels. To establish a pairwise micropayment channel, $\mathcal{A}$ and $\mathcal{B}$ each pay some amount of bitcoins into an *escrow transaction* T_e which is posted to the blockchain. This escrow transaction is on-blockchain and therefore slow ($\approx$ 10 minutes), but all subsequent transactions are off-blockchain and therefore fast ($\approx$ seconds). T_e ensures that no party reneges on an off-blockchain transaction. Suppose x bitcoins are paid into T_e. T_e offers these x bitcoins to be spent under condition: "The spending transaction is signed by both $\mathcal{A}$ and $\mathcal{B}$". Then, the spending transaction T_r has the form: "a bitcoins are paid to $\mathcal{A}$ and b bitcoins are paid to $\mathcal{B}$" where a and b reflect the agreed-upon balance of bitcoins between $\mathcal{A}$ and $\mathcal{B}$.

Once T_e is confirmed on the blockchain, $\mathcal{A}$ and $\mathcal{B}$ can transfer funds between themselves off-blockchain by signing a spending transaction T_r. Importantly, T_r is *not* posted to the blockchain. Instead, the existence of T_r creates a credible threat that either party can claim their allocated bitcoins by posting T_r to the blockchain; this prevents either party from reneging on the allocation reflected in T_r. To continue to make off-blockchain payments, $\mathcal{A}$ and $\mathcal{B}$ just need to sign a new transaction T_r' that reflects the new balance of bitcoins a' and b'. Micropayment channels have mechanisms that ensure that this later transaction T_r' always supersedes an earlier transaction T_r. Our protocol applies generically to any micropayment channel with such a mechanism, *e.g.,* Lightning Network (Poon and Dryja, 2015), Duplex Micropayment Channels (DMC) (Decker and Wattenhofer, 2015).

Micropayment Channel Networks. Micropayment channel networks are designed to avoid requiring each *pair* of parties to pre-establish a *pairwise* micropayment channel between them. Indeed, such a requirement would be infeasible, since it requires each pair of users to lock funds into many different escrow transactions T_e on the blockchain. Instead, suppose a pair of users $\mathcal{A}$ and $\mathcal{B}$ are connected by a path of users with established pairwise micropayment channels (*i.e.,* $\mathcal{A}$ has a channel with $\mathcal{A}_1$, $\mathcal{A}_1$ has a channel with $\mathcal{A}_2$, ..., $\mathcal{A}_{m-1}$ has a channel with $\mathcal{A}_m$, $\mathcal{A}_m$ has a channel with $\mathcal{B}$). Then, the path of users can run a protocol to transfer funds from $\mathcal{A}$ to $\mathcal{B}$. However, it will not suffice to simply have each user $\mathcal{A}_i$ create a transaction paying the next user $\mathcal{A}_{i+1}$ in the path, since a malicious user $\mathcal{A}_k$ could steal funds by failing to create a a transaction for $\mathcal{A}_{k+1}$. Instead, the Lightning Network and DMC use a protocol based on *hash timelocked contracts* or HTLCs. A transaction T is an HTLC if it offers bitcoins under the condition: "The spending transaction must contain the preimage of y and be confirmed within timewindow tw", where $y = H(x)$ and x is a random value, *i.e.,* the preimage. We say that T is *locked under the preimage of y.*

Micropayment channels use HTLCs as follows. Suppose the existing balance between $\mathcal{A}$ and $\mathcal{B}$ is a bitcoin for $\mathcal{A}$ and b bitcoin for $\mathcal{B}$. Now suppose that $\mathcal{A}$ wants to transfer ϵ bitcoin to $\mathcal{B}$, updating the balances to $a - \epsilon$ and $b + \epsilon$. First, $\mathcal{B}$ chooses a random value x, computes $y = H(x)$, and announces y to everyone in the path. Then, $\mathcal{A}$ asks each pair of parties $(\mathcal{A}_i, \mathcal{A}_{i+1})$ on the path to transfer ϵ bitcoin *locked under the preimage of* y using the micropayment channel from $\mathcal{A}_i$ to $\mathcal{A}_{i+1}$. The mechanics of the transfer between $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$ are as follows. Suppose the existing balance between $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$ is c bitcoin for $\mathcal{A}_i$ and d bitcoin for $\mathcal{A}_{i+1}$. Then $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$ jointly sign a new spending transaction T'_r of the form "$c - \epsilon$ bitcoins are paid to $\mathcal{A}_i$ and $d + \epsilon$ bitcoins are paid to $\mathcal{A}_{i+1}$" under the condition "the spending transaction contains the preimage of y within timewindow tw". Once $\mathcal{A}$ sees that all the transactions on the path have been signed, it releases the preimage x to the path and the funds flow from $\mathcal{A}_i$ to $\mathcal{A}_j$. If any user refuses to sign a transaction, the timelock tw allows all signing users to reclaim their funds. The timelock is decremented along the path to prevent race conditions. This entire protocol occurs off-blockchain, with x and the HTLCs creating a credible threat that users can reclaim their funds if they are posted to the blockchain.

4.5.2 Anonymizing Micropayment Channel Networks

As a strawman for anonymous transactions in micropayment channel networks, we can replace the hash lock with the transaction contracts conditions in that we use in $T_{\mathit{offer}(V \to btc)}$ and $T_{\mathit{offer}(btc \to V)}$ (see Section 4.4). The protocol assumes paths of intermediate channels $(path_1, path_2)$ connecting $\mathcal{A}$ to $\mathcal{I}$ and $\mathcal{I}$ to $\mathcal{B}$ respectively, and has *Setup* phase as in our original on-blockchain protocol.

1. $\mathcal{A}$ chooses a random serial number sn, hashes it to $h = H(sn)$ and sends h to $\mathcal{B}$.

2. $\mathcal{B}$ uses h to lock a path of micropayment channels ($path_2$) to $\mathcal{I}$ under the condition: "The spending transaction must provide a *valid* voucher V with serial number whose hash is equal to h within time window tw_2."

3. $\mathcal{A}$ blinds the serial number sn to obtain $\overline{sn}$. $\mathcal{A}$ asks $\mathcal{B}$ to confirm that each party on $path_2$ from $\mathcal{A}$ to $\mathcal{I}$ has properly locked the path. Then, $\mathcal{A}$ asks $\mathcal{I}$ to lock a path $path_1$ of micropayment channels between $\mathcal{A}_i$ and $\mathcal{I}$ under the condition: "The spending transaction must provide a valid blind signature on the blinded serial number $\overline{sn}$ within time window tw_1" where $tw_1 > tw_2$.

4. $\mathcal{I}$ then reveals $\overline{\sigma}$ to every party on $path_1$, unlocking the path from $\mathcal{I}$ to $\mathcal{A}$. $\mathcal{A}$ obtains $\overline{\sigma}$, unblinds it to σ and thus obtains the voucher $V = (\sigma, sn)$. $\mathcal{A}$ sends $V = (\sigma, sn)$ to $\mathcal{B}$ who releases it to every party on path $path2$, unlocking the path from $\mathcal{I}$ to $\mathcal{B}$.

We again need the notion of an epoch. Since we do not have blocks to coordinate these epochs, we instead use synchronized clocks. We break an epoch of q seconds into three equal divisions of $\frac{q}{3}$ seconds long. $path_2$ is set up in first division, $path_1$ is set up second and $path_1$ and $path_2$ are resolved in third division. Also, we can add anonymous fee vouchers to this protocol, since fee vouchers are redeemed out of band (Section 4.4.1).

4.5.3 Anonymity Analysis

Of the properties in Section 4.2.1, our off-blockchain scheme only supports set anonymity within an epoch (as discussed in Section 4.4.2) when $\mathcal{I}$ is *honest-but-curious.* That is, $\mathcal{I}$ follows the protocol without aborting or denying services to other payers and payees, but is still curious to learn which payer is paying which payee. However, we still support fair-exchange against a malicious $\mathcal{I}$ (Section 4.6) as well as set-anonymity within an epoch against malicious third parties.

We only support anonymity against honest-but-curious $\mathcal{I}$ because we cannot use fresh ephemeral addresses in this off-blockchain context. This follows because choosing a fresh address amounts to establishing a fresh micropayment channel. Because this requires a fresh escrow transaction T_e to be posted on the blockchain (taking $\approx$ 10 minutes), it obviates the speed benefits of the off-blockchain scheme. Recall that $\mathcal{B}$ discards its ephemeral address in order to recover from an epoch where a malicious $\mathcal{I}$ de-anonymized the payment from $\mathcal{A}$ to $\mathcal{B}$ by aborting or denying service (Section 4.4.2).

We have also given up on anonymity transparency. Because transactions are no longer posted on the blockchain, even users that participate in the protocol cannot learn the size or membership of their anonymity set.

Proxy Addresses. We need the notion of *proxy addresses* to ensure that no parties (other than $\mathcal{I}$) can break anonymity by behaving maliciously. Notice that in a micropayment channel network, a malicious user $\mathcal{A}_i$ along the path $path_1$ from $\mathcal{A}$ to $\mathcal{I}$ can abort the protocol by refusing to create the appropriate transactions. Now if $\mathcal{A}_i$ is also on the path $path_2$ from $\mathcal{I}$ to $\mathcal{B}$, then $\mathcal{A}_i$ can abort the protocol and de-anonymize $\mathcal{A}$ and $\mathcal{B}$ in the same way that $\mathcal{I}$ can. To prevent this attack, we need to make sure that $\mathcal{I}$ is the only party that is on both $path_1$ and $path_2$. The idea is that every user $\mathcal{B}$ of our system has an additional proxy address $Addr_B^{px}$, and uses this address to establish, just once, a (reusable) micropayment channel directly to $\mathcal{I}$. This ensures that $path_2$ consists of only $\mathcal{I}$ and $Addr_B^{px}$. Then, $\mathcal{B}$ will receive payments to its proxy address $Addr_B^{px}$ using the strawman protocol of Section 4.5.2 in a one epoch. In the subsequent epoch, $\mathcal{B}$ will rerun the strawman protocol to transfer funds from $Addr_B^{px}$ (acting as user $\mathcal{A}$) to its long-lived address $Addr_B$ (acting as user $\mathcal{B}$).

Intersection Attacks. The lack of anonymity set transparency and the use of proxy addresses implies that only $\mathcal{I}$ can observe the full membership of the anonymity set during each epoch. As payments between proxy $Addr_B^{px}$ and identity $Addr_B$ addresses occur in contiguous epochs, $\mathcal{I}$ could use an intersection attack (Bissias et al., 2014) to infer their relationship. Other adversaries only observe off-blackchain transactions flowing through them.

4.6 Security Analysis

Fair-Exchange Our schemes prevent parties from stealing from each other.

1. The $btc{\rightarrow}V$ fair exchange (Section 4.3) ensures that (1) $\mathcal{I}$ cannot steal $\mathcal{A}$'s bitcoin without issuing her a valid voucher V, and (2) $\mathcal{A}$ cannot refuse to pay $\mathcal{I}$ a bitcoin upon receiving a V. (Fair exchange properties (i) and (iii) from Section 4.2.2.)

2. The $V{\rightarrow}btc$ fair exchange ensures that $\mathcal{B}$ cannot steal $\mathcal{I}$'s bitcoins without actually redeeming V. Also, $\mathcal{I}$ cannot refuse to redeem a $V = (sn, \sigma)$ that it issued to $\mathcal{A}$. This follows because, as discussed in Section 4.4, $\mathcal{I}$ commits to the redemption of V when it posts $T_{\mathit{offer}(V \rightarrow btc)}$ (which contains h, where $h = H(sn)$). Moreover, recall that $T_{\mathit{fulfill}(btc \rightarrow V)}$ is transaction where (a) $1 + w$ bitcoin are transferred from $\mathcal{A}$ to $\mathcal{I}$, and (b) $\mathcal{I}$ issues V by providing the blind signature $\overline{\sigma}$. Since $T_{\mathit{offer}(V \rightarrow btc)}$ is posted to the blockchain before $T_{\mathit{offer}(V \rightarrow btc)}$, it follows that $\mathcal{A}$ does not pay $\mathcal{I}$ for V until $\mathcal{I}$ has committed to redeeming V. (Fair exchange properties (ii) and (iv) from Section 4.2.2.)

3. $\mathcal{I}$ cannot prevent $\mathcal{B}$ from redeeming V by issuing V just before $T_{\mathit{offer}(V \rightarrow btc)}$ expires. This follows because $\mathcal{A}$ choose tw_1 such that $tw_1 > tw_2$ which ensures

that $T_{\mathit{offer}(btc \to V)}$ expires earlier that $T_{\mathit{offer}(V \to btc)}$. This way, if $\mathcal{I}$ takes too long to issue $T_{\mathit{offer}(btc \to V)}$, $\mathcal{A}$ will have already reclaimed her refunds. (Fair exchange property (ii) from Section 4.2.2.)

Unforgeability and Double-spending. Unforgeability follows from the underlying blind signature scheme, which ensures that only the intermediary $\mathcal{I}$ can issue vouchers $V = (sn, \sigma)$. Moreover, vouchers cannot be double-spent because if $\mathcal{I}$ has previously seen $h = H(sn)$, $\mathcal{I}$ will refuse to post $T_{\mathit{offer}(V \to btc)}$.

DoS and Sybil Resistance. Both our on- and off-blockchain schemes support anonymous fee vouchers, and thus resist DoS and sybil attacks (Section 4.4.1).

Chapter 5

Tumblebit: An Untrusted Bitcoin-compatible Anonymous Payment Hub

In this chapter, we will discuss (Heilman et al., 2017) in which we developed new tumbler protocols for Bitcoin. This work was written in collaboration with Leen AlShenibr, Foteini Baldimtsi, Alessandra Scafuro and Sharon Goldberg.

5.1 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 (Meiklejohn et al., 2013; Ron and Shamir, 2013), and a blockchain surveillance industry (Limited, 2016; Inc, 2016), that have demonstrated weaknesses in Bitcoin's anonymity properties. As a result, a new market of anonymity-enhancing services has emerged (Möser and Böhme, 2016; Grams, 2016; wikipedia, 2016); for instance, 1 million USD in bitcoins are funneled through JoinMarket each month (Möser and Böhme, 2016). 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 $\mathcal{I}$) 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 (Noubir and Sanatinia, 2016). 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 (Stone, 2015), 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* T, to enhance anonymity. Every payment made via TumbleBit is backed by bitcoins. We use cryptographic techniques to guarantee Tumbler 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 $(\mathcal{I}_1, ..., \mathcal{I}_Q)$ 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 T, and complete in seconds. (Our implementation[1] completed a payment in 1.2 seconds, on average, when T was in New York and $\mathcal{A}$ and $\mathcal{I}$ were in Boston.)

TumbleBit Overview. TumbleBit replaces on-blockchain payments with off-blockchain puzzle solving, where Alice $\mathcal{A}$ pays Bob $\mathcal{I}$ by providing $\mathcal{I}$ with the solution to a puzzle. The puzzle z is generated through interaction between $\mathcal{I}$ and T, and

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

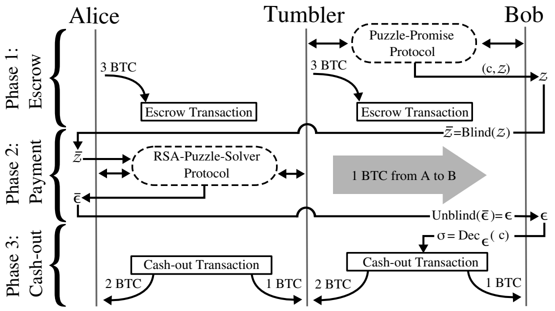


Figure 5·1: Overview of the TumbleBit protocol.

solved through an interaction between $\mathcal{A}$ and T. Each time a puzzle is solved, 1 bitcoin is transferred from Alice $\mathcal{A}$ to the Tumbler T and finally on to Bob $\mathcal{I}$.

The protocol proceeds in three phases; see Figure 5·1. In the on-blockchain *Escrow Phase*, each payer Alice $\mathcal{A}$ opens a payment channel with the Tumbler T by escrowing Q bitcoins on the blockchain. Each payee Bob $\mathcal{I}$ also opens a channel with T. This involves (1) T escrowing Q bitcoins on the blockchain, and (2) $\mathcal{I}$ and T engaging in a *puzzle-promise* protocol that generates up to Q puzzles for $\mathcal{I}$. 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 T to learn the solution to a puzzle $\mathcal{I}$ provided. Finally, the *Cash-Out Phase* closes all payment channels. Each payee $\mathcal{I}$ uses his Q' solved puzzles (*aka,* TumbleBit payments) to create an on-blockchain transaction that claims Q' bitcoins from T's escrow. Each payer $\mathcal{A}$ also closes her escrow with 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 T does not collude with other TumbleBit users, then 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 (Chaum, 1983a). 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 $\mathcal{I}$. Finally $\mathcal{I}$ 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 T playing the role of the Chaumian Bank. However, while Tumbler 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 $\mathcal{I}$). In this mode, TumbleBit is run as in Figure 5·1 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 T, can link one of the k transfers that were successfully completed during the epoch to a specific pair of payer and payee $(\mathcal{A}, \mathcal{I})$.

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 T in *fair exchange*[2] for an RSA exponentiation of a "puzzle" value z under T's secret key. Fair exchange prevents a cheating 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 (Maxwell, 2016)'s fair-exchange protocol for solving 16x16 Sudoku puzzles (Section 5.8))—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 $\mathcal{I}$) and the user requesting its solution (*e.g.,* payer Alice $\mathcal{A}$).

[2] True fair exchange is impossible in the standard model (Pagnia and Gartner, 1999) and thus alternatives have been proposed, such as gradual release mechanisms, optimistic models, or use of 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. Our proofs are in the full version (Heilman et al., 2016a).

5.1.1 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 (Peck, 2016) has highlighted the security risks of complex scripting functionalities. Moreover, the Bitcoin community is currently debating (Faife, 2017) whether to deploy a solution ("segregated witnesses" (Wuille, 2015)) that corrects Bitcoin's transaction malleability issue 1). TumbleBit, however, remains secure even if this solution is not deployed as explained in Appendix C.1.

No coordination. In contrast to earlier work (Maxwell, 2013a; Ruffing et al., 2014), if Alice $\mathcal{A}$ wants to pay Bob $\mathcal{I}$, she need not interact with any other TumbleBit users. Instead, $\mathcal{A}$ and $\mathcal{I}$ need only interact with the Tumbler and each other. This lack of coordination between TumbleBit users makes it possible to scale our system.

Performance. We have implemented our TumbleBit system in C++ and python,

trusted third party. We follow prior works that use Bitcoin for fair exchange (Andrychowicz et al., 2014; Kumaresan and Bentov, 2014; Kumaresan et al., 2015) and treat the blockchain as a trusted public ledger. Other works use the term Contingent Payment or Atomic Swaps (Maxwell, 2011; Back et al., 2014).

Scheme	Prevents Theft	Anonymity	Resists DoS	Resists Sybils	Min. Mix Time	Bitcoin Compatible	No Coordination?
Coinjoin	✓	small set	×	×	1 block	✓	×
Coinshuffle	✓	small set	×	×	1 block	✓	× (p2p network)
Coinparty	2/3 users honest	✓	some[1]	✓ (fees)	2 blocks	✓	×
XIM	✓	✓	✓	✓ (fees)	hours	✓	× (uses blockchain)
Mixcoin	TTP accountable	× (TTP)	✓	✓ (fees)	2 blocks	✓	✓
Blindcoin	TTP accountable	✓	✓	✓ (fees)	2 blocks	✓	✓
CoinSwap	✓	× (TTP)[2]	✓	✓ (fees)	2 blocks	✓	✓
BSC	✓	✓	✓	✓ (fees)	3 blocks	×	✓
TumbleBit	✓	✓	✓	✓ (fees)	2 blocks	✓	✓

Table 5.1: 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.

using LibreSSL as our cryptographic library. We have tumbled payments from 800 payers to 800 payees; the relevant transactions are visible on the blockchain. 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 5.8).

5.1.2 Related Work

TumbleBit is related to work proposing new anonymous cryptocurrencies (*e.g.,* Zerocash (Miers et al., 2013; Ben Sasson et al., 2014), Monero (Monero, 2016) or Mimblewimble (Jedusor, 2016)). 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 (Decker and Wattenhofer, 2015) and the Lightning Network (Poon and Dryja, 2015). 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 $\mathcal{I}$, TumbleBit's intermediary T cannot.

The protocol discussed in Chapter 4 Blindly Signed Contracts published as (Heilman et al., 2016c) proposed a protocol that adds anonymity to micropayment channel networks. TumbleBit is implemented and Bitcoin compatible, while Blindly Signed Contracts is not. Moreover, Blindly Signed Contracts requires both Alice $\mathcal{A}$ and Bob $\mathcal{I}$ to interact with the Tumbler T as part of every off-blockchain payment. Thus, the Tumbler could correlate the timing of its interactions with $\mathcal{A}$ and $\mathcal{I}$ in order to link their payments. Meanwhile, TumbleBit eliminates this timing channel by only requiring interaction between $\mathcal{A}$ and T (and $\mathcal{A}$ and $\mathcal{I}$) during an off-blockchain payment (see Figure 5·1 and Section 5.7.2).

TumbleBit is also related to concurrent work proposing Bolt (Green and Miers, 2016), 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 (Miers et al., 2013; Ben Sasson et al., 2014).

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 5.7.3). 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 5.1.

Blindcoin (Valenta and Rowan, 2015), and its predecessor Mixcoin (Bonneau et al., 2014), 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 (Maxwell, 2013b) 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 (Ziegeldorf et al., 2015) is another decentralized solution, but it is secure only if 2/3 of the users are honest.

CoinShuffle (Ruffing et al., 2014) and CoinShuffle++(Moreno-Sanchez et al., 2016) build on CoinJoin (Maxwell, 2013a) to provide a decentralized tumbler that prevents bitcoin theft. Their anonymity properties are analyzed in (Meiklejohn and Orlandi, 2015). CoinShuffle(++) (Ruffing et al., 2014; Moreno-Sanchez et al., 2016) 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 (Bissias et al., 2014; Bonneau et al., 2015), limiting scalability; neither (Ruffing et al., 2014) nor (Moreno-

Sanchez et al., 2016) 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 (Bonneau et al., 2015; Tschorsch and Scheuermann, 2016). TumbleBit sidesteps these scalability limitations by not requiring coordination between mix users.

XIM (Bissias et al., 2014) builds on fair-exchange mixers like (Barber et al., 2012). 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 (Heilman et al., 2016c) shares the same fair-exchange properties and anonymity properties. However, unlike TumbleBit, (Heilman et al., 2016c) is not compatible with Bitcoin and does not provide an implementation. Also, (Heilman et al., 2016c) requires three blocks to be confirmed on the blockchain, while TumbleBit requires two.

After this work was first posted, Dorier and Ficsor began an independent TumbleBit implementation.[3]. TumbleBit was integrated as a privacy protocol into the Breeze wallet (Jamie, 2017) and the Hidden Wallet (Ficsór, 2017). The Hidden Wallet (now renamed to the Wasabi Wallet) continued this line of research on Bitcoin privacy and now uses a protocol of their own invention called WabiSabi (Ficsór et al., 2021).

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

5.2 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:

- $T_{\textit{offer}}$: One party $\mathcal{A}$ offers to pay bitcoins to any party that can sign a transaction that meets some condition $\mathcal{C}$. The $T_{\textit{offer}}$ transaction is signed by $\mathcal{A}$.
- $T_{\textit{fulfill}}$: This transaction points to $T_{\textit{offer}}$, meets the condition $\mathcal{C}$ stipulated in $T_{\textit{offer}}$, and contains the public key of the party $\mathcal{I}$ receiving the bitcoins.

$T_{\textit{offer}}$ is posted to the blockchain first. When $T_{\textit{fulfill}}$ is confirmed by the blockchain, the bitcoins in $T_{\textit{fulfill}}$ flow from the party signing transaction $T_{\textit{offer}}$ 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 (Research, 2010)—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 (Todd, 2014)), 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 higher than tw. Section 5.8.1 details the scripts used in our implementation.

[4] This is why (Heilman et al., 2016c) is not Bitcoin-compatible. (Heilman et al., 2016c) 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.

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 5·1.

5.3 TumbleBit: An Unlinkable Payment Hub

Our goal is to allow a *payer*, Alice $\mathcal{A}$, to unlinkably send 1 bitcoin to a *payee*, Bob $\mathcal{I}$. 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 $\mathcal{I}$ and anonymity could be harmed using the techniques of (Meiklejohn et al., 2013; Ron and Shamir, 2013; Biryukov et al., 2014b). Instead, TumbleBit funnels payments from multiple payer-payee pairs through the Tumbler T, using cryptographic techniques to ensure that, as long as T does not collude with TumbleBit's users, then no one can link a payment from payer $\mathcal{A}$ to payee $\mathcal{I}$.

5.3.1 Overview of Bob's Interaction with the Tumbler

We overview TumbleBit's phases under the assumption that Bob $\mathcal{I}$ 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. In our full version (Heilman et al., 2016a) we also discuss how Bob can receive multiple payments of denomination 1 bitcoin each.

TumbleBit has three phases (Fig 5·1). 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.,* $\approx$ 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 $\mathcal{I}$ that wants to receive payments) during the upcoming Payment Phase runs the escrow phase with T. The escrow phase has two parts:

(a) Payee $\mathcal{I}$ asks the Tumbler T to set up a payment channel. T escrows 1 bitcoin on the blockchain via a 2-of-2 escrow transaction (Section 5.2) denoted as $T_{\text{escr}(T,\mathcal{I})}$ stipulating that 1 bitcoin can be claimed by any transaction signed by both T and $\mathcal{I}$. $T_{\text{escr}(T,\mathcal{I})}$ is timelocked to time window tw_2, after which T can reclaim its bitcoin. Similarly, the payeer $\mathcal{A}$ escrows 1 bitcoin in a 2-of-2 escrow with T denoted as $T_{\text{escr}(\mathcal{A},T)}$, timelocked for time window tw_1 such that $tw_1 < tw_2$.

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

$$z = f_{RSA}(\epsilon, e, N) = \epsilon^e \mod N \tag{5.1}$$

where (e, N) is the TumbleBit RSA public key of the Tumbler 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

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

where σ is the Tumbler's ECDSA-Secp256k1 signature on the transaction $T_{\mathrm{cash}(T,\mathcal{I})}$ which transfers the bitcoin escrowed in $T_{\mathrm{escr}(T,\mathcal{I})}$ from T to $\mathcal{I}$. (We use ECDSA-Secp256k1 for compatibility with the Bitcoin protocol.) Thus, the solution to a puzzle z enables $\mathcal{I}$ to claim 1 bitcoin from T. To prevent misbehavior by the Tumbler T, our puzzle-promise protocol requires 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 5.6.

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

$$\overline{z} = r^e z \mod N. \tag{5.2}$$

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

$$\epsilon = \overline{\epsilon}/r \mod N \tag{5.3}$$

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

3: Cash-Out Phase. Bob $\mathcal{I}$ uses the puzzle solution ϵ to decrypt the ciphertext c. From the result $\mathcal{I}$ can create a transaction $T_{\text{cash}(T,\mathcal{I})}$ that is signed by both T and $\mathcal{I}$. $\mathcal{I}$ posts $T_{\text{cash}(T,\mathcal{I})}$ to the blockchain to receive 1 bitcoin from 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 T announces its RSA public key (e, N) and Bitcoin address $Addr_T$, 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.

5.3.2 Overview of Alice's Interaction with the Tumbler

We now focus on the puzzle-solving protocol between $\mathcal{A}$ and the Tumbler 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 T by escrowing Q bitcoins in an on-blockchain transaction $T_{\text{escr}(\mathcal{A},T)}$. Each escrowed bitcoin can pay 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 T by posting a transaction $T_{\text{cash}(\mathcal{A},T)}(j)$ that reflects the new allocation of bitcoins; namely, that T holds j bitcoins, while $\mathcal{A}$ holds $Q - j$ bitcoins. The details of Alice $\mathcal{A}$'s interaction with T, which are based on a technique used in micropayment channels (Narayanan et al., 2016, p. 86), are as follows:

1: Escrow Phase. Alice $\mathcal{A}$ posts a 2-of-2 escrow transaction $T_{\text{escr}(\mathcal{A},T)}$ to the

[6] Such a proof could be provided using the GQ identification protocol (Guillou and Quisquater, 1988) made non-interactive using the Fiat-Shamir heuristic (Fiat and Shamir, 1986) in the random oracle model.

blockchain that escrows Q of Alice's bitcoins. If no valid transaction $T_{\text{cash}(\mathcal{A},T)}$ 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 T. For each payment, $\mathcal{A}$ and T engage in an off-blockchain puzzle-solver protocol (see Sections 5.5.1,5.5.3).

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

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

Remark: Scaling Bitcoin. A similar (but more elaborate) technique can be applied between $\mathcal{I}$ and T so that only *two* on-blockchain transactions suffice for Bob $\mathcal{I}$ to receive an arbitrary number of off-blockchain payments. Details are in the full version (Heilman et al., 2016a). Given that each party uses *two* on-blockchain transactions to make multiple off-blockchain playments, Tumblebit helps Bitcoin scale.

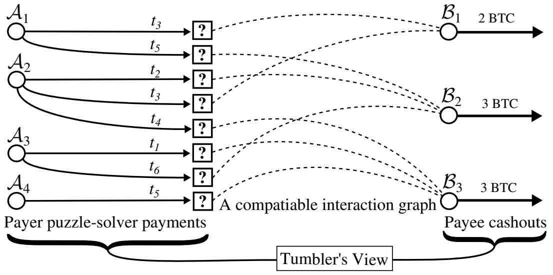


Figure 5·2: Our unlinkability definition: The Tumblers view and a compatible interaction multi-graph.

5.3.3 TumbleBit's Security Properties

Unlinkability. We assume that the Tumbler T does not collude with other users. The *view* of T consists of (1) the set of escrow transactions established between (a) each payer $\mathcal{A}_j$ and the Tumbler ($\mathcal{A}_j \overset{\text{escrow},a_i}{\longrightarrow} T$) of value a_i and (b) the Tumbler and each payee $\mathcal{I}_i$ ($T \overset{\text{escrow},b_i}{\longrightarrow} \mathcal{I}_i$), (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 $\mathcal{I}_i$ during the Cash-Out Phase.

An *interaction multi-graph* is a mapping of payments from payers to payees (Figure 5·2). 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 $\mathcal{I}_i$. An interaction graph is *compatible* if it explains the view of the Tumbler T, *i.e.,* the number of edges incident on $\mathcal{I}_i$ is equal to the total number of bitcoins cashed out by $\mathcal{I}_i$. Unlinkability requires all compatible interaction graphs to be equally likely. Anonymity therefore depends on the number of compatible interaction graphs.

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

A high-level proof of TumbleBit's unlinkability is in Section 5.7, and the limitations of unlinkability are discussed in Section 5.7.3.

Balance. The system should not be exploited to print new money or steal money, *even when parties collude.* As in (Green and Miers, 2016), 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 5.7.

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 T does not trust Alice $\mathcal{A}$ and Bob $\mathcal{I}$, 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 T, she pays for both the Q escrowed in transaction $T_{\text{escr}(\mathcal{A},T)}$ and for its transaction fees. Meanwhile, when the Tumbler T and Bob $\mathcal{I}$ establish a payment channel, the Q escrowed bitcoins in $T_{\text{escr}(T,\mathcal{I})}$ are paid in the Tumbler T, but the transaction fees are paid by Bob $\mathcal{I}$ (Section 5.3.1). Per (Bissias et al., 2014), fees raise the cost of an DoS attack where $\mathcal{I}$ starts and aborts many parallel sessions, locking 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.

5.4 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 5.3 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 $\mathcal{I}$ receives payments at an ephemeral bitcoin address $Addr_B$ chosen freshly for the epoch.

5.4.1 Anonymity Properties

As a classic tumbler, TumbleBit has the same *balance* property, but stronger anonymity: *k-anonymity within an epoch* (Heilman et al., 2016c; Bissias et al., 2014). Specifically, while the blockchain reveals which payers and payees participated in an epoch, no one (not even the Tumbler 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 5.3.3): 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 T aborts an payment made by payer $\mathcal{A}_j$. We deal with this by requiring $\mathcal{I}$ to uses an *ephemeral* Bitcoin address $Addr_B$ in each epoch. As in (Heilman et al., 2016c), Bob $\mathcal{I}$ discards $Addr_B$ if (1) the Tumbler T maliciously aborts $\mathcal{A}_j$'s payment in order to infer that $\mathcal{A}_j$ was attempting to pay $\mathcal{I}$ (see Section 5.8.3); or (2) k-anonymity was too small. (In case (2), $\mathcal{I}$ can alternatively re-tumble the bitcoin in $Addr_B$ in a future epoch.)

Remark: Intersection attacks. While this notion of k-anonymity is commonly used in Bitcoin tumblers (*e.g.,* (Bissias et al., 2014; Heilman et al., 2016c)), 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 (Ganta et al., 2008)). See also (Bissias et al., 2014; Meiklejohn and Orlandi, 2015).

DoS and Sybil Attacks. We use fees to resist DoS and Sybil attacks. Alice again pays for both the Q escrowed in transaction $T_{\text{escr}(\mathcal{A},T)}$ and for its transaction fees. However, we run into a problem if we want Bob $\mathcal{I}$ to pay the fee on the escrow transaction $T_{\text{escr}(T,\mathcal{I})}$. Because Bob $\mathcal{I}$ 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 $\mathcal{I}$ will have to pay the Tumbler T out of band. The anonymous fee vouchers described in (Heilman et al., 2016c) provide one way to address this, which also has the additional feature that payers $\mathcal{A}$ cover all fees.

5.5 A Fair Exchange for RSA Puzzle Solving

We now explain how to realize a Bitcoin-compatible *fair-exchange* where Alice $\mathcal{A}$ pays Tumbler T one bitcoin iff the T provides a valid solution to an RSA puzzle. The Tumbler 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

$$\epsilon = f_{RSA}^{-1}(y, d, N) = y^d \bmod N \tag{5.4}$$

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". (The full version (Heilman et al., 2016a) reviews the related work (Maxwell, 2011; Maxwell, 2016; Kumaresan and Bentov, 2014; Banasik et al., 2016).) Section 5.5.1 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 (Maxwell, 2016)'s protocol for solving 16x16 Sudoku puzzles (Section 5.8)). Also, the use of RSA means that our protocol supports solving *blinded* puzzles (see equation (5.2)), and thus can be used to create an unlinkable payment scheme. Section 5.5.3 shows how our protocol is integrated into TumbleBit's Payment Phase. Implementation results are in Table 5.2 of Section 5.8.2.

5.5.1 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 5.5.3 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 (Maxwell, 2011): Tumbler 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 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 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. Thus, instead of asking T to provide just *one* (c, h) pair, T will be asked to provide $m + n$ pairs (Step 3). Then, we use cut and choose: $\mathcal{A}$ asks 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

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

Alice $\mathcal{A}$ Input: Puzzle y		Tumbler 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 $\mathsf{key}_i \in \{0,1\}^{\lambda_1}$ – $c_i = H^{\mathsf{prg}}(\mathsf{key}_i) \oplus s_i$ Commit to the keys: $h_i = H(\mathsf{key}_i)$
	$\xleftarrow{c_1, \dots, c_{m+n}}$ $\xleftarrow{h_1, \dots, h_{m+n}}$	
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 $h_i = H(\mathsf{key}_i)$ Decrypt $s_i = H^{\mathsf{prg}}(\mathsf{key}_i) \oplus c_i$ Verify $s_i = \rho_i \mod N$ Abort if any check fails.	$\xleftarrow{\mathsf{key}_i\ \forall i \in F}$	
8. Post transaction T_{puzzle} T_{puzzle} offers 1 bitcoin within timewindow tw_1 under condition “the fulfilling transaction is signed by 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 $s_j = H^{\mathsf{prg}}(\mathsf{key}_j) \oplus c_j$ If s_j is s.t. $(s_j)^e = \beta_j \mod N$, Obtain solution $s_j / r_j \mod N$ which is $y^d \mod N$.		

Figure 5·3: 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 $\lambda_1 = 128$.

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 T cannot predict which pairs $\mathcal{A}$ asks it to open, T can only cheat with very low probability $1/\binom{m+n}{n}$.

However, we have a problem. Why should 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 T to open will open to "fake values" rather than "real" puzzles. Before 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 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 T, since $\mathcal{A}$ has only paid for the solution to a single puzzle, but has tricked T into solving multiple puzzles. We solve this problem in Step 9: once $\mathcal{A}$ posts T_{puzzle}, she proves to 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 T verifies this, T agrees to post T_{solve} which reveals m of the k values that open "real" (c, h) pairs (Step 10).

5.5.2 Fair Exchange

Fair exchange exchange entails the following: (1) *Fairness for 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 $\mathcal{A}$:* T will earn 1 bitcoin iff $\mathcal{A}$ obtains a correct solution.

We prove this using the real-ideal paradigm (Goldreich et al., 1987). We call the ideal functionality $\mathcal{F}_{\mathsf{fair-RSA}}$ and present it the full version (Heilman et al., 2016a). $\mathcal{F}_{\mathsf{fair-RSA}}$ acts like a trusted party between $\mathcal{A}$ and T. $\mathcal{F}_{\mathsf{fair-RSA}}$ gets a puzzle-solving request (y, 1 bitcoin) from $\mathcal{A}$, and forwards the request to T. If T agrees to solve puzzle y for $\mathcal{A}$, then T gets 1 bitcoin and $\mathcal{A}$ gets the puzzle solution. Otherwise, if T refuses, $\mathcal{A}$ gets 1 bitcoin back, and T gets nothing. Fairness for T is captured because $\mathcal{A}$ can request a puzzle solution only if she sends 1 bitcoin to $\mathcal{F}_{\mathsf{fair-RSA}}$. Fairness for $\mathcal{I}$ is captured because T receives 1 bitcoin only if he agrees to solve the puzzle. The following theorem is proved in the full version (Heilman et al., 2016a):

Theorem 5.5.1. *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 that f_{RSA} with parameters (N, e) is a permutation and a proof of knowledge for the associated secret key d.*

If the RSA assumption holds in $\mathbf{Z}_N^*$, and if functions H^{prg}, H are independent random oracles, there exists a negligible function ν, such that protocol in Figure 5·3 securely realizes $\mathcal{F}_{\mathsf{fair-RSA}}$ in the random oracle model with the following security guarantees. The security for T is $1 - \nu(\lambda)$ while security for $\mathcal{A}$ is $1 - \frac{1}{\binom{m+n}{n}} - \nu(\lambda)$.

5.5.3 Solving Many Puzzles and Moving Off-Blockchain

To integrate the protocol in Figure 5·3 into TumbleBit, we have to deal with three issues. First, if TumbleBit is to scale Bitcoin (Section 5.3.2), then Alice $\mathcal{A}$ must be able to use only *two* on-blockchain transactions $T_{\text{escr}(\mathcal{A},T)}$ and $T_{\text{cash}(\mathcal{A},T)}$ to pay for the an *arbitrary* number of Q puzzle solutions (each worth 1 bitcoin) during the Payment Phase; the protocol in Figure 5·3 only allows for the solution to a single puzzle. Second, per Section 5.3.2, the puzzle-solving protocol should occur entirely off-blockchain; the protocol in Figure 5·3 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 $T_{\text{escr}(\mathcal{A},T)}$ to the blockchain that escrows Q bitcoins, (per Section 5.3.2). $T_{\text{escr}(\mathcal{A},T)}$ 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 T.

Payment Phase. Alice $\mathcal{A}$ can buy solutions for up to Q puzzles, paying 1 bitcoin for each. Tumbler 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 5·3 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 5·3 is *not posted* to the blockchain. Instead, Alice $\mathcal{A}$ forms and signs transaction T_{puzzle} and sends it to the Tumbler T. Importantly, Tumbler T does *not* sign or post this transaction yet.

(2) Transaction T_{puzzle} points to the escrow transaction $T_{\text{escr}(\mathcal{A},T)}$; T_{puzzle} changes its balance so that 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 5·3: "the fulfilling transaction is signed by T and has preimages of $h_j \forall j \in R$."

(Suppose that T deviates from this protocol, and instead immediately signs and post T_{puzzle}. Then the bitcoins in $T_{\text{escr}(\mathcal{A},T)}$ 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) 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 5·3, the Tumbler 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 $T_{\text{cash}(\mathcal{A},T)}$ (per Section 5.3.2). $T_{\text{cash}(\mathcal{A},T)}$ points to the escrow transaction $T_{\text{escr}(\mathcal{A},T)}$ and reflects the new balance between $\mathcal{A}$ and T.

At the end of the i^{th} payment, the Tumbler T should have two new signed transactions from Alice: $T_{\text{puzzle}}(i)$ and $T_{\text{cash}(\mathcal{A},T)}(i)$, each reflecting the (same) balance of bitcoins between 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 $T_{\text{cash}(\mathcal{A},T)}(i)$?

In this case, the Tumbler immediately begins to cash out, even without waiting for the Cash-Out Phase. Specifically, Tumbler T holds transaction $T_{\text{puzzle}}(i)$, signed by $\mathcal{A}$, which reflects a correct balance of i bitcoins to T and $Q - i$ bitcoins to $\mathcal{A}$. Thus, T signs $T_{\text{puzzle}}(i)$ and posts it to the blockchain. Then, T claims the bitcoins locked in $T_{\text{puzzle}}(i)$ by signing and posting transaction T_{solve}. As in Figure 5·3, T_{solve} fulfills T_{puzzle} by containing the m preimages $k_j \forall j \in R$. The bitcoin in $T_{\text{escr}(\mathcal{A},T)}$ will be transferred to T_{puzzle} and then to T_{solve} and thus to the Tumbler T. The only harm done is that T posts two longer transactions $T_{\text{puzzle}}(i)$, $T_{\text{solve}}(i)$ (instead of just $T_{\text{cash}(\mathcal{A},T)}$), which require higher fees to be confirmed on the blockchain. (Indeed, this is why we have introduced the $T_{\text{cash}(\mathcal{A},T)}(i)$ 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 T has a transaction $T_{\text{cash}(\mathcal{A},T)}(j)$

signed by Alice, the Tumbler T just signs and post this transaction to the blockchain, claiming its j bitcoins.

5.6 Puzzle-Promise Protocol

We present the puzzle-promise protocol run between $\mathcal{I}$ and T in the Escrow Phase. Recall from Section 5.3.1, that the goal of this protocol is to provide Bob $\mathcal{I}$ 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 $T_{\text{cash}(T,\mathcal{I})}$ which transfers the bitcoin escrowed in $T_{\text{escr}(T,\mathcal{I})}$ from T to $\mathcal{I}$. Meanwhile the RSA-puzzle z hides the encryption key ϵ per equation (5.1).

If Tumbler 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, T cannot just reveal the signature σ directly, because Bob could use σ to claim the bitcoin escrowed in $T_{\text{escr}(T,\mathcal{I})}$ 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 T to compute many puzzle-promise pairs (c_i, z_i), and have Bob $\mathcal{I}$ test that some of the pairs are computed correctly. As in Section 5.5.1, 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 $\mathcal{I}$ knows that *at least one* of the unopened pairs is correctly formed. However, how does $\mathcal{I}$ know which puzzle z_i is correctly formed? Importantly, $\mathcal{I}$ can only choose one puzzle z_i that he will ask Alice $\mathcal{A}$ to solve during TumbleBit's Payment Phase (Section 5.3.1). 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 $\mathcal{I}$ wishes to obtain only a single payment of denomination 1 bitcoin; the protocol as described in Figure 5·4 and Section 5.6.1 suffices to run TumbleBit as a classic tumbler. We discuss its security properties in Section 5.6.2 and implementation in Section 5.8.2. In the full version (Heilman et al., 2016a), we show how to modify this protocol so that it allows $\mathcal{I}$ to receive arbitrary number of Q off-blockchain payments using only two on-blockchain transactions.

5.6.1 Protocol Walk Through

$\mathcal{I}$ 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 T as $\beta_1, ..., \beta_{m+n}$. T the evaluates each β_i to obtain a puzzle-promise pair (c_i, z_i) (Step 5).

Next, $\mathcal{I}$ needs to check that the η "fake" (c_i, z_i) pairs are correctly formed by T (Step 8). To do this, $\mathcal{I}$ needs T to provide the solutions ϵ_i to the puzzles z_i in fake pairs. T reveals these solutions only after $\mathcal{I}$ has proved that the η pairs really are fake (Step 7). Once this check is done, $\mathcal{I}$ knows that 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 $\mathcal{I}$ 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 T provide $\mathcal{I}$ with $\mu - 1$ quotients (Step 9). This solves our problem since knowledge of $\epsilon = \epsilon_{j_1}$ allows $\mathcal{I}$ 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 $\mathcal{I}$ 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 $T_{\text{escr}(T,\mathcal{I})}$ offers 1 bitcoin in exchange for a ECDSA-Secp256k1 signature under an *ephemeral* key PK_T^{eph} used *only once* during this protocol execution with this specific payee $\mathcal{I}$. Thus, even if $\mathcal{I}$ gets many signatures, only one can be used to form the cash-out transaction $T_{\text{cash}(T,\mathcal{I})}$ that redeems the bitcoin escrowed in $T_{\text{escr}(T,\mathcal{I})}$.

5.6.2 Security Properties

We again capture the security requirements of the puzzle-promise protocol using real-ideal paradigm (Goldreich et al., 1987). The ideal functionality $\mathcal{F}_{\textsf{promise-sign}}$ is presented the full version (Heilman et al., 2016a). $\mathcal{F}_{\textsf{promise-sign}}$ is designed to guarantee the following properties: (1) *Fairness for T:* Bob $\mathcal{I}$ learns nothing except signatures on *fake* transactions. (2) *Fairness for $\mathcal{I}$:* If T agrees to complete the protocol, then Bob $\mathcal{I}$ obtains at least one puzzle-promise pair. To do this, $\mathcal{F}_{\textsf{promise-sign}}$ acts a trusted party between $\mathcal{I}$ and T. Bob $\mathcal{I}$ 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 T signatures on any messages. (This provides property (2).) Then, if Tumbler T agrees, $\mathcal{F}_{\textsf{promise-sign}}$ provides Bob $\mathcal{I}$ with signatures on each "fake" transaction only. (This provides property (1).) The following theorem is proved the full version (Heilman et al., 2016a):

Theorem 5.6.1. *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 that f_{RSA} with parameters (N, e) is a permutation and a proof of knowledge for the associated secret key d.*

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 5·4 securely realizes the $\mathcal{F}_{\textsf{promise-sign}}$ functionality. The

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

Bob $\mathcal{I}$		Tumbler T. **Secret** input: d
		1. Set up $T_{\text{escr}(T,\mathcal{I})}$ Sign but do not post transaction $T_{\text{escr}(T,\mathcal{I})}$ timelocked for tw_2 offering one bitcoin under the condition: "the fulfilling transaction must be signed under key PK_T^{eph} and under key $PK_{\mathcal{I}}$."
2. Prepare μ Real Unsigned $T_{\text{cash}(T,\mathcal{I})}$. For $i \in 1, \ldots, \mu$: Choose random pad $\rho_i \leftarrow \{0,1\}^\lambda$ Set ${T_{\text{cash}(T,\mathcal{I})}}^i = \text{CashOutTFormat}(\rho_i)$ $ht_i = H'({T_{fulfill}}^i)$.	$\xleftarrow{T_{\text{escr}(T,\mathcal{I})}}$	
3. Prepare Fake Set. For $i \in 1, \ldots, \eta$: Choose random pad $r_i \leftarrow \{0,1\}^\lambda$ $ft_i = H'(\text{FakeFormat}\|\|r_i)$.		
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	$\xrightarrow{\beta_1 ... \beta_{\mu+\eta}}$	
Choose salt $\in \{0,1\}^\lambda$ Compute: $h_R = H(\text{salt}\|\|R)$ $h_F = H(\text{salt}\|\|F)$	$\xrightarrow{h_R, h_F}$	5. Evaluation. For $i = 1, \ldots, \mu + \eta$: ECDSA sign β_i to get $\sigma_i = \text{Sig}(SK_T^{eph}, \beta_i)$ 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.,* $z_i = (\epsilon_i)^e \mod N$
	$\xleftarrow{(c_1, z_1), ... (c_{\mu+\eta}, z_{\mu+\eta})}$	
6. Identify Fake Set.	$\xrightarrow{R, F}$	
	$\xrightarrow{r_i \ \forall i \in F}$	
	$\xrightarrow{\text{salt}}$	7. Check Fake Set. Check $h_R = H(\text{salt}\|\|R)$ and $h_F = H(\text{salt}\|\|F)$ For all $i \in F$: verify $\beta_i = H'(\text{FakeFormat}\|\|r_i)$ 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.,* $\mathsf{ECDSA\text{-}Ver}(PK_T^{eph}, H'(ft_i), \sigma_i) = 1$ Abort if any check fails	$\xleftarrow{\epsilon_i \ \forall i \in F}$	
	$\xleftarrow{q_2, ..., q_\mu}$	9. Prepare Quotients. For $R = \{j_1, ..., j_\mu\}$: set $q_2 = \frac{\epsilon_{j_2}}{\epsilon_{j_1}}, ..., q_\mu = \frac{\epsilon_{j_\mu}}{\epsilon_{j_{\mu-1}}}$
10. Quotient Test. For $R = \{j_1, ..., j_\mu\}$ check equalities: $z_{j_2} = z_{j_1} \cdot (q_2)^e \mod N$ ... $z_{j_\mu} = z_{j_{\mu-1}} \cdot (q_\mu)^e \mod N$ Check that $\forall i \in [m]$, $z_{j_i} \in \mathbf{Z}_N^*$ Abort if any check fails		11. Post transaction $T_{\text{escr}(T,\mathcal{I})}$ on blockchain
12. Begin Payment Phase. Set $z = z_{j_1}$. Send $\bar{z} = z \cdot (r)^e$ to Payer $\mathcal{A}$		

Figure 5·4: Puzzle-promise protocol when $Q = 1$. $(d, (e, N))$ are RSA keys of the tumbler T. (Sig, ECDSA-Ver) is an ECDSA-Secp256k1. 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, as used in Bitcoin's "hash-and-sign" paradigm. H^{shk} is SHA-512. CashOutTFormat is the unsigned portion of a transaction. ρ_i used to ensure sufficent entropy.

security for T is $1-\nu(\lambda)$ while security for $\mathcal{I}$ is $1-\frac{1}{\binom{\mu+\eta}{\eta}}-\nu(\lambda)$.

5.7 TumbleBit Security

We discuss TumbleBit's unlinkability and balance properties. See Section 5.3.3 for DoS/Sybil resistance.

5.7.1 Balance

The balance was defined, at high-level, in Section 5.3.3. We analyze balance in several cases.

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

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

First, the fairness of the puzzle-promise protocol guarantees that any $\mathcal{I}_i^*$ learns only (c, z) pairs; thus, by the unforgeability of ECDSA signatures and the hardness of solving RSA puzzles, $\mathcal{I}^*$ cannot claim any bitcoin at the end of the Escrow Phase.

Next, the fairness of the puzzle-solver protocol guarantees that if 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 $\mathcal{I}_i^*$ use the solved puzzles to claim bitcoins from T. By the unforgeability of ECDSA signatures (and assuming that the blockchain prevents double-spending), all colluding $\mathcal{I}_i^*$ cash-out no more than $\min(t, \sum_j SP_j)$ bitcoin in total, where t is the total number of bitcoins escrowed by T across all $\mathcal{I}_i^*$.

Case $\mathcal{I}_i^*$ and T collude. Now suppose that $\mathcal{I}_i^*$ and 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 $\mathcal{I}_i^*$ is the exchange of a puzzle (and its solution). No other secret information about $\mathcal{A}_j$ is revealed to $\mathcal{I}_i^*$. Thus, $\mathcal{I}_i^*$ cannot add any additional information to the view of T, that T can use to harm fairness for $\mathcal{A}_j$.

We do note, however, that an irrational Bob $\mathcal{I}_i^*$ 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 T for a correct solution ϵ^* to puzzle z^*. As such, Bob $\mathcal{I}_i$ 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 $\mathcal{I}_i$, *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 $\mathcal{I}_i$ himself. As such, we argue that such an attack is of no importance.

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

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

5.7.2 Unlinkability

Unlinkability is defined in Section 5.3.3 and must hold against a T that *does not collude* with other users. We show that *all* interaction multi-graphs $\mathcal{G}$ compatible with T's view are equally likely.

First, note that all TumbleBit payments have the same denomination (1 bitcoin). Thus, T^* cannot learn anything by correlating the values in the transactions. Next, recall from Section 5.3.1, that all users of TumbleBit coordinate on phases and epochs. Escrow transactions are posted at the same time, during the Escrow Phase only. All $T_{\text{escr}(T,\mathcal{I})}$ cash-out transactions are posted during the Cash-Out Phase only. All payments made from $\mathcal{A}_i$ and $\mathcal{I}_j$ occur during the Payment Phase only, and payments involve no direct interaction between T and $\mathcal{I}$. 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 $\mathcal{I}_i$. Even if the Tumbler T^* decides to cash-out with $\mathcal{A}_j$ before the Payment Phase completes (as is done in Section 5.5.3 when $\mathcal{A}_j$ misbehaves), all the $\mathcal{I}_i$ 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 $\mathcal{I}_i$ (see Section 5.3.1, equation (5.2)).

Finally, we assume secure channels, so that T^* cannot eavesdrop on communication between $\mathcal{A}_j$'s and $\mathcal{I}_i$'s, and that T^* cannot use network information to correlate

$\mathcal{A}_j$'s and $\mathcal{I}_i$'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 T^*, are equally likely.

5.7.3 Limitations of Unlinkability

TumbleBit's unlinkability (see Section 5.3.3) is inspired by Chaumian eCash (Chaum, 1983a), and thus suffers from similar limitations. (The full version (Heilman et al., 2016a) discusses the limitations of Chaumian eCash (Chaum, 1983a) in more detail.) In what follows, we assume that Alice has a single Bitcoin address $Addr_{\mathcal{A}}$, and Bob has Bitcoin address $Addr_{\mathcal{I}}$.

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.

If we can prevent ceiling attacks (*e.g.,* by requiring Bob to initiate every interaction with Alice) then Bob's puzzle z *cannot* be linked to *any* payee's Bitcoin address $Addr_{\mathcal{I}_1}, ..., Addr_{\mathcal{I}_\iota}$, even if Alice and the Tumbler collude; see the full version (Heilman et al., 2016a).

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

$$\overline{\overline{z}} = (r')^e \cdot \overline{z} \mod N. \tag{5.5}$$

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 5.3.3), 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.

Intersection attacks. Our definition of unlinkability applies only to a single epoch. Thus, as mentioned in Section 5.4.1 and (Bissias et al., 2014; Meiklejohn and Orlandi, 2015), 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.

5.8 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 $\mathcal{I}$). 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.

5.8.1 Protocol Instantiation

We instantiated our protocols with 2048-bit RSA. The hash functions and signatures are instantiated as described in the captions to Figure 5·3 and Figure 5·4.[7]

[7] There were slight difference between our protocols as described in this chapter and the implementation used in some of the tests. In Figure 5·3, $\mathcal{A}$ reveals blinds $r_j \forall j \in R$ to T, our implementation

Choosing m and n in the puzzle-solving protocol. Per Theorem 5.5.1, the probability that 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 5·3. 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 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 5.6.1 also allows 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 5·4 (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) (Andresen, 2014b) template. To format transaction T_{offer} (per Section 5.2) as 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

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 5·4; these are two 256-bit hash outputs and thus should not significantly affect performance. We have since removed these differences.

Table 5.2: Average performance of RSA-puzzle-solver and classic tumbler, in seconds. (100 trials) running between New York (NY), Boston (BOS), and Tokyo (TOK).

	Compute Time	Running Time (BOS-NY)	RTT (BOS-NY)	Running Time (BOS-TOK)	RTT (BOS-TOK)	Bandwidth
RSA-puzzle-solving	0.398	0.846	0.007949	4.18	0.186	269 KB
In clear	0.614	1.190	0.008036	5.99	0.187	326 KB
B over Tor	0.614	3.10	0.0875	8.37	0.273	342 KB
Both over Tor	0.614	6.84	0.0875	10.8	0.273	384 KB

Table 5.3: 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

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 the full version (Heilman et al., 2016a).

5.8.2 Off-Blockchain Performance Evaluation

We evaluate the performance for a run of our protocols between one payer Alice $\mathcal{A}$, one payee Bob $\mathcal{I}$ and the Tumbler 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).

Table 5.4: 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

Puzzle-solver protocol (Table 5.2). The total network bandwidth consumed by our protocol was 269 Kb, which is roughly 1/8th the size of the "average webpage" per (the Internet Archive, 2015) (2212 Kb). Next, we test the total (off-blockchain) computation time for our puzzle-solver protocol (Section 5.5.1) by running both parties ($\mathcal{A}$ and T) on the Boston machine. We test the impact of network latency by running $\mathcal{A}$ in Boston and T in Tokyo, and then with 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 5.2, we see the protocol completes in < 4 seconds, with running time dominated by network latency. Indeed, even when $\mathcal{A}$ and T are very far apart, our 2048-bit RSA puzzle solving protocol is still faster than (Maxwell, 2016)'s 16x16 Sudoku puzzle solving protocol, which takes 20 seconds.

TumbleBit as a classic tumbler (Table 5.2). Next, we consider classic Tumbler mode (Section 5.4). We consider a scenario where $\mathcal{A}$ and $\mathcal{I}$ 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 $\mathcal{I}$ in Boston and T in Tokyo, and (2) $\mathcal{A}$ and $\mathcal{I}$ in Boston and T in New York. To prevent the Tumbler T for linking $\mathcal{A}$ and $\mathcal{I}$ via their IP address, we also tested with (a) $\mathcal{I}$ connecting to T over Tor, and (b) both $\mathcal{A}$ and $\mathcal{I}$ connected through Tor. Per Table 5.2, 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 $\mathcal{A}$, $\mathcal{I}$ and 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 (≈ 20 minutes).

Performance of TumbleBit's Phases. (Table 5.3) Next, we break out the performance of each of TumbleBit's phases when $Q = 1$. We start by measuring compute time by running all $\mathcal{A}$, $\mathcal{I}$ and T on the Boston machine. Then, we locate each party on different machines. We first set A in Toronto, B in Boston and 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 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.

5.8.3 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 5.4 details the size and fees[8]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 $T_{\text{escr}(\mathcal{T},\mathcal{I})}$ 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 Phase, and 1 block for the Cash Out Phase. A Bitcoin

[8] We use a lower transaction fee rate of 15 Satoshi/byte (see Table 5.4) 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 5.5.3.

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 5.4). 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 ≈ 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 $T_{\text{escr}(\mathcal{T},\mathcal{I})}$ was timelocked for 15 blocks. All escrow transactions $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ are funded by TXID and all escrow transactions $T_{\text{escr}(\mathcal{T},\mathcal{I})}$ are funded by TXID. Two payer-payee pairs completed the protocol successfully. For the remaining three pairs, some party aborted the protocol:

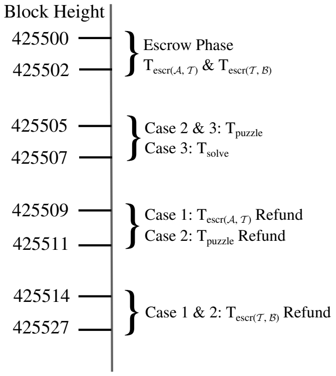

Block Height
425500
425502
Escrow Phase
$T_{\text{escr}(\mathcal{A},\mathcal{T})}$ & $T_{\text{escr}(\mathcal{T},\mathcal{B})}$
425505
425507
Case 2 & 3: T_{puzzle}
Case 3: T_{solve}
425509
425511
Case 1: $T_{\text{escr}(\mathcal{A},\mathcal{T})}$ Refund
Case 2: T_{puzzle} Refund
425514
425527
Case 1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

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

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. The Tumbler $\mathcal{T}$ reclaims its bitcoins from his payment channel with Bob $\mathcal{I}_1$ escrow transaction $T_{\text{escr}(\mathcal{T},\mathcal{I})}$ via a refund transaction after the timelock expires.

Case 2: The Tumbler aborts the puzzle-solver protocol by posting the transaction T_{puzzle} but refusing to provide the transaction T_{solve}. No payment completes from $\mathcal{A}_2$ to $\mathcal{I}_2$. Instead, $\mathcal{A}_2$ reclaims her bitcoin from T_{puzzle} via a refund transaction after the timelock in T_{puzzle} expires. Tumbler reclaims its bitcoins from its payment channel with Bob $\mathcal{I}_2$ via a refund transaction after the timelock on the escrow transaction $T_{\text{escr}(\mathcal{T},\mathcal{I})}$ expires.

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 5.5.3). 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 $\mathcal{I}_3$ 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 5.4.

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 $\mathcal{A}_1, \mathcal{A}_2$ were trying to pay payees $\mathcal{I}_1, \mathcal{I}_2$. However, anonymity of $\mathcal{A}_1, \mathcal{A}_2, B_1, B_2$ remains unharmed, since $\mathcal{I}_1$ and $\mathcal{I}_2$ were using ephemeral Bitcoin addresses they now discard to safeguard their anonymity (see Section 5.4.1).

Appendix A

Appendix: Improving the Transparency of the RPKI

A.1 Local consistency check

Relying parties use the following local consistency check (Section 2.6.4).

Initial sync. The initial connection to the repository is as in the current RPKI: all objects are downloaded and validated. If a relying party cannot obtain the valid current manifest of an RC R, or any objects logged in the manifest (as in Section 2.4.1), it raises a *missing information alarm* that blames RC R or the communication path to the publication point of R.

Subsequent syncs.

In the current design of the RPKI, updates to a relying party's local cache are performed all at once. We, instead, require incremental processing of updates: a relying party updates its local cache one publication point and one consecutive manifest change at a time. Updates are performed only for manifests issued by valid RCs. Incremental updates allow us to avoid race conditions than can occur when authorities update their manifests in parallel; see Counterexample 2. (We do allow relying parties to sync to publication points in any order, and to parallelize updates of publication points that are not in an ancestor-descendant relationship.) New RCs are an exception to this rule: the entire subtree rooted at new a RC should be downloaded and validated immediately to allow for quick issuance of new objects.

When a relying party obtains an updated state of a publication point, it reconstructs all the intermediate states of the publication point, and then compares pairs of consecutive states (indicated by consecutive manifest numbers) to make sure that (a) all new or modified objects are valid, and (b) all deletions and overwrites received proper consent. If not, it issues one of the middle alarms in Table 2.3. Details are in our tech report.

Appendix B

Appendix: Eclipse Attack on Bitcoin's P2P Network

B.1 A Useful Lemma

Lemma B.1.1. *If k items are randomly and independently inserted into n buckets, and X is a random variable counting the number of non-empty buckets, then*

$$E[X] = n\left(1 - (\tfrac{n-1}{n})^k\right) \approx n(1 - e^{-\frac{k}{n}}) \tag{B.1}$$

Proof. Let $X_i = 1$ if bucket i is non-empty, and $X_i = 0$ otherwise. The probability that the bucket i is empty after the first item is inserted is $(\frac{n-1}{n})$. After inserting k items

$$\Pr[X_i = 1] = 1 - \left(\tfrac{n-1}{n}\right)^k$$

It follows that

$$E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \Pr[X_i = 1] = n(1 - (\tfrac{n-1}{n})^k)$$

(B.1) follows since $(\frac{n-1}{n}) \approx e^{-1/n}$ for $n \gg 1$. □

B.2 Overwriting the New Table

How should the attacker send `ADDR` messages that overwrite the `new` table with "trash" IP addresses? Our "trash" is from the unallocated Class A IPv4 address block 252.0.0.0/8, designated by IANA as "reserved for future use" (IANA, 2015); any connections these addresses will fail, forcing the victim to choose an address from

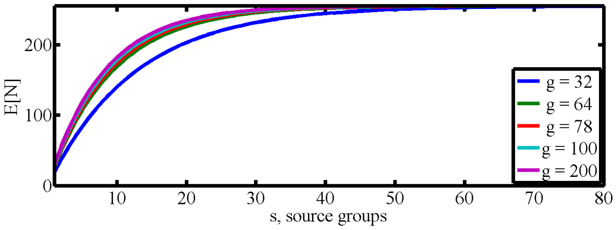


Figure B·1: $E[N]$ vs s (the number of source groups) for different choices of g (number of groups per source group) when overwriting the `new` table per equation (B.2).

`tried`. Next, recall (Section 3.2.2) that the pair *(group, source group)* determines the bucket in which an address in an `ADDR` message is stored. Thus, if the attacker controls nodes in s different groups, then s is the number of *source groups.* We suppose that nodes in each source group can push `ADDR` messages containing addresses from g distinct groups; the "trash" 252.0.0.0/8 address block give an upper bound on g of $2^8 = 256$. Each group contains a distinct addresses. How large should s, g, and a be so that the `new` table is overwritten by "trash" addresses?

B.2.1 Infrastructure strategy

In an infrastructure attack, the number of source groups s is constrained, and the number of groups g is essentially unconstrained. By Lemma B.1.1, the expected number of buckets filled by a s source groups is

$$E[N] = 256(1 - (\tfrac{255}{256})^{32s}) \tag{B.2}$$

We expect to fill ≈ 251 of 256 `new` buckets with $s = 32$.

Each (group, source group) pair maps to a unique bucket in `new`, and each bucket in `new` can hold 64 addresses. Bitcoin eviction is used, and we suppose each `new` bucket is completely full of legitimate addresses that are older than all the addresses inserted by the adversary via `ADDR` messages. Since all a addresses in a particular (group, source group) pair map to a single bucket, it follows that the number of addresses that actually stored in that bucket is given by $E[Y_a]$ in the recurrence relation of equations of (3.5)-(3.6). With $a = 125$ addresses, the adversary expects to overwrite $E[Y_a] = 63.8$ of the 64 legitimate addresses in the bucket. We thus require each source group to have 32 peers, and each peer to send `ADDR` messages with 8 distinct groups of $a = 125$ addresses. Thus, there are $g = 32 \times 8 = 256$ groups per source group, which is exactly the maximum number of groups available in our trash IP address block. Each peer sends exactly one `ADDR` message with $8 \times 125 = 1000$ address, for a total of $256 \times 125 \times s$ *distinct* addresses sent by all peers. (There are 2^{24} addresses in the 252.0.0.0/8 block, so all these addresses are distinct if $s < 524$.)

B.2.2 Botnet strategy

In a botnet attack, each of the attacker's t nodes is in a distinct source group. For $s = t > 200$, which is the case for all our botnet attacks, equation (B.2) shows that the number of source groups $s = t$ is essentially unconstrained. We thus require each peer to send a single `ADDR` message containing 1000 addresses with 250 distinct groups of four addresses each. Since $s = t$ is so large, we can model this by assuming that each (group, source group) pair selects a bucket in `new` uniformly at random, and inserts 4 addresses into that bucket; thus, the expected number of addresses inserted per bucket will be tightly concentrated around

$$4 \times E[B(250t, \tfrac{1}{256}] = 3.9t$$

For $t > 200$, we expect at least 780 address to be inserted into each bucket. From equations (3.5) and (3.6), we find $E[Y_{780}] \approx 64$, so that each `new` bucket is likely to be full.

Appendix C

Appendix: TumbleBit

C.1 Details of our Bitcoin Scripts

Figure C·1 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) (Andresen, 2014b) template, and discuss why TumbleBit protocol is not affected by the *transaction malleability* issue (Andrychowicz et al., 2015) 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* (Wuille, 2015), but as of this writing it has not been fully deployed (Faife, 2017). TumbleBit, however, remains secure even in the absence of segregated witness.

Interaction between Tumbler and Bob.

The right side of the Figure C·1 presents the transactions used for the interaction between the Tumbler T and Bob $\mathcal{I}$. The script in the $T_{\text{escr}(T,\mathcal{I})}$ transaction offers 1 bitcoin to a fulfilling transaction satisfies the condition $(\mathcal{I} \wedge T) \vee (T \wedge tw_2)$, *i.e.,* a transaction that is either (1) signed by both $\mathcal{I}$ and T, or (2) signed by T and posted to the blockchain after timewindow tw_2. Condition (2) is time-locked 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 $T_{\text{escr}(T,\mathcal{I})}$, 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 $T_{\text{escr}(T,\mathcal{I})}$ 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 $T \wedge \mathcal{I}$—checking if a valid $T_{\text{cash}(T,\mathcal{I})}$ has been posted that is signed by both Tumbler T and Bob $\mathcal{I}$. The redeem script above is hashed and its hash is stored in $T_{\text{cash}(T,\mathcal{I})}$. (This ensures that the transaction conform to the to the Pay-To-Script-Hash (P2SH) (Andresen, 2014b) template.)

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

Meanwhile, if Bob $\mathcal{I}$ refuses to post $T_{\text{cash}(T,\mathcal{I})}$ before the timewindow tw_2 ends, then the Tumbler T can reclaim the bitcoin escrowed in $T_{\text{escr}(T,\mathcal{I})}$ by posting a refund transaction $T_{\text{refund}(T,\mathcal{I})}$. (See the right side of Figure C·1.) When $T_{\text{refund}(T,\mathcal{I})}$ 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
```

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

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

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

The second reason is the *transaction malleablity* issue (Andrychowicz et al., 2015) of the current Bitcoin protocol. Suppose that Bob $\mathcal{I}$ mauls[1] his signature on transaction $T_{\text{escr}(T,\mathcal{I})}$ before it is posted to the blockchain, causing the TXID for $T_{\text{escr}(T,\mathcal{I})}$ to change from the TXID value expected by T. This has no effect on the Tumbler's ability to post the refund transaction $T_{\text{refund}(T,\mathcal{I})}$. Specifically, before posting $T_{\text{refund}(T,\mathcal{I})}$, the Tumbler need only find $T_{\text{escr}(T,\mathcal{I})}$ on the blockchain, hash $T_{\text{escr}(T,\mathcal{I})}$ to obtain its TXID, and use this TXID when it forming his refund transaction $T_{\text{refund}(T,\mathcal{I})}$. By contrast, suppose our protocol had instead somehow required Bob to participate in forming $T_{\text{refund}(T,\mathcal{I})}$ *before* $T_{\text{escr}(T,\mathcal{I})}$ had been posted to the blockchain. Then a malicious Bob could give the Tumbler a valid $T_{\text{refund}(T,\mathcal{I})}$ on $T_{\text{escr}(T,\mathcal{I})}$. Then, Bob could maul the signatures on $T_{\text{escr}(T,\mathcal{I})}$, and then post the mauled $T_{\text{escr}(T,\mathcal{I})}$ to the blockchain. $T_{\text{escr}(T,\mathcal{I})}$ would still be a valid transaction, but the $T_{\text{refund}(T,\mathcal{I})}$ held by the Tumbler would be useless, because $T_{\text{refund}(T,\mathcal{I})}$ no longer points to $T_{\text{escr}(T,\mathcal{I})}$ (because Bob has mauled the TXID of $T_{\text{escr}(T,\mathcal{I})}$).

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

Interaction between Tumbler and Bob.

The left side of the Figure C·1 presents the transactions used for the interaction between Alice $\mathcal{A}$ and the Tumbler T. The script in the $T_{\text{escr}(\mathcal{A},T)}$ transaction offers 1 bitcoin to a fulfilling transaction satisfies the condition $(\mathcal{A} \wedge T) \vee (T \wedge tw_1)$. The redeem script for this transaction is identical to the one used in $T_{\text{escr}(T,\mathcal{I})}$, 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. $T_{\text{cash}(\mathcal{A},T)}$ from Figure C·1 is formed analogously to $T_{\text{cash}(T,\mathcal{I})}$, and the refund $T_{\text{refund}(\mathcal{A},T)}$ pointing to $T_{\text{escr}(\mathcal{A},T)}$ is formed analogously to $T_{\text{refund}(T,\mathcal{I})}$.

Recall from Section 5.5.3, that in the puzzle-solver protocol, Alice forms and signs T_{puzzle} and sends it to the Tumbler. Transaction T_{puzzle} fulfils $T_{\text{escr}(\mathcal{A},T)}$ via the condition $(\mathcal{A} \wedge T)$. Thus, (just like $T_{\text{cash}(\mathcal{A},T)}$), a valid transaction T_{puzzle} should contain (1) a hash of redeem script for $T_{\text{escr}(T,\mathcal{I})}$, and (2) input values that include the required signatures from $\mathcal{A}$ and T. If a valid T_{puzzle} is posted to the blockchain, Alice's bitcoin escrowed in $T_{\text{escr}(\mathcal{A},T)}$ is transferred to T_{puzzle}. However, this bitcoin remains locked up in T_{puzzle} until T_{puzzle} fulfilled by a transaction the meets the condition $(T \wedge \mathcal{A} \forall j \in R : h_j = H(k_j)) \vee (\mathcal{A} \wedge tw1)$ 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 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 5·3) and is signed by 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 T's public key. The preimages $k_1, ..., k_{15}$ are such that $H(k_\ell) = h_\ell$ per Figure 5·3.

Per Section 5.5.3, however, if all parties are cooperative, the Tumbler 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 T, the Tumbler can claim the bitcoin escrowed in $T_{\text{escr}(\mathcal{A},T)}$. To do this, 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 T can claim his bitcoin even if Alice stops communicating with T. Notice, however, if T decides to unilaterally claims a bitcoin by posting T_{solve}, the Tumbler T necessarily reveals the puzzle solution (see Section 5.5.1). Therefore, Alice gets what she paid for even if she stops cooperating with the Tumbler 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 T, then T_{puzzle} points to the $T_{\text{escr}(\mathcal{A},T)}$ transaction which is already confirmed by the blockchain and thus cannot be mauled.

Finally, recall from Section 5.5.3 that if T becomes uncooperative, 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 $T_{\text{refund}(\mathcal{A},T)}$ that points at T_{puzzle}. (See Figure C·1.) Specifically, $T_{\text{refund}(\mathcal{A},T)}$ 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 $T_{\text{refund}(\mathcal{A},T)}$ without any help from the Tumbler. Once again, this matters because the refund transactions must be posted when T becomes uncooperative, and must still be valid even in the face of transaction malleability (*i.e.,* if T mauls the TXID for the transaction fulfilled by $T_{\text{refund}(\mathcal{A},T)}$.)

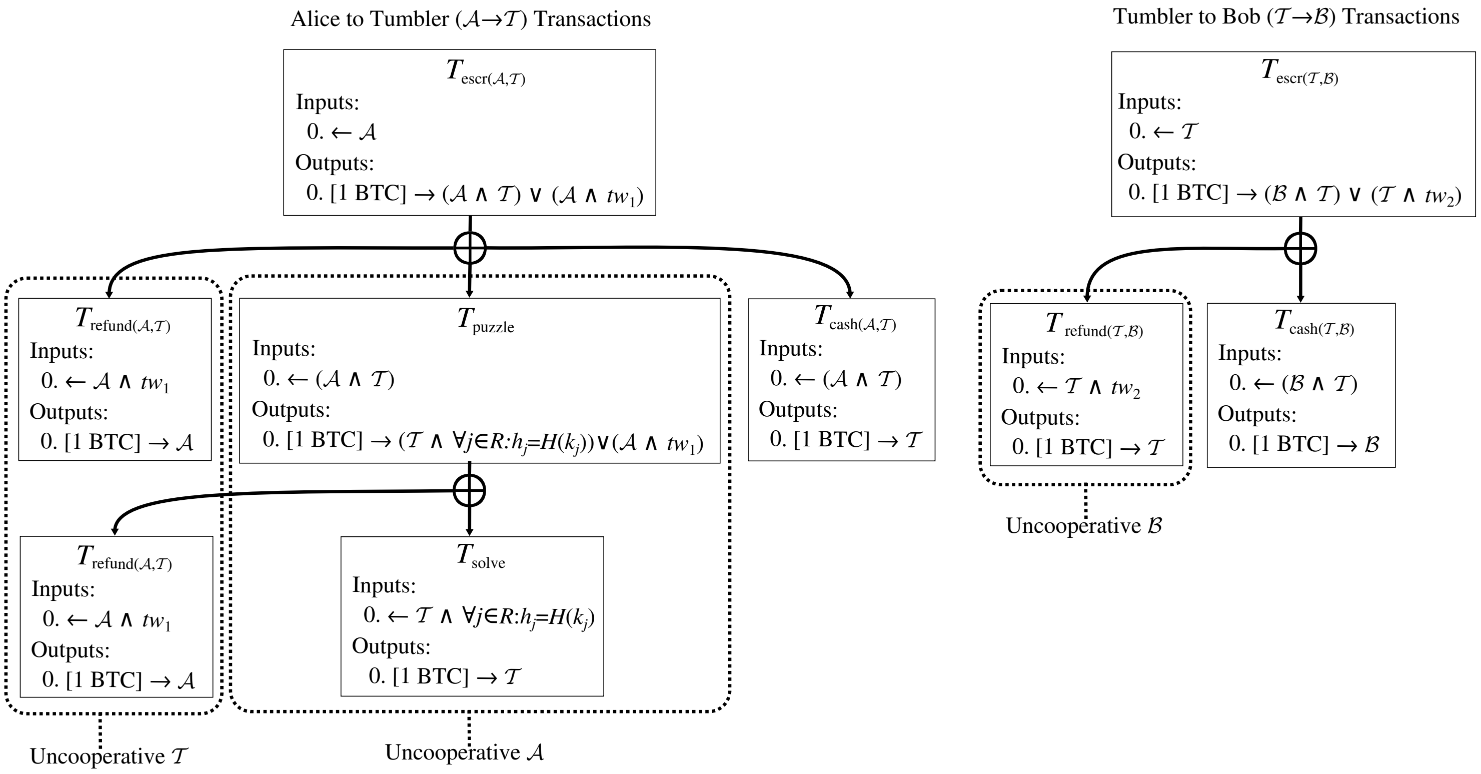

Alice to Tumbler ($\mathcal{A}\rightarrow\mathcal{T}$) Transactions
$T_{\text{escr}(\mathcal{A},\mathcal{T})}$
Inputs:
0. $\leftarrow \mathcal{A}$
Outputs:
0. [1 BTC] $\rightarrow (\mathcal{A} \wedge \mathcal{T}) \vee (\mathcal{A} \wedge tw_1)$
$T_{\text{refund}(\mathcal{A},\mathcal{T})}$
Inputs:
0. $\leftarrow \mathcal{A} \wedge tw_1$
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] $\rightarrow \mathcal{B}$
Uncooperative $\mathcal{B}$

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

C.2 TumbleBit transactions on Bitcoin's Blockchain

We ran the TumbleBit protocol to achieve a classic-tumbler for 800 users(*i.e.,* $\aleph =$ anonymity-set $= 800$). In this run of protocol. For readability we refer to each txid with a number given as follows:

1. fd51bd844202ef050f1fbe0563e3babd2df3c3694b61af39ac811ad14f52b233
2. 8520da7116a1e634baf415280fdac45f96e680270ea06810512531a783f0c9f6
3. cc6ce9f949f84f35aa33ed24c40c3a1a9d2988a69b97b95a4ca00526ae176794
4. a2819ac0112d272467b8c2c9f62a5ff1742108fb3d2da2b763bcfcd0c7d2d504
5. 0d26f9a51d3f580a8d4665e80135d7402cb3512f04cb30081ce2a5b37669beff
6. 016ddd812b4b55c3fd5b6e19a2f0930ba5842fa60c753d73d4dabb1f5d407f77
7. 4b961d9d3355f4940277aafbeb28ee10834c4eac73b2b541dd784c08936fd5b7
8. 5b71e3eba88c2760f4edf7c50a1dcfc3ff3437742dc725c3797a8bd0593462e6
9. 607dbdd7863200235c26d3e943e1acb4e70077eadb6b25448c15ef87eb65925d
10. e16a086fa132130c6e3db4b1764fa6e2b1b167fd44d28062ec891e0588b1a212
11. dbe59befda9f050ad4f4de20d288a35fddb389f0dc210b6444f52d99f0c5cdca
12. 72f01960d3b856a1c34bd914da0c570b367f0b3134d3a14fe9468c5deb393fc0

For the 800 user tumble the puzzle-promise escrow fund transactions all spend from txid:1 and the puzzler-solver escrow fund transactions all spend from txid:2.

Our second run of only 10 users demonstrates how our fair exchange properties are enforced in the face of cooperative or malicious parties.

If Alice or the Tumbler refuse to cooperate after both escrow fund transactions (TXID:3, TXID:4) has been confirmed, Alice reclaims her bitcoins via a refund transaction (TXID:5) after the timelock of tw1 blocks expires and then later the Tumber reclaims its bitcoins via a refund transaction (TXID:3) after a timelock of tw2 blocks expires). If the Tumbler posts the preimage fund (TXID:6) and refuses to provide Alice the preimage spend, Alice reclaims her bitcoins via a refund transaction (TXID:7) after the timelock tw1 on the preimage fund expires and the Tumbler reclaims its bitcoins after the timelock tw2 on puzzle-promise escrow (TXID:8) expires. If the however the Tumbler has provided Alice the solution to the puzzle and Alice refuses to sign the puzzle-solver escrow spend, the Tumbler posts the preimage fund (TXID:9) and the preimage spend (TXID:10) and claims its bitcoins.

The puzzle-promise escrow fund transactions all spend from TXID:11 and the puzzler-solver escrow fund transactions all spend from TXID:12.

References

Aiello, W., Ioannidis, J., and McDaniel, P. (2003). Origin authentication in interdomain routing. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 165–178. ACM.

Aleman, M. (2021). El salvador makes bitcoin legal tender. *Associated Press*. Accessed: 2021-06-27.

Amante, S. (2012). Risks associated with resource certification systems for internet numbers.

Anceaume, E., Busnel, Y., and Gambs, S. (2013). On the power of the adversary to solve the node sampling problem. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XI*, pages 102–126. Springer.

Anderson, D. (2012). Splinternet behind the great firewall of china. *Queue*, 10(11):40.

Andresen, G. (2014a). CVE-2013-5700: Remote p2p crash via bloom filters. `https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures`. Accessed: 2014-02-11.

Andresen, G. (2014b). BIP-0016: Pay to Script Hash. *Bitcoin Improvement Proposals*.

Andriesse, D. and Bos, H. (2014). An analysis of the zeus peer-to-peer protocol.

Andrychowicz, M., Dziembowski, S., Malinowski, D., and Mazurek, L. (2014). Secure multiparty computations on bitcoin. In *IEEE S&P*, pages 443–458.

Andrychowicz, M., Dziembowski, S., Malinowski, D., and Mazurek, Ł. (2015). On the malleability of bitcoin transactions. In *International Conference on Financial Cryptography and Data Security*, pages 1–18. Springer.

Apostolaki, M., Zohar, A., and Vanbever, L. (2017). Hijacking bitcoin: Routing attacks on cryptocurrencies. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 375–392. IEEE.

Austein, R., Huston, G., Kent, S., and Lepinski, M. (2012). *RFC 6486: Manifests for the Resource Public Key Infrastructure (RPKI)*. Internet Engineering Task Force (IETF). `http://tools.ietf.org/html/rfc6486`.

Awerbuch, B. and Scheideler, C. (2006). Robust random number generation for peer-to-peer systems. In *Principles of Distributed Systems*, pages 275–289. Springer.

AWS (2014). Amazon web services elastic ip. `http://aws.amazon.com/ec2/faqs/#elastic-ip`. Accessed: 2014-06-18.

azure (2014). Microsoft azure ip address pricing. `http://azure.microsoft.com/en-us/pricing/details/ip-addresses/`. Accessed: 2014-06-18.

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

Bahack, L. (2013). Theoretical bitcoin attacks with less than half of the computational power (draft). *arXiv preprint arXiv:1312.7013*.

Bakker, A. and Van Steen, M. (2008). Puppetcast: A secure peer sampling protocol. In *European Conference on Computer Network Defense (EC2ND)*, pages 3–10. IEEE.

Ballani, H., Francis, P., and Zhang, X. (2007). A study of prefix hijacking and interception in the Internet. In *SIGCOMM'07*.

Banasik, W., Dziembowski, S., and Malinowski, D. (2016). Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts. Cryptology ePrint Archive, Report 2016/451.

Barber, S., Boyen, X., Shi, E., and Uzun, E. (2012). Bitter to Better - How to Make Bitcoin a Better Currency. In *Financial Cryptography and Data Security*. Springer.

Ben Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., and Virza, M. (2014). Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Security and Privacy (SP)*, pages 459–474.

Biryukov, A., Khovratovich, D., and Pustogarov, I. (2014a). Deanonymisation of clients in Bitcoin P2P network. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 15–29. ACM.

Biryukov, A., Khovratovich, D., and Pustogarov, I. (2014b). Deanonymisation of Clients in Bitcoin P2P Network. In *ACM-CCS*, pages 15–29.

Biryukov, A. and Pustogarov, I. (2014). Bitcoin over tor isn't a good idea. *arXiv preprint arXiv:1410.6079*.

Bissias, G., Ozisik, A. P., Levine, B. N., and Liberatore, M. (2014). Sybil-resistant mixing for bitcoin. In *Workshop on Privacy in the Electronic Society*, pages 149–158.

Bitcoin Wiki (2015). Confirmation. `https://en.bitcoin.it/wiki/Confirmation`.

Bitcoin Wisdom (2015). Bitcoin difficulty and hash rate chart. `https://bitcoinwisdom.com/bitcoin/difficulty`.

Bitnode (2014). Bitnode.io snapshot of reachable nodes. `https://getaddr.bitnodes.io/nodes/`. Accessed: 2014-02-11.

bitpay (2014). Bitpay: What is transaction speed? `https://support.bitpay.com/hc/en-us/articles/202943915-What-is-Transaction-Speed-`.

blockchain.io (2015). Average transaction confirmation time. `https://blockchain.info/charts/avg-confirmation-time`.

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

Boneh, D., Lynn, B., and Shacham, H. (2001). Short signatures from the weil pairing. In *ASIACRYPT*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532.

Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J. A., and Felten, E. W. (2015). SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *IEEE - SP*.

Bonneau, J., Narayanan, A., Miller, A., Clark, J., Kroll, J., and Felten, E. (2014). Mixcoin: Anonymity for bitcoin with accountable mixes. In *Financial Cryptography and Data Security*.

Bortnikov, E., Gurevich, M., Keidar, I., Kliot, G., and Shraer, A. (2009). Brahms: Byzantine resilient random membership sampling. *Computer Networks*, 53(13):2340–2359.

Brands, S. (1993). Untraceable off-line cash in wallets with observers (extended abstract). In *CRYPTO*.

btcwiki (2014a). Bitcoin: Common vulnerabilities and exposures. `https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures`. Accessed: 2014-02-11.

btcwiki (2014b). Bitcoin wiki: Double-spending. `https://en.bitcoin.it/wiki/Double-spending`. Accessed: 2014-02-09.

Bush, R. (2012a). *Responsible Grandparenting in the RPKI.* Internet Engineering Task Force Network Working Group. `http://tools.ietf.org/html/draft-ymbk-rpki-grandparenting-02`.

Bush, R. (2012b). *RPKI-Based Origin Validation Operation.* Internet Engineering Task Force Network Working Group. `http://tools.ietf.org/html/draft-ietf-sidr-origin-ops-19`.

Bush, R. (2013). *RPKI Local Trust Anchor Use Cases.* Internet Engineering Task Force (IETF). `http://www.ietf.org/id/draft-ymbk-lta-use-cases-00.txt`.

Butler, K., Farley, T., McDaniel, P., and Rexford, J. (2010). A survey of BGP security issues and solutions. *Proceedings of the IEEE.*

Cachin, C. and Samar, A. (2004). Secure distributed dns. In *Dependable Systems and Networks, 2004 International Conference on*, pages 423–432. IEEE.

CAIDA (2014a). AS to Organization Mapping Dataset.

CAIDA (2014b). Routeviews prefix to AS Mappings Dataset for IPv4 and IPv6.

Camenisch, J., Hohenberger, S., and Lysyanskaya, A. (2005). Compact e-cash. In *EUROCRYPT.*

Cao, T., Yu, J., Decouchant, J., Luo, X., and Verissimo, P. (2020). Exploring the monero peer-to-peer network. In *International Conference on Financial Cryptography and Data Security*, pages 578–594. Springer.

CarnaBotnet (2012). Internet census 2012.
`http://internetcensus2012.bitbucket.org/paper.html`.

Castro, M., Druschel, P., Ganesh, A., Rowstron, A., and Wallach, D. S. (2002). Secure routing for structured peer-to-peer overlay networks. *ACM SIGOPS Operating Systems Review*, 36(SI):299–314.

Chaum, D. (1983a). Blind signature system. In *CRYPTO.*

Chaum, D. (1983b). Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203. Springer.

CoinmarketCap (2021). Coinmarketcap bitcoin july 27 2021. link.

Communications Security, Reliability and Interoperability Council III (CSRIC) (2011). Secure bgp deployment. *Communications and Strategies.*

Cooper, D., Heilman, E., Brogle, K., Reyzin, L., and Goldberg, S. (2013). On the risk of misbehaving RPKI authorities. *HotNets XII.*

Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and Polk, W. (2008). *RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.* Internet Engineering Task Force (IETF). http://tools.ietf.org/html/rfc5280.

Courtois, N. T. and Bahack, L. (2014). On subversive miner strategies and block withholding attack in bitcoin digital currency. *arXiv preprint arXiv:1402.1718*.

Cowie, J. (2010). Rensys blog: China's 18-minute mystery. http://www.renesys.com/blog/2010/11/chinas-18-minute-mystery.shtml.

Crosby, S. A. and Wallach, D. S. (2009). Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, pages 317–334.

Davis, C. R., Fernandez, J. M., Neville, S., and McHugh, J. (2008). Sybil attacks as a mitigation strategy against the storm botnet. In *3rd International Conference on Malicious and Unwanted Software, 2008.*, pages 32–40. IEEE.

de Beaupre, A. (2013). ISC Diary: Multiple Banking Addresses Hijacked. http://isc.sans.edu/diary/BGP+multiple+banking+addresses+hijacked/16249.

Decker, C. and Wattenhofer, R. (2013). Information propagation in the bitcoin network. In *IEEE Thirteenth International Conference on Peer-to-Peer Computing (P2P)*, pages 1–10. IEEE.

Decker, C. and Wattenhofer, R. (2015). A fast and scalable payment network with bitcoin duplex micropayment channels. In *Stabilization, Safety, and Security of Distributed Systems*, pages 3–18. Springer.

Delgado-Segura, S., Bakshi, S., Pérez-Solà, C., Litton, J., Pachulski, A., Miller, A., and Bhattacharjee, B. (2019). Txprobe: Discovering bitcoin's network topology using orphan transactions. In *International Conference on Financial Cryptography and Data Security*, pages 550–566. Springer.

Dillon, J. (2013). Bitcoin-development mailinglist: Protecting bitcoin against network-wide dos attack. bitcoin-dev/2013-July/002896.html. Accessed: 2021-10-7.

Dingledine, R., Hopper, N., Kadianakis, G., and Mathewson, N. (2014). One fast guard for life (or 9 months). In *7th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2014)*.

Donet, J. A. D., Pérez-Sola, C., and Herrera-Joancomartí, J. (2014). The bitcoin p2p network. In *Financial Cryptography and Data Security*, pages 87–102. Springer.

Dorier, N. (2016). Ntumblebit: tumblebit implementation in .net core. https://github.com/NTumbleBit/NTumbleBit.

Durumeric, Z., Wustrow, E., and Halderman, J. A. (2013). ZMap: Fast Internet-wide scanning and its security applications. In *Proceedings of the 22nd USENIX Security Symposium*.

Ekparinya, P., Gramoli, V., and Jourjon, G. (2018). Impact of man-in-the-middle attacks on ethereum. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 11–20. IEEE.

Evans, C., Palmer, C., and Sleevi, R., editors (2013). *Public Key Pinning Extension for HTTP*. IETF Web Security, Internet-Draft. `http://tools.ietf.org/html/draft-ietf-websec-key-pinning-09`.

EvilKnievel (2015). Bug bounty requested: 10 btc for huge dos bug in all current bitcoin clients. Bitcoin Forum. `https://bitcointalk.org/index.php?topic=944369.msg10376763#msg10376763`.

Eyal, I. (2014). The miner's dilemma. *arXiv preprint arXiv:1411.7099*.

Eyal, I. and Sirer, E. G. (2014). Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*, pages 436–454. Springer.

Faife, C. (2017). Will 2017 bring an end to bitcoin's great scaling debate? `http://www.coindesk.com/2016-bitcoin-protocol-block-size-debate/`.

FCC (2013). Working group 6, secure bgp deployment, final report. Technical report, FCC CSRIC Working Group 6.

Feld, S., Schönfeld, M., and Werner, M. (2014). Analyzing the deployment of bitcoin's p2p network under an as-level perspective. *Procedia Computer Science*, 32:1121–1126.

Fiat, A. and Shamir, A. (1986). How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*.

Ficsór, Á. (2017). Goodbye breeze wallet, hello hidden wallet! medium.com.

Ficsór, Á. (2021a). Dumplings: software to create reproducible coinjoin statistics from blockchain data.

Ficsór, Á. (2021b). Private correspondence with Adám Ficsór.

Ficsór, Á., Kogman, Y., Ontivero, L., and Seres, I. A. (2021). Wabisabi: Centrally coordinated coinjoins with variable amounts. *IACR Cryptol. ePrint Arch.*, 2021:206.

Ficsór, Á. and TDevD (2017). Zerolink: The bitcoin fungibility framework. github.com.

Finney, H. (2011). Bitcoin talk: Finney attack. `https://bitcointalk.org/index.php?topic=3441.msg48384#msg48384`. Accessed: 2014-02-12.

Ganta, S. R., Kasiviswanathan, S. P., and Smith, A. (2008). Composition attacks and auxiliary information in data privacy. In *ACM SIGKDD*, pages 265–273.

Gassko, I., Gemmell, P., and MacKenzie, P. D. (2000). Efficient and fresh cerification. In Imai, H. and Zheng, Y., editors, *Public Key Cryptography*, volume 1751 of *Lecture Notes in Computer Science*, pages 342–353. Springer.

Goldberg, S., Schapira, M., Hummon, P., and Rexford, J. (2010). How secure are secure interdomain routing protocols? In *SIGCOMM'10*.

Goldman, E. (2006). Sex.com: An update. `http://blog.ericgoldman.org/archives/2006/10/sexcom_an_updat.htm`.

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

Gould, D. (2021). Github: chaincase the only privacy preserving bitcoin wallet for ios. `https://github.com/chaincase-app/Chaincase`.

Grams (2016). Helixlight: Helix made simple. `https://grams7enufi7jmdl.onion.to/helix/light`.

Green, M. and Miers, I. (2016). Bolt: Anonymous Payment Channels for Decentralized Currencies. *Cryptology ePrint Archive 2016/701.*

Guillou, L. C. and Quisquater, J.-J. (1988). A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory. In *EUROCRYPT*.

Heilman, E. (2016a). Bitcoin Peer Forger (BPF). github. https://github.com/EthanHeilman/bitcoin_peer_forger.

Heilman, E. (2016b). (Bitcoin PR 8282) net: Feeler connections to increase online addrs in the tried table. github. https://github.com/bitcoin/bitcoin/pull/8282.

Heilman, E. (2016c). (Bitcoin PR 9037) net: Add test-before-evict discipline to addrman. github. https://github.com/bitcoin/bitcoin/pull/9037.

Heilman, E. (2018). (Bitcoin PR 12626) Limit the number of IPs addrman learns from each DNS seeder . github. .

Heilman, E., Alshenibr, L., Baldimtsi, F., Scafuro, A., and Goldberg, S. (2016a). TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub. Cryptology ePrint Archive 2016/575.

Heilman, E., Alshenibr, L., Baldimtsi, F., Scafuro, A., and Goldberg, S. (2017). Tumblebit: An untrusted bitcoin-compatible anonymous payment hub. In *Network and Distributed System Security Symposium*.

Heilman, E., Baldimtsi, F., and Goldberg, S. (2016b). Blindly signed contracts: Anonymous on-blockchain and off-blockchain bitcoin transactions. In *International conference on financial cryptography and data security*, pages 43–60. Springer.

Heilman, E., Baldimtsi, F., and Goldberg, S. (2016c). Blindly Signed Contracts: Anonymous On-Blockchain and Off-Blockchain Bitcoin Transactions. In *Workshop on Bitcoin and Blockchain Research at Financial Crypto*.

Heilman, E., Cooper, D., Reyzin, L., and Goldberg, S. (2014). From the consent of the routed: Improving the transparency of the rpki. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 51–62.

Heilman, E., Kendler, A., Zohar, A., and Goldberg, S. (2015a). Eclipse attacks on bitcoin's peer-to-peer network. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 129–144.

Heilman, E., Kendler, A., Zohar, A., and Goldberg, S. (2015b). Eclipse attacks on bitcoin's peer-to-peer network (full version). Technical Report 2015/263, ePrint Cryptology Archive, `http://eprint.iacr.org/2015/263.pdf`.

Hildrum, K. and Kubiatowicz, J. (2003). Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In *Distributed Computing*, pages 321–336. Springer.

Hlavacek, T., Cunha, I., Gilad, Y., Herzberg, A., Katz-Bassett, E., Schapira, M., and Shulman, H. (2020). Disco: sidestepping rpki's deployment barriers. In *Network and Distributed System Security Symposium (NDSS)*.

Hughes, E. (1993). A cypherpunk's manifesto. *Crypto anarchy, cyberstates, and pirate utopias*, pages 81–83.

Huston, G., Loomans, R., and Michaelson, G. (2012a). *RFC 6481: A Profile for Resource Certificate Repository Structure*. Internet Engineering Task Force (IETF). `http://tools.ietf.org/html/rfc6481`.

Huston, G. and Michaelson, G. (2012). *RFC 6483: Validation of Route Origination Using the Resource Certificate Public Key Infrastructure (PKI) and Route Origin Authorizations (ROAs)*. Internet Engineering Task Force (IETF). `http://tools.ietf.org/html/rfc6483`.

Huston, G., Michaelson, G., and Kent, S. (2012b). *RFC 6489: Certification Authority (CA) Key Rollover in the Resource Public Key Infrastructure (RPKI)*. Internet Engineering Task Force (IETF). `http://tools.ietf.org/html/rfc6489`.

Huston, G., Rossi, M., and Armitage, G. (2011). Securing BGP: A literature survey. *Communications Surveys & Tutorials, IEEE*, 13(2):199–222.

IANA (2015). Iana ipv4 address space registry. `http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xhtml`.

Inc, C. (2016). Chainalysis: Blockchain analysis. `https://www.chainalysis.com/`.

IPGlider (2017). (Monero PR 1701) Add anchor connections. github. https://github.com/monero-project/monero/pull/1701.

Jamie, R. (2017). Breeze wallet integrates trustless payment hub tumblebit. news.bitcoin.com.

Jedusor, T. E. (2016). Mimblewimble.

Jesi, G. P., Montresor, A., and van Steen, M. (2010). Secure peer sampling. *Computer Networks*, 54(12):2086–2098.

Johnson, B., Laszka, A., Grossklags, J., Vasek, M., and Moore, T. (2014). Game-theoretic analysis of ddos attacks against bitcoin mining pools. In *Financial Cryptography and Data Security*, pages 72–86. Springer.

Karame, G., Androulaki, E., and Capkun, S. (2012). Two bitcoins at the price of one? double-spending attacks on fast payments in bitcoin. *IACR Cryptology ePrint Archive*, 2012:248.

Kent, S., Lynn, C., and Seo, K. (2000). Secure border gateway protocol (S-BGP). *J. Selected Areas in Communications*, 18(4):582–592.

Kent, S. and Ma, D. (2017). *RFC 8211: Adverse Actions by a Certification Authority (CA) or Repository Manager in the Resource Public Key Infrastructure (RPKI).* Internet Engineering Task Force (IETF). `http://tools.ietf.org/html/rfc8211`.

Kent, S. and Mandelberg, D. (2013). *Suspenders: A Fail-safe Mechanism for the RPKI.* Internet Engineering Task Force (IETF). `http://tools.ietf.org/html/draft-kent-sidr-suspenders-00`.

King, L. (2014). Bitcoin hit by 'massive' ddos attack as tensions rise. *Forbes*.

Koshy, P., Koshy, D., and McDaniel, P. (2014). An analysis of anonymity in bitcoin using p2p network traffic. In *Financial Cryptography and Data Security*, pages 468–485. Springer.

Kroll, J. A., Davey, I. C., and Felten, E. W. (2013). The economics of bitcoin mining, or bitcoin in the presence of adversaries. In *Proceedings of WEIS*, volume 2013.

Kuerbis, B. (2013). Keep your pants on: Governments want suspenders for secure routing. Internet Governance Project.

Kumaresan, R. and Bentov, I. (2014). How to use bitcoin to incentivize correct computations. In *ACM-CCS*.

Kumaresan, R., Moran, T., and Bentov, I. (2015). How to use bitcoin to play decentralized poker. In *ACM-CCS*.

Lab, D. R. (2015). rcynic software. `http://trac.rpki.net`.

LACNIC (2013). RPKI looking glass.
`www.labs.lacnic.net/rpkitools/looking_glass/`.

Laszka, A., Johnson, B., and Grossklags, J. (2015). When bitcoin mining pools run dry. *2nd Workshop on Bitcoin Research (BITCOIN)*.

Laurie, B., Langley, A., and Kasper, E. (2013). Certificate transparency. *Network Working Group Internet-Draft, v12, work in progress. http://tools. ietf. org/html/draft-laurie-pki-sunlight-12*.

Ledesma, L. (2021). Bitcoin trading volumes returned as price spiked over 40k. *Yahoo Finance*. Accessed: 2021-06-27.

Lepinski, M., editor (2012). *BGPSEC Protocol Specification*. IETF Network Working Group, Internet-Draft. Available from `http://tools.ietf.org/html/draft-ietf-sidr-bgpsec-protocol-04`.

Lepinski, M. and Kent, S. (2012). *RFC 6480: An Infrastructure to Support Secure Internet Routing*. Internet Engineering Task Force (IETF). `http://tools.ietf.org/html/rfc6480`.

Li, D., Zou, H., and Shao, Q. (2018). Technical report: Rpki monitor and visualizer for detecting and alerting for rpki errors. nternet DNS Beijing Engineering Research Center, ZDNS.

Limited, E. E. (2016). Elliptic: The global standard for blockchain intelligence. `https://www.elliptic.co/`.

Liu, S. and Hopwood, D. (2018). (ZIP: 201), Network Peer Management for Overwinter. github. https://github.com/zcash/zips/blob/main/zip-0201.rst.

Lychev, R., Goldberg, S., and Schapira, M. (2013). Is the juice worth the squeeze? BGP security in partial deployment. In *SIGCOMM'13*.

Manderson, T., Vegoda, L., and Kent, S. (1973). *RFC 6491: Resource Public Key Infrastructure (RPKI) Objects Issued by IANA"*. Internet Engineering Task Force (IETF). `http://tools.ietf.org/html/rfc6491`.

Marcus, Y., Heilman, E., and Goldberg, S. (2018). Low-resource eclipse attacks on ethereum's peer-to-peer network. *IACR Cryptol. ePrint Arch.*, 2018:236.

Maxwell, G. (2011). Zero Knowledge Contingent Payment. *Bitcoin Wiki.*

Maxwell, G. (2013a). CoinJoin: Bitcoin privacy for the real world. *Bitcoin-talk.*

Maxwell, G. (2013b). CoinSwap: transaction graph disjoint trustless trading. *Bitcoin-talk.*

Maxwell, G. (2016). The first successful Zero-Knowledge Contingent Payment. Bitcoin Core.

Meiklejohn, S. and Orlandi, C. (2015). Privacy-Enhancing Overlays in Bitcoin. In *Lecture Notes in Computer Science*, volume 8976. Springer Berlin Heidelberg.

Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., Voelker, G., Savage, S., and McCoy, D. (2013). A fistful of bitcoins: Characterizing payments among men with no names. In *ACM-SIGCOMM Internet Measurement Conference, IMC.*

Miers, I., Garman, C., Green, M., and Rubin, A. D. (2013). Zerocoin: Anonymous distributed e-cash from bitcoin. In *IEEE Security and Privacy (SP)*, pages 397–411.

Miller, A., Litton, J., Pachulski, A., Gupta, N., Levin, D., Spring, N., and Bhattacharjee, B. (2015). Discovering bitcoin's network topology and influential nodes. Technical report, University of Maryland.

Misel, S. (1997). "Wow, AS7007!". Merit NANOG Archive. `www.merit.edu/mail.archives/nanog/1997-04/msg00340.html`.

mmitech (2013). Ghash.io and double-spending against betcoin dice. Bitcoin Forum. `https://bitcointalk.org/index.php?topic=327767.0`.

Mohapatra, P., Scudder, J., Ward, D., Bush, R., and Austein, R. (2013). *RFC 6811: BGP prefix origin validation.* Internet Engineering Task Force (IETF). `http://tools.ietf.org/html/rfc6811`.

Monero (2016). Monero, `https://getmonero.org/home`.

Moreno-Sanchez, P., Ruffing, T., and Kate, A. (2016). P2P Mixing and Unlinkable P2P Transactions. *Draft.*

Möser, M. and Böhme, R. (2016). Join Me on a Market for Anonymity. *Workshop on Privacy in the Electronic Society.*

Mueller, M. and Kuerbis., B. (2011). Negotiating a new governance hierarchy: An analysis of the conflicting incentives to secure internet routing. *Communications and Strategies*, 1(81):125–142.

Mueller, M., Schmidt, A., and Kuerbis., B. (2013). Internet security and networked governance in international relations. *International Studies Review*, 15(1):86–104.

Murdoch, S. J. and Anderson, R. (2008). *Access Denied: The Practice and Policy of Global Internet Filtering*, chapter Tools and technology of Internet filtering, pages 57–72. MIT Press.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. metzdowd.com.

namecoin (2015). Github: namecoin repository. `https://github.com/namecoin/namecoin`.

Narayanan, A., Bonneau, J., Felten, E., Miller, A., and Goldfeder, S. (2016). *Bitcoin and cryptocurrency technologies*. Princeton University Pres.

Natoli, C. and Gramoli, V. (2017). The balance attack or why forkable blockchains are ill-suited for consortium. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 579–590. IEEE.

Nayak, K., Kumar, S., Miller, A., and Shi, E. (2016). Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 305–320. IEEE.

NIST (2013a). RPKI deployment monitor. nist.gov.

NIST (2013b). Workshop on Improving Trust in the Online Marketplace. `http://www.nist.gov/itl/csd/ct/ca-workshop-agenda2013.cfm`.

nopara73 (2017). Introducing hiddenwallet : full block spv tumblebit wallet — testing release. `https://medium.com/hackernoon/introducing-hiddenwallet-full-block-spv-tumblebit-wallet-testing-release-1054a15a9bb1`.

Noubir, G. and Sanatinia, A. (2016). Honey onions: Exposing snooping tor hsdir relays. In *DEF CON 24*.

OpenSSL (2014). TLS heartbeat read overrun (CVE-2014-0160).
`https://www.openssl.org/news/secadv_20140407.txt`.

Osipkov, I., Vasserman, E. Y., Hopper, N., and Kim, Y. (2007). Combating double-spending using cooperative p2p systems. In *27th International Conference on Distributed Computing Systems (ICDCS'07)*, pages 41–41. IEEE.

Osterweil, E., Amante, S., Massey, D., and McPherson, D. (2011). The great ipv4 land grab: resource certification for the ipv4 grey market. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 12. ACM.

Osterweil, E., Manderson, T., White, R., and McPherson, D. (2012). Sizing estimates for a fully deployed rpki. Technical report, Verisign Labs Technical Report.

Pagnia, H. and Gartner, F. C. (1999). On the impossibility of fair exchange without a trusted third party.

Peck, M. (2016). DAO May Be Dead After $60 Million Theft. *IEEE Spectrum, Tech Talk Blog.*

Peterson, A. (2013). Researchers say u.s. internet traffic was re-routed through belarus. that's a problem. *Washington Post*.

Pilosov, A. and Kapela, T. (2009). Stealing the internet.

Piscitello, D. (2012). Guidance for preparing domain name orders, seizures & takedowns. Technical report, ICANN.

Piscitello, D. (2013). The value of assessing collateral damage before requesting a domain seizure. Technical report, ICANN.

Plohmann, D. and Gerhards-Padilla, E. (2012). Case study of the miner botnet. In *Cyber Conflict (CYCON), 2012 4th International Conference on*, pages 1–16. IEEE.

Poon, J. and Dryja, T. (2015). The bitcoin lightning network: Scalable off-chain instant payments. Technical report, Technical Report (draft). https://lightning.network.

Project, I. G. (2011). In important case, RIPE-NCC seeks legal clarity on how it responds to foreign court orders. www.internetgovernance.org.

Qiu, J., Zhihong, T., Jianwei, Y., and Shuofei, T. (2011). Design, implementation and optimization of network access control system based on routing diffusion. In *Information Technology and Artificial Intelligence Conference (ITAIC), 2011 6th IEEE Joint International*, volume 2, pages 121–125. IEEE.

Rackspace (2014). Rackspace: Requesting additional ipv4 addresses for cloud servers. www.rackspace.com/knowledge_center. Accessed: 2014-06-18.

Ramasubramanian, V. and Sirer, E. G. (2004). The design and implementation of a next generation name service for the internet. *ACM SIGCOMM Computer Communication Review*, 34(4):331–342.

Rensys Blog (2008). Pakistan hijacks YouTube. `http://www.renesys.com/blog/2008/02/pakistan_hijacks_youtube_1.shtml`.

Research, C. (2010). Sec 2: Recommended elliptic curve domain parameters.

RIPE (2013). RPKI validator. `http://localcert.ripe.net:8088/trust-anchors`.

RIPE (2014). Ripestat. `https://stat.ripe.net/data/announced-prefixes`.

RIPE (2015). Latest delegations. `ftp://ftp.ripe.net/pub/stats/ripencc/delegated-ripencc-extended-latest`.

RIPE (2015). RIPE RIS raw data. `http://www.ripe.net/data-tools/stats/ris/ris-raw-data`.

RoadTrain (2013). Bitcoin-talk: Ghash.io and double-spending against betcoin dice. `https://bitcointalk.org/index.php?topic=321630.msg3445371#msg3445371`.

Ron, D. and Shamir, A. (2013). Quantitative analysis of the full bitcoin transaction graph. In *Financial Cryptography and Data Security*, pages 6–24. Springer.

Rosenfeld, M. (2014). Analysis of hashrate-based double spending. *arXiv preprint arXiv:1402.2009*.

Rossow, C., Andriesse, D., Werner, T., Stone-Gross, B., Plohmann, D., Dietrich, C. J., and Bos, H. (2013). Sok: P2pwned-modeling and evaluating the resilience of peer-to-peer botnets. In *IEEE Symposium on Security and Privacy*, pages 97–111. IEEE.

Ruffing, T., Moreno-Sanchez, P., and Kate, A. (2014). Coinshuffle: Practical decentralized coin mixing for bitcoin. In *ESORICS*, pages 345–364. Springer.

Saxena, A., Misra, J., and Dhar, A. (2014). Increasing anonymity in bitcoin. In *Financial Cryptography and Data Security*, pages 122–139. Springer.

Schneier, B. and Kelsey, J. (1997). Automatic event-stream notarization using digital signatures. In *Security Protocols*, pages 155–169. Springer.

Shomer, A. (2014). On the phase space of block-hiding strategies. *IACR Cryptology ePrint Archive*, 2014:139.

Shrishak, K. and Shulman, H. (2020). Limiting the power of rpki authorities. In *Proceedings of the Applied Networking Research Workshop*, pages 12–18.

Singh, A., Ngan, T.-W. J., Druschel, P., and Wallach, D. S. (2006). Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM*.

Sipa (2015). (Dogecoin) Always use a 50% chance to choose between tried and new entries. github. https://github.com/dogecoin/dogecoin/commit/c6a63ceeb4956.

Sit, E. and Morris, R. (2002). Security considerations for peer-to-peer distributed hash tables. In *Peer-to-Peer Systems*, pages 261–269. Springer.

Spider, R. (2015). Rpki spider. `http://rpkispider.verisignlabs.com/`.

Stock, B., Gobel, J., Engelberth, M., Freiling, F. C., and Holz, T. (2009). Walowdac: Analysis of a peer-to-peer botnet. In *European Conference on Computer Network Defense (EC2ND)*, pages 13–20. IEEE.

Stockinger, J., Haslhofer, B., Moreno-Sanchez, P., and Maffei, M. (2021). Pinpointing and measuring wasabi and samourai coinjoins in the bitcoin ecosystem. *arXiv preprint arXiv:2109.10229*.

Stone, J. (2015). Evolution Downfall: Insider 'Exit Scam' Blamed For Massive Drug Bazaar's Sudden Disappearance. *international business times*.

Surfnet (2013). RPKI dashboard. `http://rpki.surfnet.nl/validitytables.html`.

Szabo, N. (1997). Formalizing and securing relationships on public networks. *First Monday*, 2(9).

the Internet Archive (2015). Http Archive: Trends. `http://httparchive.org/trends.php`.

The President's National Security Telecommunications Advisory Committee (2011). Nstac report to the president on communications resiliency.

Todd, P. (2014). BIP-0065: OP CHECKLOCKTIMEVERIFY. *Bitcoin Improvement Proposal*.

Tran, M., Choi, I., Moon, G. J., Vu, A. V., and Kang, M. S. (2020). A stealthier partitioning attack against bitcoin peer-to-peer network. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 894–909. IEEE.

Tran, M., Shenoi, A., and Kang, M. S. (2021). On the routing-aware peering against network-eclipse attacks in bitcoin. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

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

Urdaneta, G., Pierre, G., and Steen, M. V. (2011). A survey of dht security techniques. *ACM Computing Surveys (CSUR)*, 43(2):8.

Valenta, L. and Rowan, B. (2015). Blindcoin: Blinded, accountable mixes for bitcoin. In *FC*.

Vasek, M., Thornton, M., and Moore, T. (2014). Empirical analysis of denial-of-service attacks in the bitcoin ecosystem. In *Financial Cryptography and Data Security*, pages 57–71. Springer.

Views, R. (2015). University of oregon route views project.
`http://www.routeviews.org/`.

Wählisch, M., Maennel, O., and Schmidt, T. (2012). Towards detecting BGP route hijacking using the RPKI. In *Poster: SIGCOMM'12*, pages 103–104. ACM.

Wang, L., Park, J., Oliveira, R., and Zhang, B. (2013). Internet topology collection.
`http://irl.cs.ucla.edu/topology/`.

White, R. (2003, expired). Deployment considerations for secure origin BGP (soBGP). draft-white-sobgp-bgp-deployment-01.txt.

wikipedia (2016). Bitcoin Fog.

Wu, L., Hu, Y., Zhou, Y., Wang, H., Luo, X., Wang, Z., Zhang, F., and Ren, K. (2021). Towards understanding and demystifying bitcoin mixing services. In *Proceedings of the Web Conference 2021*, pages 33–44.

Wuille, P. (2015). Segregated witness and its impact on scalability
`https://www.youtube.com/watch?v=NOYNZB5BCHM`.

Xing, Q., Baosheng, W., and Xiaofeng, W. (2018). Bgpcoin: blockchain-based internet number resource authority and bgp security solution. *Symmetry*, 10(9):408.

Zhang, S. and Lee, J.-H. (2019). Double-spending with a sybil attack in the bitcoin decentralized network. *IEEE transactions on Industrial Informatics*, 15(10):5715–5722.

Zhang, X., Hsiao, H.-C., Hasker, G., Chan, H., Perrig, A., and Andersen, D. G. (2011). SCION: scalability, control, and isolation on next-generation networks. In *IEEE Security and Privacy (SP)*.

Zhou, L., Schneider, F. B., and Van Renesse, R. (2002). COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems (TOCS)*, 20(4):329–368.

Ziegeldorf, J. H., Grossmann, F., Henze, M., Inden, N., and Wehrle, K. (2015). Coinparty: Secure multi-party mixing of bitcoins. In *CODASPY*.

CURRICULUM VITAE

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]