Eclipse Attacks on Bitcoin's Peer-to-Peer Network

March 20, 2015.

Ethan Heilman[*] *Alison Kendler*[*] *Aviv Zohar*[†] *Sharon Goldberg*[*]
[*]*Boston University* [†]*Hebrew University*

Abstract

We present eclipse attacks on bitcoin's peer-to-peer network. Our attack allows an adversary controlling a sufficient number of IP addresses to monopolize all connections to and from a victim bitcoin node. The attacker can then exploit the victim for attacks on bitcoin's mining and consensus system, including N-confirmation double spending, selfish mining, and adversarial forks in the blockchain. We take a detailed look at bitcoin's peer-to-peer network, and quantify the resources involved in our attack via probabilistic analysis, Monte Carlo simulations, measurements and experiments with live bitcoin nodes. Finally, we present countermeasures, inspired by botnet architectures, that are designed to raise the bar for eclipse attacks while preserving the openness and decentralization of bitcoin's current network architecture.

1 Introduction

While cryptocurrency has been studied since the 1980s [17, 18, 20], bitcoin is the first to see widespread adoption with a variety of exchanges, banks, startups and point-of-sale systems contributing to its now-vibrant ecosystem. One key reason for bitcoin's success is its baked-in decentralization. Instead of relying on a central bank or government to regulate currency, bitcoin uses a decentralized network of nodes that use computational proofs-of-work to reach consensus on the state of bitcoin's distributed ledger of transactions, *aka.,* the *blockchain*. In his white paper [41], Satoshi Nakamoto 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.,* [8, 22, 29, 30, 35, 39–41, 47, 49], less attention has been paid to the peer-to-peer network used to broadcast information between bitcoin nodes (see Section 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 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 paper, we present and quantify the resources required for *eclipse attacks* [19, 50, 51] on bitcoin nodes with public IPs. In an eclipse attack, 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 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). With high probability, the victim then forms all eight of its outgoing connections to attacker-controlled addresses, and the attacker also monopolizes victim's 117 incoming connections.

Our eclipse attack uses extremely low-rate TCP con-

nections, so the main challenge for the attacker is to obtain a sufficient number of IP addresses (Section 4). We consider two attack types: (1) infrastructure attacks, modeling that 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 4) measurements, (Section 5) and experiments on our own live bitcoin nodes (Section 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 4.1.4). 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 eclipse a victim with over 85% probability in the *worst case*; 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 [3]) and honestly followed the peer-to-peer protocol, they could eclipse a target with probability about $(\frac{8192}{7200+8192})^8 = 0.6\%$.

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 [2]: (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 countermeasures that preserve openness by allowing unsolicited incoming connections, while raising the bar for eclipse attacks (Section 7). Today, an attacker with enough addresses can eclipse *any* victim that accepts incoming connections (once the victim restarts). Our countermeasures ensure that, with high probability, if a victim stores enough addresses of legitimate nodes that accept incoming connections, then the victim cannot be overwhelmed by an eclipse attack *regardless of the number of IP addresses the attacker controls.*

Disclosure. We have disclosed our results to the bitcoin developers, and are working with them on patches implementing our countermeasures.

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 an x-fraction of miners can engineer blockraces by hording blocks discovered by an eclipsed miner, and releasing blocks to both the eclipsed and non-eclipsed miners only once a competing block has been found. Thus, the x-fraction of eclipsed miners waste effort on blocks that never become part of the blockchain.

Splitting mining power. Eclipsing an x-fraction of miners eliminates their mining power from the rest of network, making it easier to launch mining attacks (*e.g.,* the 51% attack [41]). To hide the change in mining power under natural variations [13], miners could be eclipsed slowly or intermittently.

Selfish mining. With selfish mining [8, 22, 30, 49], 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.

0-confirmation double spend. A double-spend attack allows an attacker to spend some bitcoins multiple times. 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 [12]. These transactions are used when it is inappropriate to wait the 5-10 minutes typically needed to for a block confirmation [14], *e.g.,* in retail point-of-sale systems like BitPay [4], or online gambling sites like Betcoin [46]. To launch a double-spend attack against the merchant [36], 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 [46], and this attack is as effective as a Finney attack [32], but uses eclipsing instead of mining.

N-confirmation double spend. 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 [12]. By eclipsing the merchant and an x-fraction of miners, the attacker can launch N-confirmation double-spending attacks. The attacker's transaction T is sent to the x-fraction of eclipsed miners, who incorporate it into their (obsolete) view of the blockchain. The attacker shows this view of blockchain to the eclipsed merchant, receives the goods, and then sends both the merchant and eclipsed miners the (non-obsolete) view of blockchain from the non-eclipsed mi ners. The eclipsed miners' blockchain is orphaned, and the attacker obtains goods without paying.

Other attacks exist, *e.g.,* a transaction hiding attack on SPV clients [10]. Section 8 has other related work.

2 Bitcoin's Peer-to-Peer Network

We start by describing bitcoin's peer-to-peer network. This discussion is based on bitcoind version 0.9.3, the most current release from 9/27/2014 to 2/16/2015; its networking code has been largely unchanged since 2013. This client was originally written by Satoshi Nakamoto, and has near universal market share; on 2/11/2015, 97% of nodes tracked by Bitnode.io run bitcoind [3].

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 *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 [10, 11] instead. We now describe how nodes propagate and store network information, and how they select outgoing connections.

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 nodes that are active on the bitcoin network. The DNS seeder obtains these addresses by periodically crawling the bitcoin network. As of this writing, the bitcoin network has six DNS seeders, and information from DNS seeders is only used in two cases. The first case is when a new node joins the bitcoin network for the first time; it first attempts to connect the DNS seeders to obtain a list of active IPs, and otherwise fails over to a hardcoded list of about 600 IP addresses. The second case is when a node that is part of the bitcoin network restarts, and reconnects to new peers; here, the DNS 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 `ADDR` messages. An `ADDR` message is solicited *only* upon establishing a outgoing connection with a peer; the peer responds with an `ADDR` message 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.

2.2 Storing network information

Public IPs are stored in a node's `tried` and `new` tables.

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.[2] 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.

[1] This is a configurable, but we assume that nodes use the default.

[2] For IPv6 addresses matching prefix 2001:0470* (which is allocated to Hurricane Electric), the group is the /36 IP prefix that contains IPv6 address of the peer. For all other IPv6 addresses,the group is the /32 IP prefix that contains IPv6 address of the peer. For OnionCat Tor addresses, the group is the first 4 bits of the OnionCat address.

When a node successfully connects to a peer, the peer's address is inserted into the appropriate `tried` bucket. If the bucket this 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.

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 to whether to select an address from `tried` or `new`, where

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

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

$$\pi(r,\tau) = \max(1, \tfrac{1.2^r}{1+\tau}) \quad (2)$$

else, reject the address and return to (i). The acceptance probability $\pi(r,\tau)$ 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.[3]

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

3 The Eclipse Attack

Our eclipse attack is for a victim with a public IP. Our attacker (1) populates the `tried` table with the addresses of its own 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 [45]) or as "reserved for future use" by [33] (*e.g.,* 252.0.0.0/8). We fill `new` with "trash" because, unlike attacker addresses, "trash" is not a scarce resource. The attack persists until (3) the victim node restarts and chooses new outgoing connections (per Section 2.3). With high probability, the victim establishes all eight outgoing connections to adversarial addresses; all of these addresses will be from the `tried` table, since the victim cannot connect to the "trash" in the `new` table. Finally, the attacker will (5) occupy the victim's remaining 117 incoming connections. Below, we detail each step of the attack.

3.1 Populating `tried` and `new`

We exploit the following key aspects of the bitcoin peer-to-peer network to fill `tried` and `new`:

First, 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 2.2).

Second, a node accepts large unsolicited `ADDR` messages from its peers. These addresses are inserted directly into the `new` table without testing their connectivity (Section 2.2). Thus, when our attacker connects to

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

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.

Third, nodes only rarely solicit network information from peers and DNS seeders (Section 2.1). Thus, while the attacker populates the victim's tried and new tables, the victim almost never counteracts the flood of adversarial information by querying legitimate peers or seeders.

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. Perhaps the most predictable 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. Moreover, updating is often *not* optional, since some updates solve critical security issues; 2013 saw three critical security vulnerabilities, and in 2014 the heartbleed bug [43] triggered a software upgrade. Also, since the community needs to be notified about an upgrade in advance, the attacker could watch always for notifications and then commence its attack [1]. Power-cycling the node will also cause a restart; for example, [10] found that a node with a public IP has a 25% chance of going offline after 10 hours. Packets-of-death like CVE-2013-5700 [5] (patched in 2013), DDoS [37, 55] or memory exhaustion [10] attacks can also force a restart.

3.3 Selecting outgoing connections

Our attack succeeds if, upon restart, the victim makes eight 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 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 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. These addresses have timestamps that are updated at 20 minute or longer intervals (Section 2.2). We assume the worst case, so that these timestamps are fresh (*i.e.,* $\tau = 0$) when the victim restarts. A typical victim has least eight such addresses, corresponding to eight outgoing connections.

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 (2); here we use the fact that the adversary's addresses are no older than τ_a, the length of the round. If r 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

$$g(i,f,f',\tau_a,\tau_\ell) = (1-p(i,\tau_a))\cdot f + (1-p(i,0))\cdot f' + (1-p(i,\tau_\ell))\cdot(1-f-f')$$

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

$$q(f,f',\tau_a,\tau_\ell) = \sum_{r=1}^{\infty} p(r,\tau_a)\cdot f\prod_{i=1}^{r-1} g(i,f,f',\tau_a,\tau_\ell) \quad (3)$$

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 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 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. With random selection, the adversary needs to fill $f = 98.66\%$ of the tried table to have a 90% success probability. Today, the adversary has a 90% success probability even if it only fills $f = 72\%$ of tried, as long as it invests $\tau_\ell = 48$ hours into the attack, with $\tau_a = 27$ minute rounds.

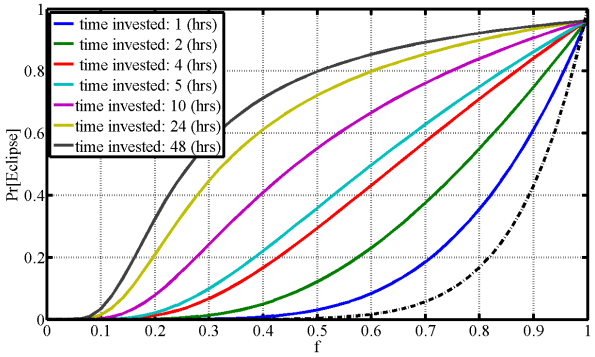

Figure 1: Probability of eclipsing a node $q(f, f', \tau_a, \tau_\ell)$[8] (equation (3)) vs f the fraction of adversarial addresses in `tried`, for different values of time invested in the attack τ_ℓ. Round length is $\tau_a = 27$ minutes, and $f' = \frac{8}{64\times64}$. The dotted line shows the probability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

3.4 Monopolizing the eclipsed victim

Our attack involves establishing and occupying TCP connections to a victim node. 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. Instead, an attacker can write a custom TCP stack and open an arbitrary number of TCP connections. A custom stack (*e.g.,* that of ZMAP [28]) requires minimal CPU and memory, and is typically bottlenecked only by bandwidth, since computers can generate traffic much faster than they can send it. The bandwidth costs of our attack's TCP connections are 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 117 remaining incoming connections. To prevent others from connecting to the victim, these 125 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 117 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 117 TCP connections costs $\frac{164\times117}{80\times60} = 3.9975$ 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* [25], 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.

4 How Many Attack Addresses?

Section 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 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:

Infrastructure attack (Section 4.1). 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 4.1.4 discusses organizations that can launch this attack.

Botnet attack (Section 4.2). The attacker holds several IP addresses, each in a *distinct* group. This models attacks by a botnet or other network of hosts scattered in diverse IP address blocks (*e.g.,* PlanetLab [21]). Section 4.2.1 explains why many botnets have enough IP address diversity for this attack.

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

4.1 Infrastructure attack

The attacker holds addresses in s distinct *groups*. We use mathematical modeling to determine how many groups s the attacker needs (Section 4.1.1), and how many addresses per group (Section 4.1.2). We validate our results with simulations, discuss implications (Sections 4.1.3) and show who can launch this attack (Section 4.1.4).

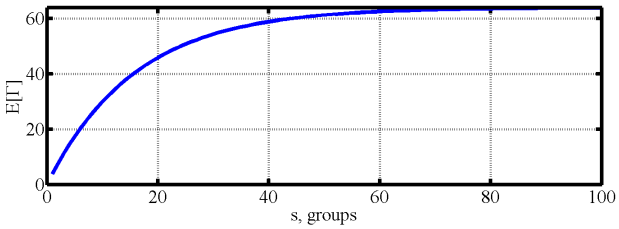

Figure 2: Infrastructure attack. $E[\Gamma]$ (expected number of non-empty buckets) in `tried` vs s (number of groups).

4.1.1 How many groups?

We can model the process of populating `tried` (per Section 2.2) by supposing that four independent hash functions map each of the s groups to one of 64 buckets in `tried`. Thus, letting $\Gamma \in [0,64]$ be a random variable that counts the number of non-empty buckets in `tried`, we use Lemma A.1 from Appendix A to find that

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

Figure 2 plots $E[\Gamma]$; we expect to fill 55.5 of 64 buckets with $s = 32$, and all but one bucket with $s > 67$ groups.

4.1.2 How many addresses per group?

We first find how many distinct addresses *hash* to a given a bucket, given s distinct groups and t distinct addresses per group. Then, we work out how many of the addresses hashing to a bucket will actually be *stored* in the bucket, by considering how the *bitcoin eviction* discipline evicts old addresses in favor of new ones.

How many addresses hash to a bucket? Recall that each group makes 4 uniform random draws of one of 64 possible buckets in `tried`. Considering a single bucket i, the probability that a single group hashes to bucket i is

$$\tfrac{1}{\alpha} = 1-(\tfrac{63}{64})^4 \approx \tfrac{1}{16} \tag{5}$$

If G_i counts the number of distinct groups hashing to bucket i, then G_i is binomially distributed[4] as $G_i \sim B(s, \frac{1}{\alpha})$. If $G_i = g$ groups hash to bucket i, and each group contains t addresses hashing to up to 4 distinct buckets, then the number of addresses hashing to bucket i is the random variable $A_i \sim B(gt, \frac{1}{4})$, with distribution

$$\Pr[A_i = a] = \sum_{g=0}^{s} \Pr[B(gt,\tfrac{1}{4}) = a]\Pr[B(s,\tfrac{1}{\alpha}) = g] \tag{6}$$

How many addresses are stored in a bucket? Now that we know that A_i addresses hash to bucket i, we need to figure out how many of these addresses will actually be *stored* in the bucket. This depends on the state of the buckets prior to the attack (*i.e.,* are they full or empty?), and the way old addresses are evicted. We consider a variety of situations.

1. Initially empty. We start with a best-case analysis, supposing that bucket i is initially empty. Since a bucket can store at most 64 addresses, the number of addresses stored in bucket i is the random variable $\min(64, A_i)$. But what is really of interest to us is the following quantity, which can easily be determined from equation (6) and evaluated numerically:

$$E[\min(64, A_i) | G_i > 0] \tag{7}$$

This gives the expected number of addresses that end up getting stored in bucket i, given that *at least one group hashes to bucket i*. This is the quantity of interest, since if no groups map to bucket i, we cannot fill the bucket by increasing t, the number of addresses per group; instead we must increase s, the number of groups.

2. Bitcoin eviction. Now we take a worst-case approach and suppose that bucket i is completely 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

$$E[Y_a | Y_{a-1}] = Y_{a-1} + 1 - (\tfrac{Y_{a-1}}{64})^4 \tag{8}$$

$$E[Y_1] = 1 \tag{9}$$

where the second equation follows because the bucket is initially full of legitimate addresses. We have just derived a recurrence relation for $E[Y_a]$! We just solve the recurrence numerically, and find the $E[Y_a] > 63$ for $a \geq 101$; thus, the adversary can expect to overwrite 63 of the 64 legitimate addresses in the bucket after inserting 101 unique addresses. Finally, we obtain the expected number of addresses stored in bucket i, given that at least one group hashes to bucket i, by taking

$$\sum_{a=0}^{st} E[Y_a]\Pr[A_i = a | G_i > 0] \tag{10}$$

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

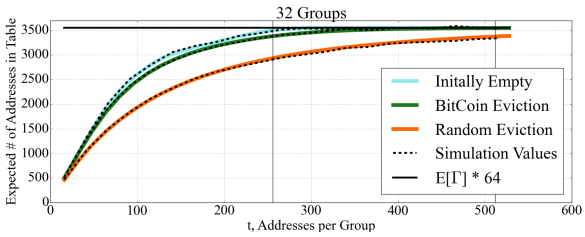

Figure 3: Infrastructure attack with $s = 32$ groups: the expected number of addresses stored in `tried` for different scenarios vs the number of addresses per group t. Values were computed by taking the product of equation (4) and equation (7) or (10), and confirmed by Monte Carlo simulations. The horizontal line assumes all $E[\Gamma]$ buckets per (4) are full.

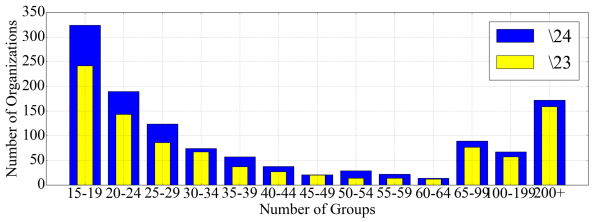

Figure 4: Histogram of the number of organizations with s groups. For the /24 data, we require $t = 256$ addresses per group; for /23, we require $t = 512$.

which we can by compute numerically by combining the recurrence relation for $E[Y_a]$ in equations (8)-(9) with the distribution of A_i from equation (6).

3. Random eviction. We once again assume that bucket i is completely full of legitimate addresses, but now we assume that each time an address is inserted it evicts a randomly-selected address. (This is not actually what bitcoin does, but we analyze it for comparison.) If Y_a is defined as above, then Lemma A.1 gives

$$E[Y_a] = 64(1 - (\tfrac{63}{64})^a) \tag{11}$$

and substituting equation (11) into (10) we get the expected number of addresses stored in bucket i, given that at least one group hashes to bucket i.

4. Exploiting multiple rounds. Our eclipse attack proceeds in *rounds*; in each round the attacker repeatedly inserts each of the t addresses in each of his s groups 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 stored in bucket i is given by equation (7); with multiple rounds the attack performs as in the best-case scenario.

4.1.3 How full is the `tried` table?

We determine the fraction f of `tried` filled by adversarial addresses. While it's not difficult to precisely determine the exact value of $E[f]$ from the equations we already derived, we keep things simple by taking the product of $E[\Gamma]$ and the expected number of addresses stored per bucket, given that at least one group maps to that bucket, for the scenarios of Section 4.1.2. We plot this estimate in Figure 3, which also shows the results of Monte Carlo simulations of the process of inserting addresses into `tried`, plotting the average of 100 trials. The horizontal lines in Figure 3 show what happens if each of $E[\Gamma]$ buckets per equation (4) are completely full.

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. 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. Referring back to Figure 1, if the round is $\tau_a = 27$ minutes long, and the attacker invests $\tau_\ell = 5$ hours in the attack, this attacker eclipses the victim with probability greater than 80%. The attack is almost as effective as in the bitcoin-eviction scenario with only one round; meanwhile, one round is much less effective when random eviction is used.

4.1.4 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 [53] and AS to prefix dataset [54] from July 2014, supplementing our data with information from the RIPE database [42]. We determined how many groups (*i.e.,* addresses in the same /16 IPv4 address block) and addresses per group are allocated to each organization. Figure 4 indicates that there are hundreds of organizations that have over $s = 32$ groups with at least $t = 256$ addresses per group; if these organizations invest $\tau_\ell = 5$ hours into an attack with a round length of $\tau_a = 27$ minutes, then their probability of eclipsing a victim is 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 Tele-

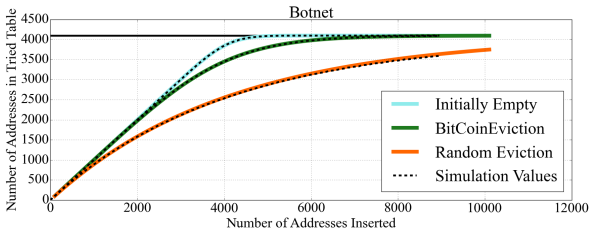

Figure 5: Botnet attack: the expected number of addresses stored in `tried` for different scenarios vs the number of addresses (bots) t. Values were computed from equations (12), (13) and (14), and confirmed by Monte Carlo simulations.

com), 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 ($s = 41$), and hundreds of other organizations.

4.2 Botnet attack

The botnet attacker holds t addresses in distinct groups. We therefore model each address as hashing to a uniformly random bucket in `tried`, and the number of addresses hashing to each bucket is distributed as $B(t, \frac{1}{64})$. How many of the 64×64 possible entries in `tried` can the botnet attacker occupy?

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

$$64E[\min(64, B(t, \tfrac{1}{64}))] \tag{12}$$

2. Bitcoin eviction. Suppose that all 64 buckets are full of legitimate addresses, and bitcoin eviction is used. The expected number of addresses stored in all 64 buckets is

$$64 \sum_{a=1}^{t} E[Y_a] \Pr[B(t, \tfrac{1}{64}) = a] \tag{13}$$

where $E[Y_a]$ comes from the recurrance relation (8)-(9).

3. Random eviction. Now we suppose that all 64 buckets are full of legitimate addresses, and that, instead of using bitcoin eviction, each time an address is inserted into a full bucket, it evicts a randomly selected address. We apply Lemma A.1 to find the expected number of addresses stored in all buckets is

$$E[Y] = 4096(1 - (\tfrac{4095}{4096})^t) \tag{14}$$

4. Exploiting multiple rounds. The attacker can exploit multiple rounds in order to increase the number of addresses it stores in `tried`. After sufficient rounds, the expected number of addresses is given by equation (12).

4.2.1 Who can launch a botnet attack?

Botnets have been known to attack bitcoin [35, 37, 55]. The Miner botnet, for example, had 29,000 hosts with public IPs that mined bitcoins [44]. While some botnet infestations concentrate in a few IP address ranges [52], it is important to remember that our eclipse attack requires no more than ≈ 6000 addresses in distinct groups; many botnets are orders of magnitude larger [48]. For example, while the Walowdac botnet was mostly in ranges 58.x-100.x and 188.x-233.x [52], these ranges create $42 \times 2^8 + 55 \times 2^8 = 24832$ bitcoin groups, which is more than enough for our attacks. The Carna botnet has published a list of its IPs [16]; randomly sampling from the list 5000 times, we find that 1250 bots give on average 402 distinct groups, enough to attack our live bitcoin nodes (Section 6). Furthermore, Figure 2 indicates that an infrastructure attack with more than $s = 200$ groups easily fills every bucket in `tried`; thus, with $s > 400$ groups, the attack performs as in Figure 5, even if multiple bots are in the same group.

4.3 Summary: infrastructure or botnet?

Figures 3, 5 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 (12)). By contrast, an infrastructure attacker needs 16,000 addresses, consisting of $s = 63$ groups (equation (4)) with $t = 256$ addresses per group.

However, per Section 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, 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 this requires only 3000 bots, or an infrastructure attack of $s = 20$ groups and $t = 256$ addresses per group (5120 addresses). To put this in context, if 3000 bots joined today's bitcoin network (that contains no more than 7200 public-IP nodes [3]) and honestly followed the peer-to-peer protocol, they could eclipse a target with probability about $(\frac{3000}{7200+3000})^8 = 0.006\%$.

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. Also, several people donated their peers files to us on 2/15/2015, and where appropriate we analyze those as well. Note, however, that there is evidence of wide variations in metrics for nodes

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

of different ages and in different regions [36]; as such, our analysis (Section 3-4) and some of our experiments (Section 6) focus on worst-case scenarios, where tables are initially full of fresh addresses.

Number of connections. Our attack requires the victim to have available slots for incoming connections. Figure 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 [11] 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 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.

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 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`.

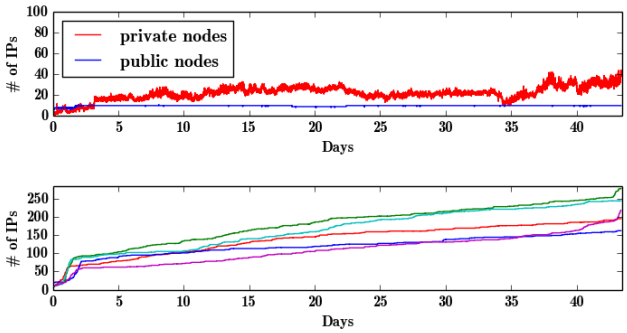

Figure 6: (Top) Connections vs time for one node. (Bottom) addresses in `tried` vs time for all our nodes.

Freshness of `tried`. Even those few addresses in `tried` are not especially fresh. Table 1 shows the age distribution of the addresses in `tried` for our nodes and from donated peers lists. 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 1 also shows that a small fraction of addresses in `tried` were online when we tried connecting to them on 2/17/2015.[5] This suggests further vulnerability to eclipse attacks, because if most legitimate addresses in `tried` are offline when a victim tries to connect, the victim is likely to connect to an adversarial address.

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) 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" [33]

[5] 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 1.

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

Table 2: Summary of our experiments.

IPs in 240.0.0.0/8-249.0.0.0/8 as 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`).

Initial conditions. We try various initial conditions:

1. Worst case. In the worst-case scenario of Section 4, 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 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 2). Results are in Table 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 3-4). 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 present predictions from Sections 3.3, 4. The *attack addrs* column gives the expected number of addresses in `new` (per Appendix B) and `tried`. For `tried`, we optimistically assume that the attacker runs his attack for enough rounds so that the expected number of addresses in `tried` is given by equation (4) multiplied by (7) for the infrastructure attack, and (12) for the botnet. The final column predicts success per Section 3.3 using *experimental values* for 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 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 1). Thus, Section 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 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;[6] 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.

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, by ensuring that a victim storing enough legitimate information cannot, with high probability, be overwhelmed by an eclipse attack. More formally: (1) If the victim has h legitimate addresses in `tried` before the attack, and a p-fraction of them accepting 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 (15). (2) If the victim's oldest outgoing connection is to a legitimate peer before the attack, then the eclipse attack *fails* if that peer accepts incoming connections when the victim restarts. Our countermeasures are inspired by botnet architectures (Section 8), and designed to be faithful to bitcoin's network architecture.

1. Deterministic random eviction. We suggest replacing the bitcoin eviction discipline as follows: just as each address deterministically hashes to a single bucket in `tried` and `new` (Section 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 4.1.2). Instead, addresses stored in `tried` is governed by equations (10) and (14); per Figures 3,5, this reduces the attack addresses 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). Meanwhile, if addresses were selected at random from `tried` and `new`, 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.

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

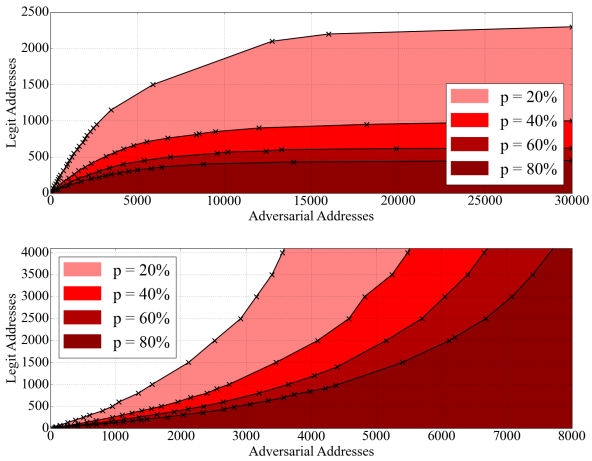

Figure 7: The area below each curve corresponds to a number of bots a that can eclipse a victim with probability at least 50%, given that the victim initially has h legitimate addresses in `tried`. We show one curve per churn rate p. (Top) With test before evict. (Bottom) Without.

These countermeasures harden the network without increasing storage or communication. We also ensure that random selection does not significantly increase the time to establish eight outgoing connections upon restart, by (shortly) introducing *feeler connections*. However, these two countermeasures still allow an attacker with enough addresses to overwrite all data in `tried`. Thus:

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

$$\Pr[\text{eclipse}] = f^8 < \left(1 - \frac{p \times h}{64 \times 64}\right)^8 \qquad (15)$$

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 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 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 two extra outgoing connections dedicated to testing randomly-selected addresses in `new`. If connection succeeds, the address is moved from `new` to `tried`, per the rules for inserting 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. Finally, we propose a countermeasure that is orthogonal to those above:

5. Anchor connections. Inspired by Tor entry guard rotation rates [26], 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.

We are working with the bitcoin developers on implementing countermeasures to our attacks as patches to bitcoind. See also other countermeasures in Appendix C.

8 Related Work

The bitcoin peer-to-peer network. Recently, a number of works have considered how bitcoin's peer-to-peer network can delay or prevent block propagation [24] or be used to deanonymize bitcoin users [10, 11, 38]. These works discuss aspects of bitcoin's peer-to-peer networking protocol, with [10] providing an excellent description of `ADDR` message propagation; we focus instead on the structure of the `tried` and `new` tables, and the way timestamps affect address storage and selection (Section 2). [11] shows that nodes connecting over Tor can be eclipsed by a Tor exit node. Other work has mapped bitcoin peers to autonomous systems [31], geolocated peers and measured churn [27], and used side channels to learn the bitcoin network topology [10].

Peer-to-peer and botnet architectures. Eclipse attacks affect many other peer-to-peer networks [19, 50, 51], and their close cousin, the botnet [48, 56]. A series of works have considered sampling peers in unstructured peer-to-peer networks subject to Byzantine attacks *e.g.,* [6, 9, 15, 34]. Our goals are less stringent; rather than requiring that each node is *equally likely* to be sampled by an honest node, we just want to mitigate against eclipse attacks on initially well-connected nodes. We are therefore inspired by botnet architectures, which have similar goals. Rossow *et al.* [48] 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 7). Storm uses test-before-evict [23], which we have also recommended for bitcoin. Zeus [7] disallows connections from multiple IP in the same /20, and regularly clean tables by testing if peers are online; our feeler connections are similar.

Implications of eclipse attacks. See Section 1.1.

9 Conclusion

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. 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 *worst case*. Moreover, a 400-node botnet sufficed for our own (non-worst-case) bitcoin nodes. Finally, we proposed countermeasures that make eclipse attacks more difficult while still preserving bitcoin's openness and decentralization. We have disclosed our results to the bitcoin developers, and are working with them on countermeasures to our attacks.

Acknowledgements

We thank Foteini Baldimtsi and Wil Koch for comments on this draft, various bitcoin users for donating their peers files to us, and the bitcoin core devs for helpful discussions. This work is supported by NSF award 1350733.

References

[1] Bitcoin: Common vulnerabilities and exposures. `https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures`. Accessed: 2014-02-11.

[2] Bitcoin wiki: Double-spending. `https://en.bitcoin.it/wiki/Double-spending`. Accessed: 2014-02-09.

[3] Bitnode.io snapshot of reachable nodes. `https://getaddr.bitnodes.io/nodes/`. Accessed: 2014-02-11.

[4] Bitpay: What is transaction speed? `https://support.bitpay.com/hc/en-us/articles/202943915-What-is-Transaction-Speed-`. Accessed: 2014-02-09.

[5] CVE-2013-5700: Remote p2p crash via bloom filters. `https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures`. Accessed: 2014-02-11.

[6] Emmanuelle Anceaume, Yann Busnel, and Sébastien Gambs. 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, 2013.

[7] Dennis Andriesse and Herbert Bos. An analysis of the zeus peer-to-peer protocol, April 2014.

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

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

[10] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. Deanonymisation of clients in Bitcoin P2P network. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 15–29. ACM, 2014.

[11] Alex Biryukov and Ivan Pustogarov. Bitcoin over tor isn't a good idea. *arXiv preprint arXiv:1410.6079*, 2014.

[12] Bitcoin Wiki. Confirmation. `https://en.bitcoin.it/wiki/Confirmation`, February 2015.

[13] Bitcoin Wisdom. Bitcoin difficulty and hash rate chart. `https://bitcoinwisdom.com/bitcoin/difficulty`, February 2015.

[14] blockchain.io. Average transaction confirmation time. `https://blockchain.info/charts/avg-confirmation-time`, February 2015.

[15] Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. Brahms: Byzantine resilient random membership sampling. *Computer Networks*, 53(13):2340–2359, 2009.

[16] Carna Botnet. Internet census 2012. `http://internetcensus2012.bitbucket.org/paper.html`, 2012.

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

[18] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In *EUROCRYPT*, 2005.

[19] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S Wallach. Secure routing for structured peer-to-peer overlay networks. *ACM SIGOPS Operating Systems Review*, 36(SI):299–314, 2002.

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

[21] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[22] Nicolas T Courtois and Lear Bahack. On subversive miner strategies and block withholding attack in bitcoin digital currency. *arXiv preprint arXiv:1402.1718*, 2014.

[23] Carlton R Davis, Jose M Fernandez, Stephen Neville, and John McHugh. Sybil attacks as a mitigation strategy against the storm botnet. In *3rd International Conference on Malicious and Unwanted Software, 2008.*, pages 32–40. IEEE, 2008.

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

[25] John Dillon. Bitcoin-development mailinglist: Protecting bitcoin against network-wide dos attack. `http://sourceforge.net/p/bitcoin/mailman/message/31168096/`, 2013. Accessed: 2014-02-11.

[26] Roger Dingledine, Nicholas Hopper, George Kadianakis, and Nick Mathewson. One fast guard for life (or 9 months). In *7th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2014)*, 2014.

[27] Joan Antoni Donet Donet, Cristina Pérez-Sola, and Jordi Herrera-Joancomartí. The bitcoin p2p network. In *Financial Cryptography and Data Security*, pages 87–102. Springer, 2014.

[28] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *Proceedings of the 22nd USENIX Security Symposium*, August 2013.

[29] Ittay Eyal. The miner's dilemma. *arXiv preprint arXiv:1411.7099*, 2014.

[30] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*, pages 436–454. Springer, 2014.

[31] Sebastian Feld, Mirco Schönfeld, and Martin Werner. Analyzing the deployment of bitcoin's p2p network under an as-level perspective. *Procedia Computer Science*, 32:1121–1126, 2014.

[32] Hal Finney. Bitcoin talk: Finney attack. `https://bitcointalk.org/index.php?topic=3441.msg48384#msg48384`, 2011. Accessed: 2014-02-12.

[33] IANA. Iana ipv4 address space registry. `http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xhtml`, January 2015.

[34] Gian Paolo Jesi, Alberto Montresor, and Maarten van Steen. Secure peer sampling. *Computer Networks*, 54(12):2086–2098, 2010.

[35] Benjamin Johnson, Aron Laszka, Jens Grossklags, Marie Vasek, and Tyler Moore. Game-theoretic analysis of ddos attacks against bitcoin mining pools. In *Financial Cryptography and Data Security*, pages 72–86. Springer, 2014.

[36] Ghassan Karame, Elli Androulaki, and Srdjan Capkun. Two bitcoins at the price of one? double-spending attacks on fast payments in bitcoin. *IACR Cryptology ePrint Archive*, 2012:248, 2012.

[37] Leo King. Bitcoin hit by 'massive' ddos attack as tensions rise. *Forbes `http://www.forbes.com/sites/leoking/2014/02/12/bitcoin-hit-by-massive-ddos-attack-as-tensions-rise/`* December 2 2014.

[38] Philip Koshy, Diana Koshy, and Patrick McDaniel. An analysis of anonymity in bitcoin using p2p network traffic. In *Financial Cryptography and Data Security*. 2014.

[39] Joshua A Kroll, Ian C Davey, and Edward W Felten. The economics of bitcoin mining, or bitcoin in the presence of adversaries. In *Proceedings of WEIS*, volume 2013, 2013.

[40] Aron Laszka, Benjamin Johnson, and Jens Grossklags. When bitcoin mining pools run dry. *2nd Workshop on Bitcoin Research (BITCOIN)*, 2015.

[41] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[42] RIPE NCC. Ripestat. `https://stat.ripe.net/data/announced-prefixes`, October 2014.

[43] OpenSSL. TLS heartbeat read overrun (CVE-2014-0160). `https://www.openssl.org/news/secadv_20140407.txt`, April 7 2014.

[44] Daniel Plohmann and Elmar Gerhards-Padilla. Case study of the miner botnet. In *Cyber Conflict (CYCON), 2012 4th International Conference on*, pages 1–16. IEEE, 2012.

[45] RIPE. Latest delegations. `ftp://ftp.ripe.net/pub/stats/ripencc/delegated-ripencc-extended-latest`, 2015.

[46] RoadTrain. Bitcoin-talk: Ghash.io and double-spending against betcoin dice. `https://bitcointalk.org/index.php?topic=321630.msg3445371#msg3445371`, 2013. Accessed: 2014-02-14.

[47] Meni Rosenfeld. Analysis of hashrate-based double spending. *arXiv preprint arXiv:1402.2009*, 2014.

[48] Christian Rossow, Dennis Andriesse, Tillmann Werner, Brett Stone-Gross, Daniel Plohmann, Christian J Dietrich, and Herbert Bos. Sok:

P2pwned-modeling and evaluating the resilience of peer-to-peer botnets. In *IEEE Symposium on Security and Privacy*, pages 97–111. IEEE, 2013.

[49] Assaf Shomer. On the phase space of block-hiding strategies. *IACR Cryptology ePrint Archive*, 2014:139, 2014.

[50] Atul Singh et al. Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM*. Citeseer, 2006.

[51] Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *Peer-to-Peer Systems*, pages 261–269. Springer, 2002.

[52] Ben Stock, Jan Gobel, Markus Engelberth, Felix C Freiling, and Thorsten Holz. Walowdac: Analysis of a peer-to-peer botnet. In *European Conference on Computer Network Defense (EC2ND)*, pages 13–20. IEEE, 2009.

[53] CAIDA UCSD. AS to Organization Mapping Dataset, July 2014.

[54] CAIDA UCSD. Routeviews prefix to AS Mappings Dataset for IPv4 and IPv6, July 2014.

[55] Marie Vasek, Micah Thornton, and Tyler Moore. Empirical analysis of denial-of-service attacks in the bitcoin ecosystem. In *Financial Cryptography and Data Security*, pages 57–71. Springer, 2014.

[56] Ping Wang, Lei Wu, Baber Aslam, and Cliff Changchun Zou. A systematic study on peer-to-peer botnets. In *ICCCN'09*, pages 1–8. IEEE, 2009.

A A Useful Lemma

The following simple lemma is plays a central role in the probabilistic analysis used in this paper.

Lemma A.1. *Suppose k items are randomly and independently inserted into n buckets. Let X be a random variable counting the number of non-empty buckets. Then*

$$E[X] = n\left(1-(\tfrac{n-1}{n})^k\right) \approx n(1-e^{-\frac{k}{n}}) \qquad (16)$$

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

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

Equation (16) follows from the approximation $(\frac{n-1}{n}) \approx e^{-1/n}$ for large n. □

B Overwriting the New Table

We now describe how the attacker should send `ADDR` messages in order to overwrite the `new` table with "trash" IP address. We use "trash" addresses from the unallocated Class A IPv4 address block 252.0.0.0/8, designated by IANA as "reserved for future use" [33]; any connections these addresses will fail, forcing the victim to choose an address from `tried`.

Recall from Section 2.2 that the pair *(group, source group)* determines the bucket in which an address in an `ADDR` message is stored. We suppose that the attacker controls bitcoin nodes in s different groups, where *group* defined as the /16 IPv4 prefix containing the node's IP; thus, 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. Our upper bound on g is $2^8 = 256$, since we use the "trash" 252.0.0.0/8 address block. 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.1 Infrastructure strategy

In an infrastructure attack, the number of source groups s is constrained, and the number of groups is essentially unconstrained. By Lemma A.1, the expected number of buckets filled by a s source groups is

$$E[N] = 256(1-(\tfrac{255}{256})^{32s}) \qquad (17)$$

With $s = 32$, we expect to fill about 251 of 256 `new` buckets, and 235 buckets with $s = 20$.

Each (group, source group) pair maps to a unique bucket in `new`, and each bucket in `new` can hold 64 addresses. The bitcoin eviction discipline 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 given by equations (8)-(9). Taking $a = 125$ addresses means the adversary expects to overwrite 63.8 of the 64 legitimate addresses in the bucket.

We thus require each source group to contain 32 peers, and each peer is entrusted with sending `ADDR` messages

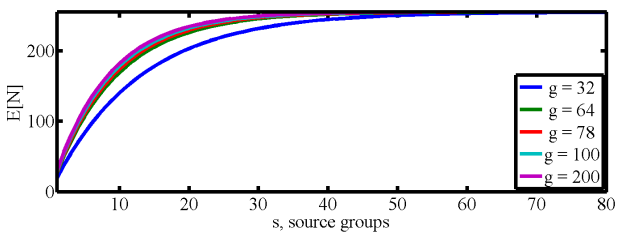

Figure 8: $E[N]$ vs s (the number of source groups) for different choices of g (number of groups per source group) when overwriting the `new` table.

containing 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` msg containing $8 \times 125 = 1000$ address. A total of $256 \times 125 \times s$ *distinct* addresses are sent by all peers. (There are a total of 2^{24} addresses in the 252.0.0.0/8 block, so as long as $s < 524$, all of these addresses are distinct.)

B.2 Botnet strategy

In a botnet attack, each of the attacker's t bitcoin peers is in a distinct source group. For $s = t > 200$, which will be the case for all of our botnet attacks, we can see from equation (17) that the number of source groups $s = t$ is essentially unconstrained. We 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 values of $t > 200$, we expect at least 780 address to be inserted into each bucket. Using the recurrence relation of equations (8) and (9), we find $E[Y_{780}] \approx 64$, so that each bucket in `new` is likely to be completely full.

C More Countermeasures

To continue the discussion of Section 7, we propose further countermeasures against our eclipse attacks:

6. More buckets. Among the most obvious countermeasures 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 (4) 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. This tweak may be worthwhile for bitcoin nodes with cheap storage.

7. More outgoing connections. Figure 6 indicates our test bitcoin nodes had at least 65 connections slots available, and [11] 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 running the risk that the network will run out of connection capacity. Requiring twelve outgoing connections instead of eight (in addition to the two feeler connections and two anchor connections), for example, decreases the success probability of the eclipse attack from f^8 to f^{12}. To put this in context, to achieve a success probability of 50%, the infrastructure attacker now needs 46 groups, and the botnet needs 11796 bots. While this improvement is not as dramatic as those of Section 7, it is still a simple way to raise the bar for eclipse attacks.

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 5). To make this more concrete, note that a node requests `ADDR` messages from its outgoing peer 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 in return, 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 [25]. We suggest a node accept only a limited number of connections from the same IP address.