PhD-FSTC-2015-31
The Faculty of Sciences, Technology and Communication

DISSERTATION

Presented on 12/06/2015 in Luxembourg
to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
EN INFORMATIQUE

by

Ivan Pustogarov
Born on 4 February 1986 in Moscow (Russia)

Deanonymisation techniques for Tor and Bitcoin

Abstract

This thesis is devoted to *low-resource off-path* deanonymisation techniques for two popular systems, Tor and Bitcoin. Tor is a software and an anonymity network which in order to confuse an observer encrypts and re-routes traffic over random pathways through several relays before it reaches the destination. Bitcoin is a distributed payment system in which payers and payees can hide their identities behind pseudonyms (public keys) of their choice. The estimated number of daily Tor users is 2,000,000 which makes it arguable the most used anonymity network. Bitcoin is the most popular cryptocurrency with market capitalization about 3.5 billion USD.

In the first part of the thesis we study the Tor network. At the beginning we show how to remotely find out which Tor relays are connected. This effectively allows for an attacker to reduce Tor users' anonymity by ruling out impossible paths in the network. Later we analyze the security of Tor Hidden Services. We look at them from different attack perspectives and provide a systematic picture of what information can be obtained with very inexpensive means. We expose flaws both in the design and implementation of Tor Hidden Services that allow an attacker to measure the popularity of arbitrary hidden services, efficiently collect hidden service descriptors (and thus get a global picture of all hidden services in Tor), take down hidden services and deanonymize hidden services.

In the second part we study Bitcoin anonymity. We describe a generic method to deanonymize a significant fraction of Bitcoin users and correlate their pseudonyms with their public IP addresses. We discover that using Bitcoin through Tor not only provides limited level of anonymity but also exposes the user to man-in-the middle attacks in which an attacker controls which Bitcoin blocks and transactions the user is aware of. We show how to fingerprint Bitcoin users by setting an "address cookie" on their computers. This can be used to correlate the same user across different sessions, even if he uses Tor, hidden-services or multiple proxies.

Finally, we describe a new anonymous decentralized micropayments scheme in which clients do not pay services with electronic cash directly but submit proof of work shares which the services can resubmit to a crypto-currency mining pool. Services credit users with tickets that can later be used to purchases enhanced services.

To my parents.

Acknowledgements

It is a pleasure for me to thank all people from whom I received support and encouragement over the past four years. I am deeply grateful to my adviser Prof. Alex Biryukov for his wisdom, patience, and guidance.

I would like to express my sincere gratitude to Dr. Aaron Johnson and Dr. Thorsten Holz for agreeing to serve as jury members, and to A.-Prof. Jean-Sébastien Coron for charing the jury.

I am grateful to my co-authors and friends: Ralf-Philipp Weinmann and Dmitry Khovratovich. It was really great to work with you.

I very much appreciate the help of LACS secretaries Fabienne Schmitz and Isabelle Schroeder with administrative matters.

Finally I would like to express my gratitude to the University of Luxembourg for outstanding research conditions.

Ivan Pustogarov, May 2015

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter we introduce the reader to the topic of anonymity. We briefly describe the history of anonymity networks and pseudonymous digital currencies thus giving the historical context in which Tor and Bitcoin were conceived.

Contents

1.1 Foreword

Privacy and anonymity are a subtle matter. Humans are social creatures and in order to establish new social connections they need (and want) to share information about themselves. On the other hand without a doubt the need for privacy is a basic human need: we all need to stay alone sometimes. We feel discomfort when somebody is prying into our affairs. It even changes our behaviour as we act differently when we know that somebody might be watching us. These two contradictory human needs make research in privacy protection particularly difficult.

Privacy is the ability of individuals to control information about themselves (i.e. whether it should be shared, who it should be shared with, and when it should be shared). The personal information that an individual tries to keep control of is called *private information.* Obvious examples are health conditions, age, salary, social status, habits. Information that was previously private may become *public* if it is leaked outside of the initial group and might uncontrollably spread among other individuals. Private information can have value as it can be used in a number of different ways such as targeted advertising, identity theft, blackmailing, etc. Thus there will always be parties interested in collecting it.

Private information can be learned either directly (by stealing, questioning or coercing) or indirectly by observing person's actions by means of surveillance. Surveillance can be prevented through encryption and anonymity. Encryption provides a way for better privacy by staving off reading the content of electronic communications. Nonetheless, even if data is encrypted, an observer is still able to analyze routing meta-data such as IP-address, mail destination addresses and web-site domains which can provide enough information about an individual. Often the mere fact of communication can reveal enough information and put the user on the "list of suspects". In addition *financial privacy* can easily be violated by governments by inquiring financial institutions which process the payments: conventional electronic payment systems (such as credit cards or PayPal) are normally tied to the holder's identity. This is why *anonymity of communications* plays an important role in protecting privacy. In practice anonymity is achieved by that an individual cannot be distinguished among a group (preferably large) of other individuals called *anonymity set*.

This thesis gives a systematical security analysis of two widely used solutions to protect one's privacy of communications and financial privacy through anonymity: Tor and Bitcoin[1]. Studying deployed widely used systems has a clear motivation. First, these systems have a large number of users and losing anonymity for them has physical consequences and not theoretical. Second, a deployed system needs to take into account much more small details than a purely theoretical design[2]. The vulnerabilities and design flaws found during the study should be avoided when designing new better systems.

1.2 Historical overview

> *Computerization is robbing individuals of the ability to monitor and control the ways information about them is used ... The foundation is being laid for a dossier society, in which computers would be used to infer individuals' life-styles, habits, whereabouts, and associations from data collected in ordinary consumer transactions. – 'Security without Identification: Transaction Systems to Make Big Brother Obsolete' by David Chaum, 1985.*

Privacy was a concern long before the advent of Internet, since ever communications were sent over public postal service or telephone/telegraph systems. However without automatic collection and processing of information, creating profound

[1]In fact Bitcoin is a *pseudonymous* payment system. Nonetheless after it was released and deployed many relied on its properties to achieve anonymity and de-facto it became a system for carrying out anonymous transactions (e.g. the infamous Silkroad market place actively used Bitcoin, see Section 1.2.4).

[2]Compare it with a cipher. While the algorithm may be strong, implementations can make it vulnerable to side-channel attacks.

profiles of large number of people was too expensive. Proliferation of computer networks, increased capacity of storage devices along with the decrease in their prices made this process significantly cheaper and easier in the late 1980s. And given the ease with which the information can be copied and exchanged between collecting parties, the danger of leaking private information moved to a different level.

The reason why electronic communications are so susceptible to privacy violation lies in the network design. When the first computer networks were created, anonymity and privacy were not among the design goals and this legacy obviously affects our present. It was the case for military networks such as Autodin (provided email-like services in 1961) and Arpanet (created in 1969), networks born in academic institutions (Usenet in 1980, and Bitnet in 1981), and community created networks as Fidonet (Tom Jennings sets up the first system in 1984). It was also not a design goal for World Wide Web in 1991.

The danger of privacy violation was still understood among both academia and non-academia researchers. One of the pioneers in this field was David Chaum. In 1981, he published paper "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms" [5] where he described *layered encryption* and *mixnets* which would become the basis of many anonymous tools. Two year later, in 1983, he published "Blind signatures for untraceable payments" [6] introducing blind signature which can be used in anonymous digital currencies. In 1988, Chaum presented *dining-crypotgraphers* nets (or DC-nets), a different from mixnets approach for anonymous communications with stronger anonymity guarantees.

These ideas were picked up by the Cypherpunk movement which started with the corresponding mailing list in 1992. The Cypherpunk's vision is reflected in the Crypto Anarchist Manifesto [7] by Timothy C. May and Cypherpunk Manifesto [8] by Eric Hughes, 1993. In particular,

> ...We the Cypherpunks are dedicated to building anonymous systems. We are defending our privacy with cryptography, with anonymous mail forwarding systems, with digital signatures, and with electronic money... – by Eric Hughes, 1993.

> ...Interactions over networks will be untraceable, via extensive rerouting of encrypted packets... – by Timothy C. May, 1992.

The goal was to achieve better privacy through anonymous communications and anonymous digital currencies (which still had to be created).

1.2.1 Anonymous communications

The research on anonymous communications[3] went mainly in two directions: mixnet-based anonymous remailers for delay-tolerant traffic and onion routing for real-time traffic[4]. The work in these two directions was being done in parallel. Both mixnets

[3]We put aside anonymous publishing systems as Freehaven [52] or Freenet [53]

[4] Overall, the literature on anonymous communication systems is vast. However the goal of this section is to provide the historical context and not to give a complete survey of anonymity protocols. An excellent bibliography can be found at [73].

and onion routing systems use layered encryption to break the link between the sender and the recipient of a message[5].

Anonymous remailers. Anonymous remailers have their origins in the cypherpunks mailing list. In 1993, Johan Helsingius launched a basic remailer system (also Called Type 0 remailer). In order to send an anonymous mail a user would add an extra header to e-mail indicating the final destination and send it to `anon.penet.fi`, which then striped off identifying information and forwarded the mail to the final destination. Such a system does not provide a strong anonymity: (1) there is just one point of failure; (2) this point stores connecting client's information; (3) mails sent to the remailer are not encrypted.

In 1995, Lance Cottrell created Mixmaster [64] remailer system in which several remailers could be chained using layered encryption. In addition traffic analysis was made harder by delaying packets and by breaking them into chunks of constant size.

Mixminion [13] is another remailer protocol published by Mathewson, Dingledine, and Danezis in 2003. Mixminion added several important features such as (1) *integrated directory servers* which maintain and distribute the list of available remailer servers; (2) *Exit policies*, a mechanism which allows a remailer to specify which addresses and by which methods a mix node is prepared to deliver messages.

Remailers were an elegant and simple solution for sending electronic mails anonymously. They however were too specialized to become widespread outside of the cypherpunk community.

Onion Routing. In contrast to remailers the work on onion routing began at U.S. Naval Research Laboratory. In late 1995 (the same year Lance Cottrell created Mixmaster), Goldschlag, Reed and Syverson began work on a project to separate identification from routing in low-latency communication [10, 24]. Onion routing employs layered encryption, but it is different from mix networks in a number of ways[6]. First, onion routing does not use mixing. Second essential differences is that "public keys are used to lay a cryptographic circuit of symmetric keys, which is then used to pass data" [24]. In addition "...onion routing network designs are for carrying bidirectional low-latency traffic over cryptographic circuits while public mixnets are designed for carrying unidirectional high-latency traffic in connectionless messages" [24].

The onion routing design paper [10] was published in 1996 and the research and development of onion routing system proceeded until 1999 and resulted in two generations of the system. The work on Onion Routing development was temporarily suspended after 1999.

[5]Ignoring some details, a three-hop layered encryption through R_1, R_2, R_3 looks like this:

$$E_{PK_{R_1}}(N_1, R_2, E_{PK_{R_2}}(N_2, R_3, E_{PK_{R_3}}(N_3, A, M)))$$

where $E_{PK_{R_i}}(...)$ denotes encryption with R_i's public key; N_i is either a random string in mixnets or session-key metrial in onion routing; A is the receiver's address and M is the message.

[6]See [24], section 3.1 for a detailed discussion on the difference between the two.

The work resumes in 2002, and in 2004 the Tor design paper [14] is published in which the original onion routing idea is enhanced by Diffie-Hellman based circuit building, Directory authorities, and other improvements. In 2006, a non-for-profit "The Tor Project" is founded. By 2015 Tor is the largest[7] deployed anonymity network comprising about 7,000 relays and estimated number of users daily 2,000,000.[8].

The development of the main Tor client software [84] is coordinated by the Tor Project [76], a non-profit organization. The funding sources of Tor are diverse but the main contributor is the US government. The primary goal of Tor is to provide the public with an easy to use low-latency tool to defend against network surveillance and circumvent censorship.

Though initial purpose the Onion Routing project was to protect U.S. intelligence online communications, the Tor software is now distributed under BSD license[9] and is used by ordinary people, businesses, political activists, whistleblowers, and media. While Tor is not illegal anywhere in the world, the access to the network is blocked in some countries (e.g. China, Republic of Belarus).

Hidden Services. Hidden services were a part of Onion routing project since very beginning. They were designed to resist DoS and physical attacks against Web- and other services. Specifically, hidden services allow running an Internet service (e.g. a Web site, SSH server, etc.) in such a way that clients of the service do not know its actual IP address. This is achieved by routing all communication between the client and the hidden service through a *rendezvous point* which connects anonymous circuits from the client and the server. *Tor Hidden Services* [83] are improved version of hidden services which were introduced and deployed in 2004.

Since Tor added support for hidden services in 2004, many of them have emerged; some enable freedom of speech (New Yorker's Strongbox [75], Wikileaks [86]) while others allow for the exchange of contraband (the Silk Road market place [25]) or are used by botnets (Skynet [55]) for hiding the location of command and control centers. More mundane services such as the DuckDuckGo search engine [49] also exist as Tor hidden services.

DC-nets. Another Chaum's idea, DC-nets, received considerably less research attention. The basic idea of dc-nets can be introduced in several different ways, here we adopt the description[10] from [15]. Assume that Alice and Bob want to publish their messages a and b in such a way that an external observer should not be able to determine who published which message. Alice and Bob share two secret keys k_0 and k_1, and a random bit x. Then they publish the following messages:

if x = 0	Alice:	$A_0 = k_0 \oplus a$,	$A_1 = k_1$
	Bob:	$B_0 = k_0$,	$B_1 = k_1 \oplus b$
if x = 1	Alice:	$A_0 = k_0$,	$A_1 = k_1 \oplus a$
	Bob:	$B_0 = k_0 \oplus b$,	$B_1 = k_1$

[7]The second largest anonymity network I2P [59] is relatively small compared to Tor.

[8]This estimation includes botnet-malware infected computers.

[9]A free software license.

[10]Though this description does not explain where the name "dining cryptographers" comes from, but in our opinion it better explains how the system can be used in practice.

Then one can compute messages a and b from $A_0 \oplus B_0$ and $A_1 \oplus B_1$, but which expression results in whose message remains private. When extended to multiple parties the protocol provides strong anonymity guarantees, however there are several obstacles (e.g. jamming and collisions) which makes implementations impractical for interactive communications. At the time of writing only highly experimental software exits [46].

1.2.2 Cryptocurrencies: from blind signatures to Bitcoin

One can define digital currency is a medium of exchange which can be used to buy goods and relies on a computer network to transmit and verify transactions. Digital currencies based on public key cryptography, *cryptocurrencies*, differ from other types of currencies as they are able to provide pseudonymity naturally. There are several notable examples of pseudonymous cryptocurrencies.

Ecash. In 1983, David Chaum introduced blind signatures and *ecash* [45], an anonymous payment system based on them. A bank could issue blindly signed coins, which were not bound to a specific person. This made the coins anonymous. Ecash was implemented by "DigiCash" company in 1989. Despite some initial success the company went bankrupt in 1998 (one of the possible reasons is the wide adoption of credit cards for online payments).

Ecash was a centralized digital currency as there is one problem which is not easy to solve without introducing a trusted third party: double spending. In contrast to physical cash, electronic coins are very easy to copy. Thus the common solution for a merchant is to consult with the bank to make sure that a coin was not previously spent.

B-money. In 1998 Wei Dai in the Cypherpunk mailing list described B-money [36], a design for *distributed digital currency*. It included several important ideas: (1) senders and receivers hide their identities behind public keys; (2) Anyone can create money by broadcasting the solution to a previously unsolved computational problem; (3) In order to transfer money, a user needs to create a transaction with the receiver's public key and the amount; the user then signs and broadcasts the transaction; (4) the information about who has how much money is kept in a distributed fashion on a set of servers. B-money protocol was however never formally published.

Bitcoin.

> Governments are good at cutting off the heads of a centrally controlled networks like Napster, but pure P2P networks like Gnutella and Tor seem to be holding their own. – Satoshi Nakamoto (from cryptography mailing list), 7th Nov 2008.

B-money protocol left out the details on how the servers should come into consensus on the "correct" transaction history. Bitcoin the design paper [65] of which was published in November 2008 under the pseudonym Satoshi Nakamoto proposed an elegant solution to this problem[11]. Bitcoin is a digital currency that relies on cryptography and a broadcast peer-to-peer network for double-spending prevention instead of a trusted third party. The heart of Bitcoin is blockchain, an electronically stored ledger of all ever existed valid transitions. The transactions are stored in timestamped blocks. In order to create a new valid block (and thus timestamp the outstanding transactions) a peer produces a Proof-of-Work. Each new block refers to the previous one thus comprising the chain of blocks. Double spending is prevented by that each transaction is timestamped by the block it is contained in. In order to double spend a coin, an attacker would need to redo the Proof-of-Work for all blocks after the corresponding transaction. With each new block, the corresponding peer also generates and earns bitcoins. Bitcoin's pseudonimty comes from the fact that users of the system are identified by Bitcoin addresses which are essentially hashes of public keys. Each user can create as many public keys as she wants locally.

Bitcoin is now accepted as a currency by many companies from online retailer Overstock to exotic Virgin Galactic. At the time of writing, about 14,000,000 bitcoins were generated, and the exchange rate is 233 USD for one Bitcoin[12]. At the time of writing Bitcoin comprised of about 6,000 servers, and the estimated number of clients was 100,000.

The development of the main Bitcoin client[13] is currently coordinated by a set of 5 core developers. There also exists Bitcoin Foundation the purpose of which is to fund[14] the development of the Bitcoin Core client and to promote Bitcoin technology. Though Bitcoin Foundation does not directly control the development of Bitcoin Core client, one of the core developers Gavin Andersen is a member of the board of directors. The legal status of Bitcoin varies from country to country (e.g. it is banned in Russia), however in general the reception is positive.

1.2.3 Examples of centralized anonymous digital currencies.

It is worth to mention two notable examples of centralized anonymous digital currencies. Their centralized nature made them vulnerable to attacks from the U.S. Government and allowed the police to arrest and bring indictments against the owners, seize reserves and servers.

eGold. eGold was founded in 1996 by Douglas Jackson and Barry Downey. By 2009 it had 5 million registered accounts [50]. The units of payment were grams of gold. It was a centralized system. Pseudonymity of eGold came from the fact that creators of accounts could use any name or label they wished[15]. The system

[11]In January 2009, Satoshi Nakamoto announces the first code release of Bitcoin (at `sourceforge.net`).

[12]The exchange rate remains very volatile however.

[13]Bitcoin Core.

[14]Mainly by donation made by companies the main business of which depends on the Bitcoin technology.

[15]However since the history of all transactions was stored on company's server, transaction flow graph analysis was possible. This allowed one to deanonymize large amount of inattentive/unlucky criminals (they just had to make a transaction with an unreliable party).

was shut down after a highly controversial court trial (see [47]): it was accused of counterfeiting and money laundering.

Liberty Reserve. Liberty Reserve was founded in 2006. It also allowed clients to create anonymous accounts: only a (pseudo)name, date of birth and an email address were required. The account could be charged using traditional means: a credit card, bank wire, postal money order, etc. Liberty Reserve was seized by the U.S. Government which claimed that it acted[16] as *a financial hub of the cyber-crime world.* [60].

1.2.4 The synergy: Silkroad marketplace

hahaha! great idea – user genjix at `bittalk.org`*, March 1, 2011.*

**hug* Hugs not drugs... no wait, hugs AND drugs! – Dread Pirate Roberts, 20th April 2012 at Silkroad Hidden Service Forum.*

Silkroad is a notable example of a successful symbioses of Tor Hidden Services and Bitcoin. Silk Road was an anonymous online market available as a Tor Hidden Service and launched in February 2011 (it was first announced at Bitcoin forum on the 1st of March 2011 by user "silkroad"). Silkroad was operated by a person (or a group of people) under the pseudonym "Dread Pirate Roberts"[17]. It was a market place where buyers had to register an account and the operator of the market place was charging a fee from each seller. There was a small detail which differentiated Silkroad from other ebay.com-like shopping Web-site and which made using Tor Hidden Service a prerequisite: it was a platform for selling illegal drugs. Given the obvious dangers in running such a shopping Web-site, the Cypherpunks' ideas came in handy. The only way to access Silkroad was through a Tor Hidden Service and buyers and sellers conducted all transactions with bitcoins. All operations were done through the SilkRoad escrow service: buyers' bitcoins were held by the SilkRoad until the order had been received.

The major public attention was brought to Silkroad after the Gawker blog's article [41]. It brought more traffic to the Web-site but also attracted more attention from FBI. At the beginning of 2013, there were several arrests and convictions related to Silkroad (mainly due to the police intercepted drugs in mails). On the 2nd October 2013, 29 years old Ross Ulbricht, the alleged owner of Silkroad was arrested in San Francisco [71]. At the time of this writing he faces 30 years to life in prison. It seems though that the FBI investigators did not attack Tor or Bitcoin but used rather traditional methods[18].

[16] It is not clear if fighting criminals was the real goal behind the seizure of both eGold and Liberty Reserve (see [47] for more on this topic).

[17] A character from "The Princes Bride" Hollywood movie.

[18] According the article in Wired [88], the FBI investigators were working to build a relationship with Dread Pirate Roberts with the help of an undercover agent. Ulbricht also made a number of mistakes that allowed FBI to tie him to Silk Road, e.g.*using the name "altoid" to post messages advertising Silk Road to a forum and then using that same name to post to a Bitcoin forum seeking workers for a Bitcoin startup. In the latter message, "altoid" told would-be job applicants to contact him at rossulbricht@gmail.com.*

1.3 Thesis structure

This thesis can logically be divided into three parts. Chapters 3 and 4 comprise the first part and describe attacks on Tor and Tor Hidden Services. Part 2 consists of Chapter 5 and describes deanonymisation techniques for Bitcoin and provides analysis of Bitcoin-over-Tor bundle. Part 3 consists of Chapter 6 only and describes an anonymous micropayment scheme.

- In Chapter 2, we provide the necessary information about Tor and Bitcoin internals.

- In Chapter 3, we describe two ways to probe the connectivity of a Tor relay. This gives an attacker access to the topological information about the Tor network. We demonstrate how the resulting leakage of the Tor network topology can be used and present attacks to trace back a user from an exit relay to a small set of potential entry nodes.

- In Chapter 4, we expose flaws both in the design and implementation of Tor's hidden services that allow an attacker to measure the popularity of arbitrary hidden services, take down hidden services, deanonymize hidden services, and efficiently collect the addresses of existing hidden services. We also propose a method for opportunistic deanonymisation of Tor Hidden Service clients. Using the discovered techniques we study (1) two botnets which use Tor hidden services for command and control channels; (2) Silk Road market place, a hidden service used to sell drugs and other contraband.

  In addition we we analyse the landscape of Tor hidden services. We study 39824 hidden service descriptors collected on 4th of Feb 2013: we scanned them for open ports; in the case of 3050 HTTP services, we analyzed and classified their content.

- In Chapter 5, we present an efficient method to link users' Bitcoin addresses to their IP addresses. Further in this chapter we show that using Bitcoin over Tor does not solve the anonymity problem. Even worse we show that combining Tor and Bitcoin creates a new attack vector. A low-resource attacker can gain full control of information flows between all users who chose to use Bitcoin over Tor. Moreover, we show how an attacker can fingerprint users and then recognize them and learn their IP addresses when they decide to connect to the Bitcoin network directly.

- In Chapter 6, we propose a new decentralized anonymous micropayments scheme which can be used to reward Tor relay operators. Tor clients do not pay Tor relays with electronic cash directly but submit proof of work shares which the relays can resubmit to a crypto-currency mining pool. Relays credit users who submit shares with tickets that can later be used to purchase improved service. Both shares and tickets when sent over Tor circuits are anonymous.

1.4 Remarks on methodology

Attacks described in this thesis are based on flaws discovered by a careful manual analysis of Tor and Bitcoin specifications [81, 37] and the corresponding source code (as in many cases the specifications left out many important details). Since in this thesis we attack concrete systems it is important to answer the following questions: how fundamental are the attacks? How hard is it to fix them? Do they have an impact on similar systems?

The attacks based on topology information leakage from Chapter 3 are generic and are likely to be applicable to other onion routing systems. They are based on the ability of an attacker to find relays' connections. For Tor, there are two ways to achieve this: one is Tor-specific and another is a more general timing-based method. We would expect other systems to have similar sidechannel vulnerabilities. Topology-based attacks are also hard to prevent as it is hard to keep the network fully connected due to scalability reasons.

Attacks on Tor Hidden Services described in Chapter 4 are a mixture of implementation specific attacks (efficient harvesting of onion addresses), attacks based on deliberate design decision (tracking and DoSing hidden services), and more fundamental traffic confirmation attacks. The latter class of attacks is especially important for future hidden service designs and seems to be very hard to fix.

Attacks on Bitcoin are applicable to a large number of other cryptocurrencies: there are hundreds of them at the time of writing and most are similar to and derived from Bitcoin. Moreover Bitcoin-over-Tor attacks work not only for Tor but for any other proxies and VPN's. Most of the attacks on Bitcoin described in Chapter 5 are due to design oversights and require moderate effort to be fixed.

1.5 Ethical considerations

The attacks described in this thesis sometimes required carrying out experiments on live systems but in no case we actually deanonymized real users: we discarded all identifying data after the experiments. Moreover our goal was at no time to perform a full deanonymization of any target that was not under our control but rather to show that this would be possible.

In case of Bitcoin and Tor, the experiments on the real networks were necessary to collect statistics required to estimate the success rate of the attacks. For all experiments we used our own Tor relays and Bitcoin peers; our relays/peers were limited in bandwidth and were running only for a short time. We believe that no users were harmed.

In case of Tor Hidden Services the described attacks could be simulated in dedicated simulators such as Shadow [27]. However, at the time of experiments deployed hidden services were not well studied and there was no statistics about the number of hidden services or their usage. Henceforth, we believe experiments (that did not intentionally cause degradation of the network and its services) on the live Tor network were worthwhile and necessary to enhance the scientific understanding of Hidden Services.

Chapter 2

Tor and Bitcoin

In this chapter we introduce the reader to the inner workings of the Tor and Bitcoin protocols.

Contents

2.1 Tor anonymity network

For many people, the first approach to hiding their identity is a public proxy server. This however has serious limitations: the owner of the proxy can be forced to reveal any logs potentially stored – or even worse, the server may turn out to be a honeypot. A better solution is to forward traffic through a chain of network nodes, so-called *relays*. A set of such relays working according to some protocol is called *anonymity network*.

2.1.1 Architecture

Tor is a volunteer-based low-latency anonymity network built on ideas of onion routing. Fig. 2.1 shows its basic architecture. Tor anonymizes user traffic by forwarding it through a *circuit* of *Tor Relays* in such a way that each relay in the

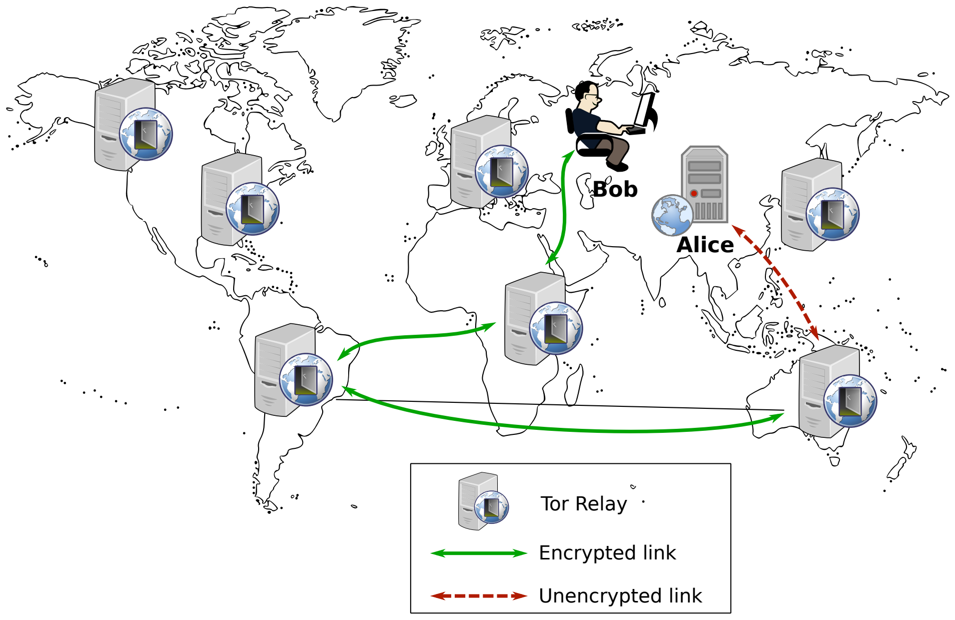


Figure 2.1: Tor anonymity network

circuit knows only the immediate transmitter of the message and the immediate receiver of the message. Using Tor makes it more difficult for non-global adversaries[1] to trace Internet activity for TCP applications. Tor tries hard to achieve low traffic latency to provide a good user experience, thus sacrificing some anonymity for performance. To keep latency low and network throughput high, Tor relays do not delay incoming messages and do not use padding.

Tor is a semi-decentralized network comprising *Tor relays* which forward client traffic and nine *Tor authorities* which maintain the list, *Consensus*, of other Tor relays and distribute this list to clients[2]. Each Tor relay is uniquely identified by an RSA public key. When Bob wants to connect to Alice through Tor, he downloads the *Consensus* from the Tor authorities. Bob then randomly chooses three Tor relays from the list.

After Bob chose a path consisting of three relays it agrees on a Diffie-Hellman key with the first (*Entry* or *Guard*) node in the path. Then he completes a Diffie-Hellman key exchange handshake with the second *Middle* node by using the first node as a proxy. In this way the Middle node does not know the real initiator of the handshake. The user repeats the same procedure with the third (*Exit*) node but uses the chain of Entry and Middle nodes as proxies. When the client wants to connect to a server on the Internet, he packs his messages in fixed 512-bytes sized *cells* and encrypts each cells with the three keys. When the message travels along the circuit, each relay strips off one layer of encryption. This process is shown in Fig. 2.2. This scheme allows one to break linkability between the sender of the

[1]A global passive adversary is a type of adversary who can observe all the traffic in the network.
[2]The IP addresses of authorities and their public keys are hard-coded into the Tor code.

message and its destination.

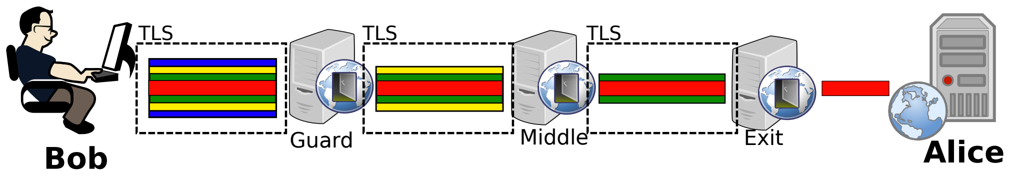


Figure 2.2: Layered encryption

2.1.2 Tor Circuits

Circuit construction. A Tor client builds a circuit one hop at a time. First, the user sets up a TLS connection with the Entry node and negotiates a Diffie-Hellman (DH) key using `COMMAND_CREATE` and `COMMAND_CREATED` cells[3], see Fig. 2.3. This creates a one-hop circuit.

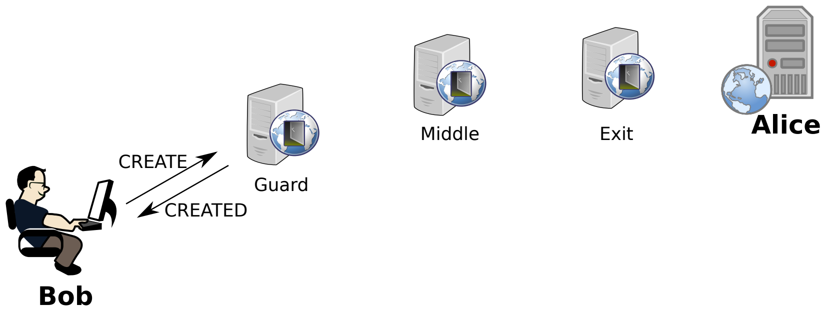


Figure 2.3: Circuit creation. First step

The client extends the circuit to the Middle node through the Guard node (see Fig. 2.4): he sends a `RELAY COMMAND_EXTEND` cell to the guard in which he specifies the IP address of the middle router, its digest, and the first part of DH key exchange encrypted by the middle node's public key. Once the guard node receives the cell, it establishes a TLS connection with the middle node and sends it the encrypted part of the DH handshake. The middle node decrypts it and replies with the second part of DH exchange which is forwarded to the client within a `RELAY COMMAND_EXTENDED` cell. In this way the second hop of the circuit is established.

If during the circuit construction process the middle node rejects the connection, the guard node sends a `COMMAND_DESTROY` cell, specifying the error code, so that the client is forced to choose another sequence of relay nodes and try to construct a new circuit. If the circuit is extended successfully up to the middle node, the rest of the circuit is established in the same way. After the circuit has been built, the client can start transmitting and receiving data over this circuit. All TCP connections of

[3]Tor protocol messages are called "cells".

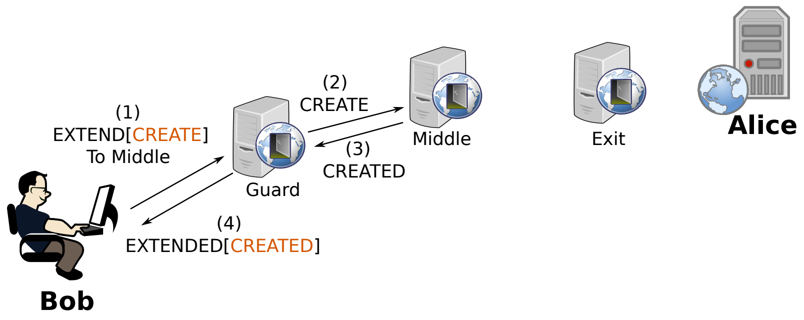


Figure 2.4: Circuit creation. Second step

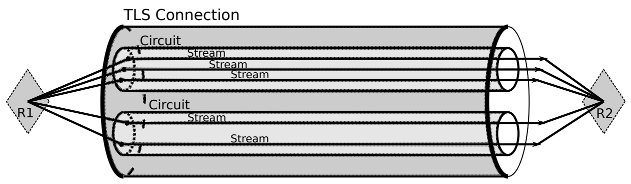


Figure 2.5: Circuits and streams multiplexing

the user's application are translated into Tor streams which are multiplexed over the circuit.

Using the initially chosen circuit for a long time makes profiling attacks easier: the longer the duration of the circuit, the more time the attacker has to reveal it. For this reason, circuits older than 10 minutes are not allowed to carry new streams (for new streams a new circuit should be constructed). After 10 minutes a circuit dies unless it carries a long-lived stream. In the latter case, the lifetime of the circuit equals the lifetime of the long-lived stream. In other words, a circuit is not destroyed until at least one stream is attached to it. In a similar way, a TLS connection between two Tor relays is not closed if it carries at least one circuit. A TLS connection without circuits between two Tor routers lives for three minutes. There is one exception to the rule. A circuit which has never carried a stream (a *clean* circuit[4]) lives for 1 hour.

When a pair of Tor routers or a Tor router and a client have several circuits between them, they try to tunnel them over a single TLS connection. In Figure 2.5, communication between two Tor routers is shown. The routers use a single TLS connection (which is also called Onion Routing connection) which carries a number of circuits, two in this picture (which may belong to different end users). Multiple streams of one user may be multiplexed over a single circuit.

Consensus. The list of all Tor relays is distributed by the Tor authorities in the *consensus* document. The consensus is updated once an hour by the directory

[4]Once a new stream is attached to the circuit, it is marked as "dirty"

authorities and remains valid for three hours. Every consensus document has a "valid-after" (VA) time, a "fresh-until" (FU) time and a "valid-until" (VU) time. The "valid-after" timestamp denotes the time at which the Tor authorities published the consensus document. The consensus is considered fresh for one hour (until "fresh-until" has passed) and valid for two hours more (until "valid-until" has passed). According to the implementation[5] clients download the next consensus document in (FU + 45 mins; VU - 10 mins) interval.

Tor Exit policy. In order to access a Web resource anonymously through a Tor circuit, the Exit relay (the final relay in the circuit) should allow establishing connections outside the Tor network. This makes Exit relay operators open to numerous abuses. In order to make their life easier, Tor allows them to specify an Exit Policy: a list of IP addresses and ports to which the Exit node is willing to establish connections and which destination are prohibited. When a client establishes a circuit, he chooses only those Exit nodes which allow connections to the corresponding IP addresses[6] and port ranges.

Guard nodes. To keep latency low and network throughput high, Tor relays do not delay incoming messages and do not use padding. This makes Tor susceptible to traffic confirmation attacks: if an attacker is able to sniff both ends of a communication, she is able to confirm that the user communicated with the server. If the first hop of a circuit is chosen at random then the probability that a malicious node is chosen as the first hop (and thus allow the attacker to learn the IP address of the user) converges to one with the number of circuits. Due to this each Tor client initially selects a fixed *Guard* node and whenever a new circuit is established this node is used for the first hop[7]. A Guard node is used for a period from 30 to 60 days (the exact duration is chosen randomly), after which a new Guard node is chosen.

Load balancing. In order to achieve better performance Tor implements a load balancing mechanism. Each relay in the Consensus is assigned a bandwidth weight and the probability for a relay to be chosen by a client is roughly proportional to that weight [80]. In order to assign weights to relays, Tor authorities conduct active measurements by building two-hop circuits to specific URL's and measure download times [69]. This way relays get clients depending on their current load.

Bandwidth management. In order to limit bandwidth usage, a Tor relay implements token bucket algorithm [70]. The relay's operator can specify the token rate

[5]Version 0.2.3.25.

[6]Note that usually at the time the path is selected, only the domain name is known.

[7] Before Tor client version 0.2.4.23 [82] released on 28 July 2014, the number of Guard nodes was 3 and for a newly constructed circuit a client chose the first node in the circuit from this small set. Stable release 0.2.4.23 introduced a new *NumEntryGuards* consensus parameter, which made the number of entry guards configurable. This parameter was set to 1 on 19 Aug 2014 [77].

(which specifies the average incoming/outgoing bandwidth usage) and the bucket size (which specifies the burst).

Tor stream timeout policy. Tor provides SOCKS interface for applications willing to connect to the Internet anonymously. Each connection to the SOCKS port by an application is called a *stream.* For each new stream Tor tries to attach it either to an existing circuit or to a newly built one. It then sends a `BEGIN` cell down the circuit to the corresponding Exit node asking it to establish a connection to the server requested by the application. In order to improve user's quality of service, if Tor does not receive a reply from the Exit node within 10 or 15 seconds[8], it drops the circuit and tries another one. If none of the circuits worked for the stream during 2 minutes, Tor gives up on it and sends a SOCKS general failure error message.

2.1.3 Hidden services

Tor Hidden Services (or simply Tor HS) allow users to hide their locations while offering TCP services. Tor HS architecture is shown in Fig. 2.6 and is comprised of the following components:

- Internet service which is available as Tor hidden service;
- Client, which wants to access the Internet service;
- Introduction points (IP): Tor relays chosen by the hidden service and which are used for forwarding management cells necessary to connect the Client and the hidden service at the Rendezvous point;
- Hidden service directories (HSDir): Tor relays at which the hidden service publishes its descriptors and which are communicated by clients in order to learn the addresses of the hidden service's introduction points;
- Rendezvous point (RP): a Tor relay chosen by the Client which is used to forward all the data between the client and the hidden service.

Hidden service side. In order to make an Internet service available as a Tor hidden service, the operator (Bob) configures his Tor Onion Proxy (OP) which automatically generates a new RSA key pair. The first 10 bytes of the SHA-1 digest[9] of the RSA public key become the identifier of the hidden service. The OP then chooses a small number of Tor relays as introduction points and establishes a new introduction circuit to each one of them (step 1 in Fig. 2.6).

As the next step (step 2), Bob's OP generates two service descriptors with different IDs, determines which hidden service directories among the Tor relays are responsible for his descriptors and uploads the descriptors to them. A hidden

[8]Tor waits for 10 seconds for the first two attempt and 15 seconds for the subsequent attempts.
[9]ASN.1 encoded.

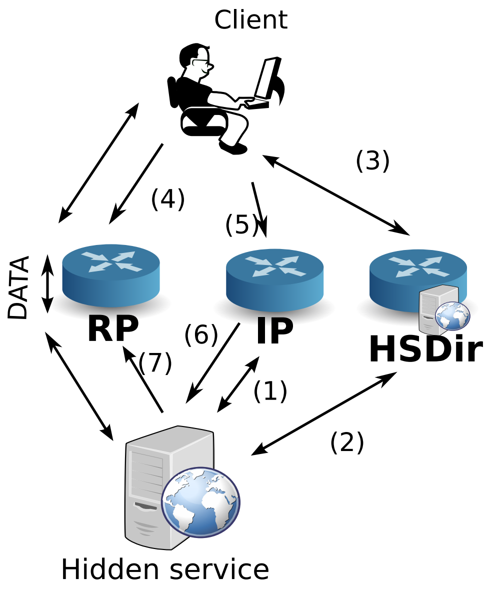


Figure 2.6: Tor hidden services architecture

services directory is a Tor relay which has an HSDir flag. A Tor relay needs to be up for at least 96 hours and should have a "Stable" flag[10] to become an HSDir[11]. The hidden service descriptors contain the descriptor ID, the list of introduction points and the hidden service's public key.

Client side. Using traditional means (e.g. e-mail, blog/forum post, etc.) Bob advertises the *onion address* of his hidden service. Onion address is a hostname of the form "z.onion", where z is the base-32 encoded hidden service identifier described above. A client (Alice) then computes the descriptor IDs of the hidden service (see below) and the list of responsible hidden service directories and fetches the descriptors from them (step 3).

In order to establish a connection to the Bob's hidden service Alice first builds a circuit (step 4) to a randomly chosen Tor relay which becomes the *rendevous point* (RP). This is done be sending a `RELAY_COMMAND_ESTABLISH_RENDEZVOUS` cell to RP. The body of that cell contains a Rendezvous Cookie (RC). The rendezvous cookie is an arbitrary 20-byte value, chosen randomly by Alice's OP. Alice chooses a new rendezvous cookie for each new connection attempt. Upon receiving a `RELAY_COMMAND_ESTABLISH_RENDEZVOUS` cell, the RP associates the RC with the circuit that sent it. Alice builds a separate circuit to one of Bob's chosen introduction points, and sends it a `RELAY_COMMAND_INTRODUCE1` cell containing the IP address and the fingerprint of the rendezvous point, the hash of the public key of the hidden

[10]Ignoring some details, a router is "Stable" if either its mean time between failures (MTBF) is at least the median for known routers or its MTBF corresponds to at least 7 days.

[11]Before Tor version 0.2.6.6 released on 24 March 2015, the time required to get the HSDir flag was 25 hours and "Stable" flag was not needed.

service (PK_ID), and the rendezvous cookie (step 5).

If the introduction point recognizes PK_ID as the public key of a hidden service it serves, it sends the body of the cell in a new `RELAY_COMMAND_INTRODUCE2` cell down the corresponding circuit (step 6).

When Bob's OP receives the `RELAY_COMMAND_INTRODUCE2` cell, it decrypts it using the private key of the corresponding hidden service and extracts the rendezvous point's nickname as well as the rendezvous cookie. Bob's OP builds a new Tor circuit to the rendezvous point, and sends a `RELAY_COMMAND_RENDEZVOUS1` cell along this circuit, containing RC (step 7). Subsequently, the rendezvous point passes relay cells, unchanged, from each of the two circuits to the other.

In this way, the client knows only the rendezvous point. Neither does the hidden service learns the actual IP address of the client nor does the client learn the IP address of the hidden service.

Choosing responsible HSDirs. Bob determines if a hidden services directory is responsible for storing his descriptor based on the descriptor's ID and the directory's fingerprint[12].

Descriptor identifiers change periodically every 24 hours and are computed as follows:

```
descriptor-id = H(public-key-id || secret-id-part)
secret-id-part = H(descriptor-cookie || time-period ||
replica-index)
```

The field `descriptor-cookie` is an optional field. If present, it prevents non-authorized clients from accessing the hidden service. The field *time period* denotes the number of days since the epoch[13]. This is used to make the responsible directories change periodically. The *replica index* is used to create different descriptors identifiers so that the descriptor is distributed to different parts of the fingerprint range.

After computing the descriptor identifiers, Bob determines which directory nodes are responsible for storing his descriptor replicas. To do this he sorts their fingerprints alphabetically and arranges them in a closed fingerprint circle (Fig.2.7. He chooses the three closest relays in positive direction (fingerprint value of them is greater than the fingerprint value of the hidden service).

According to the Tor implementation[14], a hidden service generates and publishes two replicas of its descriptor which results in 2 sets of 3 hidden service directories with consecutive fingerprints.

As an example, consider the circle of fingerprints depicted in Figure 2.7 and assume that one of the hidden service descriptor IDs is between fingerprints of relays HSDir$_{k-1}$ and HSDir$_k$. In this case the hidden service directories serving the descriptor are relays with fingerprints HSDir$_k$, HSDir$_{k+1}$, and HSDir$_{k+2}$. The

[12]Each Tor relay is identified by SHA-1 digest of its public key. We call this digest the relay's fingerprint.

[13]The number of days from 1 January 1970, 00:00 UTC (the Unix epoch).

[14]Version 0.2.6.10.

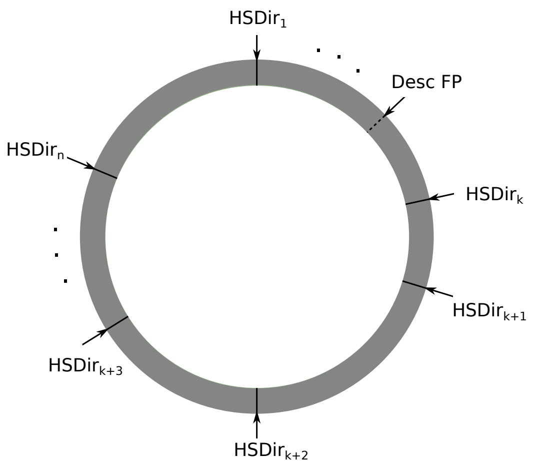


Figure 2.7: Tor hidden services fingerprints circular list

fingerprint of HSDir$_k$ is the first following the descriptor ID. We call this HSDir relay the *first responsible hidden service directory* for the descriptor ID.

Encrypted descriptors. The protocol as described above provides no client authentication: everyone who knows the onion address is able to connect to the corresponding hidden service. But it is also possible to restrict access to the hidden service to clients with a previously received secret key only. A hidden service operator would encrypt the introduction points in the HS descriptor using a symmetric "descriptor cookie" and distribute it among clients outside of Tor. The contact information for a hidden service in this case look like this [83]:

```
v2cbb2l4lsnpio4q.onion Ll3X7Xgz9eHGKCCnlFH0uz
```

Upon downloading a hidden service descriptor, only clients who know the "descriptor cookies" are able to decrypt the introduction points and send a `RELAY_COMMAND_INTRODUCE1` cell.

2.2 Bitcoin

2.2.1 Architecture

Bitcoin network consists of interconnected peers which re-broadcast valid transactions received from clients (see Fig. 2.8). Payers and payees (Alice and Bob in Fig. 2.8) of the Bitcoin system are identified by public keys[15]. Each peer stores the

[15]In practice one uses hashes of these public keys.

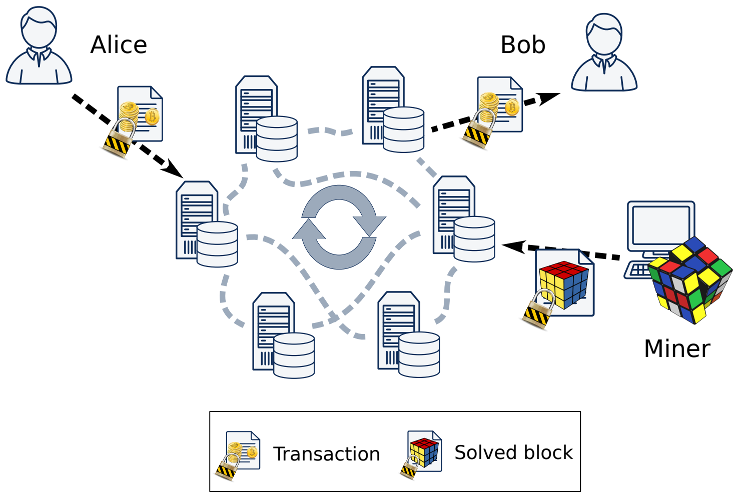


Figure 2.8: How Bitcoin works

received transactions in a (separate) database. When a client receives bitcoins he can verify that they were not spent before by contacting a (small) subset of peers. Clients also need not be always online as they can request transactions from peers at any time.

2.2.2 Transactions and Blockchain

Coins and transactions. By convention, a new portion of 25 coins[16] starts with every newly generated block[17] (see below). The first, *coinbase*, transaction $T_{coinbase}$ in the block "transfers" these coins to the public key P_A of the block's creator (Alice). Alice then can broadcast transactions to transfer the ownership of the coins. In other words coins are created with a new block and transactions are singed messages in which the owner of public key P_X announces "I transfer the ownership of these coins to public key P_Y".

In more details, when Alice decides to transfer some of her coins to Bob, she needs to generate a transaction T_1, sign it with her private key and broadcast it to the network. This signed transaction[18] would consist of the hash $H_{coinbase}$ of the coinbase transaction and Bob's public key P_B. When Bob then decides to pay Carol with these coins he would generate a signed by *his* private key transaction which would contain hash H_1 of transaction T_1 and Carol's public key. Since each new transaction refers to the previous one Carol can follow the chain and verify that it ends with the coinbase transactions.

[16]This amount halves every 4 years. When the Bitcoin was launched it was 50 bitcoins.

[17] This is why Bitcoin participants constantly search for new valid blocks and keep the network running. This also provides a way to initially distribute coins into circulation, since there is no central authority to issue them.

[18]Bitcoin allows one to create very complicated transactions, but for the sake of clarity we consider only very simple transaction with one input and one output.

Blockchain and coin generation. As it is trivial for Alice to generate two different transactions which transfer the same coins to both Bob and Carol, Bitcoin includes a subprotocol using which a client/peer can decide which out of two transaction histories is the "correct" one. Transactions in the transaction history of a peer are stored in blocks. Each block contains some transactions, a timestamp[19] (of when this block was generated) and the hash of the previous block. Thus the transaction history is organized as a chain of blocks or a *blockchain* (see Fig 2.9). The blockchain which requires more computational power to construct it, is considered the "correct" one.

In more detail, each block contains a header and transaction data[20]. The 80-byte header Head contains the 256-bit hash of the previous block H_{i-1}, the timestamp (in seconds) T_i, the 32-bit nonce N_i (used to generate blocks), the hash TX_i of the transaction data, and the difficulty parameter d_i. To be valid, the double-hash of the block header must be smaller (as an integer) than a certain value, which is a linear function of the difficulty parameter:

$$H_i = \text{SHA-256}(\text{SHA-256}(H_{i-1}||T_i||TX_i||d_i||N_i||))) < f(d_i).$$

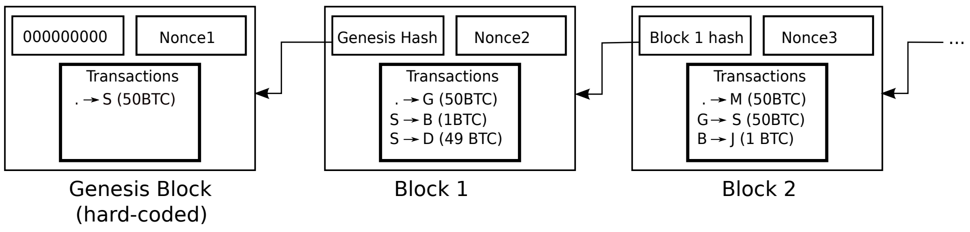


Figure 2.9: Bitcoin blockchain

In order to create a new *valid block* a peer (which we call a *miner*, see Fig. 2.8) needs to provide *proofs of work* (PoW). The Bitcoin miners first collect all transactions not yet included into a block. Then they generate the header fields and exhaustively try different nonces, timestamps, and other parameters in order to obtain a valid block. Whenever a block is created, a miner broadcasts it to the network, so that each node attaches it into its internal block chain. The difficulty parameter is adjusted automatically by the network so that the network generates one block every 10 minutes on the average.

When a payee receives a transaction it checks the blockchain if the same coins were already spent previously. Once a transaction is buried under sufficient number of blocks it becomes computationally impractical to revert it. Note that transactions that transfer ownership of a coin can be in different blocks and it is the responsibility of the Bitcoin client to scan all the blocks and extract the necessary transactions.

[19]In order to get a timestamp, a miner consults an NTP server.

[20]All these conditions are strictly enforced, and a block not conforming to them is discarded immediately.

2.2.3 Bitcoin P2P network

Peers of the Bitcoin network connect to each other over an unencrypted TCP channel. There is no authentication functionality in the network, so each node just keeps a list of IP addresses associated with its connections.

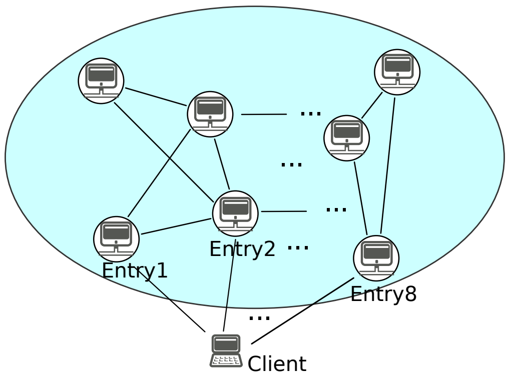


Figure 2.10: Bitcoin P2P network

Though official Bitcoin software[21] does not explicitly divide its functionality between clients and servers, Bitcoin peers can be grouped into those which can accept incoming connections (servers) and those which can't (clients), i.e. peers behind NAT or firewall, etc. At the time of writing there were about 6,000 reachable servers while the estimated number of clients was about 100,000.

By default[22] Bitcoin peers (both clients and servers) try to maintain 8 outgoing connections. In addition, Bitcoin servers can accept up to 117 incoming connections (thus having up to 125 connections in total). If any of the 8 outgoing connections drop, a Bitcoin peer tries to replace them with new connections. If none of the 8 outgoing connections drop, the peer will stay connected to them until it is restarted. In case of a client, we call the 8 nodes to which it establishes connections *entry nodes* (see Fig. 2.10). A Bitcoin server accepts any number of connections from a single IP address as long as the treshold for the total number of connections is not reached.

Bootstrapping. Bitcoin supports three mechanisms for bootstrapping. First, each Bitcoin peer keeps a database of IP addresses of peers previously seen in the network. This database survives between Bitcoin client restarts. This is done by dumping the database to the hard drive every 15 minutes and on exit (as we will see later this facilitates setting a cookie on the user's computer). Second, when a Bitcoin clients starts, it tries to populate its address database by resolving 6 hard-coded hostnames[23]. Finally, as a fallback if no addresses can be found at all, after

[21]Bitcon core version 0.9.2 from 13 June 2014.

[22]Here and below in this section we consider Bitcoin reo version 0.9.2.

[23]If Tor is used, Bitcoin does not explicitly ask Tor to resolve them but rather asks it to establish connections to these hostnames: when applications communicate with Tor they can either ask Tor to establish a connection to a hostname by sending a `CONNECT` command or to resolve a hostname

60 seconds of running the Bitcoin client uses a list of 600 hard-coded IP addresses.

Choosing outgoing connections For each address in the address database, a Bitcoin peer maintains statistics which among other things includes when the address was last seen in the network, if a connection to this address was ever established before, and the timestamp of such connection. All addresses in the database are distributed between so called buckets. There are 256 buckets for "new" addresses (addresses to which the Bitcoin client has never established a connection) and 64 for "tried" addresses (addresses to which there was at least one successful connection). Each bucket can have at most 64 entries (which means that there can be at most 20480 addresses in the database). When a peer establishes outgoing connections, it chooses an address from "tried" buckets with probability $p = 0.9 - 0.1n$, where n is the number of already established outgoing connections. If an address is advertised frequently enough it can be put into up to 4 different "new" buckets. This obviously increases its chances to be selected by a user and to be transferred to a "tried" bucket.

There are some peculiarities when using Bitcoin over Tor. If Tor is not used, the addresses for outgoing connections are taken from the addresses database only. In case Tor is used, every second connection is established to hard-coded DNS hostnames. These DNS hostnames are called "oneshots" and once the client establishes a connection to such a hostname it requests a bunch of addresses from it and then disconnects and never tries to connect to it again.

Self-discovery. After the startup a Bitcoin peer discovers its own IP addresses, which includes not only its network interfaces addresses but also the IP address as it is seen from the Internet (in the majority of cases for NAT users it resolves to an IP address of the peer's ISP). In order to discover the latter, the peer issues a `GET` request to two hard-coded web-sites which reply with the address. For each address obtained by the discover procedure, the peer assigns a score. Local interfaces initially get score 1, the external IP address gets score of 4 (in case the external IP address coincides with one of the local addresses the scores a summed). When a client establishes an outgoing connection to a remote peer, they first exchange `VERSION` messages and the client advertises its address with the highest score. The remote peer then uses the addresses propagation algorithm described below. The client repeats the same procedure for the remaining 7 outgoing connections.

Peer discovery. The Bitcoin protocol implements an address propagation mechanism to help peers to discover other peers in the P2P network. Each Bitcoin peer maintains a list of addresses of other peers in the network and each address is given a timestamp which determines its freshness. Bitcoin peers periodically broadcast their addresses in the network. Peers can request addresses from this list from each other using `GETADDR` messages and unsolicitely advertise addresses known to them

by sending a `RESOLVE` command.

using `ADDR` messages[24].

Bitcoin nodes recognize three types of addresses: IPv4, IPv6, and OnionCat [67]. For each type of addresses the peer maintains a state variable indicating if this address family is reachable or not. An address family is considered *reachable* by a node if the node has a network interface associated with same address family. Otherwise the address family is marked as *unreachable*. These state variables become important when Bitcoin is configured to connect to the network over a Tor proxy: the only address type which is accepted from other peers is OnionCat type. Curiously, this results in that all IPv4 and IPv6 addresses obtained from oneshots are dropped and the client uses its original database. The opposite case also holds: if Tor is not used, onion addresses are not stored in the address database[25].

Whenever a Bitcoin node receives an `ADDR` message it decides individually for each address in the message if to forward it to its neighbours. It first checks if (1) the total number of addresses in the corresponding `ADDR` message does not exceed 10, and (2) the attached timestamp is no older than 10 minutes. If either of these two checks fails, the address is not forwarded; otherwise the address is scheduled for forwarding[26] to two of the node's neighbours in case the address is reachable and to one neighbour only if it is non-reachable. Limiting the number of neighbours to which an address is forwarded reduces the total amount of traffic in the Bitcoin P2P network.

In order to choose neighbours to which to forward an address, a Bitcoin node does the following. For each of its neighbours it computes a hash of a value composed of the following items: address to be forwarded, a secret salt, current day, and the memory address of the data structure describing the neighbour. The exact expression for the hashed value is of little importance for our attacks. The only thing which we need to emphasize is that the hash stays the same for 24 hours. The peer then sorts the list of its neighbours based on the computed hashes and chooses the first entry or two first entries (which depends on the reachability of the address). In the following text we call such nodes *responsible nodes* for the address.

The actual transmission of the scheduled `ADDR` messages does not happen immediately. Every 100 milliseconds one neighbour is randomly selected from the list of all peer's neighbours and the queue for outgoing `ADDR` messages is flushed for this node only. We call the node chosen at the beginning of a 100 milliseconds round *trickle node* and the procedure as a whole as *trickling*.

Consider an example on Fig. 2.11. Assume that node `N0` gets an `ADDR` message with one address A_0 from node `N3` and that node `N0` schedules to forward it to nodes `N1` and `N2` (i.e. these nodes are *responsible nodes* for address A_0). In round 1, node `N1` is chosen as a trickle node and the address is forwarded to this node while the delivery to `N2` is still pending. After 100 milliseconds in round 2, `N3` is

[24]One `ADDR` message can contain any number of address, however messages containing more than 1000 addresses are rejected on the remote side.

[25]While peers drop newly arrived addresses, they can still use addresses which are already in their database.

[26]By scheduling a transmission we mean that the node puts the corresponding message to the outgoing queue but does not yet make the actual transmission.

chosen as the trickle node thus no actual transmission happens at this stage. After another 100 milliseconds in round 3, `N2` is chosen as the trickle node and address A_0 is finally sent to it. Choosing a trickle node causes random delays at each hop during an address propagation.

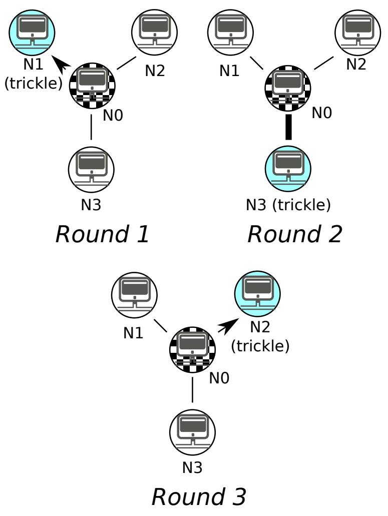


Figure 2.11: Trickling of `ADDR` messages

Finally for each connection, a Bitcoin peer remembers addresses that were forwarded over this connection. Before a peer forwards an address, it first checks if the same address was already sent over the connection. This history is cleared every 24 hours. An important note is that the history of sent addresses is kept per connection and not per IP, i.e. if a Bitcoin peer reconnects, its history will be cleared. The total number of addresses a Bitcoin peer can store is limited by 20480. Whenever new addresses arrive at a peer they replace old ones. In addition when a peer receives a `GETADDR` messages it sends back 23% of the number of addresses it stores but not more than 2500 addresses.

Anti-DoS protection. To avoid denial-of-service attacks, the Bitcoin protocol minimizes the amount of information forwarded by peers. Valid blocks and transactions are relayed whereas invalid blocks are discarded. Moreover, Bitcoin implements a reputation-based protocol with each node keeping a penalty score for every connection. Whenever a malformed message is sent to the node, the latter increases the penalty score of the connection and bans the "misbehaving" IP address for 24 hours when the penalty reaches the value of 100.

Bitcoin peers as Tor hidden services Tor hidden services are service-agnostic in the sense that any TCP-based service can be made available as a Tor hidden service. This is used by Bitcoin which uses Onioncat address format to represent an onion address as an IPv6 address: the first 6 bytes of an OnionCat address are fixed and set to `FD87:D87E:EB43` and the other 10 bytes are the hex version of the onion address (i.e. base32 decoded onion address after removing the ".onion" part).

Transaction propagation. Forwarding a transaction from one peer to another involves several steps. First the sender transmits an `INVENTORY` message with the hash of the transactions. Second, the receiver checks if he already has a valid transction with the same hash. If the checks passes, the receiver requests the actual transaction by sending a `GETDATA` message. The sender then transmits the transaction in a `TRANSACTION` message. When the receiver gets the transaction he advertises it to its peers in another `INVENTORY` message.

When a client generates a transaction he schedules[27] it for forwarding to all of its neighbours. It then computes a hash of a value composed of the transaction hash and a secret salt. If the computed hash has two last bits set to zero the transaction is forwarded[28] immediately to all the 8 entry nodes. Otherwise a queue of a neighbour for outgoing transactions is flushed when the neighbour becomes the trickle node (the same as with `ADDR` messages). Obviously $\frac{1}{4}$ of all transaction are forwarded immediately in average.

When a transaction is received it is scheduled for the delivery to all peer's neighbours as described above. As with `ADDR` messages, a Bitcoin peer maintains history of forwarded transactions for each connection. If a transaction was already sent over a connection it will not be resent again. A Bitcoin peer keeps all received transaction in a memory pool. If the peer received a transaction with the same hash as one in the pool or in a block in the main block chain, the received transaction is rejected.

2.2.4 Mining-pools and altcoins

Initially Bitcoin mining was carried out by CPU. As Bitcoin was becoming more popular and new miners were entering the game, the difficulty raised significantly and mining has moved to GPU. Then with the rise of the Bitcoin price, dedicated ASICs which provided orders of magnitude higher hash rates were developed. The probability to mine a block with a consumer-level GPU became very low at this stage. For miners without powerful dedicated hardware it takes prohibitively long time (years) before they can make a return. Such miners solve this problem by joining their resources in a mining pool. Participants of a mining pool all together generate blocks much faster and receive a portion of the block reward on a consistent basis.

Each miner in a mining pool tries to solve a block with a much lower than the original difficulty. Such simpler block is called a *share*. With some probability the share will also have a solution with the original difficulty in which case the pool mines a block. The block reward is then divided among the participants based on

[27]By scheduling we mean that the node puts the transaction into the outgoing queue of the corresponding connection.

[28]More precisely the peer sends an `INVENTORY` message with the hash of the transaction.

the work they contributed. There exist pools which track crypto-currencies market prices and automatically switch to mining the most profitable crypto-currency.

Since Bitcoin's inception in 2009, a number of alternative currencies has appeared (called *Altcoins*). They all share the same basic principles of Bitcoin, but differ in PoW algorithms, difficulty adjustment rules, and amount of coins generated per block. Bitcoin along with Altcoins form the crypto currency market where they can be exchanged into the usual fiat currencies. Currencies based on ASIC-resistant PoW functions from Password Hashing Competition (PHC) [68] have been recently proposed.

2.2.5 Bitcoin Testnet

Bitcoin Testnet is an alternative Bitcoin network with separate block chain. It is used for testing and testnet coins have no value. The main purpose of the testnet is to allow application developers or bitcoin testers to experiment, without having to use real bitcoins or worrying about breaking the main bitcoin chain.

For the time of experiments (May 2014) the number of running Bitcoin servers in the testnet fluctuated between 230 and 250, while the estimated average degree of the nodes was approximately 30.

Chapter 3

Deanonymizing Tor Connections Using Topology Leaks

The goal of onion routing is to guarantee that each relay in the client's circuit (including the destination server) knows only the adjacent hops in the path. In case of Tor, this effectively means that for the Exit node, the probability of correctly guessing the Guard node is $\frac{1}{n}$, where n is the number of Guards in the Tor network. This only holds for a fully connected network however. While Tor is usually considered as such, in reality it is not and not all entry and exit nodes are connected via three hop paths at a given point of time.

In this chapter we first present two ways to reveal the connectivity of nodes in the Tor network: one using canonical connections which are a part of the Tor specification; the other is a more generic technique, namely a timing attack on the connection establishment between two relays.

Second, we present attacks which are based on the *connectivity scanning* approach. The first attack allows one to identify the guard node which was used in a circuit carrying a long-lived connection – such as an SSH session or a large file download. The second attack, which we have chosen to call *differential scan attack*, uses recurrent connections to reveal all guard nodes of a user. Though revealing a user's Guard does not allow an attacker to immediately determine the actual originator of the connection, it tells her where to attack next.

Contents

3.1 Revealing Tor connectivity dynamics

3.1.1 Canonical Connectivity Scanning

We will now show the first technique to find out if there is a TLS connection between two Tor relays. To explain how this works, we first have to delve into details of the Tor specification. To save resources, when a relay R_1 gets a circuit extend request to R_2 (identified by its fingerprint FP_{R_2} and IP address IP_{R_2}), it should reuse already existing TLS connection to R_2 (if it exists). A naïve approach would be to use the first created connection. But in this case an attacker can redirect unmodified user's traffic through her node and mount a byte counting attack. Instead the Tor protocol implements connection reuse in the following way.

When R_1 establishes its first connection to R_2, it marks the connection as *canonical* if IP_{R_2} in the extend request coincides with R_2's[1] IP address in the Consensus. Once a canonical connection is established, R_1 ignores the IP addresses for all future extend requests to R_2, and uses the IP address from the consensus instead.

We noticed that Canonical connections give an attacker a convenient way to determine how routers in the Tor network are connected to each other. When sending a `RELAY EXTEND` cell, the circuit originator specifies both the identity fingerprint and the IP address of the router he wants to extend the circuit to. Assume that the attacker wants to figure out whether router R_1 is connected to router R_2. In order to do this, the attacker constructs a Tor `RELAY EXTEND` cell with FP_{R_1} as fingerprint and an IP address from non-private range with an unreachable port (port 1 for example) and sends it to R_2. When the cell is received, the reaction of router R_1 depends on whether it has a connection to router R_2:

- If R_1 has a canonical connection to R_2 (it should be noted that if a connection exists it is almost always canonical), router R_1 ignores the IP address from the forged `RELAY_EXTEND` cell and uses the already established TLS connection, extends the circuit and sends back `RELAY_EXTENDED` cell.

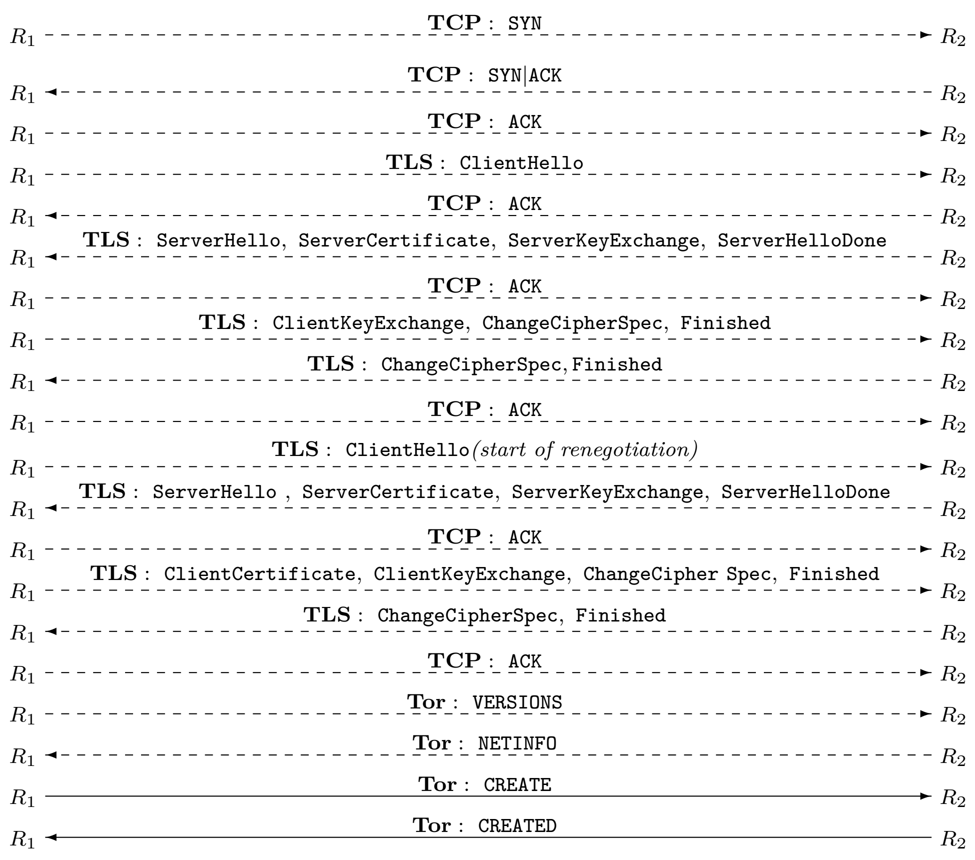
- If R_1 does not have a connection to R_2 then it tries to make a new TLS connection using the address from the received cell. Obviously, the connection attempt is refused which causes router R_1 to send a `DESTROY` cell to the attacker.

By inspecting the cell the attacker receives back from router R_1, she can determine whether router R_1 is connected to router R_2. Evidently, the attacker can probe router R_1 for connection with any router contained in the consensus.

[1]Identified by its fingerprint.

3.1.2 Connectivity probing via timing attacks

We now consider the second, somewhat less powerful approach for determining whether two relays are already connected. When a client extends a circuit from relay R_1 to relay R_2, the time until he received the `RELAY EXTENDED` reply from R_2 depends on whether a TLS connection between R_1 and R_2 is already set up or whether it needs to be established first. In the later case, both the additional network and the cryptographic latency are considerable.



R_1 R_2
TCP : SYN
TCP : SYN|ACK
TCP : ACK
TLS : ClientHello
TCP : ACK
TLS : ServerHello, ServerCertificate, ServerKeyExchange, ServerHelloDone
TCP : ACK
TLS : ClientKeyExchange, ChangeCipherSpec, Finished
TLS : ChangeCipherSpec, Finished
TCP : ACK
TLS : ClientHello *(start of renegotiation)*
TLS : ServerHello , ServerCertificate, ServerKeyExchange, ServerHelloDone
TCP : ACK
TLS : ClientCertificate, ClientKeyExchange, ChangeCipher Spec, Finished
TLS : ChangeCipherSpec, Finished
TCP : ACK
Tor : VERSIONS
Tor : NETINFO
Tor : CREATE
Tor : CREATED

Figure 3.1: Tor circuit setup. The last two steps are performed always. Steps marked with dashed lines are performed only when there is no TLS-connection between R_1 and R_2.

A TLS connection setup between Tor relays can cause huge delays, especially if version 2 or above of the handshake protocol is used. This delay is caused by network latency and the large number of protocol steps until the `CREATE` cell can be sent (see Figure 3.1 for details). If a TLS connection needs to be set up to create a circuit, a delay on the order of 7.5 round-trip times is added to the circuit creation until the `CREATE` cell is received by R_1. Approximately 6.5 round trips are required for the TLS connection setup alone, another round-trip for the v2 handshake. By sending multiple `RELAY EXTEND` requests and comparing the time it takes for the first one to arrive versus subsequent ones, we can determine whether a relay is connected to another relay. This has been confirmed with experiments. The disadvantage of this method is that network jitter as well as cell forwarding delays by the relay scanned can add significant amounts of noise which makes the method less reliable. Moreover, in contrast to the method described in the previous subsection, this method will really establish TLS connections to all routers that are scanned and not just prolong the lifetimes of the connections that are already open.

3.2 Attacking Tor using connectivity dynamics

3.2.1 Tracing long-lived streams

Tor is used by many people to establish long-lived SSH sessions, download very large files (sometimes using file-sharing applications, even though this is frowned upon) and to communicate over instant messaging networks. The use-cases described above imply long-lived TCP-streams which necessarily create long-lived TLS-connections between Tor routers which are used to carry the stream. Thus, we show how an attacker knowing the exit node of a long-lived TCP-stream can link it with the guard node using our scanning techniques[2].

One-Hop Attack

In this attack, we assume that the attacker controls one or more very fast exit routers which see a significant fraction of the traffic exiting the Tor network, thus she gets access to pseudonyms of the users (ex. cookies, logins). This is not an unrealistic scenario; some organizations have control over sizable portions of the total exit traffic: according to the consensus in February 2012, 7.2% of total exit capacity were provided by the Chaos Computer Club, 5.9% by Torservers.net and 5.4% by Formless Networking LLC. The attacker is curious to connect the pseudonyms with guard nodes of users that pass through her Exit relays. Assume that one of the attacker's nodes E (see Figure 3.2) is selected as the exit node of a circuit. By looking at the traffic pattern, the attacker will be able to infer that the connection to the exit node is likely to be of long-lived type. The attacker then starts the attack:

1. The attacker starts scanning the middle node M for connectivity using either of the techniques described in the previous section. The set of connected nodes necessarily includes the guard node G in question and makes up its initial anonymity set.

[2]One important note is that in the current Tor protocol, the connections between two routers which last more than 7 days are marked as "bad" for new circuits and no new circuits can be added to such connections. However persistent circuits inside these connections are not closed and will continue running. At the same time we cannot see these persistent OR connections anymore using our probing techniques after 7 days have elapsed.

2. Next, the attacker continues with the connectivity scanning of the middle node for several hours or even days in hope that the majority of the nodes of the initial anonymity set will disconnect (nodes with dash lines on Figure 3.2.)

3. The attack stops when the anonymity set of the guard node is considerably reduced or when the user closes the long-lived TCP-stream.

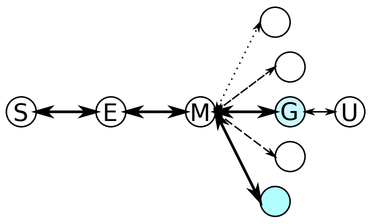


Figure 3.2: One-hop attack against long-lived connections

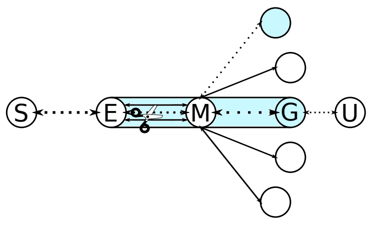


Figure 3.3: Differential scanning attack

When the attack is finished, the user's guard node will be in the resulting anonymity set (node *G* and another node with the solid line on Figure 3.2) along with some number of other connections that can be considered as "noise". The attacker may also infer extra information from the speed of the connection, which will indicate whether the middle or the guard node are the bottleneck for the traffic of the long-lived circuit; this helps her to further shrink the set of candidates for the guard node since it allows to discard very active routers from the list of candidate guard nodes.

Two-Hop Attack

This attack does not required from the attacker to control any relays in the Tor network and can be performed by a server (or an attacker close to the server) who tries to reveal the guard nodes of pseudonymous users connecting to the server. The attack starts from connectivity scanning of the exit node (similar to one-hop attack) in order to reduce the anonymity set of the middle node. After having narrowed down the set sufficiently, the candidate middle nodes are scanned resulting in the anonymity set of the guard node. The attack might be successful if either middle or guard nodes are low-bandwidth which might be inferred from the connection latency by the attacker. We also assume that exit node is medium or low-bandwidth. The difficulty in the two-hop attack comes from the fact that many middle nodes reachable from the exit node would come from a set of active routers with many connections. This will result in hundreds of candidate guard nodes even after several days of scanning. This effect happens due to "immortal"connections formed between active routers, which we will describe in Section 3.3. In spite of its simplicity, the described attack is quite powerful since:

(i) it does not require control over any relays in the Tor network. The attacker merely probes relays (probing could be also done from a distributed set of addresses);

(ii) it is cheap in terms of bandwidth: in order to scan one router the aggregated amount of traffic that needs to be sent and received is less than 5 MBytes (for the size of 3000 routers of the Tor network in February 2012);

(iii) it is fast: the average time of scanning one router is 20 seconds and scanning of different routers can be easily parallelized (again for the size of the network in 2012).

Experimental results

In order to estimate how efficient the attacks can be in the wild, we used Python to implement a rudimentary Tor client which provides basic functionality. The client can establish a TLS connection to an arbitrary Tor router, complete Diffie-Hellman key establishment protocol and send and receive Tor relay cells. In other words, the client is able to create and extend arbitrary chosen circuits. Using canonical connectivity scanning, our client is able to check a Tor router for connectivity with 99% of other routers in the Tor network in less than 30 seconds.

In order to check the correctness of the proposed canonical connectivity scanning, we scanned two routers under our control `omicron` and `Layercake` for five days from February 11th until February 16th, 2012. During the experiment the routers had bandwidth weights in the range [500 - 1500] for `omicron` and in the range [15000-55000] for `layercake` which means that the later was in the top 10% set of fastest and thus most frequently chosen routers. Both relays had Guard flags and did not have Exit flags. Since the routers were operated by us, we could gather the real time statistics directly from them using the Tor control port. We then compared the results from the canonical connectivity scan and from the control port. Figure 3.4 shows the number of persistently connected Tor routers over time, i.e. those routers which were connected to our routers at the start of the experiment and never disconnected during the experiment. The close match of the results as shown on Figure 3.4 demonstrates that canonical connections scanning provides reliable results. The slight difference in the results is explained by the difference of scanning frequency: for canonical connection scanning, each sample cannot be taken faster than every three minutes (i.e. the lifetime of an idle Tor TLS connection); the data from the routers' control port however was fetched every ten seconds. According to Figure 3.4, for the router with bandwidth weight 1500 (`omicron`), the number of persistently connected routers decayed from 303 to 20 in just 12 hours. This matches with our prediction from Section 3.3.1. It then took 4 days for another 18 routers to disconnect. Among two remaining connections, there was one which we established by ourselves and which we tried to identify. The decay rate of persistent connections of the high-bandwidth router (layercake) looks similar: the number of persistent connections drops sharply from 1116 to 300 in 12 hours and then decays slowly. We tested canonical connection scanning on several

Tor routers not under our control. The results for one such router with bandwidth weight in range [2040-2190] are shown on Figure 3.5. We observed a very similar behaviour: a large number of connections dropped quickly, and the rest decayed slowly. After two days of scanning, we found 12 persistent connections.

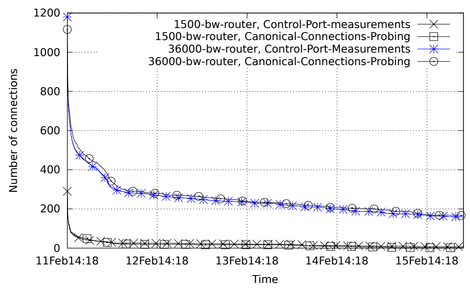


Figure 3.4: Decay rate of Persistent connections. Canonical connection scan vs. direct measurements

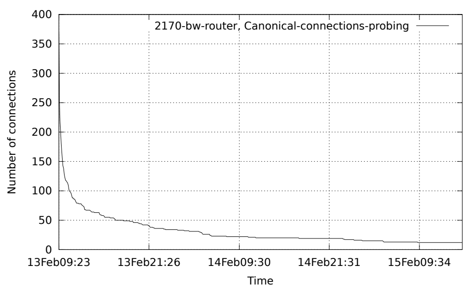


Figure 3.5: Persistent connections decay rate for a random router

3.2.2 Differential scan attack

Attack description

Consider a user which periodically checks some Web server or a web service that instructs the user's browser to periodically re-establish streams. Google Mail for instance builds a series of short-lived (around 2 minutes) TCP sessions. Another example are news web sites with auto-refresh contents. In this section, we describe an attack on such kind of recurrent connections. The aim of the attacker is to find at least one of the guard nodes[3] of a pseudonymous user (identified by a cookie or a login credential) that uses such a service for several days. Note that this attack does not require a single long-lived circuit or session. It just requires that a Tor client is connected to the Tor network for non-negligible amount of time within the span of a month (as long as the guards are still valid).

Similar to Section 3.2.1, in this attack, the attacker has control over a significant fraction of the exit capacity of the Tor network. Assume that a user visits a Web server S (see Figure 3.3) that causes recurrent connections to occur. Ten minutes after the first connection, his initial circuit should expire and the user's Tor client will try to build a new circuit. Given a sufficient number of exit nodes controlled by the attacker, the circuit will include one of the attacker's exit nodes E. Once the exit node receives incoming traffic destined to the web server it executes the following sequence of steps:

[3]In February - March 2012, when the attacks were implemented, each Tor client had a set of three Guard nodes.

1. The exit node E observing the stream to the web server determines the middle node M of the circuit that caused the stream to be established and transmits it to the attacker.

2. The attacker probes the connectivity of M and remembers the list of routers connected to it (nodes connected to M both with dash and solid lines on Figure 3.3).

3. E sends a `DESTROY` cell[4] down the circuit which leads to the circuit termination. The circuit termination may lead to the connection termination between the middle node and the user's guard node with some probability which can be estimated using expressions from Section 3.3.2.

4. The attacker waits for three minutes and starts the scan of M again.

5. The attacker computes the difference between the sets obtained via the first and the second scans, i.e. he determines connections which were present in the first list but absent in the second (node G and another node with dash line.) We say that we have a differential with node G and M if G is in the difference.

6. The attacker then repeats steps 1-3 each time one of her exit nodes is chosen for the recurrent connection.

7. Once an attacker has performed the above steps often enough, and given that the circuit closure event caused the connections closure frequently, she can discover the user's Guard nodes: the probability of having the guard node in the difference should converge to $1/N$ (where N is the number of the user's Guard nodes).

This attack may be further enhanced by scanning the full network at regular and frequent intervals. Then if the connection to the malicious Exit arrives shortly after the full network scan, the attacker will have additional differential connectivity information in order to filter the noise. Our experiments have shown that the full network scan can be done in 3 minutes using 20 hosts (using Amazon EC2 service, a day of full network scans with 3 minutes between scans costs around 80 USD).

A similar but less stealthy approach can be used to track any user's connection. Assume that a user connecting to a server chose one of the attacker's exit node. This allows the attacker to incorporate a small piece of code in each HTML document requested by the user, which artificially creates recurrent connections. Specifically the user can be redirected to an arbitrary address and port. Note that in the Tor network, aggregated exit bandwidth for different port is different, thus by choosing the appropriate port range, the attacker can increase the probability that her exit node is chosen.

[4]If the attacker wants to be more stealthy she can just wait until the circuit expires by itself.

Experimental results[5]

We have implemented a proof of concept version of our differential scanning technique and have tested it using sets of paths generated by a modified version of the Tor client – this client does not create any circuits but simply outputs randomly generated paths with user-specified constraints. These paths are then used to build circuits through the control port of the Tor daemon. After a circuit has been built, a scan is conducted, then the circuit is torn down, the program waits for 200 seconds and scans again. To perform experiments more quickly we have implemented this in a parallelized manner on Amazon's EC2 platform so that many (non-interfering) experiments can be conducted in parallel. As a first experiment, we used only one guard node with capacity of 36500 and allowed for middle nodes with capacity of 1600 or lower in the consensus[6]. For 150 paths, 125 successful differential scans were performed. In these, the guard node we had selected appeared at position 1 in the list of most frequently occurring guard relays in the difference sets, having been counted 58 times.

1. `C37B234FAD013453B90375EB55864FEBC876104A`: 58 `(PPrivCom052)` bw=36500
2. `CA1CF70F4E6AF9172E6E743AC5F1E918FFE2B476`: 35 `(spfTOR3)` bw=29800
3. `0B7ED44C67DBE50313F0B32BD335D093D0474CE8`: 33 `(bauruine2)` bw=117000
4. `847B1F850344D7876491A54892F904934E4EB85D`: 31 `(tor26)` bw=20
5. `DB8C6D8E0D51A42BDDA81A9B8A735B41B2CF95D1`: 30 `(rainbowwarrior)` bw=81300
6. `173B220F9F32F39086D5661274A47485EDA26131`: 29 `(TorExitProgressbar9)` bw=650
7. `1603DFE9FC373ECDA39046FADB5A76B87A4BA36B`: 27 `(StickItToTheMan)` bw=46800
8. `1F52D692FA2C21B23FAD4D711A7BF17BAE2673DF`: 26 `(alice)` bw=7170
9. `47916CAB5878C810E7EF71A316D37FC823CC7F52`: 26 `(CCN)` bw=53100
10. `95A0D58710EA9B61DAD3A01CAD3BE77DACA76BEF`: 25 `(OccupyMyPants)` bw=30300

This shows that differential probing works in practice: there's a drastic reduction in the anonymity set of the guard nodes, even for high capacity guard nodes. Below is the concrete data of one of the experiments in which we had chosen guards of capacity 300, 412, and 501, constrained the capacity of the middle nodes to 30,000 and scanned different middle nodes in 134 trials[7]:

1. `A58E0F05C1939725D7247BA60BA3135DB88209BC`: 43 `(jef0lewkia)`, bw = 501
2. `D3378ABA009078158DB59E8B36B8EBB88B309BA7`: 40 `(torn0t)`, bw = 412
3. `2629979FD21BF3B522E818B73F6F8D0B5D8A5CF0`: 40 `(tapir)`, bw = 300
4. `A9C039A5FD02FCA06303DCFAABE25C5912C63B26`: 29 `(chaoscomputerclub5)`, bw = 173000
5. `FA486415B86D28CD047D10F76768E4E88A182F71`: 28 `(ZhangPoland1)`, bw = 56400
6. `131B60B9AFE6AEA60042132D648798534ABEA07E`: 28 `(wagtail)`, bw = 24400
7. `4536ED68D9DB4B2FF532AD43A632AAF600B798CC`: 27 `(Unnamed)`, bw = 116
8. `1D8625690AB9729FB2040D8194EC0D6789A4D092`: 25 `(TOR1CINIPAC)`, bw = 43900
9. `FC35DE87F6E4022693323275F6B8EE5F72FD21B5`: 24 `(Unzane)`, bw = 3160
10. `CA1CF70F4E6AF9172E6E743AC5F1E918FFE2B476`: 23 `(spfTOR3)`, bw = 28700

[5]The experiments were carried out in February - March 2012. Each users had a set of $N = 3$ Guard nodes at this time and the total Exit capacity was approximately $5 \cdot 10^6$ Kbyte/s.

[6]See Section 3.3 for justification of the choice of the bandwidths. In brief: (1) the product of bandwidths of the guard node and middle node should not exceed 300 million to avoid "immortal connections" ; (2) the attack works best when either the guard or the middle node are not high-bandwidth.

[7]`jef0lewkia` was involved in 43 circuits, `torn0t` in 45 and `tapir` in 46.

Again, although we have some spurious low-bandwidth routers in the top ten, these results show that the attack described above works well in practice. Note that in real life, the attacker will perform scans for any circuit which has been detected to be established by a unique pseudonym of a user and for which the middle node is below a certain threshold bandwidth.

We now try to estimate how many measurements the attacker should make when low capacity guards are being used. There are 1,440 minutes in a day; this means that if the attacker is unlucky (i.e. his exit is not selected and then she needs to wait for 10 minutes until the circuit expires in order to get another chance) there are 144 measurement chances per day. The fact that attacker controlling a fraction f of the exit bandwidth tears down circuits to which she gets access, increases the number of measurement slots available to the attacker by a factor $\frac{1}{1-f}$, which for $f = 1/3$ results in $144\frac{f}{1-f} = 72$ slots. If the upper bound for the capacity of the middle node is set to 30000 then (according to Figure 3.11) there is about 40% chance for a circuit to go through such middle node. This reduces the amount of measurements to 29 per day. The attacker will continue the attack until he obtains about 40 measurements, which means the attack will run for about 1 day. Note that the attack is very successful if the bandwidth of one of the user guard nodes is below 500. There is about 3% chance that a user's client has chosen a Guard node with low capacity, i.e. $G_{min} < 500$, into his triplet of guard nodes (in February - March 2012). Thus this attack could affect more than 10,000 daily users of the Tor network.

3.3 Analysis of the attacks

3.3.1 Long-lived connections

In section 3.2.1, one could notice that after a relatively short period of scanning time, a substantial number of routers which were connected at the beginning of the scan disconnected. The decay rate of the number of persistently connected routers becomes very low after. In other words, when the number of connections drops to some value, the reduction rate of the anonymity set of the guard node an attacker is trying to identify becomes negligible. This value can be considered as a threshold for this attack which we try to estimate in this section. There are two main reasons of why a connection between a pair of routers may last very long and thus increase the anonymity set:

1. This pair of routers is a part of a long-lived circuit similar to the one the attacker tries to identify, i.e. a circuit which is used for an application which requires a long-lived TCP-stream;

2. The circuit creation rate over this connection is high and there is always at least one circuit inside this connection which prevents it from closing. Such *immortal* connections form if the product of bandwidths of the two routers exceeds a certain threshold as will be shown below.

First, we estimate the number of connections of the first type. Figures 3.6 and 3.7 show circuit duration distributions over a connection between two high bandwidth routers (`layercake` with bandwidth weight of 35300 and `bouazizi` with bandwidth weight of 69700 for 13 of Feb 2012). Figures 3.8 and 3.9 show circuit duration distributions over a connection between a high bandwidth router and a non-high bandwidth router (`omicron` with bandwidth 491 for 13 of Feb 2012 and `layercake`). Life-times of circuits have two clear peaks at around 10 and 60 minutes due to properties of the Tor protocol: renewal time of "dirty" circuits and the lifetime of "clean" circuits which have never been marked as "dirty". According to the measurement, circuits with life-time longer than 2 hours constitute less than 1.5% of the total number of circuits. From this we can assume that the majority of long-lived connections in Tor are of the seconds type, i.e. formed by high circuits creation rate over these connections. Another observation is that the anonymity set of persistent streams is small compared to the anonymity set of non-persistent streams.

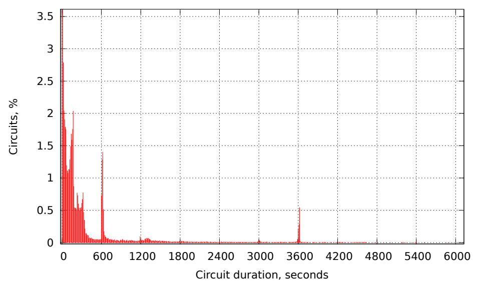


Figure 3.6: Circuit duration probability density function between two high bandwidth routers

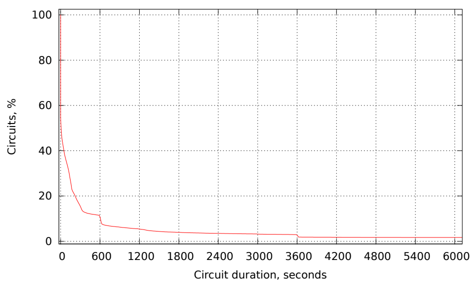


Figure 3.7: Circuit duration distribution function between two high bandwidth routers

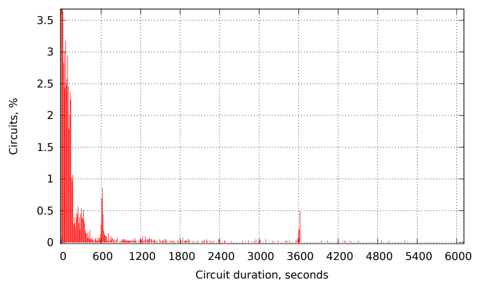


Figure 3.8: Circuit duration probability density function between a high bandwidth and non-high-bandwidth routers

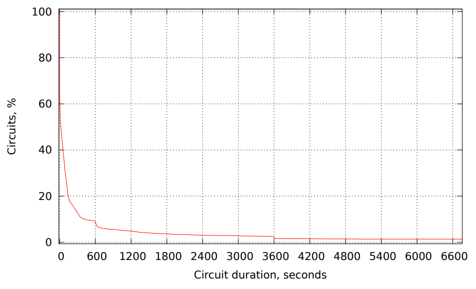


Figure 3.9: Circuit duration distribution function between a high bandwidth and non-high-bandwidth routers

To estimate the number of long-lived connections of the second type, we observed several active (i.e. high-bandwidth) guard Tor routers under our control and measured client circuits arrival rate. Figure 3.10 shows the number of new circuit per ten seconds gathered during two days on one of our active routers. We observed that:

- Circuits arrive according to non-homogeneous Poisson process.
- Assuming that client circuit arrival rate is proportional to the guard router's bandwidth, we estimate an average circuit arrival rate R in the whole Tor network to be about 900 circuits per second (in February 2012). In the expressions below one can also use the value of circuit arrival rate for the specific time of the day instead of the average value.
- The average circuit duration time t_{avg} is about 200 seconds which varies only slightly for routers with different bandwidth weights.

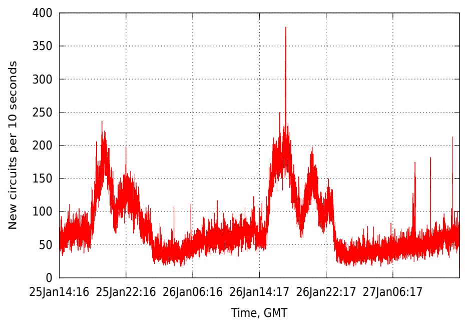


Figure 3.10: Circuit arrival rate for an active high bandwidth router

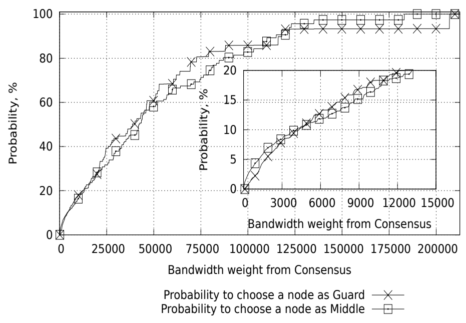


Figure 3.11: Probability for a node to be chosen as a guard and a middle node

We now estimate how likely it is for a pair of routers A and B to be connected with an "immortal" connection. Note that a TLS-connection between Tor relays is closed only if no circuits were carried over this connection for $t_{idle} = 180$ seconds. We assume that (1) circuit duration follows exponential distribution with parameter $\mu = \frac{1}{t_{avg}+t_{idle}}$; (2) circuits between routers A and B arrive according to Poisson distribution with rate $\lambda_{a,b} = p_{A,B} \cdot R$. Here $p_{A,B}$ is the probability of routers A and B to form an edge in a new circuit[8].

$$p_{A,B} = 2 \cdot \frac{bw_A bw_B}{bw_{total}} \left(\frac{1}{bw_{guards}} + \frac{1}{bw_{exit}} \right),$$

[8]This expression for $p_{a,b}$ is an approximation since it does not take into account all peculiarities of the Tor path selection algorithm, in particular, the expression ignores weights which are assigned to a relay based on its position in the circuit and its flags. We compared our approximation with the precise calculation and found that simpler approximation is sufficient for our purposes and makes the analysis easier to understand.

where bw_{guards} is the total bandwidth of guard nodes, bw_{exit} is the total bandwidth of exit nodes, bw_{total} is the total bandwidth of the whole Tor network, bw_A and bw_B are bandwidths of routers A and B respectively.

We model a connection between A and B as a Birth-and-Death process (or more exactly as a simplest trunking problem, see e.g. [4], section XVII.7) with $\lambda_n = \lambda_{a,b} = \lambda$ and $\mu_n = n\mu$ (the system is in state i if there are i circuits inside the connection). We are interested in stationary probability $p_0^{(A,B)}$ of state 0, i.e. to have zero circuits between A and B. The well-known result for simplest trunking problem: $p_n^{(A,B)} = e^{-\lambda\backslash\mu} \cdot \frac{(\lambda\backslash\mu)^n}{n!}$, and thus $p_0^{(A,B)} = e^{-\lambda\backslash\mu}$ A connection between A and B almost never closes if p_0 is close to 0. Using this expression we find that immortal connections are formed between routers of bandwidth $> 17,500$ (or routers with product of bandwidths above 300 million, in February 2012). By bandwidth we mean not the advertised bandwidth but actual figures from the Consensus computed by Tor authorities' bandwidth measurements and used in the Tor code to choose routers for the circuits. Given the bandwidth of a router, an attacker can estimate the number of immortal connections that it has and decide whether it is worthwhile to perform the attack.

Figure 3.13 shows complementary cumulative bandwidth distribution of Tor relays along with the share (i.e. the percentage of total number of Tor relays) of persistent connections for each bandwidth[9] during March and February 2012. For example, if an attacker decides to scan a Tor relay with bandwidth weight of 5000, she can expect that this relay has about 1% of "immortal" connections. Given 3000 Tor relays, this yields the anonymity set of 30 relays. This kind of prediction corresponds well with the experimental results obtained in section 3.2.1. If $bw < 1300$, the attack should give the unique solution[10]. Note that although only few routers have large percentage of immortal connections, these routers are high-bandwidth and and are selected more frequently.

In order to give a first order approximation of how long we should wait until a persistent connection is detectable among other "non-immortal" connections, we collected connection duration statistics from Tor routers operated by us for 7 days[11]. Figure 3.12 shows the connection duration distribution for two pairs of routers: medium-to-medium bandwidth (lower curve), medium-to-high bandwidth. For medium-to-high only 5% of connections have duration of more than three hours. In the case of medium-to-medium bandwidth routers (see Fig. 3.12), only 5% of connections between them have duration of more than 1 hour. In ten hours, 99% of all non-immortal connections should disconnect for both cases. Thus, we expect that if a persistent connection under observation has a duration of more then 10 hours, the probability of its successful identification depends mostly on the number

[9]Note that bandwidth distribution can be approximated by the Pareto distribution with minimal value $x_m = 350$ and exponent $\alpha = 0.85$.

[10]For 11th of February 17:00, 2012, there were 2388 nodes out of 2897 with bandwidth less than 1300. Their aggregated capacity was 371,159 out of 9,458,556 total capacity of the whole Tor network.

[11]The logs we obtained were stored on computers with full-disk encryption behind the firewall of our academic institution.

of immortal connections.

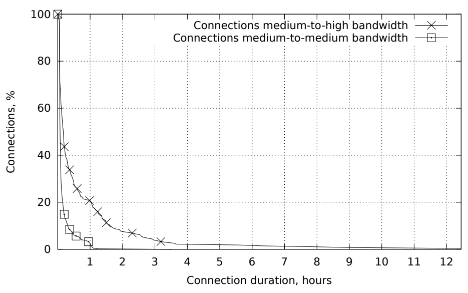


Figure 3.12: Connection duration distribution

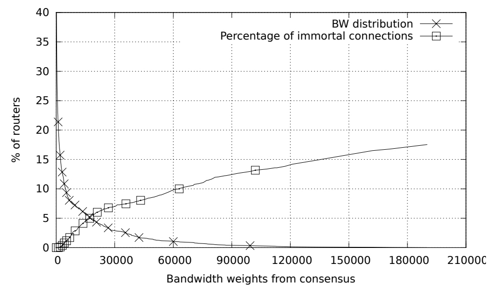


Figure 3.13: Tor bandwidth distribution and share of immortal connections

3.3.2 Differential scanning attack

In this section, we explore the limits of the differential scan attack. Assume that an attacker tries to reveal a guard node g by establishing/dropping circuits $\{1, ..., k\}$ which leads to scanning of a set of middle nodes $M = \{m_1, m_2, ..., m_k\}$. Let T denote the set of all Tor relays and $|T| = n$. Then we define $d : M \times T \longrightarrow \{0, 1\}$ in the following way:

$$d(m_i, r) = \begin{cases} 1 & \text{if we observed a differential between Tor relays } m_i \text{ and } r \text{ for circuit } i \\ 0 & \text{otherwise.} \end{cases}$$

Recall that we say that there is a differential between m_i and r if there was a connection between m_i and r during the first scan and this connection closed during the second scan. Probability to have a differential between m_i and guard node g equals the probability that there are no circuits between m_i and g at the moment of the second scan which is $p_0^{(m_i,g)}$ (see the previous section).

In regard to false positives, for some Tor relay $r \neq g$, $d(m_i, r) = 1$ if: (a) at the time of the first scan, there is a connection between m_i and r; (b) there is no connection at the time of the second scan. Assuming these two event are independent, the probability of a differential between m_i and r is: $(1 - p_0^{(m_i,r)}) \cdot p_0^{(m_i,r)}$.

The overall success of the attack depends on: (1) $Signal = \sum_{i=1}^{k} d(m_i, g)$, i.e. number of differentials with guard node g , and (2) $Noise_{r_j} = \sum_{i=1}^{k} d(m_i, r_j)$, number of differentials with some other Tor relay r_j, $j = 1, ..., n$. We then use signal-to-noise ratio $SNR = \frac{Signal}{\max_j\{Noise_{r_j}\}}$ as a measure of the success of the attack.

To demonstrate how the above expressions work, we used the set of 125 middle nodes from the experiment described in Section 3.2.2 with bandwidth weights equal or less then 1600. Figure 3.14 shows: (a) the expected Signal of the guard node against its bandwidth; (b) the expected Noise of a Tor relay against its bandwidth.

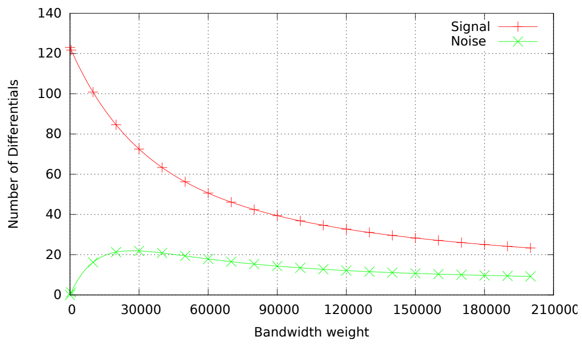


Figure 3.14: Signal and Noise for differential scan

As can be seen from the figure, for low-bandwidth nodes the signal is close to its maximum value. This happens since for this type of node, the probability that the connection between it and a middle node carries just one circuit is very high. Low circuit arrival rate of a low-bandwidth relay also implies the low value of noise since the probability to have a connection between it and a middle node is low.

3.4 Potential countermeasures and conclusion

A potential countermeasure against the canonical connection scan is to abolish canonical connections. Of course this must be done while preserving the circuit multiplexing feature. An obvious approach is to not have a set of connections that are identified by the fingerprint as a primary key but rather by both the fingerprint and the IP address of the relay. This prevents the attack, but needs to be weighed against the fact that incorporating this fix gives an attacker a new way to perform denial-of-service by resource exhaustion against Tor relays.

The countermeasures for obtaining relay connectivity by using timing informations is not straightforward; experiences in side-channel cryptanalysis have shown that simple countermeasures like adding randomized delays can often be defeated. At the same time, a fully connected graph for the Tor network – i.e. having each relay connected to all the other relays at all times – probably is too expensive from a performance standpoint. The balance to strike here is to add sufficient noise to make timing attacks unreliable to attackers.

Since our connectivity revealing techniques are orthogonal to the existing attacks described in the literature, they can be used to improve many of them substantially. Indeed, during the times when the number of Tor routers was small, several attacks were available to adversaries. These attacks allowed to link the exit and entry nodes of a user's circuit. However, once the number of Tor routers grew, those attacks became too expensive in terms of required bandwidth and time. This is because for those attacks to be successful, exhaustive probing of each link in the Tor network was required. Given a way to determine the real connectivity of Tor network, these attacks can become practical again since the amount of links to be probed is significantly reduced.

Chapter 4

Anonymity Analysis of Tor Hidden Services

Tor allows people not only to access information anonymously but also to provide services anonymously. This kind functionality, *responder privacy*, can be achieved with Tor by making TCP services available as *hidden services*.

In this chapter we analyze the security of Tor hidden services. We look at them from different attack perspectives and provide a systematic picture of what information can be obtained with very inexpensive means. We focus both on attacks that allow to censor access to targeted hidden services as well as on deanonymization of operators and clients of hidden services.

We also study *deployed hidden services*. First, we analyze and classify hidden services collected during our experiments. Second, we apply our findings to a botnet which makes its command and control center available to bots as a Tor hidden service (we extracted its onion address by analyzing a malware sample) and extrapolate its size by counting the number of hidden service requests.

Contents

4.1 Shadowing

According to Tor Directory specification [78], section 3.4.2, the maximum number of Tor relays on a single IP address that Tor authorities include to the Consensus document is 2. This restriction is enforced by the directory authorities when they cast their votes for the consensus. If more than two relays are running on the same IP address, only two relays with the highest-most measured bandwidth will appear in the consensus document. This prevents an attacker from performing the Sybil attack described by Bauer et al. [17], in which an attacker floods the network with dummy Tor relays.

However, by inspecting the Tor source code[1] we noticed that while only two relays per IP appear in the Consensus, all running relays are monitored by the authorities; more importantly, statistics on them is collected, including the uptime which is used to decide which flags a relay will be assigned.

We call relays appearing in the consensus *active relays* and those which run at the same IP address but do not appear in the consensus *shadow relays*. Whenever one of the active relays becomes unreachable and disappears from the consensus, one of the shadow relays becomes active, i.e. appears in the consensus. Interestingly, this new active relay will have all the flags corresponding to its real run time and not to the time for which it was in the consensus. We call this technique *shadowing*. **From February 2013 Tor implements a countermeasure against the shadowing technique.**

4.2 Bandwidth inflation

The path selection algorithm [80] of Tor selects nodes at random, with a probability proportional to the bandwidth advertised for the node in the consensus document. Hence it is of interest to an attacker to artificially inflate the bandwidth of her nodes, in order to increase the chance of of them being included in the path.

In the current design, Tor authorities actively measure the bandwidth. A subset of directory authorities operate a set of bandwidth scanners [69], which periodically choose two-hop exit circuits and download predefined files from a particular set of IP addresses (according to the source code of bandwidth scanners from summer 2014, there are two such IP addresses). The bandwidth of a relay shown in the

[1]Tor version 0.2.2.37 from June 2012.

consensus depends on the self-reported bandwidth B_{rep} and the bandwidth measurement reports $B_{measured}$ by the Tor authorities. The weak point of this approach is the fact that the scanning can be reliably detected by relays that want to cheat. To inflate bandwidth and attacker would provide more bandwidth for authorities' measurement streams while throttling bandwidth for all other streams. This results in a high bandwidth value shown in the consensus while keeping the traffic expenses at a low level.

When doing bandwidth measurements, authorities establish two-hop circuits. Thus it is sufficient for cheating non-exit nodes to provide more bandwidth for streams which originate at IP addresses of authorities and throttle all other streams. As an improvement the attacker can take into account that for bandwidth measurements authorities download files which are known. Taking this into account, the attacker can drop circuits which carry a traffic pattern inconsistent with these downloads.

During experiments we were able to inflate the bandwidth of our relays more than ten fold; while the consensus showed bandwidth values of approximately 5000 kBytes/sec per relay, they only provided 400 kBytes/sec of real bandwidth to Tor clients each.

4.3 Catching and tracking hidden service descriptors

In this section we study the security of descriptor distribution procedure for Tor hidden services. We show how an attacker can gain complete control over the distribution of the descriptors of a particular hidden service. This undermines their security significantly: before being able to establish a connection to a hidden service, a client needs to fetch the hidden service's descriptor; unless it has it cached from a prior connection attempt. Thus, should the attacker be able to control the access to the descriptors, the hidden service's activity can be monitored or it can be made completely unavailable to the clients. We apply our finding to several deployed hidden services and we start this section with the description of these hidden services.

4.3.1 Examples of hidden services analyzed

A botnet using hidden services: In April 2012, an "ask me anything" thread (AMA) on the social news website Reddit appeared in which an anonymous poster, allegedly a malware coder and botnet operator, claimed to be operating a botnet with its command and control center running as a Tor hidden service [33]. The malware installed on the clients was described to be a modified version of ZeuS.

Subsequently, a thread on the tor-talk mailing list appeared [62] in which apparently the same botnet was discussed. We obtained samples of this malware and found the properties of the malware matched: just like described in the AMA thread, it was using a modified UnrealIRC 3.2.8.1 server[2] for one of the command

[2]Unfortunately, not the backdoored distribution by ac1db1tch3z

and control channels and included a Bitcoin miner. This was the first publicly documented instance of a botnet in the wild using Tor hidden services. While there had been a talk given at DEFCON in 2011 [21] about how hidden services could be used to protect botnets from takedowns of their command and control structure, previously no such malware had been observed.

Interestingly, not one but two hidden services were operated for command and control: the standard HTTP based channel[3] that ZeuS uses for command as well as an IRC based one[4]. Furthermore, the malware creates a hidden service (on port 55080) on each install, which allows the botnet operator to use the infected machine as a SOCKS proxy for TCP connections through the hidden service. While the hidden service is constantly running, the command to enable SOCKS proxy functionality needs to be given through the IRC command and control channel. In the version of the malware we analyzed, a Tor v0.2.2.35 binary was executed by injecting it into a svchost process.

In September 2012, G Data Security described a sample of apparently the same malware in a blog post [54]; a more thorough analysis of the botnet was published by Claudio Guarnieri of Rapid7 in December 2012 [55].

Black Markets on Hidden Services: A number of black markets exist on Tor hidden services. Silk Road is by far the most widely known, even triggering requests from U.S. senators to the U.S. Attorney General and the Drug Enforcement Agency (DEA) to request it to be shut down [72].

Silk Road is a market that operates mostly in contraband goods using Bitcoin as currency. According to [25] primarily narcotics and other controlled substances are sold on this platform. This study estimates the Silk Road revenue at over USD 1.9 million per month – aggregated over all sellers – with a 7.5% cut going to the Silk Road operators.

4.3.2 Controlling hidden service directories

As mentioned in chapter 2, the list of responsible hidden service directories depends on the current consensus document and the descriptor IDs of the hidden service. In this subsection, we explain how to inject relays into the Tor network that become responsible for the descriptors of the hidden service. This immediately translates into the problem of finding the right public keys, i.e. the keys with fingerprints which would be in-between the descriptor IDs of the hidden service and the fingerprint of the first responsible hidden service directory.

Figure 4.1 shows the distances between consecutive hidden service directories (in $\log_{10}$ scale) computed for a randomly picked consensus document in November 2012. The average value is 44.8 and the minimum value is 42.16. This means that we need to find a key with a fingerprint which would fall into an interval of size $10^{44.8}$ on the average. This takes just a few minutes on a modern multi-core computer.

[3]`mepogl2rljvj374e.onion:80`
[4]`eoqallfil766yox6.onion:16667`

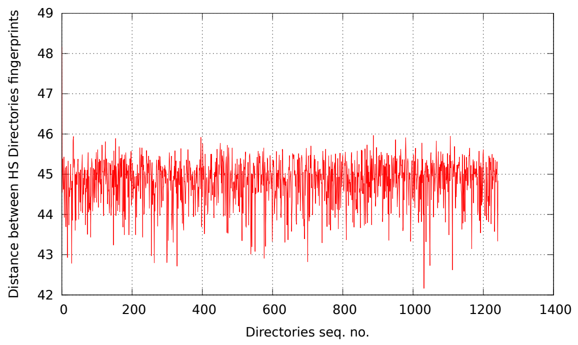


Figure 4.1: Distances between HS directories fingerprints, $\log_{10}$ scale

Just like any Tor client, an attacker is able to compute the descriptor IDs of the hidden service for any moment in the future and find the fingerprints of expected responsible HS directories. After that she can compute the private/public key pairs so that SHA-1 hash of the public keys would be in-between the descriptor ID and the fingerprint of the first responsible hidden service directory. The attacker then runs Tor relays with the computed public/private keys pairs and waits until they obtain HSDir flags (25 hours for Tor versions before 0.2.6.6, and 96 hours for later versions, i.e. after March 2015). When the attacker's relays appear in the consensus as hidden service directories, they will be used by the hidden service to upload the descriptors and by the clients to download the descriptors. In this way the attacker can gain control over all the responsible HS directories for a particular service by injecting 6 Tor relays with precomputed public keys. This allows her to censor a hidden service of her choice or gather its usage statistics.

As a proof of concept we used this approach to control one of the six hidden service directories of the discovered Tor botnet, the Silk Road hidden service, and the DuckDuckGo hidden service. We tracked these for several days and obtained the following measurements: (1) The number of requests for the hidden service descriptor per day (see Tables 4.1 and 4.2) and (2) the rate of requests over the course of a day, which is shown in Figure 4.2 (each point corresponds to the number of hidden service descriptor requests per one hour).

Column 1 of Table 4.1 and columns 2 and 4 of Table 4.2 show the number of requests for a particular hidden service descriptor per day. Columns "Total" show the total number of descriptors requests (for any hidden services descriptor) served by the hidden service directory per day. The hidden service tracked in Table 4.1 is the IRC C&C service.

Descriptors are cached by the Tor process in RAM for 24 hours. Hence, as long as a computer is not restarted, we will see at most one descriptor request every 24 hours, even if the long-lived circuit to the IRC server is repeatedly dropped; moreover suspending the computer will not cause the descriptor to be requested again. On the other hand, multiple power cycles per day lead to overcounting the size of the botnet. Hence, from Table 4.1 one can estimate that the size of the

Table 4.1: Popularity of the discovered botnet

Date	Botnet descriptor	Total
13 Jul 2012	1408	6581
14 Jul 2012	1609	2392
15 Jul 2012	1651	4715
16 Jul 2012	1448	6852
25 Jul 2012	4004	6591
26 Jul 2012	4243	4357
27 Jul 2012	4750	4985
28 Jul 2012	4880	7714
29 Jul 2012	4977	9085

Table 4.2: Popularity of Silk Road and DuckDuckGo

Date	Silk Road	Total	DuckDuckGo	Total
09 Nov 2012	19284	27363	502	2491
10 Nov 2012	15427	16103	549	5621
11 Nov 2012	15185	15785	543	3899
12 Nov 2012	15877	16723	549	10910

botnet was in the range 12,000 – 30,000 infected machines.

This is a very rough approximation since bots can request the descriptor several times per day, each time when the infected computer is turned on. By looking at the descriptor request rate against time we can infer that the bulk of the botnet resides in the European time-zone.

4.3.3 Efficient harvesting of Tor HS descriptors

It is of a particular interest to collect the descriptors of all hidden services deployed in Tor. We will show later how an attacker can use this collection to opportunis-

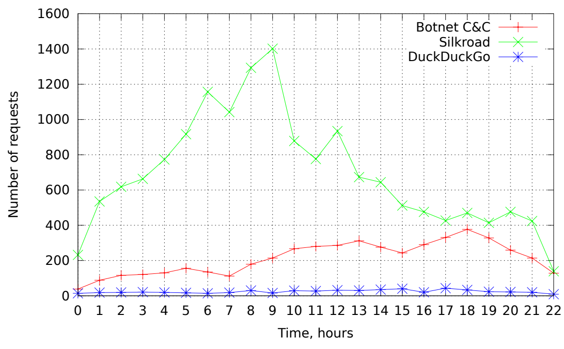


Figure 4.2: Hidden service descriptor request rate during one day (Summer 2012)

tically deanonymize any hidden service which chose one of the attacker's nodes as one of its entry guards. The IP addresses of these hidden services can be revealed in a matter of seconds using a traffic correlation attack, as we will show later. The technique for collecting descriptors is based on the shadowing technique from section 4.1. **Note that shadowing is not possible in the current versions of Tor (after version 0.2.4.10-alpha).**

It is clear that an attacker can operate several hidden service directories and collect hidden service descriptors over a long period of time. However, since there were more than 1200 hidden service directories on summer 2012 it can take the attacker significant amount of time to collect enough hidden service descriptors.

To collect the descriptors of all hidden services in a short period of time, a naïve attack requires to run many Tor relays from a non-negligible number of IP addresses. Assume that a hidden service descriptor's ID falls into some gap[5] on the fingerprint circle. The hidden service uploads its descriptor to the three hidden service directories with the next greater fingerprints. This means that each hidden services directory receives descriptors with identifiers falling into two gaps preceding the hidden service directory's fingerprint. This in turn means that the attacker needs to inject a hidden service directory into every second gap in the fingerprint circle to collect all hidden service descriptors. Thus she would need to run more than 600 Tor relays for 27 hours. This requires more than 300 IP addresses, given that the attacker is allowed to run only 2 Tor relays on a single IP address.

However, using the shadowing technique from section 4.1, we can collect the hidden service descriptors much more efficiently. In this subsection, we show how to reduce the number of IP addresses to approximately 50 (depending on the exact number of hidden service directories in the consensus). An attacker can rent 50 IP addresses and run 24 relays on each of them for 25 hours (for authorities running Tor version before 0.2.6.6) thus running 1200 Tor instances in total; 100 of them should appear in the consensus. The fingerprints of the public keys of the relays should fall into every second gap in the fingerprint circle. At the end of 25 hour time period all of the relays will have HSDir flags but only 100 of them will appear in the consensus and the rest will be shadow relays. The idea is to gradually make active relays unreachable to the Tor authorities so that shadow relays become active and thus gradually cover all gaps in the circular list during 24 hours.

It should be noted that the descriptor IDs of hidden services (and hence the responsible hidden service directories) change once per 24 hours and the time of the day when they change can be different for different hidden services. Since each hour the attacker covers only a fraction of the gaps on the fingerprint circle, the location of the descriptor can change from a gap not yet covered by the attacker to a gap already covered. Thus, if the attacker makes only one pass over the fingerprint circle during the day, she may not catch some descriptors. It will not happen if the attacker makes two passes during the day. Those descriptors location of which changed during the first pass to already covered gaps will be collected during the second pass (since they can change the location once per 24 hours only).

[5]A gap is defined to be an interval in the circular list of fingerprints between two consecutive HS directories

Another important point is that consensus document remains valid for a client for 3 hours, starting from its publication. According to the Tor implementation in summer 2012 clients can download the new consensus in (FU + 45 mins;VA - 10 mins) interval. Hence a hidden service can skip the consensus document which immediately follows its current consensus. This means the hidden service directories of the attacker should be in at least two consecutive consensus documents in order for the hidden service to learn about them.

Taken into account the aforementioned the attacker would need to control $R = \frac{N}{12*2}$ IP addresses, where N is the number of hidden service directories in the Tor network. Note that all the relays run by the attacker can be cheap since they do not have to provide high performance. Thus the attacker will have to pay only for the additional IP addresses and very little for the traffic. The IP addresses can be acquired from Amazon EC2 accounts. This results in a low-resource attack.

4.3.4 Experimental results

We performed the attack using 50 EC2 virtual instances on November 15th, 2012. During the experiment we received 59130 publication requests for different descriptor IDs. We also fetched the descriptors from the memory of running Tor instances and obtained 58389 descriptors in total[6]. Out of them there were 24703 descriptors with unique public keys. The fraction of encrypted descriptors among them was approximately 1.5%.

When computing onion addresses from the descriptors, we found the botnet C&C addresses, DuckDuckGo's hidden service and the Silk Road onion address in that set – as expected. However, we also found what looked like backup or phishing onion addresses for Silk Road, namely onion addresses with the same 8 letter prefix:

```
silkroadrlzm5thj.onion
silkroadvb5piz3r.onion
silkroadvlsu5apk.onion
silkroad5hq52m36.onion
```

Both `silkroadvlsu5apk.onion` and `silkroad5hq52m36.onion` redirected us to `silkroadvb5piz3r.onion` which is an onion-address for Silk Road that is publicly known. We were not able to connect to `silkroadrlzm5thj.onion`.

In order to verify the completeness of the harvested data we collected a sample of 120 running hidden services from public sources. Our data set missed 4 relays from this sample set. By extrapolating this result we conclude that we could have lost about 3% of hidden descriptors.

We launched a second experiment on February 4th, 2013 in order to reduce the costs of the attack. Because of the increased number of hidden services directories on that date, we used 58 EC2 instances. We also used an improved harvesting script: in addition to storing descriptors posted by hidden services we also initiated descriptors fetches from other responsible hidden services directories if a client's

[6]Note that we fetched the descriptors from memory 3 hours after the end of the experiment. This means that by that time some of our Tor relays removed a small portion of the descriptors from their memory.

request was received for an unknown descriptor. At the end of the experiment we collected 39824 unique onion addresses.

In order to reduce the experiments' costs we used the following. First both shadow and active relays had reported bandwidth of 0 Bytes/sec or 1 Bytes/sec. Since the granularity of the bandwidth values in the consensus is 1 kBytes/sec, all relays used in our attack were assigned bandwidth 0 kBytes/sec in the consensus. This means that the relays used in the attack should never be chosen by clients for purposes other than hidden services descriptors fetches. This has cut the traffic costs expenses. Secondly, we launched Tor relays participating in the harvesting from cheaper EC2 instances. In the second experiment, we used EC2 micro instances which is the cheapest option. In combination with reductions in traffic costs, this allowed us to reduce the overall price down to 57 USD.

Falling back to micro instances created performance problems however. Due to limited amount of RAM, at the end of the experiment, we could not establish SSH connections to some of EC2 instances and we had to reboot them to retrieve the data. The log files indicated that system clock jumped for several times which means that we could loose some hidden services descriptors.

This experiment had inadvertent but important side-effect on the flag calculation of Tor, of which we were notified by the Tor developers.

4.3.5 The Influence of Shadow Relays on the Flag Assignment

During the second harvesting experiment we accidentally revealed an important artifact of the flag assignment in Tor which is not obvious from the Tor specifications. Near the end of the experiment we were notified by the Tor developers that the Sybil attack had caused a spike in the number of relays assigned Fast flags and Guard flags (see Fig. 4.3)

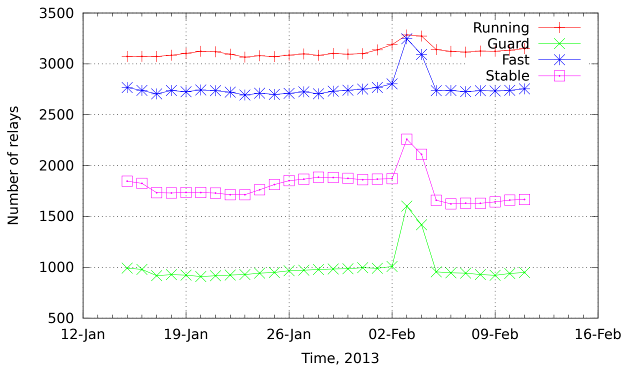


Figure 4.3: Increase in the number of Guard nodes

This happened because the shadow relays were taken into account for calculating medians of the bandwidth and the uptime. From these values, thresholds are derived that determine the flag assignment of all relays. According to the Tor specification (Fubruary 2013):

> ... A router is a possible Guard if its Weighted Fractional Uptime is at least the median for familiar active routers, and if its bandwidth is at least median or at least 250KB/s.
>
> To calculate weighted fractional uptime, compute the fraction of time that the router is up in any given day, weighting so that downtime and uptime in the past counts less.
>
> A node is familiar if 1/8 of all active nodes have appeared more recently than it...

In our second experiment, we caused the authorities to take into account 1392 shadow relays of bandwidth 0 Bytes/sec and 1 Bytes/sec. This significantly changed the medians for both bandwidth and uptime, which allowed many already running relays to get the Guard flag. During the harvesting experiment, this caused the number of Guard nodes to suddenly increase by 500.

The artifact has been patched in tor v0.2.4.10-alpha by ignoring Sybil relays when assigning flags. However, it is important to note that a more expensive version of the same Sybil attack is still possible. For example, an attacker could rent a large number of EC2 instances, running 2 Tor relays on each. This would enable attackers' Tor relays to decrease the value of the median for the Weighted Fractional Uptime as well as the bandwidth median, allowing to obtain the Guard flag for her relays much faster. For example, in order to inject 1200 Tor relays, an attacker would need to run 600 EC2 instances, spending only 288 USD per 24 hours.

4.4 Content and popularity analysis

In this section we analyze 39824 hidden service descriptors which we obtained during the harvesting experiments. We tested them for reachability, open ports and popularity. We classify the content of 1813 hidden services which were reachable at the time we crawled them and which contained more than 20 words of text.

4.4.1 Port scanning hidden services

We scanned the full collection of 39824 onion addresses for open ports at different times between 14 and 21 Feb 2013. At the time of the scans hidden service descriptors were available for 24511 addresses. In total, 22007 ports were found open on these. For hidden services for which descriptors were available, we obtained a coverage of 87% of all ports. The full coverage could not be achieved since we scanned different port ranges on different days and in a number of cases hidden services which we partially scanned on one day went off-line the day of the next scan; when scanning some hidden services we were persistently getting timeout errors.

During the scan we noticed that a large amount of hidden services did not have any open ports, however when scanned for port 55080 they returned an error message different from the usual error message. According to the Rapid7 blog post [55] port 55080 corresponds to hidden services created on computers infected by a

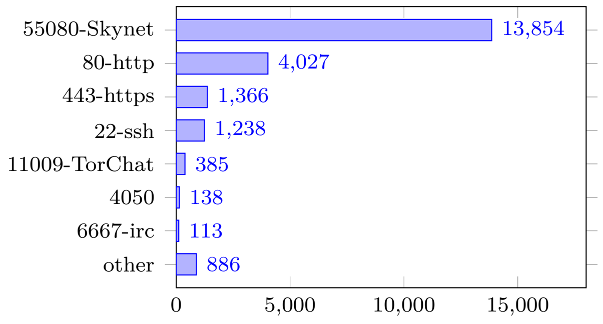


Figure 4.4: Open ports distribution

botnet malware called "Skynet". The observation is explained by the fact that the malware immediately closes any connection to this port unless it has been set up as a connection forwarder. We received such an "abnormal" error message for port number 55080 only and counted such events as open ports.

The open ports distribution is shown in Fig. 4.4. Port number 55080 is the most frequent one, found open on more than 50% of all onion addresses. This can be used to estimate the number of computers infected by "Skynet". HTTP and HTTPS services constitute 22% and SSH services are run by 5% of hidden services. Ports not shown on Fig. 4.4 have counts of less than 50 and are grouped together under "Other" label. In total we found 495 unique port numbers.

During our port scan we discovered that a number of hidden services provided HTTPS access. We discovered that in 1,225 cases the certificates were self signed and the certificates' common names did not match the requested host names. In 1,168 cases the certificate common name was `esjqyk2khizsy43i.onion` which is hosted at free onion hosting service "TorHost". We found 34 hidden services using certificates containing common names corresponding to their public DNS names, allowing for deanonymization of the service.

4.4.2 Content analysis

We analyzed the content of hidden services which provide HTTP(S) access: we classify them both according to topics and languages. We excluded port 55080 and tried to connect to the remaining 8,153 destinations (onion address:port pairs) using HTTP and HTTPS. We performed the crawl 2 months after the port scan[7]. At the time of the crawl, 7,114 ports were open – of these, we were able to connect to 6,579 using either HTTP or HTTPS. Table 4.3 shows the number of hidden services that offered HTTP or HTTPS services.

About half of the destinations were inappropriate for classification so we excluded them and ended up with 3050 destinations. In more detail we excluded: 2348 destinations which contained less then 20 words of text (this included 1092 messages from port 22 which were SSH banners); 1108 destinations at port 443

[7]We excluded all binary data such as images, executables, etc.

Port Num	**# of onion addresses**
80	3741
443	1289
22	1094
8080	4
Other	451

Table 4.3: HTTP and HTTPS access

which had corresponding copies at port 80; 73 destinations which returned an error message embedded in an HTML page.

For language detection we used "Langdetect" [66] software. The vast majority of the hidden services (84%) were in English. This is an expected result and corresponds to the statistics for the public Internet [19]. Overall we found hidden services in 17 different languages. Content was offered in the following languages besides English (each constituting less than 3%): German, Russian, Portuguese, Spanish, French, Polish, Japanese, Italian, Czech, Arabic, Dutch, Basque, Chinese, Hungarian, Bantu, Swedish.

We used the software "Mallet" [12] and the web service "uClassify" [85] for automatic topic classification. We considered only hidden services which offered pages in English (2,618 hidden services in total). Among them, 805 hidden services showed the default page of the Torhost.onion[8] free anonymous hosting service. We classified the remaining 1,813 onion addresses into 18 different categories.

Resources devoted to drugs, adult content, counterfeit (selling counterfeit products, stolen credit card numbers, hacked accounts, etc.), and weapons constitute 44%. The remaining 56% are devoted to a number of different topics: "Politics" and "Anonymity" are among the most popular (9% and 8% correspondingly). In the "Politics" category, one can find resources for reporting and discussing corruption, repressions, violations of human rights and freedom of speech, as well as leaked cables, and Wikileaks-like pages; the category "Anonymity" includes resources devoted to discussion of anonymity from both technical and political points of views as well as services which provide different anonymous services like anonymous mail or anonymous hosting.

The category "Services" includes pages which offer money laundering, escrow services, hiring a killer or a thief, etc. In "Games" one can find a chess server, lotteries, and poker servers which accept bitcoins. While making a preliminary analysis of the collected onion address names we noticed that 15 of them had prefix "silkroa" (two "official" addresses of the silkroad marketplace and the silkroad forum were among them). At least one of these addresses is a a phishing site imitating the real Silk Road login interface.

[8]torhostg5s7pa2sn.onion

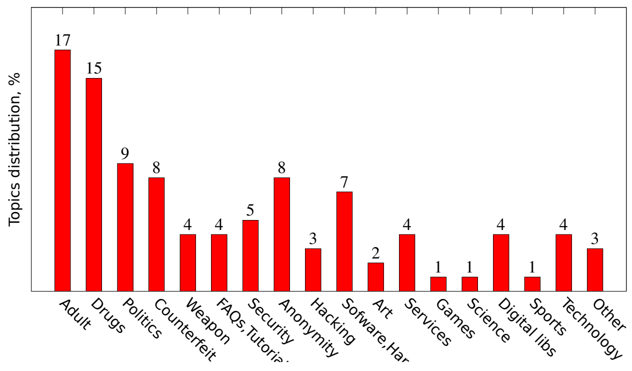


Figure 4.5: Tor Hidden services topics distribution, February 2013

4.4.3 Popularity measurement

In the previous sections we have analyzed the Tor hidden services landscape from the supply side. The analysis however will not be complete without estimating the popularity of different hidden services among clients. This becomes possible since the method we used to collect onion addresses also allows us to get the number of client requests for each of them in a 2 hour period. This can serve as an approximation of the popularity of hidden services.

During our experiments, we received a total of 1,031,176 requests for 29,123 unique descriptor IDs[9]. We used our database of onion addressees to resolve the descriptor ID requests. For each address in the list we computed corresponding descriptor IDs for each day between 28 January 2013 and 8 February in order to deal with possible wrong time settings of Tor clients. We compared this list of derived descriptor IDs with the list of client requests. In this way we resolved 6,113 descriptor IDs to 3,140 different onion addresses.

In order to explain the small fraction of resolved descriptor IDs, we ran several hidden service directories for a number of days. From the log files we could derive that 80% of the clients' requests were for non-existent descriptors (i.e. which were never published). Also only 10% of published descriptors were ever requested by clients. Given that the number of collected onion addresses is 39824, we believe that the small number of resolved descriptor IDs was caused by clients requesting descriptors which did not exist. We do not have a good explanation for this phenomenon (one explanation could be that specialized Hidden Service search engines were trying to connect to services from their databases which did not exist anymore), but this was a consistent behavior over several months.

[9]Remember that the descriptor ID is not equivalent to the onion address. While the onion address remains fixed, the descriptor ID changes every 24 hours and is derived from the onion address. Specifically, it is used to fetch hidden service's public key.

#	RQSTS	Addr	Desc	#	RQSTS	Addr	Desc
1	13714	`uecbcfgfofuwkcrd.onion`	Goldnet	22	899	`qdzjxwujdtxrjkrz.onion`	Skynet
2	11582	`arloppepzch53w3i.onion`	Goldnet	23	898	`6tkpktox73usm5vq.onion`	Skynet
3	11315	`pomyeasfnmtn544p.onion`	Goldnet	24	889	`kk2wajy64oip2***.onion`	Adult
4	7324	`lqqciuwa5yzxewc3.onion`	Goldnet	25	781	`gpt2u5hhaqvmnwhr.onion`	Skynet
5	7183	`eqlbyxrpd2wdjeig.onion`	Goldnet	26	746	`smouse2lbzrgeof4.onion`	<n/a>
6	6852	`onhiimfoqy4acjv4.onion`	<n/a>	27	694	`xqz3u5drneuzhaeo.onion`	FreedomHosting
7	6528	`saxtca3ktuhcyqx3.onion`	Goldnet	28	667	`f2ylgv2jochpzm4c.onion`	Skynet
8	4941	`qxc7mc24mj7m4e2o.onion`	<n/a>	29	585	`kdq2y44aaas2a***.onion`	Adult
9	3746	`mwjjmmahc4cjjlqp.onion`	BcMine	30	542	`4pms4sejqrryc***.onion`	Adult
10	3678	`mepogl2rljvj374e.onion`	Skynet	...	...	...	...
11	2573	`m3hjrfh4hlqc6***.onion`	Adult	34	453	`dkn255hz262ypmii.onion`	SilkRoad(wiki)
12	1950	`ua4ttfm47jt32igm.onion`	Skynet	...	...	...	...
13	1863	`opva2pilsncvt***.onion`	Adult	47	255	`dppmfxaacucguzpc.onion`	TorDir
14	1665	`nbo32el47o5cl***.onion`	Adult	...	...	...	...
15	1631	`firelol5skg6e***.onion`	Adult	62	172	`5onwnspjvuk7cwvk.onion`	BlckMrktReloaded
16	1481	`niazgxzlrbpevgvq.onion`	Skynet	...	...	...	...
17	1326	`owbm3sjqdnndmydf.onion`	Skynet	157	55	`3g2upl4pq6kufc4m.onion`	DuckDuckGo
18	1175	`silkroadvb5piz3r.onion`	Silk Road	...	...	...	...
19	1094	`candy4ci6id24***.onion`	Adult	250	30	`x7yxqg5v4j6yzhti.onion`	Onion Bookmarks
20	1021	`x3wyzqg6cfbqrwht.onion`	Skynet	...	...	...	...
21	942	`4njzp3wzi6leo772.onion`	Skynet	547	10	`torhostg5s7pa2sn.onion`	Tor Host

Table 4.4: Ranking of most popular hidden services, February 2013

Table 4.4 shows the number of requests for the most popular hidden services. We explored the five most popular addresses in more detail. Searching for them using the major search engines did not give any result – this already seemed quite strange for very popular hidden services. They only exposed port 80; connecting to them at this port returned 503 Server errors. As a next step, we tried to retrieve server-status pages, which succeeded. By analysing these pages we noticed that traffic to these servers remained constant at about 330 KBytes/sec and had about 10 client requests per second, almost exclusively POST requests. Looking at other hidden services we discovered another 4 onion addresses with the very same characteristics: they had port 80 open, they were returning 503 server errors, and had server-status page available. They also had similar traffic and client request rates. By looking at the uptime of the Apache server on the server-status pages we noticed that they could be divided into two groups with exactly same uptime within each group. From this we assumed that different hidden services lead to two physical servers. Given a huge number of requests, we made a conclusion that these hidden services belong to a very large botnet infrastructure (probably different from Skynet, we call it "Goldnet"[10]). It is also worthwhile to notice that 10 onion addresses of "Skynet" were also among the most popular hidden services (residing between 10th and 28th places).

The Skynet bitcoin pooling servers are the second most popular, just after the probable botnet. However their request rate is 4 time lower. Bitcoin mining servers are followed by resources offering adult content (there were 8 such resources among the 30 most popular hidden services). According to our results, the Silk Road market place is at 18th place with 1175 requests per 2 hours. Black Market Reloaded (another market for illegal goods) is at 62th place with 172 requests. With regard to the popularity of other hidden services, Freedom hosting is at 27th place with 694 requests, and the DuckDuckGo search engine is at 157th place with 55 requests. The public bitcoin mining pools Slush and Eligius had two and zero requests respectively.

4.5 Opportunistic deanonymisation of hidden services

The next step after getting a large collection of hidden service addresses, is to try to find their locations. The fact that an attacker always controls one side of the communication with a hidden service means that it is sufficient to sniff/control a guard of the hidden service in order to implement a traffic correlation attack and reveal the actual location of the hidden service. In particular, an attacker can:

- Given the onion address of a hidden service with unencrypted list of introduction points determine if her guard nodes are used by this hidden service.
- Determine the IP addresses of those hidden services that use the attacker's guard nodes.

[10] Since August 19, 2013 the Tor network experienced a huge increase in the number of new users. There was much speculation about the reason. On the 5th September 2013, Fox-IT published an analysis of a botnet malware which allegedly caused the spike. The onion addresses extracted from this malware coincided with "Goldnet".

- Determine if the attacker's guard nodes are used by any of the hidden services, even if the list of introduction points is encrypted.

4.5.1 Unencrypted descriptors

In order to confirm that an attacker controls a guard node of a hidden service she needs to control at least one Tor non-Exit relay. In the attack, the hidden service is forced to establishes rendezvous circuits to the rendezvous point (RP) controlled by the attacker. Upon receiving a `RELAY_COMMAND_RENDEZVOUS1` cell with the attacker's cookie, the RP generates traffic with a special signature. This signature can be identified by the attacker's middle node. We note that a special `PADDING` cell mechanism in Tor simplifies generation of a signature traffic which is discarded at the recipient side, and is thus unnoticeable to the hidden service. The steps of the attack are shown in Figure 4.6 and are as follows:

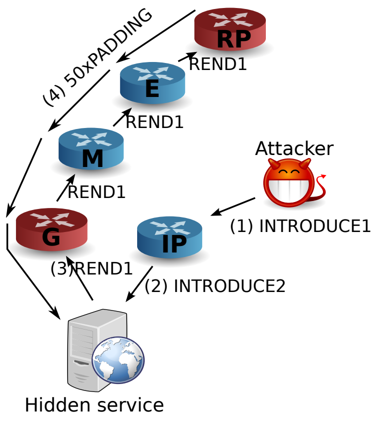


Figure 4.6: Revealing the guards

- The attacker sends a `RELAY_COMMAND_INTRODUCE1` cell to one of the hidden service's introduction points (IP) indicating the address of the rendezvous point.
- The introduction point forwards the content in a `RELAY_COMMAND_INTRODUCE2` cell to the hidden service.
- Upon receiving the `RELAY_COMMAND_INTRODUCE2` cell, the hidden service establishes a three-hop circuit to the indicated rendezvous point and sends it a `RELAY_COMMAND_RENDEZVOUS1` cell.
- When the rendezvous point controlled by the attacker receives the `RELAY_COMMAND_RENDEZVOUS1` cell, it sends 50 `PADDING` cells back along the rendezvous circuit which are then silently dropped by the hidden service.
- The rendezvous point sends a `DESTROY` cell down the rendezvous circuit leading to the closure of the circuit.

Whenever the rendezvous point receives a `RELAY_COMMAND_RENDEZVOUS1` with the same cookie as the attacker sent in the `RELAY_COMMAND_INTRODUCTION1` cell it logs the reception. At the same time, the attacker's guard node monitors the circuits passing through it. Whenever it receives a `DESTROY` cell over a circuit it checks:

1. whether the cell was received just after the rendezvous point received the `RELAY_COMMAND_RENDEZVOUS1` cell;

2. the number of the forwarded cells: 3 cells up the circuit and 53 cells down the circuit. Three cells more come from the fact that the hidden service established a circuit to the rendezvous point thus the attacker's guard node had to forward (2×`RELAY_COMMAND_EXTEND` + 1×`RENDEZVOUS1`) cells up and (2×`RELAY_COMMAND_EXTENDED` + 1×`DESTROY`) cells down. This is very important for our traffic signature since it allows us to distinguish the case when the attacker's node was chosen as the guard from the case when it was chosen as the middle.

If all the conditions are satisfied, the attacker decides that her guard node was chosen for the hidden service's rendezvous circuit and marks the previous node in the circuit as the origin of the hidden service.

In order to estimate the reliability of the traffic signature, we collected a statistics on the number of forwarded cells per circuit. We examined 748,846 circuits on our guard node. None of the circuits exhibited the traffic pattern of 3 cells up the circuit and 53 cells down the circuit. This means that the proposed traffic signature is highly reliable.

We implemented the approach to attack our own hidden service. We used a relay with a bandwidth of 500 Kbytes/s according to the consensus as the guard node and were scanning for the aforementioned traffic signature. For each received `RELAY_COMMAND_RENDEZVOUS1` cell we collected the corresponding traffic pattern and got no false positives.

4.5.2 Encrypted descriptors

If the list of introduction points is encrypted, an attacker will not be able to establish a connection to the hidden service. Hence the attack described in the previous section does not apply. However, we can use a different method to determine if some of those encrypted hidden services use a guard node controlled by us. We will not be able distinguish between hidden services with encrypted introduction points though. To achieve this goal we do the following:

- On our guard node we look for a traffic pattern characteristic for introduction circuits (we describe this traffic pattern and how unique it is later in this section).

- We discard introduction circuits which originate at the same IP address as any of the hidden services with unencrypted descriptors.

- For all remaining introduction circuits, we mark their origins as possible locations of an encrypted hidden services.

Let us describe the characteristics exhibited by introduction circuits: the main difference between general-purpose circuits and introduction circuits is their duration. A general Tor circuit stays alive either for ten minutes (if they were used by any stream), for one hour (if they did not carry any data traffic) or as long as any traffic is carried over them (this implies an open stream). In contrast, introduction circuits stay alive much longer, namely until some hop in the circuit fails or the hidden service closes the connection.

The second important difference is that after an introduction circuit is established, it does not transmit cells from the origin. On the other hand, general-purpose circuits usually transmit traffic back and forth.

Thirdly, we can use the fact that introduction circuits are always multi-hop while some general-purpose circuits are one-hop.

In order to check how good these filters are, we launched a hidden service which established two introduction circuits through a non-Guard relay controlled by us. By collecting the circuit statistics on this node for 24 hours we were able to identify our introduction circuits while having no false positives. We also did measurements on our Guard node during 24 hours and identified 14 potential introduction circuits. However, we did not check if they belonged to hidden services with unencrypted introduction points.

4.5.3 Success rate and pricing for targeted deanonymizations

In early 2012 we operated a Guard node that we rented from a large European hosting company (Server4You, product EcoServer Large X5) for EUR 45 (approx. USD 60) per month. Averaging over a month and taking the bandwidth weights into account we calculated that the probability for this node to be chosen as a Guard node was approximately 0.6% on average for each try a Tor client made that month. As each hidden service at the time of experiments chose three Guard nodes initially, we expect over 450 hidden services to have chosen this node as a Guard node[11]. Running these numbers for a targeted (non-opportunistic) version of the attack described in Section 4.5.1 shows us that by renting 23 servers of this same type would give us a chance of 13.8% for any of these servers to be chosen. This means that within 8 months, the probability to deanonymize a long-running hidden service by one of these servers becoming its Guard node is more than 90%, for a cost of EUR 8280 (approximately USD 11,000).

Take into account that this scales well: attacking multiple hidden services can be achieved for the same cost once the infrastructure is running.

4.5.4 Tracking clients

The technique for opportunistic deanonymisation of hidden service operators can be easily modified to opportunistically deanonymize HS clients.

[11]Assuming the number of hidden services at the time of experiments.

Assume that an attacker controls a responsible HS directory[12] of a hidden service. Whenever it receives a descriptor request for that hidden service, it sends it back encapsulated in a specific traffic signature which will be then forwarded to the client via its Guard node. With some probability, the client's Guard node is in the set of Guards controlled by the attacker. Whenever an attacker's Guard receives the traffic signature, it can immediately reveal the IP address of the client.

This attack has several important implications. Suppose that we can categorize users on Silk Road into buyers and sellers. Buyers visit Silk Road occasionally while sellers visit it periodically to update their product pages and check on orders. Thus, a seller tends to have a specific pattern which allows his identification. Catching even a small number of Silk Road sellers can seriously spoil Silk Road's reputation among other sellers.

As another application, one can collect IP addresses of clients of a popular hidden service and compute a map representing their geographical location. We have computed such a map for one of the "Goldnet" hidden services – in Figure 4.7.

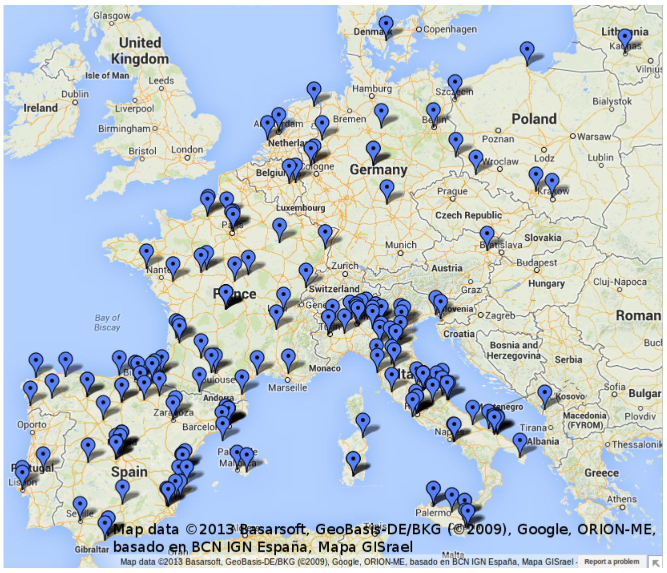

Map data ©2013 Basarsoft, GeoBasis-DE/BKG (©2009), Google, ORION-ME, basado en BCN IGN España, Mapa GISrael

Figure 4.7: Clients of a popular hidden service

4.6 Revealing Guard nodes of hidden services

In this section we present an attack to reveal the Guard nodes of a hidden service. Revealing the Guards does not immediately allow an attacker to reveal the location of the hidden service but gives her the next point of attack. This can be dangerous

[12] Remember that responsible hidden service directories of a hidden service are used to store the hidden service's descriptor (which contains its public key) for a period 24 hours. The HS directories are chosen among all Tor relays – different hidden services usually have different responsible HS directories.

for a long runing hidden service[13] as it gives an attacker sufficient amount of time either to take control over the Guard nodes or to start sniffing network traffic near the Guards. Given that Guard nodes are valid for more than a month, this may also be sufficient to mount a legal attack to recover traffic meta data for the Guard node, depending on the jurisdiction the Guard node is located in.

4.6.1 Unencrypted descriptors

We first describe the method to reveal Guards when the list of the introduction points in the HS descriptor is not encrypted. To do this, we use a technique similar to that presented in section 4.5; control over at least two Tor non-Exit relays is needed to carry it out. In the attack, the hidden service is forced to establishes many rendezvous connections to the rendezvous point (RP) controlled by the attacker in hope that some circuits pass through the second node (the middle node) controlled by the attacker. The RP generates traffic with a special signature which can be identified by the attacker's middle node. The steps of the attack are the same as in section 4.5.

Asymptotically, the probability that the attacker's middle node is chosen for the rendezvous circuit approaches 1. Whenever the rendezvous point receives a `RELAY_COMMAND_RENDEZVOUS1` with the same cookie as the attacker sent in the `RELAY_COMMAND_INTRODUCTION1` cell it logs the reception and the IP address of the immediate transmitter of the cell. At the same time, the attacker's middle node monitors the circuits passing through it. Whenever it receives a `DESTROY` cell over a circuit it checks:

1. whether the cell was received just after the rendezvous point received the `RELAY_COMMAND_RENDEZVOUS1` cell;

2. if the next node of the circuit at the middle node coincides with the previous node of the circuit at the rendezvous point;

3. whether the number of forwarded cells is exactly 2 cells up the circuit and 52 cells down the circuit.

If all the conditions are satisfied, the attacker decides that her middle node was chosen for the hidden service's rendezvous circuit and marks the previous node in the circuit as a potential Guard node of the hidden service.

We implemented the attack and ran it against two hidden services operated by us. In both cases the Guard nodes were identified correctly, without any false positives. In the first run with duration 1 hour 20 minutes, the rendezvous point received about 36 000 `RELAY_COMMAND_RENDEZVOUS1` cells in and the correct Guard nodes were identified 8, 6, and 5 times correspondingly. In the seconds case, the rendezvous point received 16 000 `RELAY_COMMAND_RENDEZVOUS1` cells in 40 minutes and the correct Guard nodes were identified 5, 2, and 1 times respectively.

[13]Silk Road's hidden service was running for almost two years.

We also used this approach to identify the Guard nodes of the botnet hidden service. Note that in the attack described in this section an attacker can use just one middle node and send the traffic signature as a client. However it requires building rendezvous circuits which makes the attack longer. The same applies to the attack presented in section 4.5.

4.6.2 Encrypted descriptors

If the list of introduction points of a hidden services is encrypted, it is not possible to make the hidden service establish rendezvous circuits (as was described in Section 4.6). In order to reveal the Guard nodes of a popular hidden service in this case, an attacker can use another attack. The condition for this attack is that the hidden service has many clients which establish long-lived (approx. 1-2 hours or longer) connections. This is the case of the botnet described in previous sections; connections to its IRC hidden service are long-lived.

Our measurements show that currently only a small fraction of all hidden services use encrypted descriptors. However we believe that this is an important case to study since encrypted descriptors offer significant additional protection and in the original draft of the Tor hidden services protocol all descriptors were supposed to be hidden.

Popularity of a hidden service, i.e. a large number of clients connecting to it, creates additional load on its Guards nodes. This changes the topological properties of the Guard nodes in terms of their degree[14] and in terms of the decay rate of persistent connections (in comparison to the case when the Guard nodes are not used by a popular hidden service). In particular:

- the degree of the Guard nodes of such a service will depend on the number of clients. The deviation of a node's degree from the expected value can serve as an indication of a popular hidden service;
- if clients make persistent connections to the hidden service (which is the case with botnet where IRC channel is used) the decay rate of the persistent connections of the HS's Guard nodes will look substantially different from that of other Guard nodes with similar bandwidth. We expect that the decay curve is much steeper in the case of normal Guard nodes.

In order to identify the Guard nodes of a popular hidden service we implement the following steps. We provide two analytical models for (1) the expected degree and (2) the expected persistent connections decay rate of a "normal" Guard node. We then use the scanning technique from Chapter 3 to determine the real degrees of the Guard nodes and their persistent connections decay rates. Finally, we compare the predicted values with those received from the measurements and single out nodes with too high degrees and too slow decay rates. The nodes we get are the candidates for the Guard nodes of the hidden service.

[14]The *degree* of a Tor relay denotes the number of TLS connections established between a given relay and other relays.

By *persistent connections decay rate* of a Tor relay we mean the following: assume that at time t_0 the relay has N TLS connections with other Tor relays. The decay rate of these connections is a function of time which shows how many of them remain connected at time t.

Expected degree of nodes in the Tor network graph

In order to derive the expected degree of a Tor relay we use results presented in Chapter 3 , Section 3.3. The probability of a TLS connection between two Tor relays A and B is computed as $p_0^{(A,B)} = e^{-\lambda_{AB} \backslash \mu}$, where $\lambda_{AB} = p_{A,B} \cdot R$ and $\mu = \frac{1}{t_{avg}+t_{idle}}$. $R = 900$ is estimated circuit arrival rate in the whole Tor network and $p_{A,B}$ is the probability of routers A and B to form an edge in a new circuit.

The expected number of open connections of a Tor relay at an arbitrary point of time is thus:

$$N_A^{avg} = \lceil \sum_{B \in T} p_0^{(A,B)} \rceil,$$

where T denotes the set of all Tor relays and $|T| = n$.

One can use either this model or compare a Tor relay's degree with the average (among relays of the same bandwidth) in order to find potential guard nodes of a hidden service. We used the technique described in Section 3.3 in order determine to which other relays a given Tor relay has established TLS connections. Figure 4.8 shows the degrees of Tor relays sorted by their bandwidth weight from the consensus.

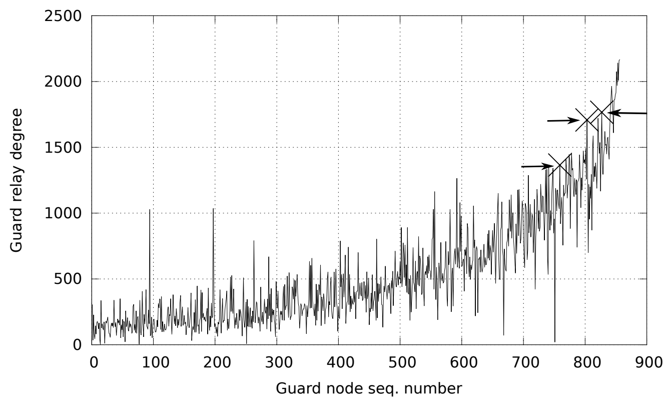


Figure 4.8: Degrees of Tor relays, Fall 2012

From this figure, one can see that there are a number of nodes which deviate significantly from the average – we call these *peak* nodes. The Guard nodes of the botnet which we determined in the previous section are marked by arrows and are among the peaks. This allows us to filter out quite many relays. However the number of peaks is still considerable. In the next section we show how to reduce the set of candidates of Guard nodes of a popular hidden service based on the persistent connection decay rate.

Decay rate of persistent connections

As mentioned in Section 3.3, for an average Tor relay the decay rate of persistent connections is steep during the first hours. This is not the case if a relay is a Guard node of a hidden service with persistently connected clients, such as the botnet's IRC command and control. In this case the decay rate will be determined by the bots going offline rather than by the bandwidth of the node.

In order to predict the decay rate of a "normal" Tor relay we use the following approach: we first find the expression for the duration of a connection between relay A and B and use it to determine the connection decay rate. We assume the following: 1) circuits arrive to the connections according to Poisson distribution (see Section 3.3); 2) the circuit arrival rate is proportional to the bandwidth of the relay; 3) the circuit duration follows an exponential distribution. Given these assumptions, we adopt a finite state Markov chain to model the connection duration. Each state of the Markov chain represents the number of circuits carried over the connection. The chain has one absorbing state 0. We are interested in the extinction time. The number of states is finite.

We assume that at the time when we observe the connections, the system is in quasistationary state, conditioned that the extinction has not occurred. Thus the initial state distribution is a quasi-stationary distribution which always exists for finite state case (see [1] ,[3]). Classical matrix theory can be used to show that a matrix containing infinitesimal transition probabilities of transient states has a dominant eigenvalue such that the corresponding left and right eigenvectors have positive entries (see [1], and [3]); the left eigenvector is the quasistationary distribution. We denote $(q_1, q_2, ..., q_N)$ as the row vector of quasistationary probabilities.

Let λ be the circuit arrival rate to a connection between two Tor relays and μ the circuit closing rate. In this case, the matrix of infinitesimal transition probabilities is:

$$\mathbf{R} = \begin{bmatrix} 0 & \mathbf{0} \\ \mathbf{a} & \mathbf{C} \end{bmatrix}, \tag{4.1}$$

where the matrix $\mathbf{C}$ corresponds to transient states $T = \{1, 2, ..N\}$ and state 0 is absorbing. The matrix $\mathbf{C}$ can be written as:

$$\begin{pmatrix} -\lambda-\mu & \lambda & 0 & 0 & \cdots & 0 & 0 \\ \mu & -\lambda-\mu & \lambda & 0 & \cdots & 0 & 0 \\ 0 & \mu & -\lambda-\mu & \lambda & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \mu & -\mu \end{pmatrix} \tag{4.2}$$

The probability of extinction of a connection between relays A and B in this case can then be derived as (we use Kolmogorov forward equations to get this result):

$$p_0^{AB}(t) = 1 - e^{-\mu q_1 t}$$

The circuit arrival rate $\lambda_{a,b}$ is computed as in the previous subsection. We set the circuit closing rate as $\mu = 1/(t_{avg} + t_{idle})$, where t_{avg} is the average duration of a circuit as we saw in Section 3.3 and $t_{idle} = 180$ seconds is the time before an idle connection would close. As we also observed in Section 3.3, t_{avg} depends only slightly on the pair of relays and is close to 200 seconds.

One can use numerical methods (see [9] for example) to compute the eigenvalues and eigenvectors. Note that for the cases when $\lambda_{a,b} < \mu$, one can approximate the values with an expression for infinite state Markov chain [2]. In the infinite case, the quasistationary probability for the system to be in state j is:

$$q_j = (1 - \beta)^2 j \beta^{j-1},$$

where $\beta = \sqrt{\frac{\lambda}{\mu}}$. Particularly,

$$q_1 = (1 - \beta)^2.$$

We apply this model to the pair of medium-bandwidth Tor relays for which the experimental data can be found in Section 3.3. The consensus bandwidths of the relays were 1850 kBytes/sec and 4280 kBytes/sec. Both were Guard and non-Exit nodes. The comparison between the model and the data obtained from the direct measurements on one of the nodes is shown in Figure 4.9.

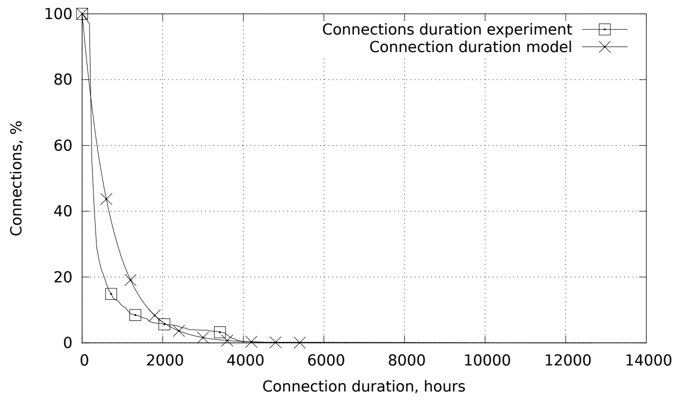


Figure 4.9: Connection duration model validation

Given the initial connections of a relay, we use the model for connection duration to compute the expected number of persistent connections at an arbitrary point of time for Tor relay A:

$$N_A^{pers}(t) = \sum_{B \in I} (1 - p_0^{AB}(t)),$$

where I is the set of initial connections of A.

Using the canonical scanning technique we obtained the persistent connections decay rate of the Tor guard nodes. We compared them with the connection decay rate predicted by the model and filtered those connections which differ from the model. Particularly, we compared the number of persistent connections after 3 hours of scanning. Out of 856 nodes 200 had a degree that exceeded the value predicted by the model. Choosing a threshold such that guard of the botnet's hidden service is included, we find that 37 nodes have a degree that is 1.4 time higher than the value predicted by the model.

Figure 4.10 shows the real decay rate of the botnet's guard nodes plotted against the theoretically predicted one. As one can see, the discrepancy is quite detectable.

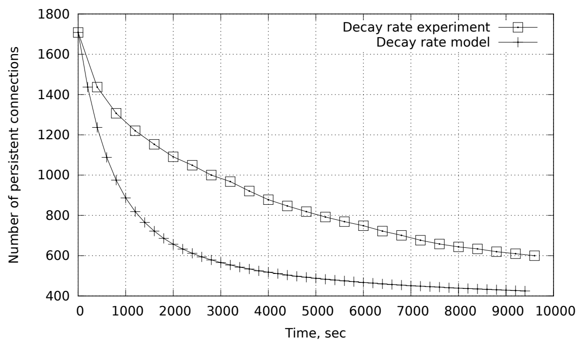


Figure 4.10: Decay rate of the botnet's guard 1

Figure 4.11 shows the observed decay rate and the predicted decay rate of another node with a degree above the average (one of the peaks in Figure 4.8). The majority of peaks from the previous section have this type of the decay rate, which is close to the theoretical predictions. This allows us to reduce the number of candidates to 29. Since we know the actual guard nodes of the botnet's hidden service from the unencrypted descriptor attack, we were able to check that indeed the correct guards appeared in this list of candidates.

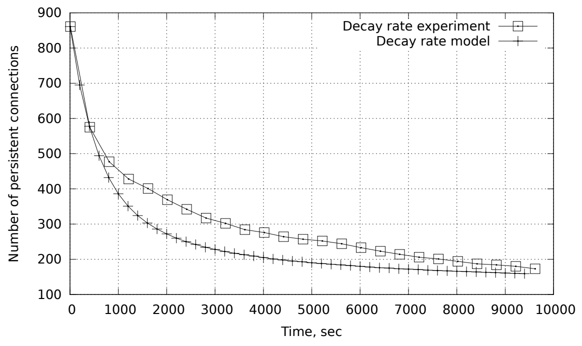


Figure 4.11: Common shape of the decay rate

4.7 Potential countermeasures

In this section we briefly describe several possible countermeasures. Harvesting can be easily prevented by making the `descriptor-cookie` authentication [20] mandatory for all hidden services and base32 encoding the value as part of the URL together with the `permanent-id`. The downside of this change is a significantly reduced usability: instead of 16 character onion addresses the user now has to deal with onion-addresses that are 42 characters long.

In order to prevent adversaries from efficiently targeting hidden service directories, an unpredictable value can be derived by the directory authorities each hour from a shared secret. Three of these values are included in the consensus – one for each of the hours the consensus is valid. The unpredictable value valid for the hour of the request is then included in the calculation of the descriptor ID and henceforth determines the place on the ring where the descriptor is stored. This makes it impossible for an attacker to precompute identity keys for time periods further ahead than 3 hours in the future.

To prevent the guard nodes being revealed, one can use an additional layer of guard nodes – guard middle nodes. This countermeasure has already been proposed in [16] but is not implemented in Tor. Note that this measure will not protect against an attacker exploiting degree anomalies of the guard nodes as described in Section 4.6.2.

A work by Tariq et al.[26] suggests that the guards compromise rate can be decreased by (1) making the guard rotation interval longer and (2) by taking into account how long nodes have been part of the network when assigning Guard flags to them. Note that this approach if not carefully implemented has a number of downsides like reduced end-user quality of experience and malicious nodes accumulating Tor users. In regard to revealing the introduction circuits, if the attacker will not be able to collect the full list of hidden service descriptors, she will not be able to distinguish between introduction circuit of hidden services with encrypted introduction points and non-encrypted.

Chapter 5

Anonymity Analysis of Bitcoin P2P Network

Bitcoin protocol has two well known properties. First, the entire Bitcoin transaction history is publicly available so anyone can see how Bitcoins travel from one pseudonym to another and potentially link different pseudonyms of the same user together [28]. This alone however does not allow mapping pseudonyms to real identities. Second, Bitcoin does not use encryption and studying the entire IP traffic of the Bitcoin peers could potentially reveal the origins of many transaction. As Bitcoin users come from different jurisdictions this type of attack is only available to a very powerful adversary which is able to conduct global-scale surveillance.

By contrast in this chapter we study the level of anonymity provided by Bitcoin against a non-global off-path adversary, i.e. an ordinary attacker with a few machines and without ability connect to clients behind NAT. We first describe a generic low-resource off-path method to deanonymize a significant fraction of Bitcoin users and correlate their pseudonyms with public IP addresses. The method explicitly targets clients (i.e. peers behind NAT or firewalls) and can differentiate nodes with the same public IP.

Second, we study an important case of using Bitcoin over Tor. We show that sending Bitcoin traffic through Tor not only provides limited level of anonymity but also exposes the user to man-in-the middle attacks in which an attacker controls which Bitcoin blocks and transactions the user is aware of.

Third, we show that Bitcoin peer discovery protocol allows an attacker to set an “address cookie” on users’ computers. This can be used to correlate the same user across different sessions, even if he uses Tor, hidden-services or multiple proxies. If the user later decides to connect to the Bitcoin network directly the cookie would be still present and would reveal his IP address. A small set of Sybil nodes (about a 100 attacker’s nodes) is sufficient to keep the cookies fresh on all the Bitcoin peers (including clients behind NATs).

Contents

5.1 Deanonymization of client in Bitcoin P2P network

A traditional approach for deanonymization would be to have a list of suspicious Bitcoin pseudonyms first and then try to find IP addresses of their owners. By contrast in the technique described in this section, we go other way round. We first collect a list of IP addresses of Bitcoin clients[1], and then for a large number of transactions we determine at which client it was generated.

The crucial idea behind the deanonymization attack is that each client can be uniquely identified by a set of nodes he connects to (entry nodes). We show that this set can be learned at the time of connection and then used to identify the origin of a transaction.

5.1.1 Learning entry nodes

We start by recalling terminology and protocol rules (see Chapter 2, section 2.2.3). We call peers that accept incoming connections *servers* and peers behind NAT's or firewalls *clients*. In order to connect to the Bitcoin network a client c needs to establish connections to Bitcoin servers. We call eight servers $E = \{e_1, \ldots, e_8\}$ through which a client connects to the Bitcoin network *entry nodes* of this client. Remember that each time c connects to e_i, it sends it its IP address[2] IP_c. The entry node then forwards the address to two randomly chosen neighbors (we call them *responsible nodes*).

[1]Each client advertises its IP address to the network according to the Bitcoin peer discovery protocol.

[2]As it is seen from the Internet.

Let us see how an attacker can learn entry nodes E of client c. The key observation is that if the attacker is connected to e_i in advance, then with some probability[3] e_i will forward IP_c to the attacker (once client c connects to e_i). This suggests the following strategy:

1. Connect to W Bitcoin servers, where W is close to the total number of servers, and listen to `ADDR` messages.

2. Each time address IP_c is received, add the sender to set E' of potential entry nodes.

In general $E' \neq E$, and this is due to the following reason. When an entry node forwards IP_c to two responsible nodes, with some probability these responsible nodes will not belong to the attacker, in which case the address will bounce in the network and will arrive at the attacker from some random (noise) node.

Noise reduction technique Our strategy of filtering noise assumes that either the client's IP was already used in the Bitcoin network, which is quite common for the clients behind NAT or the client's public IP is contained in a known list of IP addresses (e.g. within an IP range of a major ISP) which an attacker can use. If an attacker knows IP_c, he restricts its propagation using the following fact (Bitcoin core v0.9.2): *if the address had already been sent from peer c_1 to peer c_2, it will not be forwarded over this connection again.*

This suggests broadcasting IP_c (or all the addresses under investigation) to all Bitcoin servers in advance. We suggest repeating this procedure every 10 minutes, though there could be other options. This will block future propagation of IP_c via all existing connections. The attacker then needs to re-establish its connections to the Bitcoin servers (so that her connections are "unblocked").

Eventually the attacker learns a subset $E' \subseteq E$ of client's entry nodes. The exact value of $p_{addr} = \frac{|E'|}{|E|}$ depends on the number of attacker's connections, and it is computed for some parameters in Section 5.1.4. For instance, if an attacker establishes 35 connections to each potential entry node, which all had 90 connections beforehand, then he identifies 4 entry nodes out of 8 on average.

5.1.2 Deanonymizing clients

We have shown how to find entry nodes of a client. Now we describe the main phase of the deanonymization attack which consists of four steps:

1. Getting the list $\mathbf{S}$ of (almost) all Bitcoin servers. This list is regularly refreshed.

2. Composing a list $\mathbf{A}$ of IP addresses of Bitcoin clients $\mathbf{C}$.

3. Learning entry nodes of clients from $\mathbf{A}$ when they connect to the network.

[3]This probability depends on the number of the attacker's connections to e_i.

4. Listening to servers from $\mathbf{S}$ and mapping transactions to entry nodes and then to clients.

Eventually for each client $c \in \mathbf{C}$ we create a tuple $(E'_c, IP_c, \mathbf{\Pi}_c)$, where IP_c is the IP address of the client or its ISP, entry nodes E'_c distinguish clients sharing the same IP, and $\mathbf{\Pi}_c$ is the list of client's transactions. Extracting Bitcoin pseudonyms from transactions is straightforward. We now explain each step in detail.

Step 1. Getting the list of servers. This phase of the attack is rather straightforward. An attacker first collects the entire list of peers by querying all known peers with a `GETADDR` message. Each address IP in the response `ADDR` message can be checked if it is online by establishing a TCP connection and sending a `VERSION` message. If it is, IP is designated as a server. An attacker can initiate the procedure by querying a small set of seed nodes and continue by querying the newly received IP addresses. The adversary establishes m connections to each server (we suggest 50 for the size of the current Bitcoin network).

Step 2. Composing the list of Bitcoin clients. The attacker builds set $\mathbf{A}$ of IP addresses of clients whose Bitcoin pseudonyms she wants to reveal. The addresses may come from various sources. The attacker might take IP ranges of major Internet service providers, or collect addresses already advertised in the Bitcoin network. Finally, she might take some entries from the list of peers she obtained at Step 1.

Step 3. Mapping clients to their entry nodes. Now the attacker identifies entry nodes of clients that connect to the Bitcoin network. Equipped with the list $\mathbf{A}$ of addresses, the attacker runs the procedure described in Section 5.1.1. Let us estimate how many entry nodes are needed to uniquely identify a client.

For each c advertising its address in the network the attacker obtains a set $E'_c \subseteq E_c$. Since there are about $8 \cdot 10^3$ possible entry nodes out of 10^5 total peers (servers and clients together), the collisions in E'_c are unlikely if every tuple has at least 3 entry nodes:

$$\frac{10^5 \cdot 10^5}{(8 \cdot 10^3)^3} \ll 1.$$

Therefore, 3 entry nodes uniquely identify a user, though two nodes also do this for a large percent of users. We stress that it is likely that $E_{c_1} \neq E_{c_2}$ even if c_1 and c_2 share the same IP address. An attacker adds E_c to its database and proceeds to Step 4.

Step 4. Mapping transactions to entry nodes. This step runs in parallel to steps 1-3. Now the attacker tries to correlate transactions appearing in the network with sets E'_c obtained in step 2. The attacker listens for `INVENTORY` messages with transaction hashes received over all the connections that she established and for each transaction T she collects R_T — the first q addresses of Bitcoin servers that forwarded the `INVENTORY` message. She then compares E'_c with R_T (see details below), and if they match the attacker decides that transaction T belongs to client c. In our experiments we take $q = 10$.

There could be many variants for the matching procedure, and we suggest the following version.

- The attacker composes all possible 3-tuples from all sets E'_c and looks for their appearances in R_T. If there is a match, he gets a pair (c, T);
- If there is no match, the attacker consider 2-tuples and then 1-tuples. Several pairs $\{(c_i, T)\}$ can be suggested at this stage, but we can filter them with later transactions.

We made several experiments and collected some statistics to estimate the success of the attack. In our experiments on the testnet (see Chapter 2, section 2.2.5) we established 50 connections to each server, obtained 6 out of 8 entry nodes on average, and the 3-tuples were detected and linked to the client in 60% of transactions (Section 5.1.3). In the real network, where we can establish fewer connections on average, our pessimistic estimate is 11% (Section 6.2), i.e. we identify 11% of transactions.

Finally, let us consider the approach where we identify clients by 2-tuples in the top-10. As detailed in Section 6.2, for 35% of transactions the right client would be identified. However, each transaction might generate several false positives.

To estimate the false positive rate, we first calculate the average number of 2-tuples among the entry nodes we catch. For $p_{addr} = 0.34$ each 2-tuple is detected with probability $p_{addr}^2 = 0.115$, so out of 28 possible 2-tuples we detect 3.2 on average. Each top-10 suggests 45 2-tuples, and given 8000 servers at the time of experiments (May 2014) there are $\binom{8000}{2} \approx 2^{25}$ 2-tuples at all (all tuples are unordered). If we work with a database of N clients, each transaction suggests $\frac{N \cdot 45 \cdot 3.2}{2^{25}} \approx N \cdot 2^{7.3-25} = N/2^{17.7}$ candidate clients. If we track all 100,000 clients, we get the false positive rate around 0.28, which is slightly smaller than the probability 0.35 to detect the right client for a transaction. In other words, for each suggested client the probability that he is the right one is about 55%.

Remark 1. Step 4 of the attack depends on that some entry nodes of a client are among the first to forward the `INVENTORY` message with the transaction's hash. The intuition behind it is that it takes a number of steps for a transaction to propagate to the next hop. Fig. 5.1 shows steps that are required for a transaction to be propagated over two hops and received at peer `A`. When a transaction is received by a node it first runs a number of checks and then schedules the transmission. The actual transmission will happen either immediately (for 25% of transations) or with a random delay due to trickling (see Section 2.2.3). The time needed for an `INVENTORY` message to be forwarded to the attacker's node through node `Entry` is the sum of propagation delays of 4 messages (2x`INVENTORY`, 1x`GETDATA`, 1x`TRANSACTION`) plus the time node `Entry` needs to run 16 checks and possibly a random trickling delay. On the other hand the time needed for the same `INVENTORY` message to be forwarded to the attacker's node through peer `A` consists of 7 messages

(3xINVENTORY, 2xGETDATA, 2xTRANSACTION), 32 checks, and two random delays due to trickling. Finally since the majority of connections to a peer are coming from clients, one more hop should be passed before the transaction reaches an attacker's node through a wrong server. Measurements of transaction propagation delays are given in Section 6.2.

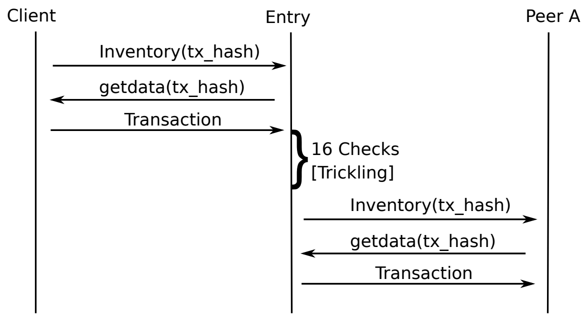


Figure 5.1: Steps necessary to forward a transaction

Based on this we expect that if a transaction generated by a client is forwarded to the entry nodes immediately, the entry nodes will be the first nodes to forward the transaction. In case when the transcation was sent sequentially with 100 ms between transmissions we still expect a fraction of entry nodes to be among the first 10 to forward corresponding INVENTORY message to one of the attacker's nodes. This fraction obviously depends on the propagation delay between Bitcoin peers. The higher the propagation delay the less significant becomes delay of 100 ms in trickling. For example if the propagation delay is 300 ms between the client and each entry node it's likely that 3 entry nodes will be among the first to forward the INVENTORY message (given that the attacker has enough connections to Bitcoin servers).

Remark 2. The attack presented in this section requires from an attacker only to be able to keep a significant number of connections to Bitcoin servers without sending large amount data. In order to make the attack less detectable an attacker might decide to establish connection to a given Bitcoin server from different IP addresses, so that all connection look like they came from different unrelated clients. The same set of IP addresses can be used for different servers.

Remark 3. The technique considered in the section provides unique identification of Bitcoin clients for the duration of a session, and thus if a client makes multiple transactions during one session they can be linked together with very high probability. Note that this is done even if the client uses totally unrelated public keys/Bitcoin wallets, which have no relation in the Bitcoin transaction graph and thus such linkage would be totally unachievable via transaction graph analysis [28, 29]. Moreover we can easily distinguish all the different clients even if they come from the same ISPs, hidden behind the same NAT or firewall address.

5.1.3 Experimental results

As a proof of concept we implemented and tested our attack on the Bitcoin testnet. For our experiments we built our own Bitcoin client, which included functionality specific for our attack – sending specific Bitcoin messages upon request or establishing various numbers of parallel connections to the same Bitcoin server, etc. When imitating clients we used the main Bitcoin client. In order to periodically get the list of all running Bitcoin servers we used an open source crawler [39].

For the time of experiments (May 2014) the number of running Bitcoin servers in the testnet fluctuated between 230 and 250, while the estimated average degree of the nodes was approximately 30. In our experiments we were imitating several different users connecting to the testnet from the same ISP's IP address and from different ISP's at different times. As an attacker we added 50 additional connections to each Bitcoin server. For each experiment in the first phase of the attack we propagated clients' addresses in the testnet 10 minutes before they started to send their transactions. In total we (as clients) sent 424 transactions.

In the first experiment we confirm our expectations that transactions are first forwarded by entry nodes and analyse the number of entry nodes that were among the first 10 to forward the transactions (i.e. we assume that the attacker correctly identified all entry nodes). We split all transactions into two sets: the first set contains 104 transactions, which were forwarded to the entry nodes immediately; the second set contains all other 320 transactions (i.e. for which trickling was used). Fig. 5.2 shows the number of entry nodes that were among the first 10 to forward the transaction to the attacker's nodes for these two sets. As expected if a transaction was immediately forwarded to all entry nodes the attacker was able to "catch" three or more of them in 99% of cases. In case of transactions from the second set, the attacker was able to "catch" 3 or more entry nodes in 70% of cases. We also observed that for the majority of transactions the first two nodes to forward the transaction to the attacker were the entry nodes.

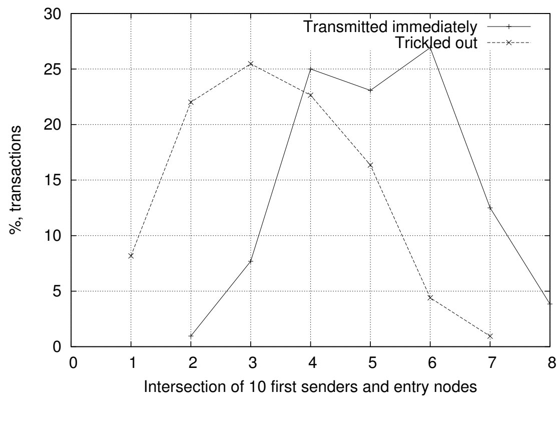


Figure 5.2: Intersection of top-10 senders and entry nodes

In the second experiment we executed all steps of the attack. In our experiment each client was successfully uniquely identified by his own set of entry nodes and on average we identified 6 entry nodes for each client. Assuming that 3 entry nodes is enough for unique identification of a client we correctly linked 59.9% of all transactions to the corresponding IP address by matching entry nodes of clients and first 10 Bitcoin servers which forwarded the transaction. We correctly glued together all transactions of the same client which were made during one session. In a bit more conservative setting we added only 20 additional nodes in which case we successfully deanonymized 41% of our transactions.

5.1.4 Analysis

The success rate of the attack presented above depends on a number of parameters, among which the most important is the fraction of attacker's connections among all the connections of client's entry nodes. The fewer the number of connections of entry nodes are, the more connections the attacker can establish and the higher chance is to deanonymise the client. In this section we analyze each step of the attack and compute success rates for some parameter sets.

Number of connections to servers

Both mapping client to entry nodes and mapping entry nodes to transactions depends on the number of connections the attacker can establish to the Bitcoin servers. Assuming the entry node had n connections and the attacker added m new connections, thus the total number of connections is $N = n + m$, the probability to receive the address at the first hop is $p_{addr}(n, N) = 1 - \frac{n}{N} \cdot \frac{n-1}{N-1}$. For a transaction which was not forwarded immediately to the peer's neighbours the probability that one of attacker's nodes is chosen as trickle node in the first round is $p_{tx} = \frac{m}{N}$. For $n = 50$, $m = 50$, $p_{addr} = 0.75$ and $p_{tx} = 0.50$. For $n = 90$, $m = 35$, $p_{addr} = 0.49$ and $p_{tx} = 0.28$. The number of connections that the adversary can establish to a server is limited by the total number of 125 connections a Bitcoin peer can have by default.

In order to see how many open connection slots Bitcoin peers have we conducted the following experiment in April 2014. For each Bitcoin server that we found we tried to establish 50 parallel connections and check the actual number of established connections[4]. Fig. 5.3 shows the distribution of number of established connections.

The experiment shows that 60% of peers allow 50 connections or more, and 80% of Bitcoin peers allowed up to 40 connections. Note that even if sufficient number of connection cannot be established to a Bitcoin peer immediately they can be established in longer term since many Bitcoin clients will eventually disconnect and thus allow new connections (according to an example disconnection rate as shown in Fig. 5.5 it might take several hours, but once an attacker got the required number of connections she can keep them as long as needed). Also note that Bitcoin servers allow any number of connections from a single IP address.

[4]We did not try establish more than 50 connections in order not to degrade the Bitcoin network performance.

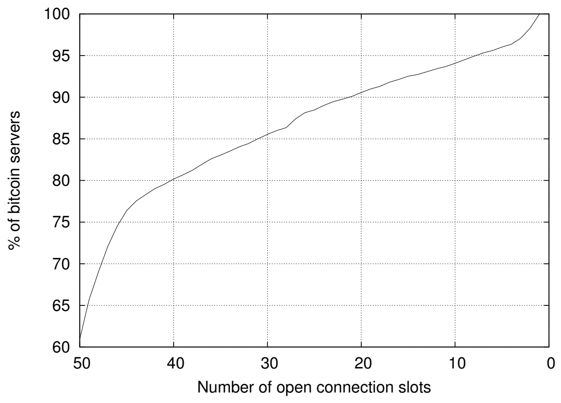


Figure 5.3: Distribution of open slots

Finally the attacker does not send much traffic over the established connections but rather listens for messages. Incoming traffic is normally free of charge if one rents a server. Thus in spite of the large number of connections that an attacker needs to establish the attack remains very cheap.

Estimating false positives

Assume that some of the steps of that attack fail. Then the first 10 peers to report the transaction to the attacker will be some random Bitcoin peers. If there is no 3-subset of these 10 that match some entry node set, then such a transaction is marked as unrecognized by an attacker. The chance that the intersection between these 10 random nodes and 8 entry nodes of some client is 3:

$$p_c = \frac{\binom{8}{3} \cdot \binom{N_s-8}{10-3}}{\binom{N_s}{10}}$$

Given that there are about $N_s = 8000$ Bitcoin servers and 100,000 Bitcoin clients, the number of incorrectly assigned transactions is negligible.

We now estimate the probability that an attacker adds a wrong entry node to the set of entry nodes of a particular client (we recall that according to the address propagation mechanism after receiving an address a peer forwards it to only two randomly chosen *responsible nodes*). For this to happen, one or more entry nodes should forward the client's address IP_c over one of non-attacker's connections, whence (since the attacker periodically propagates the client's address) at least one of responsible nodes for address IP_c should change on an entry node after the attacker last propagated IP_c.

In order to estimate this probability we collected statistics from our Bitcoin peer for 60 days from March 10 till May 10 2014. We collected information about 61,395 connections in total. Assume that the attacker propagated IP_c at time t_0, the probability that a responsible node will be different at time $t_1 = t_0 + \Delta t$ depends on the number of new connections the entry node has at t_1 and number of nodes that disconnected since t_0. Fig. 5.4 shows probability density function of the number of new connections (i.e. the incoming connections rate) for different values of Δt.

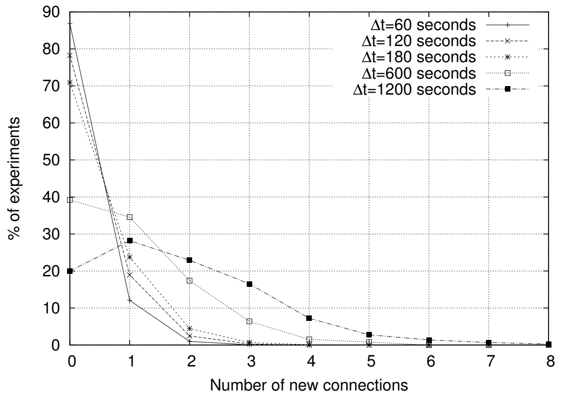


Figure 5.4: Probability density of number new connections

Fig. 5.5 shows probability density function of the number of disconnection (i.e. connection close rate) for different values of Δt.

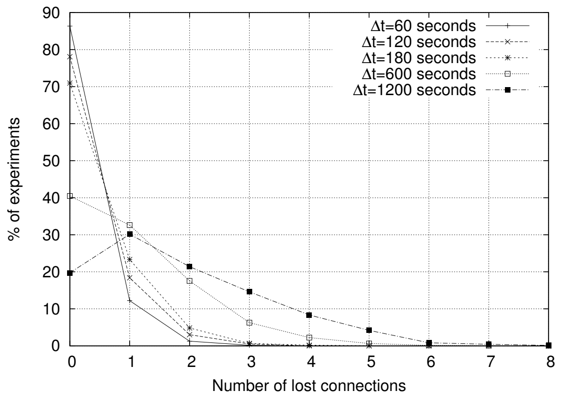


Figure 5.5: Probability density of number lost connections

We use these distributions to simulate the address propagation and calculate the probability that the client's address is forwarded by an entry node over a non-attacker's link after time Δt after the attacker sent this address over the network. We obtained probabilities for different number of attacker's and non-attacker's connections and for each connection setting and each Δt we executed 10,000 runs of the model. Fig. 5.6 shows the obtained probabilities. The number of attacker's connections is denoted by m and the number of non-attacker's connections by n.

As expected, the more connection a node has the less probable that the responsible nodes for an address will change after Δt. Another observation is that the

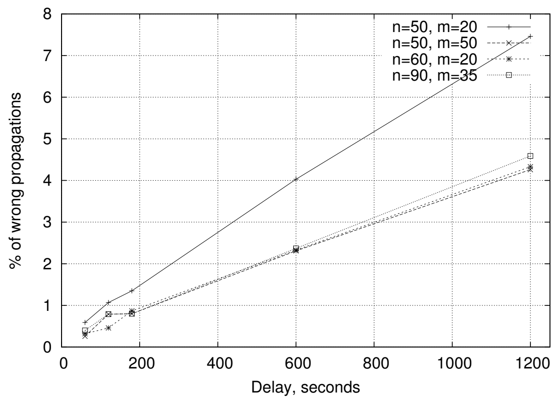


Figure 5.6: Percentage of addresses forwarded by entry node over non-attacker connections

probability of a node to forward the client's address over one of the non-attacker's connections depends on the total number of connections rather than on the fraction of attacker's connections. From Fig. 5.6 we conclude that resending client addresses over the Bitcoin network every 10 minutes seems to be a reasonable choice. Also note that even if a client's address was forwarded over a non-attacker's link, the further propagation of the address will likely stop at the next hop.

Overall success rate

The success rate $P_{success}$ of the attack depends on a number of characteristics of the real network. We propose the following method to estimate it. First, we assume that the attacker establishes all possible connections to Bitcoin servers. From the data used in Figure 5.3, we estimate the average value p_{addr}^{Avg} of the parameter p_{addr}. We did not establish more than 50 connections to avoid overloading servers, and we take a pessimistic estimation that 50 is the maximal number of attacker's connections. This yields

$$p_{addr}^{Avg} \approx 0.34.$$

Then we assume that both the testnet and the mainnet exhibit similar local topology so the probabilities $\mathbb{P}_3(L)$ for the number L of entry nodes being in top-10 are almost the same (Figure 5.2). We calculate the probabilities $\mathbb{P}_1(R)$ for the number R of entry nodes being detected out of 8 as a function of p_{addr}^{Avg}. Then we compute the total probability that the adversary detects at least $M = 3$ nodes among those appeared in top-10, and we get the following estimation (see details in the next section):

$$\mathbb{P}_{success}(3) \approx 0.13.$$

When we restrict to 2-tuples, the success rate increases to 0.37.

In the testnet we managed to achieve $p_{addr}^{Avg} = 0.86$ and the success rate for $M = 3$ being close to 60%. An attacker may achieve such high rates if he first saturates servers' connections and then gradually replaces the expired connections from other nodes with his own ones. However, this may cause degradation of QoS as some clients will be unable to connect to all their entry nodes.

Thus a careful attacker that follows the 3-tuple rule only and establishes 50 connections at maximum to each server can catch about 11% of transactions generated by clients. Given 70,000 transactions per day, this results in 7,700 transactions per day. This also means that a user needs to send 9 transactions in average in order to reveal his public IP address.

Success rate: details

In this section we describe a mathematical model that allows us to estimate the success rate of the deanonymization attack.

As inputs, we take the average probability p_{addr} over the network, which is estimated in Section 5.1.4, and the distribution of the number of entry nodes among the first 10 nodes reporting a transaction to attacker's peers (Section 5.1.3). We extrapolate the latter probability spectrum from the test net to the main net, which assumes similar network performance and the stability of the spectrum when the attacker has more or fewer connections to servers. The correctness of the extrapolation can be tested only by mounting a full-scale attack on the network, which we chose not to perform for ethical reasons.

First, we introduce two combinatorial formulas. Suppose that there are N balls. If each ball is red with probability p_a, and green with probability $1 - p_a$, then the probability that there are R red balls is

$$\mathbb{P}_1(R; N) = \binom{N}{R} p_a^R (1 - p_a)^{N-R} \tag{5.1}$$

Now assume that there are R red balls and $N - R$ green balls. Suppose that we select L balls at random out of N. The probability that there will be exactly q red balls among L chosen is computed as follows:

$$\mathbb{P}_2(q\,;\,L, R, N) = \frac{\binom{R}{q}\binom{N-R}{L-q}}{\binom{N}{L}}.$$

Now we get back to Bitcoin. If each entry node is detected with probability $p_{addr}^{Avg} = 0.34$ (Section 5.1.4), then according to Eq. (5.1) we detect R entry nodes out of 8 with the following probability spectrum:

$\mathbb{P}_1(R; 8)$:

R	Probability
0	0.0360
1	0.1484
2	0.2675
3	0.2756
4	0.1775
5	0.0732
6	0.0188
7	0.0028
8	0.0002

Based on our experiments on the Bitcoin test net (Section 5.1.3), we computed the probability to have L entry nodes among the top-10 (Table 5.1).

$\mathbb{P}_3(L)$:

L	Probability
1	0.0613
2	0.1675
3	0.2103
4	0.2323
5	0.1803
6	0.1003
7	0.0383
8	0.0096

Table 5.1: Probability that L entry nodes (out of 8) appear in the top-10 of those that forward the transaction to adversary's client.

We assume that both events are independent. Then the probability that at least M out of these L nodes we have detected (i.e. it belongs to the set of R entry nodes) is

$$\mathbb{P}_{success}(M) = \sum_{q \geq M} \sum_{L \leq 8} \sum_{R \leq 8} \mathbb{P}_2(q\,;\, L, R, 8) \cdot \mathbb{P}_1(R; 8) \cdot \mathbb{P}_3(L);$$

We have made some calculations and got the following results:

$$\sum_{L\leq 8}\sum_{R\leq 8}\mathbb{P}_2(q\,;\,L,R,8)\cdot\mathbb{P}_1(R;8)\cdot\mathbb{P}_3(L):$$

q	Probability
0	0.2513
1	0.3730
2	0.2448
3	0.0986
4	0.0267
5	0.0049
6	0.0006

$$\mathbb{P}_{success}(M):$$

M	Probability
1	0.7487
2	0.3757
3	0.1308
4	0.0323
5	0.0056
6	0.0006

Therefore, we expect to catch 3-tuples in 13% of transactions, and 2-tuples in 37% of transactions.

We applied this model to the testnet as well, and obtained that it fits our actual deanonymization results well:

Estimated p_{addr}	Deanonymization rate with 3-tuples	
	Actual	Predicted
0.64	41%	44.9%
0.86	59.9%	66.7%

Attack costs

The expenses for the attack include two main components: (1) renting machines for connecting to Bitcoin servers and listening for `INVENTORY` messages; (2) periodically advertising potential client addresses in the network. Note that if an attacker rents servers, the incoming traffic for the servers is normally free of charge. Assuming that an attacker would like to stay stealthy, she would want to have 50 different IP addresses possibly from different subnetworks. Thus she might want to rent 50 different servers. Assuming monthly price per one server 25 EUR, this results in 1250 EUR per month.

When advertising potential client addresses, the attacker is interested in that the addresses propagate in the network as fast as possible. In order to achieve this the attacker might try to advertise the addresses to all servers simultaneously. Given that there are 100,000 potential clients and the attacker needs to send 10 addresses per `ADDR` message, this results in 10,000 `ADDR` messages of 325 bytes each per Bitcoin server or (given there are 8,000 Bitcoin servers) 24.2 GB in total.

If an attacker advertises the addresses every 10 minutes and she is interested in continuously deanonymising transaction during a month, it will require sending 104,544 GB of data from 50 servers. Given that 10,000 GB per server is included into the servers price and the price per additional 1,000 GB is 2 EUR [56], the attacker would need to pay 109 EUR per month. As a result the total cost of the attack is estimated to be less than 1500 EUR per month of continuous deanonymisation.

Transaction propagation delay

In this section we measure transaction propagation delays between our high-speed server (1 Gbit/s, Intel Core i7 3GHz) located in Germany and 6,163 other Bitcoin servers. As was described in Section 5.1.2, Remark 1, it takes 3 steps to forward a transaction between two Bitcoin peers. As we are not able to obtain times when a remote peer sends an `INVENTORY` message, we skipped the first step (i.e. propagation delays of `INVENTORY` messages) and measured time differences between receptions of corresponding `INVENTORY` messages and receptions of the transactions. Note however that the size of an `INVENTORY` message is 37 bytes, while the size of a transaction which transfers coins from one pseudonym to two other pseudonyms is 258 bytes. Thus the obtained results can serve as a good approximation. For each Bitcoin server we collected 70 transactions and combined them into a single dataset (thus having 431,410 data points). Fig. 5.7 shows probability density function of the transaction propagation delay between our node and other Bitcoin servers and Fig. 5.8 shows the corresponding cumulative distribution.

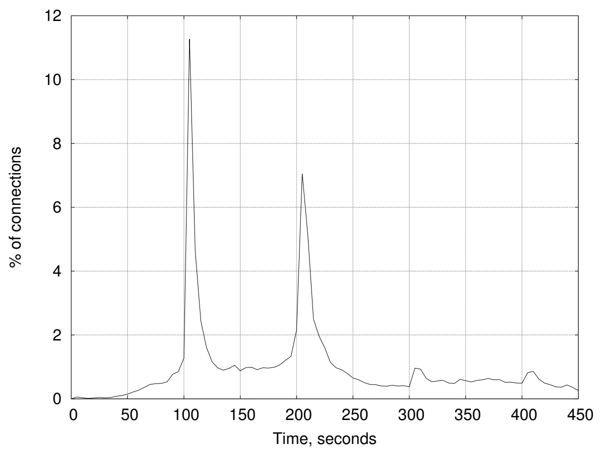


Figure 5.7: Transaction propagation delay, density

On stability of the entry nodes

In this section we estimate the stability of a client's fingerprint (the set of eight first-hop connections). According to the `bitcoind` source code (version v0.9.1), there are three reasons why an entry node can be disconnected from a client:

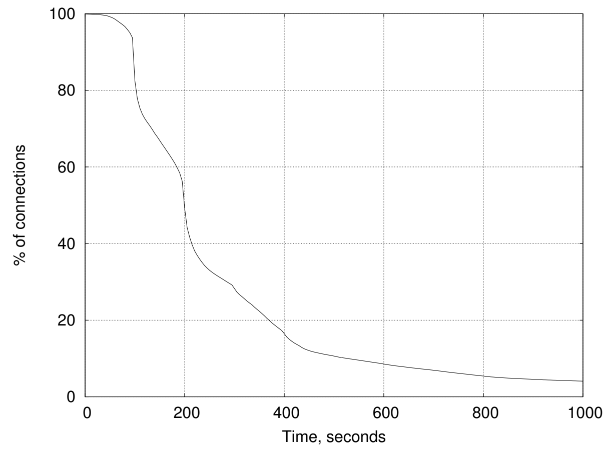


Figure 5.8: Transaction propagation delay, cumulative

- The client switched off the computer/closed Bitcoin application.
- No data was sent over a connection for 1.5 hours.
- An Entry node goes offline.

Given the number of transitions generated by the network[40], block generation rate, and addresses propagation, some data is normally sent to and from the entry nodes within 1.5 hours.

In order to estimate the probability of an entry node going off-line we we took data from `getaddr.bitnodes.io` which produces a list of running Bitcoin servers every five minutes. We analysed the data for two weeks. The probability for a node to disconnect after specific amount of time with 95% confidence interval is shown on Fig. 5.9.

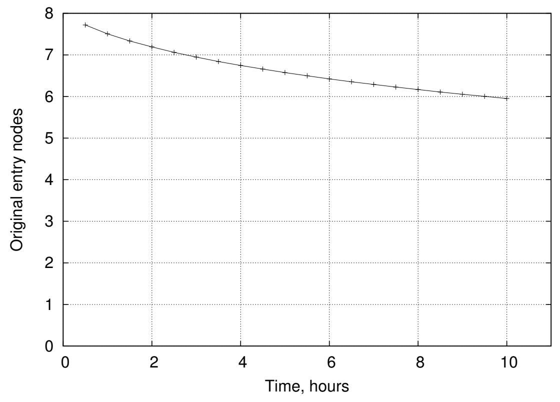


Figure 5.9: Bitcoin servers churn rate

Fig. 5.9 shows that after 2.5 hours only one node would disconnect on average and only two nodes will disconnect after 10 hours. So for the typical duration of a client session the fingerprint is very stable. In our experiment, after running our Bitcoin client for about 10 hours 3 nodes out of eight have disconnected.

The second point we address in this section is regarding the usage of VPN which is a popular recommendation for preserving anonymity in Bitcoin [32]. While protecting a user's IP, the stability of the fingerprint still allows an attacker to glue together different Bitcoin addresses of the same user. We checked the stability of the fingerprint on the Bitcoin testnet while connecting to the network:

1. via public free VPNs (`vpngate.net`);
2. via a non-free one (AirVPN).
3. via our own VPN server.

For cases 2 and 3, the stability of the fingerprint was the same as if no VPN was used. For case 1, connections to entry nodes were dropped from time to time (about every 20 mins for the main net and about every few minutes for the testnet due too absence of traffic) by the VPN servers. It's likely that free VPN servers were set with small inactivity timeouts and some limits for connection durations.

5.1.5 Countermeasures

As a possible countermeasure against client de-anonymization we propose add some random delay after the transaction. This will remove likability of transactions and will also prohibit distinguishing of different clients from the same ISP. This however will not prevent the attacker from learning the ISP of the client. One can also increase the percentage of trickled transactions from 75% to 90%, this of course will increase transaction propagation delays. Another efficient counter measure is to decrease the number of outgoing connections from 8 to 3; this however has an implication that the network becomes less connected.

5.2 Bitcoin over Tor

In the previous sections we showed that the anonymity provided by the plain Bitcoin protocol is rather low. This should encourage users to connect to the Bitcoin network through anonymizers like Tor. In this section we will show that such combination not only provides limited level of anonymity but also exposes users to man-in-the-middle attacks.

5.2.1 Disconnecting from Tor

We first show how to prohibit the Bitcoin servers to accept connections via Tor and other anonymity services. This results in clients using their actual IP addresses when connecting to other peers and thus exposing their public IP addresses. In the further text we discuss Tor, but the same method applies to other anonymity services with minor modifications.

To separate Tor from Bitcoin, we exploit the Bitcoin built-in DoS protection. Whenever a peer receives a malformed message, it increases the penalty score of the IP address from which the message came (if a client uses Tor, then the message will obviously come from one of the Tor exit nodes). When this score exceeds 100, the sender's IP is banned for 24 hours. According to the Bitcoin core implementation (version v0.9.2), there are many ways to generate a message which would cause penalty of 100 and an immediate ban, e.g. one can send a malformed loose coinbase transaction which is 60 bytes in size. It means that if a client proxied its connection over a Tor relay and sent a malformed message, the IP address of this relay will be banned.

This allows to separate any target server from the entire Tor network. For that we connect to the target through as many Tor nodes as possible. For the time of writing there were 1008 Tor exit nodes. Thus the attack requires establishing 1008 connections and sending a few MBytes in data. This can be repeated for all Bitcoin servers, thus prohibiting all Tor connections for 24 hours at the cost of a million connections and less than 1 GByte of traffic. In case an IP address of a specific Bitcoin node can be spoofed, it can be banned as well.

5.2.2 Getting in the middle

Instead of just baning Bitcoin clients from using Tor an attacker might achieve much smarter results. By exploiting Bitcoin's anti-DoS protection a low-resource attacker can force users which decide to connect to the Bitcoin network through Tor to connect exclusively through her Tor Exit nodes or to her Bitcoin peers, totally isolating the client from the rest of the Bitcoin P2P network. This means that combining Tor with Bitcoin may have serious security implications for the users: 1) they are exposed to attacks in which an attacker controls which Bitcoin blocks and transactions the users are aware of; 2) they do not get the expected level of anonymity.

The attack consists of four steps:

- Inject a number of Bitcoin peers to the network. Note that though Bitcoin allows only one peer per IP address, it does not require high bandwidth. IP addresses can be obtained relatively cheaply and on per-hour basis.
- Periodically advertise the newly injected peers in the network so that they are included into the maximum possible number of buckets at the client side.
- Inject some number of medium-bandwidth Tor Exit relays. Even a small fraction of the Exit bandwidth would be enough for the attacker as will be shown later.
- Make non-attacker's Bitcoin peers ban non-attacker's Tor Exit nodes.

We now explain each step of the attack in more detail. See section 5.2.4 for attack parameter estimation.

Injecting Bitcoin peers

This step is rather straightforward. In order to comply with Bitcoin's limitation "one peer per IP address", the attacker should obtain a large number of IP addresses. The easiest way would be to rent IP addresses on per hour basis. The market value is 1 cents per hour per IP address [74]. The important note is that the obtained IP addresses will not be involved in any abusive activity (like sending spam or DoS attacks) which makes this part of the attack undetectable.

Advertising malicious peers

The attacker is interested in that her Bitcoin peers are chosen by Bitcoin clients as frequently as possible. In order to increase by factor four the chances for her peers to be included into "tried" buckets, the attacker should advertise the addresses of her peers as frequently as possible. This mechanism would allow the attacker to inject less malicious peers. Note also that address advertisement is not logged by default and thus requires special monitoring to be noticed.

Injecting Tor Exit nodes

During this step the attacker runs a number of Exit Tor nodes. In order to get Exit flag from the Tor authorities, an attacker's Exit node should allow outgoing connections to any two ports out of ports 80, 443, or 6667. Such an open Exit policy might not be what a stealthy attacker wants. Fortunately for the attacker she can provide incorrect information about her exit policy in her descriptor and thus have Exit flag while in reality providing access to port 8333 only. The attacker can do even better, and dynamically change the exit policy of her relays so that only connections to specific Bitcoin peers are allowed. We implemented this part of the attack: while the Tor consensus indicated that our relays allowed exiting on ports 80, 443, and 8333 for any IP address, the real exit policy of our relays was accepting port 8333 for a couple of IP addresses[5].

Banning Tor Exit nodes

In this phase, the attacker exploits the built-in Bitcoin anti-DoS protection. The attacker chooses a non-attacker's Bitcoin peer and a non-attacker's Tor Exit, builds a circuit through this Exit node and sends a malformed message to the chosen Bitcoin peer (e.g. a malformed coinbase transaction which is 60 bytes in size and which causes the immediate ban for 24 hours). As soon as the Bitcoin peer receives such message it analyses the sender's IP address which obviously belongs to the Tor Exit node chosen by the attacker. The Bitcoin peer then marks this IP address as misbehaving for 24 hours. If a legitimate client then tries to connect to the same Bitcoin peer over the banned Exit node, his connection will be rejected. The attacker repeats this step for all non-attacker's Bitcoin peers and each non-attacker's Tor Exit node. This results in that a legitimate Bitcoin user is only able to connect

[5]We also allowed exiting to IP addresses used by Tor bandwidth scanners.

to Bitcoin over Tor if he chooses either one of the attacker's peers or establishes a circuit through an attacker's Exit node. We validated this part of the attack by forcing about 7500 running Bitcoin peers to ban our Exit node. To do this we implemented a rudimentary Bitcoin client which is capable of sending different custom-built Bitcoin messages.

Defeating onion peers

Bitcoin peers can be made reachable as Tor hidden services. Banning Tor Exit nodes will obviously not prevent Bitcoin clients from connecting to such peers. Nonetheless our observations show that this case can also be defeated by the attacker.

First, the current design of Tor Hidden Services allows a low-resource attacker to DoS a hidden service of her choice (see Section 4.3). It can become a problem though for a large number of hidden services: for each hidden service the attacker needs to run at least 6 Tor relays for at least 96 hours[6], 2 relays per IP address. Fortunately for the attacker the fraction of Bitcoin peers available as Tor hidden services is quite small. During August 2014 we queried address databases of reachable Bitcoin peers [38] and among 1,153,586 unique addresses (port numbers were ignored), only 228 were OnionCat addresses and only 39 of them were actually online; in November 2014 we repeated the experiment and among 737,314 unique addresses 252 were OnionCat addresses and 46 were online. This results in (1) a very small probability for a client to choose a peer available as a hidden service; (2) this makes black-holing of existing Bitcoin hidden services practical.

Second, the attacker can at almost no cost inject a large number of Bitcoin peers available as Tor hidden services. It requires running only one Bitcoin core instance and binding it with as many onion addresses as needed. Thus users will more likely connect to attacker controlled "onion" peers.

Third, as was described in Section 2.2.3, when running Bitcoin without Tor, onion addresses received from peers are silently dropped. Thus one can only obtain OnionCat addresses by either connecting to an IPv4- or IPv6-reachable peers through a proxy[7] or by specifying an onion address manually in the command line.

5.2.3 Attack vectors

The technique described in this section allows an attacker to direct all Bitcoin-over-Tor traffic through servers under her control. This creates several attack vectors which we will briefly describe in this subsection.

Traffic confirmation attack. First, it becomes much cheaper to mount a successful traffic confirmation attack. In traffic confirmation attacks, the attacker controls a fraction of Guard and Exit nodes. The attacker sees that one of her exit nodes is requested to access a particular (e.g. censored) web-site and the attacker is interested in finding out the user who made this request. The attacker sends a traffic signature down the corresponding circuit. If the attacker was lucky and the

[6]Or 25 hours if Tor authorities run Tor version 0.2.6.6 or earlier.

[7]Not necessarily Tor.

user chose one of her Guard nodes, the attacker will see the traffic signature going through this Guard to the target user. This reveals the user's IP address.

The success probability of the attack is computed as the product of two factors: the probability for the user to choose an attacker's Guard and the probability for the user to choose an attacker's Exit. Since now all exit Bitcoin-over-Tor traffic goes through the attacker, the second factor becomes 1.

Revealing Guard nodes. In case the attacker does not control the user's Guard node, he may try to find this Guard. We assume that the attacker controls a fraction of middle nodes. As before the attacker would send a traffic signature down the circuit and if none of the attacker's middle nodes detects this signature, the attacker drops the circuit. This will force the user to build another circuit. After some number of circuit tries, one of the attacker's middle nodes will finally be chosen. This middle node will know the user's Guard node. The re-identification of the user between different circuits is possible e.g. using the fingerprinting technique from section 5.3.

Revealing the guards does not immediately allow an attacker to reveal the location of the user but gives her the next point of attack. Given that guard nodes are valid for more than a month, this may be sufficient to mount a legal attack to recover traffic meta data for the guard node, depending on the jurisdiction the guard node is located in.

Linking different bitcoin addresses. Even without knowing the user's IP, the attacker can link together user's transactions regardless of pseudonyms used.

Possibility of double spending. Finally, after successfully mounting the attack described in this section the attacker controls the connectivity to the Bitcoin network for users which chose to use Tor. This increases the success rate of double-spend attacks.

In addition the attacker can defer transactions and blocks and send dead forks. In collusion with a powerful mining pool (for example 10-20% of total Bitcoin mining capacity) the attacker can create fake blocks. This enables additional possibilities for double spending, however to make this relevant the amount should exceed what such miner would be able to mine in the real Bitcoin network. Also complete alternative Bitcoin reality for all the users who access Bitcoin solely through Tor is possible. This however would come at a cost of 5-10 times slower confirmations, which after some time can be detected by the wallet software.

5.2.4 Analysis

Estimating client's delays

The steps described in section 5.2.2 imply that once a client decides to use Bitcoin network over Tor, he will only be able to do this by choosing either one of the attacker's Exit nodes or one of the attacker's Bitcoin peers. However for the attack to be practical a user should not experience significant increases in connection delays. Otherwise the user will just give up connecting and decide that Tor-Bitcoin bundle is malfunctioning. In this section, we estimate the number of Bitcoin peers and the amount of bandwidth of Tor Exit relays which the attacker needs to inject, so that the attack does not degrade the user's experience.

Once the attacker completes the steps described in section 5.2.2, for each user connecting to the Bitcoin network through Tor there are several possibilities (see Fig. 5.10).

1. The user chooses one of the attacker's Bitcoin peers. The attacker does nothing in this case: the attacker automatically gains control over the information forwarded to the user.

2. The user chooses one of the attacker's Exit nodes. The attacker can use the fact that Bitcoin connections are not encrypted and not authenticated and redirect the client's request to Bitcoin peers under her control.

3. The user chooses a non-attacker's Exit relay and a running non-attacker's Bitcoin peer. In this case, due to the ban the user's connections will be rejected. And the user will try to connect to a different Bitcoin peer.

4. The user chooses a non-attacker's Exit relay and a non-attacker's Bitcoin peer which went offline[8]. In this case the Bitcoin client will wait until the connection times-out which can be up to two minutes (see section 2.1.2). This delay on the surface looks like taking prohibitively long time. However since during these two minutes Tor rebuilds new circuits every 10-15 seconds, trying new Exits at random, it actually makes the attacker's life easier. It increases the chances that malicious Exit relay will be chosen.

Handling unreachable Bitcoin peers. Before estimating the delays we consider case 4 in more detail. Our experiments show that for a Bitcoin client which was already used several times prior to the connection over Tor, the address database contains 10,000 – 15,000 addresses and the fraction of unreachable Bitcoin peers among them is between 2/3 and 3/4. Abundance of unreachable addresses means that case 4 is the most frequent scenario for the client. Consider a client which chose an unreachable Bitcoin server and a non-attacker's Exit node.
The Exit relay can send either:[9]
1) An END cell with "timeout" error code. In case of a "timeout" message, Tor sends a "TTL expired" SOCKS error message to the Bitcoin application which then tries another Bitcoin peer.
2) An END cell with "resolve failed" error code[10]. In case of "resolve failed" message, Tor drops the current circuit and tries to connect to the unreachable Bitcoin peer through a different Exit node. After 3 failed resolves, Tor gives up and sends a "Host unreachable" SOCKS error code, which also results in Bitcoin trying a different peer.
3) The third and the most common option is that the exit relay will not send any cell at all during 10-15 seconds. As was described in the Background section that

[8]Or never really existed: Bitcoin allows storing fake addresses in client addresses database.
[9]This is based on the Tor source code analysis (v0.2.5.10) and monitoring a running Tor instance.
[10]We observed this behaviour not only for hostnames but also for IP addresses.

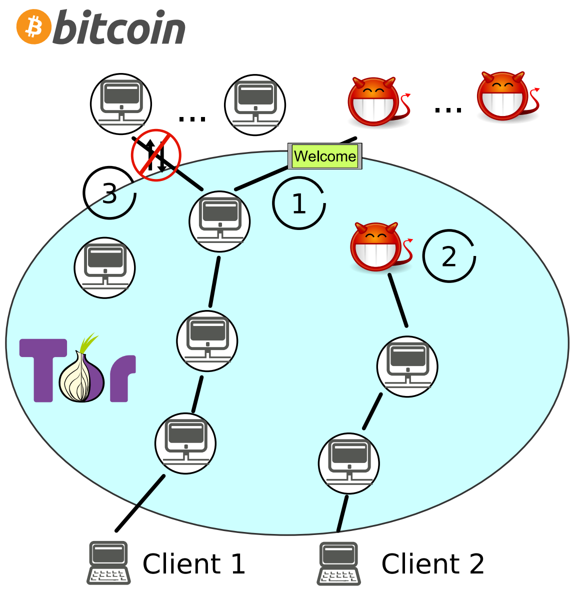


Figure 5.10: Client's state after the main steps of the attack

in case the Exit node does not send any reply within 10 or 15 seconds (depending on the number of failed tries) along the circuit attached to the stream, Tor drops the current circuit and attaches the stream to another circuit (or to a newly built one if no suitable circuits exist). In case Tor cannot establish connections during 125 seconds, it gives up and notifies Bitcoin client by sending a "General failure" SOCKS error message. Bitcoin client then tries another peer.

Delays. The facts that a) Tor tries several different circuits while connecting to unreachable peers and b) the fraction of unreachable peers in the client's database is very large, significantly increases the chances that a malicious Exit node is chosen. The attacker only needs this to happen once, since afterwards all connections to the other Bitcoin peers will be established through this Tor circuit; Bitcoin client will work even with one connection. On the other hand, unreachable nodes increase the delay before the user establishes its first connection. This delay depends on the number of attacker's Bitcoin peers and on how often the user chooses new circuits.

In order to estimate the latter, we carried out the following experiment. We were running a Bitcoin client over Tor and for each connection to an unreachable Bitcoin client we were measuring the duration of the attempt and the number of new circuits (and hence different Exit nodes). The cumulative distribution function of the amount of time a Bitcoin client spends trying to connect to an unreachable node is shown in Fig. 5.11. On the average a Bitcoin peer spends 39.6 seconds trying to connect to an unreachable peer and tries to establish a new circuit (and hence a different Exit node) every 8.6 seconds. This results in 4.6 circuits per unreachable peer on the average.

We now estimate how long it will take a user on the average to establish his first connection to the Bitcoin network. This delay obviously depends on the number of

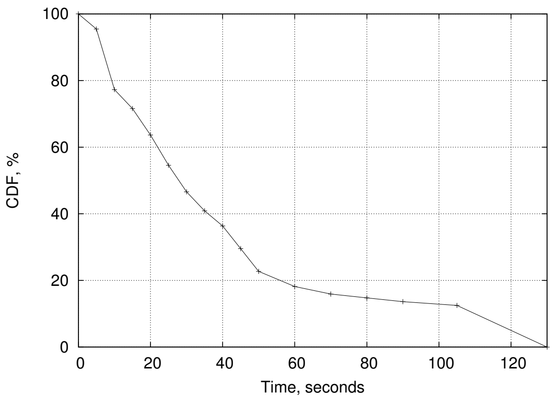


Figure 5.11: Time spent connecting to an unreachable node

the attacker's Bitcoin peers and the amount of bandwidth of her Tor Exit relays. We adopt a simple discrete time absorbing Markov chain model with only three states (see Fig. 5.12):

- State 1: the Bitcoin client tries to connect to an unreachable peer;
- State 2: the Bitcoin client tries to connect to a reachable Bitcoin peer banned by the attacker;
- State 3: the Bitcoin client tries to connect to an attacker's Bitcoin peer or chooses an attacker's Tor Exit node. State 3 is absorbing state, once it is reached, the user thinks that he connected to the Bitcoin network (while he is now controlled by the attacker).

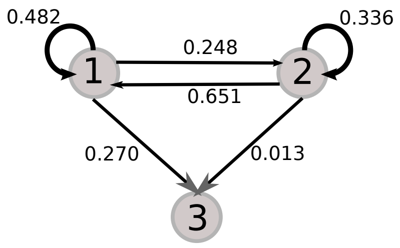


Figure 5.12: Delay before the first connection. Markov chain with probabilities for 400K of Exit capacity and 100 malicious Bitcoin peers. The client spends about 0.5 seconds in State 2 and about 40 seconds in State 1

After composing the fundamental matrix for our Markov chain, we find the average number of steps in two non-absorbing states. Taking into account the average amount of time spent by the user in each of the states (we use our experimental data here), we find the average time before the absorbing state. We compute this time for different number of Bitcoin peers and Tor Exit relay bandwidth. The results are presented in Fig 5.13. We have taken a conservative estimate that the fraction of unreachable Bitcoin peers in the client's database is $2/3 = 66\%$, also the client spends only about 0.5 seconds in State 2 and about 40 seconds in State 1.

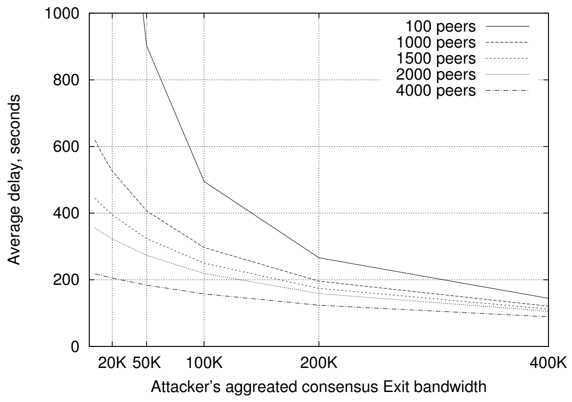


Figure 5.13: Average time before the first connection

Fig. 5.13 shows that an attacker having 100,000 of consensus Exit bandwidth and 1000 Bitcoin peers is able to carry out the attack while keeping the average delay below 5 minutes. For example an attacker controlling a small botnet can afford that many peers (she will need 1000 peers with public IPs or supporting UPnP protocol). An attacker having consensus weight of 400,000 and very few peers can decrease the average delay to about two minutes. Such a bandwidth is achievable by an economy level attacker as will be shown later in this section.

The line corresponding to 4000 attacker's Bitcoin peers in Fig. 5.13 is not as unrealistic as it may seem. Recall (see Section 2.2.3) that each Bitcoin peer address can go to up to 4 "new" buckets at client's side. This can be used by a persistent attacker to increase the choice probability for her peers by a factor 4 (in the best case) which means an attacker can have significantly less than 4000 peers.

Clients with empty addresses cache. As was pointed in Section 2.2.3, all IPv4 and IPv6 addresses received from DNS-oneshots are dropped by a Bitcoin client if Tor is used. If the addresses database of a client is empty and all the seed nodes are banned, the client can connect to hidden services only.

Attack Costs

Tor Exit nodes. During July 2014 we were running a non-Exit Tor relay for 30 USD per month. We set the bandwidth limit of the relay to 5 MB/s which resulted in traffic of less than 15GB per hour. The consensus bandwidth of this relay fluctuated between 5,000 and 10,000 units[11]. While the total weighted consensus bandwidth of all exit nodes was about 7 million units, the weighted consensus bandwidth of

[11]A unit roughly corresponds to 1 KB/s of traffic.

relays allowing exiting at port 8333 was about 5.7 million units. Assuming that we could achieve 5,000 − 10,000 units in the consensus for an Exit node this gives the probability of 0.08%-0.17% for our relay to be chosen for Exit position by a user. Given that 10 TB of traffic is included into the server's price and one has to pay 2 EUR per additional 1 TB, it would cost an attacker 360 USD to have 180 TB of traffic per month. The corresponding speed is 69 MB/s (69,000 consensus bandwidth units). By running 6 such relays the attacker can achieve 400K of bandwidth weight in total for the price below 2500 USD (2160 USD for the traffic and 240 for renting fast servers).

Thus having a consensus weight close to 400,000 is possible for an economy-level attacker. The attacker can also decide to play unfair and mount a bandwidth cheating attack which would allow her to have a high consensus weight while keeping the budget of the attack even lower [31]. This is especially possible since Bitcoin traffic by itself is rather lightweight and high bandwidth would be needed only in order to drive Tor path selection algorithm towards attacker's nodes.

Bitcoin peers. The attack described in this section suggests the attacker injects a number Bitcoin peers; at the same time Bitcoin network allows only one peer per IP address. Thus the attacker is interested in getting as many IP addresses as possible. Currently there are several options. The cheapest option would be to rent IP addresses on per hour basis. The market price for an IP address is 1 cent per hour [74]. This results in 7200 USD per 1000 IP's per month. From these computations it is clear that an attacker would do better by investing in Exit bandwidth rather than running Bitcoin peers (unless she controls a small botnet), and the only limitation for her would be not to become too noticeable.

5.2.5 Countermeasures

These attacks are very effective due to a feature of Bitcoin which allows an easy ban of Tor Exit nodes from arbitrary Bitcoin peers. One possible countermeasure against Tor-ban could be to relax the reputation-based DoS protection. For example each Bitcoin peer could have a random variable, which would decide whether to turn ON or OFF the DoS protection mechanism with probability 1/2. As a result the attacker might be able to DoS at most half of the network, but on the other hand he will not be able to ban any relays or VPNs from *all* the Bitcoin peers.

An obvious countermeasure would be to encrypt and authenticate Bitcoin traffic. This would prevent even opportunistic man-in-the-middle attacks (i.e. even if the user is unlucky to choose a malicious Exit relay). Another possible countermeasure is to run a set of "Tor-aware" Bitcoin peers which would regularly download Tor consensus and make sure that Bitcoin DoS countermeasures are not applied to servers from the Tor consensus. K. Atlas [35] implemented a similar countermeasure (which maintains *historical record of Tor exit nodes used to connect to the Bitcoin network.*)

Finally, Bitcoin developers can maintain and distribute a safe and stable list of onion addresses. Users which would like to stay anonymous should choose at least one address from this list. There currently exists a short and not up-to-date list of

Bitcoin fallback onion addresses [51]. Another advice for a user would be to run two Bitcoin nodes, one over Tor and one without, and compare their blockchains and unconfirmed transactions. This would prevent from creation of virtual reality for Tor-only users.

5.3 User fingerprinting

In this section we describe a technique which can be used to fingerprint Bitcoin users by setting an "address cookie" on their computers. A cookie can be set and checked even when the user connects to the Bitcoin network through Tor or through a chain of proxies. It can be used to correlate different transactions of the same user even across different sessions (i.e. after his computer was rebooted). If the user decides later to send a non-sensitive transaction without Tor, his fingerprint can be correlated to his IP address, thus deanonymizing all his transactions sent previously through Tor. **From April 2015 Bitcoin implements a countermeasure which prevents an attacker from requesting fingerprint from clients.**

5.3.1 Setting cookie

The fingerprinting technique is based on the Bitcoin's peer discovery mechanism. More specifically on that a Bitcoin peer stores addresses received from other peers and on that his database can be queried.

As was described in section 2.2.3 whenever a peer receives an unsolicited `ADDR` message, it stores the addresses from this message in his local database. The attacker can use this fact as follows. When a client connects to an attacker's peer, the peer sends him a unique combination of possibly fake addresses (*address cookie* or *fingerprint*; we will use these two terms interchangeably below). Unique non-existent peer addresses work best, however a more sophisticated and more stealthy adversary may use existing Bitcoin peer addresses as well (exploiting the combinatorics of the coupon collector problem). The client stores these addresses and the next time he connects to (another) malicious peer, the peer queries his address database. If the fingerprint addresses are present in the set of retrieved addresses, the attacker identifies the user.

Consider a user c and assume that one of the attacker's servers is among the user's entry nodes. The attacker executes the following steps:

1. Send a number of `GETADDR` messages to the user. The user should reply with `ADDR` messages.

2. Check the received from the client addresses if they already contain a fingerprint. If the user already has a fingerprint, stop. Otherwise go to the next step.

3. Generate a unique combination of N fake addresses FP and send them in an `ADDR` message to the client. The `ADDR` message should contain at least 11

addresses so that it is not forwarded by the client. If N is less than 11, pad the message with $11 - N$ legitimate[12] addresses.

4. If the user connects to the Bitcoin network directly (i.e. without Tor), store the correspondence between the client's IP address and his fingerprint as a tuple (FP, IP_c). If the user connects through Tor save him as (FP, NIL).

There is a detail of the Bitcoin protocol which an attacker should take into account. When a client connects to the Bitcoin network over Tor, he will accept and store in his database OnionCat addresses only (thus ignoring IPv4 addresses). It means that in case of Tor, the fingerprint generated by the attacker should consist of OnionCat addresses only. On the other hand when a client connects to the network directly, he will ignore non-IPv4/IPv6 addresses. Hence an attacker should generate a fingerprint consisting of IPv4 addresses only. This results in that an attacker needs to store 2 different types of cookies: OnionCat and IPv4. At the same time, a client does not limit the types of addresses he sends as a reply to a `GETADDR` message. This means that once a cookie was set it can be queried both over Tor and directly.

5.3.2 Extracting cookie

The remaining question is how many `GETADDR` messages an attacker needs to send to the client to learn that the database of this client contains a cookie. According to [30], section 9.2 it can be up to 80 messages to retrieve the full collection of client's addresses. However in practice we will not need to collect all the addresses in a fingerprint, which significantly reduces the number of requests. About eight `GETADDR` messages would be sufficient to retrieve about 90% of the cookie addresses. This shows that the cookie can be checked without raising suspicion.

5.3.3 Low-resource Sybil attacks on Bitcoin

A client needs to connect directly to one of the attacker's nodes in order for the attacker to set/refrech the cookie or to reveal the client's IP address and thus deanonymize his previous transactions done over Tor. Bitcoin as a peer-to-peer network is vulnerable to Sybil attacks and just operating many Bitcoin servers means that a client will sooner or later choose an entry node controlled by the attacker (i.e. in some number of sessions). However running too many servers can be costly (see section 5.2.4 for attack cost estimation). Fortunately for the attacker there are a couple of ways to prevent Bitcoin clients from using non-attacker's Bitcoin servers (and choose an attacker's one instead).

Exhausting connections limit

As described in section 2.2.3, by default a Bitcoin server accepts up to 117 connections. Once this limit is reached all new incoming connections are dropped. At

[12]By legitimate we mean that there are some Bitcoin servers running at these addresses.

the same time a Bitcoin server neither checks if some of these connections come from the same IP address[13], nor forces clients to provide proof-of-work. As a result a low-resource attacker can establish many connections to all but his Bitcoin servers[14] and occupy all free connection slots. If a client connects directly to a Bitcoin server connection slots of which are occupied, the connection will be dropped immediately, thus the client will soon end up connecting to a malicious peer. This straightforward attack has been known in the Bitcoin community.

Port poisoning attack

A less effective but much stealthier new attack exploits the following fact. Peer addresses are of the following form (IP,PORT). However when a client decides if to add a received address to the database, he does not take the port number into account. For example assume a client receives an address $(IP_0, PORT_1)$ and there is already an entry in the client's database $(IP_0, PORT_0)$. In such case the client will keep $(IP_0, PORT_0)$ and will not store $(IP_0, PORT_1)$.

The attacker can use this fact to flood with clients with addresses of legitimate Bitcoin servers but wrong port numbers. If the attacker is the first to send such addresses, the client will not be able to connect to legitimate nodes.

5.3.4 Attack vectors

Deanonymization of Bitcoin over Tor users. Consider the following case. A client uses the same computer for sending both benign Bitcoin transactions and sensitive transactions. For benign transactions the user connects to Bitcoin directly, but for sensitive transactions he forwards his traffic through a chain of Tor relays or VPNs. If an attacker implements the attack described in section 5.2.2, all client's sensitive transactions with high probability will go through attacker's controlled nodes which will allow her to fingerprint the user and record his transactions.
When the client later connects to the Bitcoin network directly to send benign transactions, he will with some probability choose an entry node controlled by the attacker (in section 5.3.3 we showed how to increase this probability). Once it happens, the attacker can query the client for the fingerprint and thus correlate his sensitive transactions with his IP address. Note that even if the attacker is not implementing the complete man-in-the-middle attack on Tor, but just injects Sybil peers and Sybil hidden services she will be able to link many sensitive transactions to the real IP addresses of users.
Linking different Tor sessions. In the case, when a client uses a separate computer (or Bitcoin data folder[15]) to connect to Bitcoin through Tor, the attacker will not be able to learn his IP address. However, the attacker will still be able to link different transactions of the same user. This can be done even across different ses-

[13]One explanation is that if clients are behind the same NAT they will share the same IP address.
[14]The list of all running Bitcoin servers can be obtained from e.g. [38].
[15]A Bitcoin data folder is a directory where Bitcoin clients store their wallets and dump IP address databases between restarts.

sions (computer restarts). This will in turn allow the attacker to correlate different Bitcoin addresses completely unrelated via transaction graph analysis.

Domino Effect. Tor multiplexes different streams of the same user over the same circuits. This means that if the source of one stream in the circuit is revealed by the fingerprinting attack, all other streams will also be deanonymized. Specifically, it is likely that a user who sends a sensitive Bitcoin transaction through Tor, will also browse a Darkweb site. Similar result was also noted in [23] but in relation to Bittorrent over Tor privacy issues. To prevent this it is recommended to enable option IsolateSOCKSAuth when running Tor (this will prevent *sharing circuits with streams for which different SOCKS authentication was provided*).

5.3.5 Analysis. Stability of Cookie

According to the Bitcoin core [48] source code (v0.9.2), at the startup when a client establishes outgoing connections he sends `GETADDR` messages, and gets back a set of addresses (typically 2,500, the maximum possible number per `GETADDR` request). Given 8 outgoing connection, the client will receive up to 20,000 non-unique addresses. These addresses can potentially overwrite the address cookie previously set by an attacker. Below we will try to estimate how this affects the stability of the cookie. Assume that an attacker managed to set an address cookie on a user's computer and disconnected (e.g. the client ended the session). The client then establishes a new session sometime later.

First note that if the user reconnects to Bitcoin over Tor and if the attacker has mounted the attack from section 5.2.2, he controls all user's traffic and the cookie is preserved. Let us now describe what happens if the client decides to connect to the Bitcoin network directly.

When a client receives an address IP_{in} he first checks if it is already contained in his database. If yes, he does nothing (thus the cookie is not damaged). In case it is a new IP address the client executes the following procedure. He computes the bucket number (see section 2.2.3) based on the peer which sent the address and the address itself. If this bucket contains a "terrible"[16] address $IP_{terrible}$, it is replaced by IP_{in}. Otherwise 4 random addresses are chosen from the bucket and the one with the oldest timestamp is replaced by IP_{in}.

In other words, in order for the incoming address IP_{in} to replace a cookie address IP_{cookie}[17] the following conditions should hold:

1. IP_{in} should not be in the user's database;
2. IP_{in} should belong to the same bucket B as IP_{cookie} and there should be no "terrible" addresses in B;
3. IP_{cookie} should be among the four randomly chosen addresses, and its timestamp should be the oldest.

[16]An address is called terrible if any of the following holds: 1) its timestamp is 1 month old or more than 10 minutes in the future; 2) 3 consecutive connections to this address failed.

[17]A cookies consists of several IP address, but in order to make the explanation simpler, we use just one address here.

These conditions as we will see below make the attacker's cookie quite stable for many hours (this also depends on the number of user sessions since at each startup the address database is refreshed).

In order to estimate the probability that a cookie address set by the attacker is preserved we conducted the following experiment. In November 2014 we queried running Bitcoin servers by sending them `GETADDR` messages. We received 4,941,815 address-timestamp pairs. Only 303,049 of the addresses were unique. This can be interpreted as that only about 6% of the addresses received by a client will not be already contained in his database (if the client re-connects immediately).

As the second step, we looked at the timestamp distribution of the non-unique address set. This distribution can serve as approximation of the distribution of address timestamps of a client's database. The results are shown in Table 5.2: 89% of addresses had a timestamp more than 3 hours in the past. Taking into account conditions stated above, it almost guarantees that the attacker's cookie will not be damaged within the first 3 hours. For 45% of addresses the timestamp was older than 10 hours (which is the duration of a working day); 9% of addresses were older than 1 week.

Address age, hours	1-CDF
3	89%
5	77%
10	45%
15	28%
24	19%
36	15%
48	13%
72 (3 days)	12%
168 (1 week)	9%

Table 5.2: Complementary Cumulative distribution function for addresses timestamps, November 2014

The results above could be summarized as follows: (1) there is a high chance that an address received by a client will already be contained in his database, which keeps the cookie intact; (2) if a cookie IP address is among the 4 nominees for erasing, it is likely that its timestamp will be fresher than that of at least one of other nominees (and thus will not be erased).

Finally we conducted the following experiment. We set a cookie consisting of 100 IPv4 addresses and monitored how stable this cookie was across different sessions. Table 5.3 shows the decay rate of the number of cookie addresses over time and sessions. Note that by session we mean that the client switches off Bitcoin software and switches it on again, which forces him to make 8 new outgoing connections and retrieve up to 20,000 addresses.
The experiment shows that even after 10 sessions (i.e. after reception of about 200,000 non-unique IP addresses) and 8 hours, one third of the fingerprint remained in the user's database (thus it will be possible to identify the client). Note that sessions 9 and 10 took 2 and 2.5 hours. On the average an attacker will need about

Session number	Time since start, hours	Remaining addresses
1	0	100
2	0.5	100
3	1	100
4	1.5	100
5	2	100
6	2.5	100
7	3	98
8	3.5	92
9	5.5	50
10	8	36

Table 5.3: Address cookie decay rate (example)

90 peers (given that at the time of writing there are about 7,000 Bitcoin servers) to become one of the client's entry nodes during any of these 10 sessions and update the fingerprint. Running this number of peers will cost the attacker less than 650 USD per month (see section 5.2.4).

In another experiment we checked that in the case of two sessions with 10 hours between sessions, our client kept 76% of the initial fingerprint addresses, and in the case of 24 hours between two sessions 55% of the initial fingerprint were kept (which again allows the user identification). In order to carry out the experiments from this section we built our own rudimentary Bitcoin server which is able to connect/accept connections to/from Bitcoin peers and is capable of sending/receiving different Bitcoin messages on demand. We used this server as a malicious Bitcoin server which sets new address cookies and checks previously set cookies. In order to simulate a user we used the official Bitcoin core software (v0.9.2). The attack from this section was experimentally verified by tracking our own clients in the real Bitcoin and Tor networks.

5.3.6 Countermeasures

With regards to the fingerprinting attack several countermeasures are possible. First, Bitcoin peers can request performing proof-of-work computation for each sent `GETADDR` message, so that it becomes computationally expensive for an attacker to query each client. Second, according to the Bitcoin core source, the only time when a client sends a `GETADDR` message is when he establishes an outbound connection. Thus ignoring `GETTADDR` requests on outbound connections will not change the usual operation of Bitcoin networking protocol and will prevent the attacker from requesting the fingerprint[18]. Finally an immediate countermeasure would be to remove the cached address database file before each session and to use only trusted hidden-services.

[18]We implemented this countermeasure and submitted the corresponding patch.

Chapter 6

Proof-of-Work as Anonymous Micropayment

In this chapter we describe a way to use the existing Tor and Bitcoin infrastructures to make anonymous transactions. Clients of a service do not pay with electronic cash directly but submit proof of work shares which the service can resubmit to a crypto-currency mining pool. The service credits users who submit shares with tickets that can later exchanged for goods. Both shares and tickets when sent over Tor circuits are anonymous. The proposed scheme has the following desirable properties: (1) it does not rely on a central bank; (2) it preserves user anonymity; (3) it removes a psychological barrier since clients do not pay directly (and thus the risk of their money being stolen is removed); (4) unlike Bitcoin transactions which have to be stored in the public blockchain, there is no reason to store shares[1].

The scheme was specifically designed to reward Tor relay operators for providing improved quality of service but the same approach can be adopted to accept mircopayments for any other service.

Contents

6.1 Proof-of-Work as payment for service

6.1.1 Design goals

The main objective of the proposed scheme is to compensate Tor relays for providing improved service and to encourage server operator's participation in the Tor

[1]Unless a share also solves a block.

network. In addition, we require the following properties. First, the scheme should not degrade the anonymity provided by Tor, i.e. it should not introduce new attack vectors. Second, it should not involve direct payments neither with fiat nor with crypto-currencies. The reason for this is that direct payment even with a digital currency like Bitcoin will reduce user privacy and may become a strong psychological obstacle for adopting a scheme for ordinary users. Third, it should not rely on secure bandwidth measurement mechanisms. Fourth, it should not involve a central bank as in [22]. Sixth, the scheme should not require from users to run a Tor relay in order to get improved service. We analyse these properties in more detail in section 6.2.

6.1.2 System design

Tor users can get improved service from a Tor relay by producing proof-of-work and sending it to the relay over an anonymous Tor circuit. The relay can then forward this proof-of-work to a crypto-currency mining pool and earn coins. Users are rewarded by relay-specific *priority tickets* which can later be exchanged at the same relay for improved service (higher bandwidth or lower latencies). Tickets are issued by relays using blind signatures [6] and exchanged between users and relays over anonymous Tor circuits. Unlike [22] we do not use any bank entity and tickets are blind-signed by relays themselves.

Setup. In the setup phase a Tor relay first chooses a mining pool, the corresponding crypto-currencies and PoW algorithms (note that the relay can choose a pool which automatically switches to the most profitable currencies). Second, the relay generates a public/private key pair which will be used in generation of priority tickets (this key pair should be different from the relay's onion and identity keys). The relay then includes this information into its descriptor. A client which plans to obtain improved service chooses relays which announce compatible PoW algorithms.

Protocol 1. Ticket Purchase: Client C obtains a priority ticket from relay R

```
1: $C \rightarrow R$ : SUBSCRIBE message.
2: $R \rightarrow C$ : JOB message.
3: $C$ : start mining a share.
4: $C$ : If share $w$ is found, generate random number $x$ and its hash $H(x)$.
5: $C \rightarrow R$ : $w$, $H(x)$.
6: $R$ : check $w$, if correct pass it to the mining pool.
7: $R \leftrightarrow C$ : Generate partially blind signature $S$ over $\{H(x), d\}$, where $d$ is an
   assigned by the relay timestamp, which specifies the current day.
8: $C$: Keeps the ticket $T_R = \{S, d, x, H(x)\}$.
```

Purchasing priority tickets. A relay will provide improved service for clients in exchange for priority tickets. Priority tickets are relay-specific which means that by default they can only be used to purchase service from the relay which issued them (see Protocol 2 if ticket exchange is required). The protocol for client C to obtain a ticket from relay R is described in Protocol 1. Prior to execution of the protocol, the client establishes an anonymous Tor circuit to the relay. All communications are carried over this circuit, including (optionally) the future client traffic. Client C registers for a new mining job with relay R and the relay sends a reply in which it specifies the PoW algorithm, difficulty per share, and data sufficient to construct a share (steps 1–2). At step 3, the client starts solving a new share. At steps 4–5 (given that the client solved the share), the client generates a random value x and its hash $H(x)$ and sends the share to R. The relay verifies the share and produces a partially blind signature S over $H(x)$ with timestamp d as an added factor according to [11]. The tuple $T = \{S, d, x, H(x)\}$ is a priority ticket which the client can later exchange for the improved service. By reducing the granularity of the timestamp to just the current date makes all clients that got tickets on the same day undistinguishable.

Buying improved service. Every ticket that a client gets can be used to transmit cells with priority access during Δt seconds through the Tor relay which issued the ticket. In order to prevent double-spending, the relay should keep history of spent tickets. To limit the size of this database tickets should expire after e.g. 48 hours.

Priority access. We suggest using Hierarchical Token Bucket Algorithm [61] to provide improved quality of service for users with priority tickets, however other options exist [18]. HTB is a simple algorithm and it is a logical step from the currently employed by Tor Token Bucket algorithm. The priority access scheme should allocate enough resources for "free" users so that people without funds to buy high-speed computers can still have reasonable QoS with Tor.

In Hierarchical Token Bucket the bandwidth is allocated to one or more classes, and when a class-allocated bandwidth is exceeded, it can temporarily "borrow" unused bandwidth from another class. Classes form a tree structure in which only leaves have corresponding packet queues. Each class C has associated guaranteed rate R_C and Ceil rate CR_C. Class C is guaranteed to have at least rate R_C. The rate of a parent class should not exceed the sum of the rates of its children classes. CR_C specifies the maximum speed that class C can have by borrowing from its parent class. Classes borrow unused bandwidth in proportion to their allocations.

Consider an example in Figure 6.1 in which a relay is willing to provide up to 10 Mpbs for Paid and Free services in total. The guaranteed rate for Free service is 2 Mpbs; the total rate for Paid service is 8 Mbps which is later divided between two different users based on the number of tickets they pay. Consider two examples. In the first example a relay does not have any paid clients in which case the relay increases the bandwidth for Free service to 4 Mbps by borrowing from the Paid class. In the second example the relay has very few free clients which consume only 1 Mbps while classes $Group_1$ and $Group_2$ require 6 Mbps and 4 Mpbs respectively. In this case class $Group_1$ will be given additional 0.625 Mpbs and class $Group_2$ will take 0.375 Mbps.

Ticket exchange. So far in the proposed scheme a client gets tickets from the same relay R_1 for which he is working, and the tickets are valid at this relay only. Such scheme works best if the client provides proof-of-work simultaneously with sending his data over Tor. Assume now that a client pre-mined priority tickets

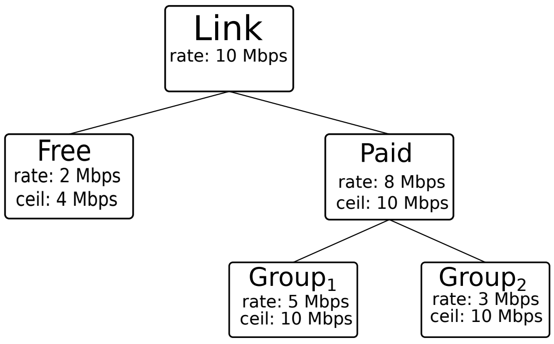


Figure 6.1: Hierarchical Token Bucket example: the link of 10 Mbps is divided between paid and free services. Free circuits will share either 2 Mpbs or 4 Mbps if there are no paid clients. Paid clients can get the whole capacity of 10 Mbps if there are no free users.

with an intention to spend them later. He might become frustrated if at the time when he decides to spend them relay R_1 is off-line. In such a case relay R_1 may team with a backup relay R_2 and ask it to accept its priority tickets. R_2 can later request payment from R_1 in crypto-coins or by redirecting his clients to mine for R_2. Protocol 2 describes how priority tickets issued to client C by relay R_1 can be spent at relay R_2. When relays R_1 and R_2 are both online they synchronise their databases of spent tickets.

Protocol 2. Ticket Exchange: C gets improved service at R_2 by providing a ticket issued by R_1

Client C obtained ticket $T_{R_1} = \{S_1, d, x, H(x)\}$ from relay R_1. R_2 is a backup relay for R_1

```
1: $C \to R_2 : T_{R_1}$
2: $R_2$ : verify signature $S_1$ and timestamp $d$.
3: $R_2$: If correct, register $T_{R_1}$ as spent (sync this with $R_1$).
4: $R_2$ : If $T_{R_1}$ is correct, provide priority access.
5: $R_2 \to R_1$ : PAYMENT_REQUEST (Once every $N$ served tickets).
```

Assume that client C has ticket $T_{R_1} = \{S_1, d, x, H(x)\}$ issued by relay R_1. The objective of the Protocol 2 is for the client to be able to get improved service from relay R_2 while preserving the following properties: (1) A colluding client and relay should not produce "free" tickets which can later be used at other relays; (2) Double spending of the same ticket at two different relays should be prevented.

"Free" tickets created by colluding client C and relay R_1 are avoided by that R_2 requests payment for each batch of N served tickets (either in crypto-coins or by delegating new mining work). We can envision that in practice relays R_1 and R_2 might be run by the same operator or by two operators, who trust each other. In the second case the amount of trust can be regulated by the size of N. In case R_1 stops paying, relay R_2 will stop accepting its tickets. In order to prevent double-spending of the same ticket at relays R_1 and R_2 they should regularly synchronise their databases of spent tickets.

Mining strategies. The operator of a Tor relay which accepts PoW shares has two possibilities. First, he can decide to do solo-mining, by making his crypto-currency address a part of `JOB` messages sent to clients in the hope that one of the submitted shares will also solve a block. This strategy requires significant computational power at a large number of Tor clients. Second, the Tor relay operator may decide to ask for work from a large mining pool and then delegate this work to clients. The operator then resubmits the shares found by the clients to the mining pool. Note that the mining pool requests the relay to generate a share of difficulty lower than the current block's difficulty in the hope that one share will also solve the block. The Tor relay may use the same strategy towards Tor clients: it may request to generate PoW with difficulty lower than that indicated by the mining pool in the hope that a client's PoW will also solve the mining pool's share. With this approach the Tor relay may regulate how many tickets are issued to different clients, proportional to their mining power.

Donations. Clients that just want to support Tor relays without requesting any bandwidth can submit shares without requesting anything back.

Implementation considerations. The scheme proposed in this paper requires several modifications to the Tor protocol and bundling Tor with crypto-currencies mining software. It introduces the following new Tor cells:

- `RELAY_MINING_REGISTER`[2]– by sending this message a user asks a Tor relay to send him mining jobs.
- `RELAY_MINING_JOB` – a Tor relay uses this message to send mining jobs to clients.
- `RELAY_TICKET` – used by Tor relays to (1) send material (blind signed) to clients for producing a priority ticket, (2) to notify backup relays about spent tickets; used by Tor clients to send priority tickets and request improved service from a relay.

In addition our scheme requires:

- Replacing currently used Token Bucket algorithm with Hierarchical Token Bucket algorithm.
- Implementing a partially blind signature module.
- Keeping track of spent tickets.
- Synchronizing a relay's spent tickets database with its backup relay.

At the client side, Tor should be bundled with a crypto-currency miner software (e.g. [34] or [43]). At the relay side, Tor should be bundled with both miner and

[2]The described message sequence borrows from Stratum protocol `http://mining.bitcoin.cz/stratum-mining`.

mining pool software (e.g. [63]). Tor control port should also be extended to enable communication between Tor and mining software. Note that it is not necessary to develop new mining software, but rather bundle existing projects with Tor.

6.2 Analysis

6.2.1 Profitability

Motivation. According to the performance statistics maintained by the Tor project[3] [79], it takes roughly between 10 and 15 seconds to download a 5MB file over the Tor network on average (which results in 333 KB/s). While such speeds are likely to be enough for general Web-surfing they might be frustrating for bulk file downloads, watching videos, or having a video conference [58]. The later types of traffic could be the reason why Tor clients may decide to get improved service from Tor relays. This might be especially true for Bittorrent users. Bittorrent over Tor has been problematic for both Bittorrent users and Tor relay operators: users did not get enough speed, and Tor operators are concerned that bulk file downloads consume a lot of bandwidth and thus decrease Quality of Service (QoS) for Web-surfing users.

Another reason why a Tor client would want to have higher capacity/lower delays is to improve QoS for his hidden services. The current version of Tor Hidden Services suffers from high delays and low speeds [57] which significantly reduces the number of users.

Choosing crypto-currencies There are more than 400 different crypto-currencies by September 2014 [44] (however only few of them achieved noticeable market capitalisation and are less susceptible to huge fluctuations in market value towards fiat currencies). According to [42] and [87] the following PoW algorithms are used in existing crypto-currencies: Blake-256, Groestl, HEFTY1, JHA, Keccak, NeoScrypt, Quark, Scrypt, Scrypt-Adaptive-Nfactor, Scrypt-Jane, SHA-256, X11, X13 (see Table 6.1).

Profitability of mining a digital currency obviously depends on the miner's hash-rate, price of electricity, the currency's difficulty, and its current market price. The miner's hash-rate can vary significantly depending on hardware. Table 6.2 shows hash-rates achievable for different algorithms on Intel Core i7-2760QM (4 cores at 2.40GHz). The table also includes maximum revenue[4] for each algorithm for the 1st of September 2014 according to [42] (averaged over multiple observations). Electricity costs are estimated to be 11 cents per day given that max power of the CPU is 45W. During the day we also observed short periods of time when the revenue jumped to 11 cents per day. Also note that hash rates achievable on GPU's can be an order of magnitude higher. We assume that an average user of our protocol does not use ASICs.

Profit estimation. In order to estimate[5] how much a Tor relay can earn using

[3]For June – September 2014.

[4]Revenue can be smaller when trying to exchange due to small market size.

[5]These are of course very rough estimates: it's not possible to learn the current hardware of Tor users, estimate the fraction of non-botnet Tor users, the number of Tor users which would be

Blake-256	BlakeBitcoin Blakecoin Dirac Electron Photon
Groestl	Diamond Groestlcoin
HEFTY1	Heavycoin Mjollnircoin
JHA	JackpotCoin
Keccak	365coin Maxcoin Slothcoin Cryptometh
NeoScrypt	Phoenixcoin
Quark	CNotes Quark Securecoin Animecoin BitQuark Diamondcoin
Scrypt	42 Alphacoin Anoncoin Auroracoin BBQCoin Bitbar Bottlecaps Casinocoin Catcoin CHNCoin CryptogenicBullion DigiByte Digitalcoin DNotes Dogecoin Earthcoin Einsteinium Emerald Fastcoin Feathercoin Franko Globalcoin Goldcoin Grandcoin HoboNickels Infinitecoin Klondikecoin Krugercoin Litecoin Luckycoin Lycancoin Megacoin Mincoin Myriadcoin-Scrypt Nautiluscoin Netcoin Noblecoin Noirbits Novacoin Nyancoin Potcoin Quatloo Razor Reddcoin RonPaulcoin Rubycoin Sexcoin Stablecoin Starcoin Tagcoin Teslacoin USDe Viacoin Worldcoin
Scrypt-Adaptive-Nfactor	Entropycoin Execoin GPUcoin Murraycoin ParallaxCoin SiliconValleyCoin Spaincoin Spots Vertcoin VirtualMiningCoin VertCoin
Scrypt-Jane (Scrypt-Chacha)	YaCoin Ultracoin Velocitycoin
SHA-256	Battlecoin Betacoin BigBullion Bitcoin Bytecoin Curecoin Devcoin eMark Fireflycoin Freicoin Ixcoin Joulecoin Mazacoin Myriadcoin-SHA-256 Namecoin OpenSourcecoin Peercoin SaveCoin Takcoin Teacoin TEKcoin Terracoin Tigercoin Unobtanium Zetacoin
X11	ConspiracyCoin Cryptcoin Darkcoin Fractalcoin GlobalDenomination Hirocoin Karma Logicoin Smartcoin Vootcoin X11coin
X13	MaruCoin BoostCoin X13Coin

Table 6.1: Proof-of-work algorithms and corresponding crypto-currencies

Hashing algorithm	Rate on Intel Core i7-2760QM	Currency	Revenue per day
Blake-256	9,6 Mh/s	Blakecoin	n/a
Groestl	1 Mh/s	Diamond	2.1
HEFTY1	128 Kh/s	Heavycoin	n/a
JHA	308 Kh/s	Jackpotcoin	2.2 cents
Keccak	5.2 Mh/s	Maxcoin	0.7 cents
Quark	300 Kh/s	CNotes	3.8 cents
Scrypt	40 Kh/s	42	0.8 cents
		Litecoin	0.65 cents
		Dogecoin	0.26 cents
Scrypt-N	20 Kh/s	Vertcoin	2.3 cents
Scrypt-Jane	360 h/s	Yacoin	n/a
SHA-256d	9.6 Mh/s	Peercoin	0.01 cents
		Bitcoin	0.008 cents
X11	360 Kh/s	Smartcoin	3.8 cents
		Darkcoin	2.5 cents
X13	104 Kh/s	Marucoin	n/a

Table 6.2: Hash rates of the proof-of-work algorithms on Intel Core i7-2760QM

the proposed scheme we first make the following assumptions:

- Among 2,000,000 daily Tor clients (according to the Tor statistics), only 500,000 are real users and the rest belong to botnets [89]. I.e. only 500,000 users can mine.
- Moreover we assume that each user's session takes about 1 hour and every user is willing to mine with a hash-rate similar to that from Table 6.2. The later implies that clients will spend 100% of CPU on mining during 1 hour period. If clients decide to use less fraction of their CPU, the revenue of a Tor relay will decrease proportionally.

Income of a Tor relay obviously depends on the number of users which establish their circuits through this relay. This in turn depends on the relay's consensus bandwidth. We consider the case in which the scheme motivates running a Tor Exit node (by September 2014 there are only about 1,000 Exits out of 6,000 Tor relays). The green line in Figure 2 shows the income of an Exit relay under the assumption that each client can mine an equivalent of 3.8 cents per 24 hours of which a fraction of 1/24 is received by the relay during a 1 hour session. For such a case top Tor relays (with consensus bandwidth 200,000 KB/s) can earn about 500 USD per month. A middle-tier relay with consensus bandwidth 10,000 KB/s can earn about 25 USD. The green line in Figure 3 shows monthly incomes assuming 11 cents per client per day (in which case a top Tor relay can earn up to 1,600 USD).

willing to mine, and the number of new (Bittorrent over Tor) users.

Running a high-bandwidth Tor relay obviously means high costs. In order to estimate the incurred costs we assume that the rental price is: 25 EUR per month for a relay with consensus weight less than 15,000; 40 EUR for weight between 15,000 and 50,000; 70 EUR for consensus weight larger than 50,000. In addition we assume that 10 TB of traffic is included into the server's price and one has to pay 2 EUR per additional 1 TB [56]. It is important to note that we consider costs which Tor relays already have regardless whether they use the proposed rewarding scheme or not. Note also that in order to compute traffic costs of a relay we take its consensus bandwidth (which represents the relay's speed in KB/s), and assume that the relay constantly transmits with such speed which results in upper bound of traffic costs.

Costs to run an Exit relay of specific bandwidth and corresponding profitability of running such a relay (given the income produced by mining clients) are shown in Figures 2 and 3 with blue and red lines. A Tor relay partially compensates its costs in case of 3.8 cents per day per client; when clients mine an equivalent of 11 cents per day, the relay's costs are lower than its income. Additional income can be used for the server upgrade or to provide better free services.

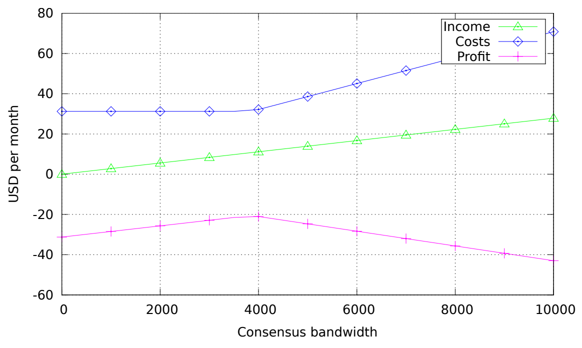


Figure 6.2: Income, costs, and profit of an Exit relay in case of 3.8 cents per day per miner

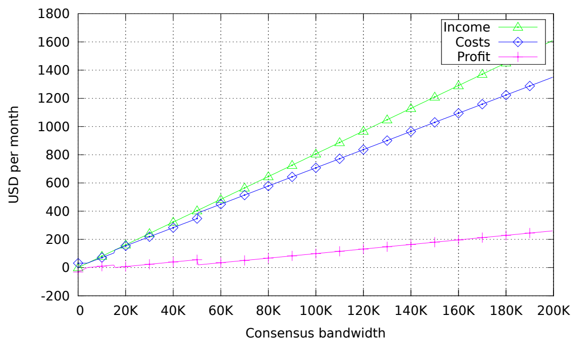


Figure 6.3: Income, costs, and profit of an Exit relay in case of 11 cents per day per miner

6.2.2 Anonymity

In this section we discuss anonymity of the proposed scheme. In Protocol 1, after client C mined a share he sends it to the corresponding Tor relay along with the hash of a random number (to be blindly signed). All communications are done over anonymous circuits, so that the Tor relay does not learn the originator of the messages (unless it is a Guard node). In addition blind signatures prevent the Tor relay from distinguishing client C from other clients. Finally shares generated by client C contain a Bitcoin address of either the Tor relay or a mining pool (the client is even not required to have a crypto-currency account), thus they don't reveal the identity of the client in spite of known attacks against Bitcoin (and hence Altcoins) anonymity [29].

A curious relay can however learn the hash rate of a client, thus it may recognize repeated connections from the same client. In order to mitigate such an attack a client is advised to randomize its hash rate. The same holds if a client decides to pre-mine bandwidth tickets from a relay. In addition, just paying at all identifies the client as somebody who pays for good service, which is a smaller set than the original set of users.

We also note that a powerful miner can try to DoS the paid traffic of a relay, by taking all the paid traffic of a relay for itself. However such behavior is not rational, since it is economically more reasonable for such miner to just earn shares in the mining pool.

Bibliography

[1] P. Mandl. “Sur le comportement asymptotique des probabilites dans les ensembles de etats d’une chaine de Markov homogeene (Russian)”. *Casopis Pest* Mat. 84 (1959), pp. 140–149. (Cit. on p. 67).

[2] E. Seneta. “Quasi-Stationary Behaviour in the Random Walk with Continuous Time”. *Australian Journal of Statistics* 8.2 (1966), pp. 92–98 (cit. on p. 68).

[3] J. N. Darroch and E. Seneta. “On Quasi-Stationary Distributions in Absorbing Continuous-Time Finite Markov Chains”. *Journal of Applied Probability* 4.1 (1967), pp. 192–196 (cit. on p. 67).

[4] W. Feller. *An Introduction to Probability Theory and Its Applications, Vol. 1, 3rd Edition.* Wiley, 1968 (cit. on p. 41).

[5] D. Chaum. “Untraceable electronic mail, return addresses, and digital pseudonyms”. *Communications of the ACM* 24 (1981), pp. 84–88 (cit. on p. 3).

[6] D. Chaum. “Blind Signatures for Untraceable Payments”. *Advances in Cryptology.* Springer US, 1983, pp. 199–203 (cit. on pp. 3, 104).

[7] T. C. May. “The Crypto Anarchist Manifesto” (1992). URL: `http://www.activism.net/cypherpunk/crypto-anarchy.html` (visited on 05/2015) (cit. on p. 3).

[8] E. Hughes. “A Cypherpunk’s Manifesto” (1993). URL: `http://www.activism.net/cypherpunk/manifesto.html` (visited on 05/2015) (cit. on p. 3).

[9] P. Pollett. “Analytical And Computational Methods For Modelling The Long-Term Behaviour Of Evanescent Random Processes”. *12th National Conference of the Australian Society for Operations Research.* 1993, pp. 514–535 (cit. on p. 68).

[10] D. Goldschlag, M. Reed, and P. Syverson. “Hiding Routing information”. *Information Hiding.* Vol. 1174. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, pp. 137–150 (cit. on p. 4).

[11] M. Abe and T. Okamoto. “Provably Secure Partially Blind Signatures”. *Advances in Cryptology – Crypto 2000.* Vol. 1880. Springer Berlin Heidelberg, 2000, pp. 271–286 (cit. on p. 105).

[12] A. K. McCallum. “MALLET: A Machine Learning for Language Toolkit”. 2002. URL: `http://mallet.cs.umass.edu` (visited on 05/2015) (cit. on p. 56).

[13] G. Danezis, R. Dingledine, D. Hopwood, and N. Mathewson. “Mixminion: Design of a Type III Anonymous Remailer Protocol”. *24th IEEE Symposium on Security and Privacy.* IEEE Computer Society, 2003, pp. 2–15 (cit. on p. 4).

[14] R. Dingledine, N. Mathewson, and P. Syverson. “Tor: The Second-Generation Onion Router”. *13th Usenix security symposium.* USENIX Association, 2004 (cit. on p. 5).

[15] P. Golle and A. Juels. “Dining Cryptographers Revisited”. *Advances in Cryptology – Eurocrypt 2004).* Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 456–473 (cit. on p. 5).

[16] L. Øverlier and P. Syverson. “Locating Hidden Servers”. *27th IEEE Symposium on Security and Privacy.* IEEE Computer Society, 2006, pp. 100–114 (cit. on p. 70).

[17] K. S. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. C. Sicker. “Low-resource routing attacks against Tor”. *Workshop on Privacy in Electronic Society.* ACM, 2007, pp. 11–20 (cit. on p. 46).

[18] J. W. Evans and C. Filsfils. *Deploying IP and MPLS QoS for Multiservice Networks: Theory & Practice.* Morgan Kaufmann Publishers Inc., 2007 (cit. on p. 105).

[19] P. Gerrand. “Estimating Linguistic Diversity on the Internet: A Taxonomy to Avoid Pitfalls and Paradoxes”. *Journal of Computer-Mediated Communication* 12.4 (2007), pp. 1298–1321 (cit. on p. 56).

[20] K. Loesing. “Privacy-enhancing Technologies for Private Services”. PhD thesis. University of Bamberg, 2009. URL: `http://www.opus-bayern.de/uni-bamberg/volltexte/2009/183/pdf/loesingopusneu.pdf` (cit. on p. 70).

[21] D. Brown. “Resilient Botnet Command and Control with Tor”. DefCon 18. 2010. URL: `https://www.defcon.org/images/defcon-18/dc-18-presentations/D.Brown/DEFCON-18-Brown-TorCnC.pdf` (cit. on p. 48).

[22] R. Jansen, N. Hopper, and Y. Kim. “Recruiting New Tor Relays with BRAIDS”. *17th Conference on Computer and Communications Security.* ACM, 2010 (cit. on p. 104).

[23] P. Manils, C. Abdelberi, S. L. Blond, M. A. Kaafar, C. Castelluccia, A. Legout, and W. Dabbous. “Compromising Tor Anonymity Exploiting P2P Information Leakage”. *CoRR* abs/1004.1461 (2010) (cit. on p. 100).

[24] P. Syverson. “A peel of onion”. *27th Annual Computer Security Applications Conference.* ACM, 2011, pp. 123–135 (cit. on p. 4).

[25] N. Christin. “Traveling the Silk Road: A measurement analysis of a large anonymous online marketplace”. *CoRR* abs/1207.7139 (2012) (cit. on pp. 5, 48).

[26] T. Elahi, K. Bauer, M. AlSabah, R. Dingledine, and I. Goldberg. "Changing of the Guards: A Framework for Understanding and Improving Entry Guard Selection in Tor". *Workshop on Privacy in the Electronic Society.* 2012, pp. 43–54 (cit. on p. 70).

[27] R. Jansen and N. J. Hopper. "Shadow: Running Tor in a Box for Accurate and Efficient Experimentation". *19th Annual Network & Distributed System Security Symposium.* Internet Society, 2012 (cit. on p. 10).

[28] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. Voelker, and S. Savage. "A Fistful of Bitcoins: Characterizing Payments Among Men with No Names". *Internet Measurement Conference.* ACM, 2013 (cit. on pp. 71, 76).

[29] D. Ron and A. Shamir. "Quantitative analysis of the full Bitcoin transaction graph". *Financial Cryptography and Data Security.* Springer, 2013 (cit. on pp. 76, 112).

[30] A. Biryukov, D. Khovratovich, and I. Pustogarov. "Deanonymisation of Clients in Bitcoin P2P Network". *Conference on Computer and Communications Security.* ACM, 2014, pp. 15–29 (cit. on p. 98).

[31] F. Thill. *Hidden Service Tracking Detection and Bandwidth Cheating in Tor Anonymity Network.* Master thesis. University of Luxembourg. 2014. URL: `https://www.cryptolux.org/images/b/bc/` (cit. on p. 96).

[32] Anonymous. *Best VPN's using Bitcoin.* URL: `https://bitcointalk.org?topic=247212.0` (visited on 05/2015) (cit. on p. 87).

[33] Anonymous. *IAmA a malware coder and botnet operator, AMA.* URL: `http://www.reddit.com/r/IAmA/comments/sq7cy/iama_a_malware_coder_and_botnet_operator_ama` (visited on 05/2015) (cit. on p. 47).

[34] *ASIC and FPGA miner in c for bitcoin.* URL: `https://github.com/ckolivas/cgminer` (visited on 05/2015) (cit. on p. 107).

[35] K. Atlas. *Historical record of Tor exit nodes used to connect to the Bitcoin.* URL: `http://www.openbitcoinprivacyproject.org/torban/www/` (visited on 05/2015) (cit. on p. 96).

[36] A. Back. *Wei Dei's "b-money" protocol.* Cypherpunks Archives. URL: `http://cypherpunks.venona.com/date/1998/12/msg00194.html` (visited on 07/2015) (cit. on p. 6).

[37] *Bitcoin Wiki.* URL: `https://en.bitcoin.it` (visited on 05/2015) (cit. on p. 10).

[38] *Bitnodes.* URL: `https://getaddr.bitnodes.io/` (visited on 05/2015) (cit. on pp. 90, 99).

[39] *Bitnodes Source Code.* URL: `https://github.com/ayeowch/bitnodes` (visited on 05/2015) (cit. on p. 77).

[40] *BlockChain.info Charts.* URL: `https://blockchain.info/charts` (visited on 05/2015) (cit. on p. 86).

[41] A. Chen. *The Underground Website Where You Can Buy Any Drug Imaginable.* URL: `http://gawker.com/the-underground-website-where-you-can-buy-any-drug-imag-30818160` (visited on 05/2015) (cit. on p. 8).

[42] *CoinWars: Crypto Currencies.* URL: `http://www.coinwarz.com` (visited on 05/2015) (cit. on p. 108).

[43] *CPU miner for bitcoin.* URL: `https://github.com/jgarzik/cpuminer` (visited on 05/2015) (cit. on p. 107).

[44] *Crypto-Currency Market Capitalizations.* URL: `http://coinmarketcap.com` (visited on 09/2014) (cit. on p. 108).

[45] *DigiCash.* URL: `http://www.chaum.com/projects/eCash/ecash.html` (visited on 07/2015) (cit. on p. 6).

[46] *Dissent accountable anonymous group communication.* URL: `http://dedis.cs.yale.edu/dissent` (visited on 07/2015) (cit. on p. 6).

[47] K. Dowd. *Contemporary Private Monetary Systems.* URL: `http://www.kevindowd.org/working-papers` (visited on 05/2015) (cit. on p. 8).

[48] *Download Bitcoin Core.* URL: `https://bitcoin.org/en/download` (visited on 05/2015) (cit. on p. 100).

[49] *DuckDuckGo Search Engine.* URL: `https://duckduckgo.com` (visited on 05/2015) (cit. on p. 5).

[50] *E-gold Statistics. Internet Archive.* Accessed on May 2015. URL: `https://web.archive.org/web/20040711020115/http://www.e-gold.com/stats.html` (cit. on p. 7).

[51] *Fallback Nodes.* URL: `https://en.bitcoin.it/wiki/Fallback%5C_Nodes` (visited on 05/2015) (cit. on p. 97).

[52] *Free Haven.* URL: `http://www.freehaven.net` (visited on 07/2015) (cit. on p. 3).

[53] *Freenet. The Free Network.* URL: `https://freenetproject.org/` (visited on 07/2015) (cit. on p. 3).

[54] *G Data Security Blog. Botnet command server hidden in Tor.* URL: `http://blog.gdatasoftware.com/blog/article/botnet-command-server-hidden-in-tor.html` (visited on 05/2015) (cit. on p. 48).

[55] C. Guarnieri. *Skynet, a Tor-powered botnet straight from Reddit.* URL: `https://community.rapid7.com/community/infosec/blog/2012/12/06/skynet-a-tor-powered-botnet-straight-from-reddit` (visited on 05/2015) (cit. on pp. 5, 48, 54).

[56] *Hetzner Online Server Auction.* URL: `https://robot.your-server.de/order/market` (visited on 05/2015) (cit. on pp. 85, 111).

[57] *Hidden Services need some love.* URL: `https://blog.torproject.org/blog/hidden-services-need-some-love` (visited on 05/2015) (cit. on p. 108).

[58] *How much bandwidth does Skype need?* URL: `https://support.skype.com/en/faq/FA1417/how-much-bandwidth-does-skype-need` (visited on 05/2015) (cit. on p. 108).

[59] *I2P. The invisible Internet Project.* URL: `https://geti2p.net` (visited on 07/2015) (cit. on p. 5).

[60] B. Krebs. *U.S. Government Seizes LibertyReserve.com.* URL: `https://krebsonsecurity.com/2013/05/u-s-government-seizes-libertyreserve-com` (visited on 05/2015) (cit. on p. 8).

[61] *Linux HTB Home Page.* URL: `http://luxik.cdi.cz/~devik/qos/htb/` (visited on 05/2015) (cit. on p. 105).

[62] D. H. Lipman. *Mailing list post: [tor-talk] vwfws4obovm2cydl.onion.* URL: `https://lists.torproject.org/pipermail/tor-talk/2012-June/024565.html` (visited on 05/2015) (cit. on p. 47).

[63] Luke-Jr. *Eloipool - FAST Python3 pool server.* URL: `https://bitcointalk.org/index.php?topic=61731.0` (visited on 05/2015) (cit. on p. 108).

[64] *Mixmaster.* URL: `http://mixmaster.sourceforge.net/` (visited on 07/2015) (cit. on p. 4).

[65] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* URL: `https://bitcoin.org/bitcoin.pdf` (visited on 07/2015) (cit. on p. 6).

[66] S. Nakatani. *Language Detection Library for Java.* URL: `http://code.google.com/p/language-detection/` (visited on 07/2015) (cit. on p. 56).

[67] *OnionCat An Anonymous VPN-Adapter.* URL: `https://www.onioncat.org/about-onioncat` (visited on 05/2015) (cit. on p. 24).

[68] *Password Hashing Competition.* URL: `https://password-hashing.net` (visited on 05/2015) (cit. on p. 27).

[69] *Projects for TorFlow.* URL: `https://trac.torproject.org/projects/tor/wiki/org/roadmaps/TorFlow` (visited on 07/2015) (cit. on pp. 15, 46).

[70] *RFC 2212. Specification of Guaranteed Quality of Service.* URL: `http://www.rfc-editor.org/rfc/rfc2212.txt` (visited on 05/2015) (cit. on p. 15).

[71] J. Roy. *Feds Raid Online Drug Market Silk Road.* URL: `http://nation.time.com/2013/10/02/alleged-silk-road-proprietor-ross-william-ulbricht-arrested-3-6m-in-bitcoin-seized/` (visited on 07/2015) (cit. on p. 8).

[72] C. E. Schumer and J. Manchin. *Press Release: Manchin Urges Federal Law Enforcement to Shut Down Online Black Market for Illegal Drugs.* URL: `http://manchin.senate.gov/public/index.cfm/2011/6/manchin-urges-federal-law-enforcement-to-shut-down-online-black-market-for-illegal-drugs` (visited on 05/2015) (cit. on p. 48).

[73] *Selected Papers in Anonymity.* URL: `http://www.freehaven.net/anonbib/date.html` (visited on 07/2015) (cit. on p. 3).

[74] *Terremark vCloud Express.* URL: http://vcloudexpress.terremark.com/pricing.aspx (visited on 05/2015) (cit. on pp. 89, 96).

[75] *The New Yorker Strongbox.* URL: http://www.newyorker.com/strongbox (visited on 05/2015) (cit. on p. 5).

[76] *Tor. Anonymity Online.* URL: https://www.torproject.org (visited on 07/2015) (cit. on p. 5).

[77] *Tor Consensuses archive.* URL: https://collector.torproject.org/archive/relay-descriptors/consensuses/ (visited on 07/2015) (cit. on p. 15).

[78] *Tor directory protocol, version 3.* URL: https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt (visited on 07/2015) (cit. on p. 46).

[79] *Tor Metrics: Performance.* URL: https://metrics.torproject.org/performance.html (visited on 05/2015) (cit. on p. 108).

[80] *Tor Path Specification.* URL: https://gitweb.torproject.org/torspec.git/tree/path-spec.txt (visited on 07/2015) (cit. on pp. 15, 46).

[81] *Tor Protocol Specification.* URL: https://gitweb.torproject.org/torspec.git/tree/tor-spec.txt (visited on 07/2015) (cit. on p. 10).

[82] *Tor Release Notes. Changes in version 0.2.4.23 - 2014-07-28.* URL: https://gitweb.torproject.org/tor.git/plain/ReleaseNotes?id=tor-0.2.4.23 (visited on 07/2015) (cit. on p. 15).

[83] *Tor Rendezvous Specification.* URL: https://gitweb.torproject.org/torspec.git/tree/rend-spec.txt (visited on 07/2015) (cit. on pp. 5, 19).

[84] *Tor source code.* URL: https://gitweb.torproject.org/tor.git (visited on 07/2015) (cit. on p. 5).

[85] *uClassify web service.* URL: http://www.uclassify.com (visited on 07/2015) (cit. on p. 56).

[86] *Wikileaks.* URL: wikileaks.org (visited on 05/2015) (cit. on p. 5).

[87] *Windows GPU Miners for the More Commonly Used Crypto Algorithms.* URL: http://cryptomining-blog.com/2595-windows-gpu-miners-for-the-more-commonly-used-crypto-algorithms (visited on 05/2015) (cit. on p. 108).

[88] K. Zetter. *How the Feds Took Down the Silk Road Drug Wonderland.* URL: http://www.wired.com/2013/11/silk-road (visited on 05/2015) (cit. on p. 8).

[89] Z. Zorz. *Massive spike of Tor users caused by Mevade botnet.* URL: http://www.net-security.org/secworld.php?id=15530 (visited on 05/2015) (cit. on p. 110).

List Of Publications

[1] A. Biryukov, I. Pustogarov, and R.-P. Weinmann. “TorScan: Tracing Long-Lived Connections and Differential Scanning Attacks”. *17th European Symposium on Research in Computer Security (ESORICS 2012).* Vol. 7459. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 469–486.

[2] A. Biryukov, I. Pustogarov, and R.-P. Weinmann. “Trawling for Tor Hidden Services: Detection, Measurement, Deanonymization”. *34th IEEE Symposium on Security and Privacy (S&P 2013).* IEEE Computer Society, 2013, pp. 80–94.

[3] A. Biryukov, D. Khovratovich, and I. Pustogarov. “Deanonymisation of Clients in Bitcoin P2P Network”. *21st ACM Conference on Computer and Communications Security (CCS 2014).* New York, NY, USA: ACM, 2014, pp. 15–29.

[4] A. Biryukov, I. Pustogarov, F. Thill, and R.-P. Weinmann. “Content and Popularity Analysis of Tor Hidden Services”. *34th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW 2014).* IEEE Computer Society, 2014, pp. 188–193.

[5] A. Biryukov and I. Pustogarov. “Bitcoin over Tor isn’t a good idea”. *36th IEEE Symposium on Security and Privacy (S&P 2015).* IEEE Computer Society, 2015, pp. 122–134.

[6] A. Biryukov and I. Pustogarov. “Proof-of-Work as Anonymous Micropayment: Rewarding a Tor Relay”. *19th International Conference Financial Cryptography and Data Security (FC 2015).* Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015, pp. 445–455.