A New Approach to Deanonymization of *Unreachable* Bitcoin Nodes

Indra Deep Mastan[1] and Souradyuti Paul[2]

[1] Indian Institute of Technology Gandhinagar
`immastan@gmail.com`
[2] Indian Institute of Technology Bhilai
`souradyuti.paul@gmail.com`

Abstract. Mounting deanonymization attacks on the *unreachable* Bitcoin nodes – these nodes do not accept incoming connections – residing behind the NAT is a challenging task. Such an attack was first given by Biryukov, Khovratovich and Pustogarov based on their observation that a node can be uniquely identified in a *single session* by their directly-connected neighbouring nodes (ACM CCS'15). However, the BKP15 attack is less effective across *multiple sessions.* To address this issue, Biryukov and Pustogarov later on devised a new strategy exploiting certain properties of address-cookies (IEEE S&P'15). Unfortunately, the BP15 attack is also rendered ineffective by the present modification to the Bitcoin client.

In this paper, we devise an efficient method to link the sessions of *unreachable* nodes, even if they connect to the Bitcoin network over the Tor. We achieve this using a new approach based on organizing the *block-requests* made by the nodes in a *Bitcoin session graph.* This attack also works against the modified Bitcoin client. We performed experiments on the *Bitcoin main network*, and were able to link consecutive sessions with a *precision* of 0.90 and a *recall* of 0.71. We also provide counter-measures to mitigate the attacks.

1 Introduction

Bitcoin works in a peer-to-peer network over which users create transactions that are stored in a distributed ledger known as the Blockchain [9]. All transactions in the Bitcoin network can be publicly viewed and analyzed. One of the most important properties of Bitcoin is its anonymity. If an adversary is able to link the transactions to its owner, then she has broken the anonymity property; this event is known as deanonymization.

Let $U = \{U_1, U_2, \ldots, U_n\}$ be the set of addresses of the user u (note that Bitcoin allows a user to have multiple addresses). If an adversary is able to find U, then she will be able to get the full transaction history of u from the Blockchain (publicly available distributed ledger). In this setting, there are two main problems associated to deanonymization. First, how to link the bitcoin addresses (or transactions) to determine U? Next, how to link U to the real identity u? Our work is in the direction of the first problem.

Motivation Deanonymization attack is performed mainly in two ways: *transaction graph* analysis and *Bitcoin network* analysis. Several papers show how to perform *transaction graph analysis* to link the transactions (and thus Bitcoin addresses) of users [8,11]. Some papers even link real world entities such as *mtgox*, *silkroad* with their Bitcoin addresses [8,10]. Biryukov, Khovratovich and Pustogarov observed that deanonymization using *transaction graph analysis* is less effective when a user makes multiple transactions using *distinct* bitcoin addresses, since such transactions might not have any relations in the graph [4].

Now we concretely discuss the main challenges in deanonymization with respect to Bitcoin network analysis. Suppose, a victim node v having the public IP address IP creates a transactions tx. Even if the adversary discovers that tx is related to IP, he is still not sure about the owner of tx, because there could be multiple Bitcoin nodes (or *users*) having the same public IP IP behind the NAT. Therefore, an adversary first needs to distinguish the nodes behind the NAT, before linking the transactions. Another challenge in the deanonymization is when a user connects to Bitcoin over Tor: the user can change its onion address in a new session, making it difficult for the attacker to trace his activities.[3]

Many of the deanonymization problems in Bitcoin network have already been solved in [4,5,7]. These attacks have the following limitations: attacks given in [7] do not apply to unreachable Bitcoin nodes; attacks given in [4] are not performed on the Bitcoin *main network*, and can not deanonymize nodes across the sessions; the *address-cookies* method given in [5] is ineffective in the *updated* version of the Bitcoin client. In short, it is not satisfactorily solved as to how to deanonymize Bitcoin nodes, when the victim nodes behind the NAT are unreachable or are using Bitcoin over Tor.

In this paper we solve this issue using a novel technique based on analysing the sequence of block-header hashes (block-ids) requested across multiple sessions by the unreachable nodes.

Related work Following are various attempts at deanonymization attacks and their limitations, in chronological order.
2014: Koshy et al. have shown how to perform deanonymization by analyzing the relay patterns of transactions [7]. They collected data over 5 months, and used statistical analysis to determine the source IP addresses of the transactions. They were able to deanonymize 1162 bitcoin addresses of *reachable* Bitcoin nodes. However, in [4], it was pointed out that their attack applied to only *reachable* nodes that constituted only 10% of all nodes.
2014: Biryukov, Khovratovich and Pustogarov gave the first attack to deanonymize transactions of nodes that are *unreachable* and hidden behind the NAT [4]. They performed experiments on the *Bitcoin Testnet*. Their attack was based on the observation that a node can be uniquely identified by its direct connections (or

[3] Tor is a circuit-based communication service which provides anonymity by relaying traffic through routers as proxies (see Sect. 2).

entry-nodes). They gave a strategy to learn the entry-nodes; however, their solution has an inherent limitation that the entry-nodes change in a new Bitcoin session. Thus, their attack can not relate the transactions created in the multiple sessions. This shows that the ability to identify a node across the sessions is an essential step of any deanonymization attack.
2015: Biryukov and Pustogarov gave a fingerprinting technique for identification of the nodes across the sessions [5]. The technique is as follows: adversary sends unique *address-cookies* to his peers in ADDR messages, the peers store IP addresses contained in *address-cookies* into their IP address tables (address-cookies are created into peers IP address tables); after some time, adversary sends GETADDR to query IP addresses in the peers IP address tables; the peers respond using ADDR; now the adversary can analyse the responses received, and check if it matches some *address-cookies*. The present modification to Bitcoin's inbuilt fingerprinting protection[4] makes *unreachable* nodes ignore the GETADDR; thus, the fingerprinting attack is prevented for *unreachable* nodes.

Our contribution Our main contribution is launching a deanonymization attack on *unreachable* Bitcoin nodes, even if they are behind the NAT, in both direct connection and proxy connection settings (running Bitcoin over Tor). Most importantly, unlike the previous attacks, our technique works against the new version of Bitcoin client [3]. Our attack is fairly generic, and does not seem to exploit any rectifiable mistake in the Bitcoin implementation. The crux of the attack is an observation that a Bitcoin node requests for blocks following a specific pattern, in particular, in the increasing order of Blockchain height. This pattern is observable even in the following scenarios: when the node is connected via Tor; and when the node is connected in multiple sessions with or without Tor. Using this observation we linked the consecutive sessions (sessions that follow each other continuously) of an unreachable node, which could then be used to link majority of the sessions. Linked sessions help in linking the transactions created in the different sessions.

The above attack has been experimentally verified in the Bitcoin main network. We have performed experiments by running Bitcoin nodes on Amazon EC2.

The main objective of the first experiment is to give a concrete measure of the quality of the attack when the victim nodes are connected to Bitcoin network directly. We ran eight sessions with the four *unreachable* nodes (therefore, a total of 32 sessions). In this experiment, we link the consecutive sessions with a *precision* of 0.90 and a *recall* of 0.71.

The objective of the second experiment is to show the performance of the attack when victim nodes connect to Bitcoin network with Tor and without Tor in different sessions. We ran six sessions with four nodes (therefore, a total of 24 sessions), where, in the first three sessions, nodes are connected to Bitcoin network over Tor, and, in the next three sessions the nodes are connected di-

[4] The modification that are done to provide inherent fingerprinting protection to Bitcoin network.

rectly to Bitcoin network. In the experiment, we link consecutive sessions with a *precision* of 1.0 and a *recall* of 0.75. To thwart this attack, we propose a counter-measure, where the blocks are requested in a random fashion.

2 Background

Here we first give necessary background of Bitcoin network, nodes and Bitcoin protocol messages. Next, we describe the Tor network.

2.1 Bitcoin network

Bitcoin network is a peer-to-peer network. A node in the Bitcoin network can have at most 8 outgoing and 117 incoming connections. The nodes get connected to each other by establishing a TCP connection.

Nodes. There are mainly two classes of Bitcoin nodes: (1) first one is based on the *reachability* criterion, and (2) the second one is based on the existence of proxy nodes in the connection. The first class of nodes is further divided into two types: (1a) nodes that accept incoming connections, we call them *reachable* nodes, and (1b) nodes that do not accept incoming connections, we call them *unreachable* nodes. Both *reachable* and *unreachable* nodes can make outgoing connections. The second class of nodes is also subdivided into three categories: (2a) nodes that connect to Bitcoin network directly, (2b) nodes that connect to Bitcoin network using Tor anonymity system, and (2c) nodes that connect to Bitcoin network sometimes with Tor and sometimes without Tor.

Bitcoin protocol messages. Bitcoin protocol uses a large number of application layer messages that are exchanged between the nodes for various purposes. Below we describe the important messages.

- `VERSION` & `VERACK`: In the beginning of a connection, two nodes exchange `VERSION` and `VERACK` messages. The `VERSION` message contains information of the Bitcoin version, best-height[5] of the sender, a random nonce, network addresses of the sender and receiver, etc. The `VERACK` message sent from the receiver denotes acknowledgement of `VERSION` message.
- `GETADDR` & `ADDR`: Every node normally holds a list of IP addresses that are working as active nodes in the network in the recent past. Using `GETADDR` message, a node X can request another node Y for the list of those IP addresses. This is done to help X find the potential active nodes in the network. In response, Y returns the requested list of IP addresses using `ADDR` message.
- `PING` & `PONG`: The `PING` is sent to check the status of the connection is alive. The `PONG` is sent in response to a `PING` message.

[5] The best-height of a node is the height of the Blockchain of the node.

- `INV`: A node advertises suitably chosen transactions and blocks it possesses to its peers using `INV` messages. It is a tuple *(count,inventory)*, where count is the size of *inventory* and *inventory* is a list of inventory vectors. Each inventory vector is a tuple *(type,hash)*, where *type* identifies the object type; i.e. transaction or block, and *hash* denote transaction-id or block-id (block header hash). The `INV` message can be issued by a node unsolicited.
- `GETDATA`: The `GETDATA` message is sent in the response to an `INV` message. It is a request to retrieve the full content of specific transactions or blocks[6]. Similar to `INV`, the `GETDATA` is also a tuple *(count,inventory)*.
- `TX`: It describes a bitcoin transaction. When a Bitcoin user create a new transaction, it broadcasts the transaction-id in `INV` message. The peers connected to the user receive the `INV` messages and get the transaction-id of new transaction, next, they request the transaction by sending the `GETDATA` message for it. Then user sends transaction in `TX` message.
- `BLOCK`: It describes a block. The `BLOCK` message sent for two different reasons: (1) sent as response to the `GETDATA` message, and (2) sent by miners to broadcast newly-mined blocks.
- `GETBLOCKS`: A `GETBLOCKS` message is exchanged between peers to tell each other the block-ids of the top block on their Blockchain, this helps in updating their Blockchain. For example, suppose node X and Y have exchanged `GETBLOCKS` message, and Blockchain height of node X is more than Y. Since `GETBLOCKS` sent from Y contains the block-id of the block at the top of Blockchain of Y, X will determine the set of blocks that Y needs in-order to update his Blockchain. Next, X sends `INV` message containing upto 500 block-ids to Y, and then Y can request the desired blocks using `GETDATA` message. This way of synchronizing Blockchain is called Blocks-First Sync.
- `GETHEADERS` & `HEADERS`: These messages are exchanged between peers to update the block headers[7]. The `GETHEADERS` message contains the block-id from where the sender wants to receive the headers. When a peer sends `GETHEADERS` message, it gets a `HEADERS` message as a response. The `HEADERS` message contains up to 2000 block headers. Note that similar to `GETBLOCKS`, the peer with higher Blockchain height sends `HEADERS` message. This way of synchronizing Blockchain is called Headers-First Sync. After updating the block headers a node can request the `GETDATA` message for the blocks.

2.2 The onion router (Tor)

Tor is a circuit-based communication service which provides anonymity by relaying traffic through routers in the Tor network as proxies [6]. Tor network is

[6] Node X advertises blocks and transactions to node Y using `INV`, where `INV` contains block header hashes (block-ids) or transaction-ids. Then Y requests specific transactions or blocks from X using `GETDATA`; such a communication is called *pull-based communication*

[7] Each block has a 80-byte block header, which contains important information such as the hash value of the previous block, the time of creation of the block, a nonce, number of transactions etc.

a distributed overlay network, which consists of approximately 7,000 volunteer-operated routers or *onion routers* (ORs) or *relays.*

When a user runs Tor client, it creates a *Tor circuit* to route the traffic through the Tor network by choosing three relays – namely entry, middle and exit – and establishes a session key with each relay. Suppose, Alice is using a Tor client to connect to Bob. When Alice starts the Tor client in her machine, it creates the following three-hop circuit.

$$\text{Alice} \leftrightarrow \text{entry relay} \leftrightarrow \text{middle relay} \leftrightarrow \text{exit relay.}$$

Next, the Tor client sends *data* to Bob by encapsulating it in three layers of encryption, using the session keys established with the relays. Each relay in the circuit removes its layer by performing decryption, and finally Bob receives data in the unencrypted form.

Each relay in the circuit knows the IP addresses of its predecessor and successor. The entry relay knows the IP address of Alice, but does not know the *data* Alice is sending. Exit relay knows the *data* sent by Alice, but does not know the IP address of Alice. Therefore, none of the relays (as well as Bob) can relate the *data* with the IP address of Alice.

The three-hop Tor circuit does not provide anonymity to Bob, because the IP address of Bob is known to Alice; however, using *Tor Hidden Service* (THS) Bob can hide its IP address from Alice while offering a TCP service, e.g. a *Bitcoin server.*

The THS is accessed through its *onion* address rather than IP address, which is of the form "x.onion", where x is the base-32 encoded THS identifier.[8] The onion address – as opposed to IP address – does not reveal geographical information. The Tor client routes data *to and from* THS using the onion address. For example, suppose Bob is running a THS, and suppose Alice wants to use it. They construct the following circuit to exchange data, where *RP* is a Tor relay, also known as *Rendezvous Point* [6].

$$\text{Alice} \leftrightarrow \text{Relay} \leftrightarrow \text{Relay} \leftrightarrow \text{RP} \leftrightarrow \text{Relay} \leftrightarrow \text{Relay} \leftrightarrow \text{Relay} \leftrightarrow \text{Bob}$$

RP connects Alice's circuit to Bob's circuit; it does not know the IP address of Alice and Bob and the *data* they exchange.

3 Peer-Representations and Sessions

Here we give important definitions required to formalise our deanonymization attack. Our main focus is to give a strategy to link the sessions of an identical node. Linking sessions of a node enables the attackers to trace the activities of the user and monitor its transactions.

In our attack model, victim nodes are assumed to be inside the NAT, whereas the adversarial nodes are outside of it. Nodes inside the NAT connect to Bitcoin

[8] Base-32 encoding is done using 32-character: twenty-six letters A to Z and six digits 2 to 7.

network mainly in two ways: Direct connection and Proxied connection (Bitcoin over Tor). The directly connected nodes share the same public IP, making distinguishing difficult outside the NAT. The Bitcoin nodes over Tor may not share the same onion address; however, they could change their onion addresses (e.g. opening new Bitcoin sessions), making tracing difficult.

Let $\mathcal{A} = \{a_i\}_{i\in[n]}$ and $\mathcal{V} = \{v_i\}_{i\in[m]}$ denote the sets of adversarial and victim nodes. Two nodes are called *peers* of each other, if they are connected. In the Bitcoin network, an attacker node a_j identifies a peer victim node v_i by assigning it a *peer-id* p_{ij}. Note that a victim node may be assigned different peer-ids by different attacker nodes, making it harder for the attacker to determine whether the peer-ids actually are of a single node or of multiple nodes behind the NAT.

To represent the victim by a single representation outside the NAT in one session, we provide a *peer-representation* based on the time at which a victim is disconnected. Let T_v denote the set of all *disconnect times* of the victim v from the Bitcoin network (Suppose, v comes online k times; therefore, $T_v = \{t_v^1, t_v^2, \dots, t_v^k\}$). Also, let $addr(v)$ be the set of public addresses by which v is identified outside the NAT. (Suppose, IP is the public IP address of the victim node v. Also, suppose that $o_v^1, o_v^2 \dots o_v^k$ are the onion addresses of v. Therefore, $addr(v) = \{IP, o_v^1, o_v^2 \dots o_v^k\}$.)

The set of all *peer-representations* of all victim nodes is defined as follows.

$$\mathcal{A}.peers = \{(a,t) : v \in \mathcal{V}, t \in T_v,\ a \in addr(v), a\ \ disconnects\ \ at\ t\}.$$

Note that $\mathcal{A}.peers$ contains the *peer-representations* different from *peer-ids*; however, to compute $\mathcal{A}.peers$, adversary needs *peer-ids*. The technical details of how $\mathcal{A}.peers$ is computed is provided in Appendix A. In the Figure 1 a pictorial representation of relation between $\mathcal{A}.peers$ and $\mathcal{V}$ is given.

A *session* (or Bitcoin session) is the collection of the Bitcoin protocol messages exchanged between the times a node connects to Bitcoin network and disconnect from it. Suppose, a victim represented by $x \in \mathcal{A}.peers$ comes online and exchanges various Bitcoin protocol messages, let S_x be the set of messages exchanged with x; therefore, S_x is a *session*. We define $\mathcal{A}.data$ as follows:

$$\mathcal{A}.data = \{S_x : x \in \mathcal{A}.peers\}.$$

The set $\mathcal{A}.data$ contains all the *sessions* of the victims. For example, the set $S_{(IP,t_{v_1}^1)}$ contains Bitcoin protocol messages exchanged with victim v_1; thus, $S_{(IP,t_{v_1}^1)}$ is a *session* of v_1. It is easy to establish a bijection $S : \mathcal{A}.peers \to \mathcal{A}.data$, which shows that for a victim $x \in \mathcal{A}.peers$, its *session* $S(x)$ is contained in $\mathcal{A}.data$. We shall use $S(x)$ and S_x synonymously. The technical details of how S_x is computed is provided in Appendix A.

4 A New Form of Deanonymization: Linking the Sessions

Here we give the necessary background for our deanonymization attack using the definitions of the *peer-representation* and *session* as described in Sect. 3. First

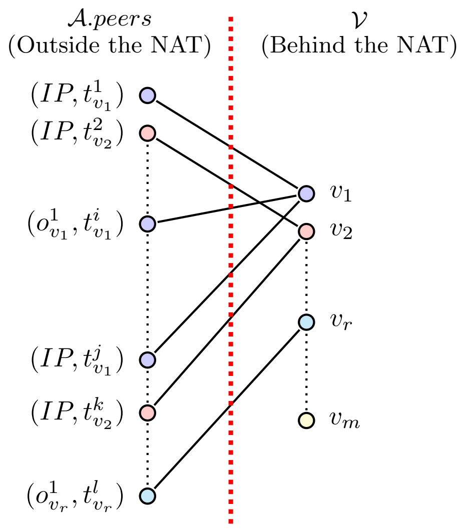

Fig. 1: Representations of victim nodes behind and outside the NAT. For example, the node v_1 is identified as $(IP, t_{v_1}^1)$ in one session (direct connection) and $(o_{v_1}^1, t_{v_1}^j)$ in another session (a Bitcoin over Tor connection), here IP and $o_{v_1}^1$ denote the public IP address and onion address of v_1; $t_{v_1}^1$ and $t_{v_1}^j$ denote the disconnect times.

we describe why linking the *sessions* is a deanonymization attack, and then we outline the major steps.

Why linking sessions is a deanonymization attack The first deanonymization attack of nodes behind the NAT was given in [4], where the adversary logs the first 10 nodes broadcasting the transaction-ids in `INV` messages, and then assigns the transactions to a node behind the NAT; however, their attack fails to link transactions created in different sessions.

We now give an example of how linking of sessions helps in deanonymizing transactions. Suppose, a user creates transactions T_1 and T_2 in sessions s_1 and s_2. After creating the transactions, he broadcasts them to his peers. Suppose, the user is connected to adversarial nodes; therefore, they receive T_1 and T_2. The adversary determines that T_1 was first broadcast in s_1, and T_2 in s_2. Next, the adversary checks if s_1 and s_2 are of identical victim. If so, she then concludes that T_1 and T_2 were created by the same user; this way he is able to link transactions created in different sessions.

Let $\gamma_i \subseteq \mathcal{A}.data$ contains the *sessions* (sets of Bitcoin protocol messages) of a victim v_i. If adversary is able to link *sessions* of v_i and compute γ_i, then he will be able to link transactions of v_i.

Linking of *sessions* also gives an additional interesting result. If adversary is able to compute γ_i, then he can also get the set of *peer-representations* α_i of the victim v_i outside the NAT. For example, in Figure 1, the node v_1 is represented by the set of *peer-representations* $\alpha_1 = \{(IP, t^1_{v_1}), (o^1_{v_1}, t^i_{v_1}), (IP, t^j_{v_1})\}$. Suppose, the adversary is able to link the sessions of v_1, and to compute γ_1, where

$$\gamma_1 = \{S(IP, t^1_{v_1}), S(o^1_{v_1}, t^i_{v_1}), S(IP, t^j_{v_1})\}.$$

Each *session* $S(x) \in \gamma_1$ is a set of Bitcoin protocol messages, that contains information on IP address or onion address of the victim, and also the time of disconnect (see Sect. 2); thus, the adversary can determine *peer-representations* x from the Bitcoin protocol messages in $S(x)$. The adversary can compute α_1 using γ_1. Hence, by linking *sessions* of v_1, it is also possible to achieved identification of v_1 when it is connected to Bitcoin network directly, and when it is connected to Bitcoin network over Tor.

Major steps In our attack, we analyse the `GETDATA` messages sent by the victims. Below are the major steps of the attack.

1. *Extracting the Block-ids:* We compute the block-ids requested by the victims in each session using the `GETDATA` messages sent by them.
2. *Linking consecutive sessions:* We take two sequences of block-ids, and determine if they are requested in the consecutive sessions of a node (linking consecutive sessions).
3. *Linking sll the sessions:* To link all the sessions of a victim, we define a *Bitcoin session graph*, where each vertex represents a sequence of block-ids requested by the victim in a session; and two vertices have an edge if they are related to the consecutive sessions. The vertices of the maximally connected component of the graph gave the sequences of block-ids requested by the victim; which in turn gives all the sessions of the victim node.

In what follows, we describe above steps in detail.

4.1 Step 1: extracting the block-ids

Here we describe the first step of our attack. We focus on the analysing the block-ids (block header hashes) requested by the victims. We first give the motivation for extracting block-ids, and then show how to extract them.

Motivation The Bitcoin protocol messages sent by a victim contain `GETDATA` messages. A `GETDATA` message contains the list of block-ids the victim *does not* have at a specific time (see Sect. 2). Each block-id is associated with a unique block, and each block has a unique height in the Blockchain. By analysing the block-ids in `GETDATA` messages issued by the victim, adversary can get two pieces of important information: First, estimate of the Blockchain height of victim at a specific time; second, the block-ids of the blocks that the victim has updated into its Blockchain. Below we describe how they are useful for adversary.

The estimate of Blockchain height of a node can help in linking the consecutive sessions of the node: if adversary gets the Blockchain height of the victim v_i when it disconnects, then, in the new session, v_i starts requesting blocks from the height achieved in previous session; the height achieved in previous session and starting height of new session of v_i are equal. Therefore, the height of the blocks requested by v_i in the beginning of a new session will be close to the height of requested blocks, when v_i disconnected in previous session; thus, adversary can compare the block-requests, and identify victim v_i in the new session.

The block-ids requested by the victims can help in distinguishing their sessions: a node does not request blocks after they are updated into its Blockchain; however, two nodes can request same blocks; thus, by comparing the block-ids requested in the two sessions, adversary can determine if sessions correspond to a single (or two different) victim node(s).

Extracting block-ids The set S_x contains the `GETDATA` messages issued by a victim whose peer-representation is x. When a victim sends `GETDATA` message to retrieve a block, the adversary sends `BLOCK` message in the response. A `GETDATA` message may contain a maximum of 50,000 entries for blocks or transaction ids [1]. Let E denote the algorithm that, given the element $S \in \mathcal{A}.data$, outputs the multiset $E(S)$ containing the block-ids of the blocks requested in S. We define the set $\mathcal{A}.SessionBid$ as follows:

$$\mathcal{A}.SessionBid = \{E(S) : S \in \mathcal{A}.data\}. \tag{1}$$

Another way of formalization is:

$$\begin{aligned}\mathcal{A}.SessionBid &= \{E(S_x) : x \in \mathcal{A}.peers\} \\ &= \{E(S_x) : x \in \bigcup_{i \in [m]} \alpha_i\}.\end{aligned}$$

Here, $\alpha_1|\alpha_2|\dots|\alpha_m$ is a disjoint partition of $\mathcal{A}.peers$, where α_i is the set of *peer-representations* of victim v_i. Let $E(S_x)$ be denoted by β_x. One should observe, if there are multiple `GETDATA` messages for an identical block, then there are multiple entries of one block-id in β_x (therefore, β_x is a multiset!). A node sends multiple `GETDATA` messages for an identical block, if the response of `GETDATA` is not received.

The set $\mathcal{A}.SessionBid$ can be computed inside the NAT (from the nodes in $\mathcal{V}$) because the `GETDATA` messages are known to both victims (nodes in $\mathcal{V}$) and the adversary (who is the set $\mathcal{A}$). For example, suppose α_i is the set of *peer-representations* of a victim v_i outside the NAT, when a `GETDATA` message sent by v_i comes out of NAT, it appears that it is sent by a *peer-representation* contained in α_i; therefore, the block-ids requested by node v_i is same as those requested by the set of peers in α_i. Let $\hat{\beta_{v_i}}$ contain the sets of block-ids requested in various sessions by victim v_i. We define $\hat{\beta_{v_i}}$ as follows:

$$\hat{\beta_{v_i}} = \{\beta_x : x \in \alpha_i\}.$$

Putting the β_{v_i} in the definition of $\mathcal{A}.SessionBid$ we get

$$\mathcal{A}.SessionBid = \bigcup_{v_i \in \mathcal{V}} \hat{\beta_{v_i}}. \tag{2}$$

We have computed the set $\mathcal{A}.SessionBid$ from the files contained in the Bitcoin's application data folder of victim nodes.[9]

4.2 Step 2: linking consecutive sessions

After extracting the block-ids, our next step is to link the *consecutive sessions*. The consecutive sessions of a node follow each other continuously. The phrase "consecutive sessions" is *only* meaningful for a single node.

Let us take two sequences of block-ids β_x and β_y in $\mathcal{A}.SessionBid$. If the adversary determines that they are requested in the consecutive sessions of a victim node; then using the inverse of *extract operation*, she can determine that S_x and S_y are consecutive sessions, where $E^{-1}(\beta_x) = S_x$ and $E^{-1}(\beta_y) = S_y$.[10] Since S_x and S_y contain information of IP address or onion address of the victim, and also the time of disconnect, the adversary can determine *peer-representations* x and y of the victim in two consecutive sessions.

We run the Algorithm 1 (also called `consecutive`), which returns *True* iff the inputs β_x and β_y are requested in consecutive sessions. The algorithm uses a fixed parameter th, which is a threshold of the number of common block-requests sent in the consecutive sessions. The correctness of the algorithm and the parameter threshold th are described in Appendices B and C. The function $H(b)$ used in `consecutive` returns the Blockchain height of the input block-id b.

4.3 Step 3: linking all sessions

In Sect. 4.2, we already described how to link the *consecutive sessions*. In this section, we describe how to link all the sessions – not necessarily consecutive – of a victim. We achieve it by constructing a *Bitcoin session graph*, and, finally, extracting the connected components in it. First, we define the *Bitcoin session*

[9] Bitcoin's application data folder: A set of data files containing the following information of the Bitcoin client: Private keys, Peer IP addresses, and various information related to the current Blockchain.

[10] E is a bijection from $\mathcal{A}.data$ to $\mathcal{A}.SessionBid$. It shows that the sequence of block-ids requested in a session is unique, which we found to be true in our experiments (see Sect. 5).

```
consecutive($\beta_x, \beta_y$)
    $h_x^s = min\{H(b) : b \in \beta_x\}$
    $h_x^e = max\{H(b) : b \in \beta_x\}$
    $h_y^s = min\{H(b) : b \in \beta_y\}$
    $h_y^e = max\{H(b) : b \in \beta_y\}$
    if $|\beta_x \bigcap \beta_y| < th$ then
        if $max\{|h_y^e - h_x^s|, |h_x^e - h_y^s|\} < |\beta_x| + |\beta_y|$ then
            return $True$
    return $False$
```

Algorithm 1: `consecutive(`β_x, β_y`)` determines, if block-ids in β_x and β_y are requested in consecutive sessions.

graph, and then we show how it will help in linking all the sessions of nodes.

Bitcoin session graph A *Bitcoin session graph* $G(\mathcal{S}, \mathcal{E})$ is defined as follows: (1) $\mathcal{S} = \{E(S) : S \in \mathcal{A}.data\}$, where E is an algorithm, which, given a session S, outputs certain data; (2) for all $(a, b) \in \mathcal{S} \times \mathcal{S}$, there is an undirected edge between a and b, *iff* a and b are data contained in the consecutive sessions of an identical node.

In our setting, $E(S)$ is a sequence of block-ids requested in `GETDATA` contained in the session $S \in \mathcal{A}.data$; therefore, $\mathcal{S} = \mathcal{A}.SessionBid$; two vertices a and b have an edge, if `consecutive(a,b)` $=$ *True*, where `consecutive` is the Algorithm 1.

Linking *all* sessions of victim nodes Here we describe a procedure that links all the sessions of the victim nodes and finally gives the set of *peer-representations* of a victim. Since the set $\mathcal{A}.SessionBid$ contains the sequences of block-ids extracted from *all* the sessions of *all* the victim nodes, therefore, there exists a subset of vertices in $\mathcal{S}$ corresponding to the sequences of block-ids requested by a victim in all its sessions.

A path in *Bitcoin session graph* is a sequence of edges, where each edge gives information of two vertices related to a victim; therefore, if two vertices have path between them, then they correspond to an identical victim. To determine the vertices related to a victim, adversary can compute the set of vertices of a maximally connected component of the *Bitcoin session graph*.[11] The graph

[11] A maximally connected component of a graph $G = (V, E)$ is a subgraph $C = (V', E')$ such that: C is connected, and, for all vertices $u \in V \setminus V'$, there is no vertex $v \in V'$ such that $(u, v) \in E$.

$G(\mathcal{S}, \mathcal{E})$ can have more than one maximally connected component, where each of them gives the set of vertices related to a victim node.

Let $\mathbb{M}$ contain the sets of vertices of the maximally connected components in the *Bitcoin session graph* $G(\mathcal{S}, E)$. The details of the constructions of the sets γ_i and α_i for $i \in [|\mathbb{M}|]$ are as follows:

The following are the *sessions* and *peer-representations* of a victim node.

$$\gamma_i = \{E^{-1}(\beta_x) : \beta_x \in c_i; c_i \in \mathbb{M}\}, \tag{3}$$

$$\alpha_i = \{S^{-1}(E^{-1}(\beta_x)) : \beta_x \in c_i; c_i \in \mathbb{M}\}. \tag{4}$$

5 Experiments

Precision and recall We ran experiments to evaluate the performance of Algorithm 1 (a.k.a `consecutive`). In particular, we compute two parameters, namely, *precision* and *recall*, whose definitions are given below. Suppose, $G = (\mathcal{S}, \mathcal{E})$ is a Bitcoin session graph (see Sect. 4.3 for definition). Let $G^* = (\mathcal{S}^*, \mathcal{E}^*)$ denote the Bitcoin session graph obtained from our experiment.

1. *precision*: This captures a measure of correct linking of the vertices.

$$precision = \frac{|\mathcal{E} \bigcap \mathcal{E}^*|}{|\mathcal{E}^*|}$$

   (The *precision* of 1 menas that the edges we have guessed are all correctly linked.)
2. *recall*: It captures a measure of how much the result is close to the best case of linking all the vertices related to consecutive sessions.

$$recall = \frac{|\mathcal{E} \bigcap \mathcal{E}^*|}{|\mathcal{E}|}$$

   (The *recall* of 1 means that, for each victim node, we have linked all its consecutive sessions. Note that, unlike *precision*, *recall* does not capture the scenario of linking sessions of two different nodes.)

Details of the experimental set-up We have done two sets of experiments to measure the performance of `consecutive` procedure. Experiment 1 had 4 victim bitcoin nodes, and each node had 8 sessions (this implies that the experiment included a total of 4 x 8 = 32 sessions); Experiment 2 had 4 victim nodes and each node had 6 sessions, implying that we experimented with a total of 4 x 6 = 24 sessions. In Experiment 1, the victims are directly connected to Bitcoin network, and in the Experiment 2, victims connect to Bitcoin network with (and without) Tor in different sessions. We believe that 32 and 24 sessions are good enough to demonstrate the proof of concept, which is the main purpose of this paper. Also, we would like to point out that, in order to obtain more realistic results, our experiments were performed on the *Bitcoin main network*, rather than on the *Bitcoin Testnet* (unlike the attack in [4]). We constructed Bitcoin session graph $G^*_1 = (\mathcal{S}^*_1, \mathcal{E}^*_1)$ in Experiment 1, and $G^*_2 = (\mathcal{S}^*_2, \mathcal{E}^*_2)$ in Experiment 2. Before giving our experimental results, below we provide the technical details of the experimental set-up.

In our experiment, we have used the following components: (a) Amazon Elastic Compute Cloud (*a.k.a* Amazon EC2), (b) a software container platform *Docker* [2], (c) a Bitcoin client *Bitcoind*, and (d) Tor client. See Figure 2 for the layers in which the components reside. Amazon EC2 is a web service that provides a cloud server on which *Ubuntu* runs. In the cloud server, we ran *Docker* to create multiple instances of the container, and each container had *Bitcoind* and Tor client in it. We used Docker because it allowed us to run instances of *Bitcoind* in the same EC2 machine, so that we had the same connection speed for all the Bitcoin nodes. The nodes running at the same speed are less vulnerable to fingerprinting attack (more on that later). The Tor client is used to connect to Bitcoin network over Tor.

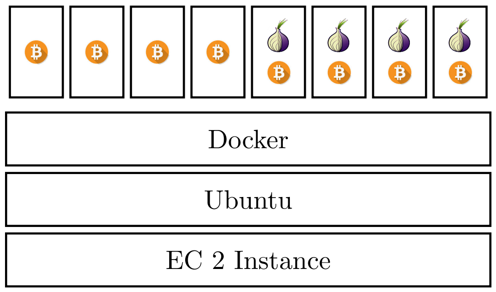

Fig. 2: Setup for Experiment 1 and 2

After we ran Bitcoin nodes, they connect to the Bitcoin network and started requesting blocks from other running peers to update their Blockchain. The session timings vary from 5 minutes to 160 minutes. Figure 3 shows the height of Blockchain at the start of each session we ran, we can see that the height of the nodes are close to each other. The nodes were running in the same EC2 instance for approximately the same time in each experiment, making their Blockchain growth very close to each other. This way of running Bitcoin nodes is a challenging scenario for deanonymization. After checking the block-ids requested in the sessions, we found that each victim requests a unique sequence of block-ids in its sessions.

Experiment 1 The main objective of the first experiment is to measure the performance when victim nodes are connected to Bitcoin network directly. We ran eight sessions for each victim in $\mathcal{V}_1 = \{v_1, v_2, v_3, v_4\}$ (total 32 sessions), then constructed the *Bitcoin session graph* to see if block-ids requested by a victim in different sessions can be linked. The major steps are as follows:

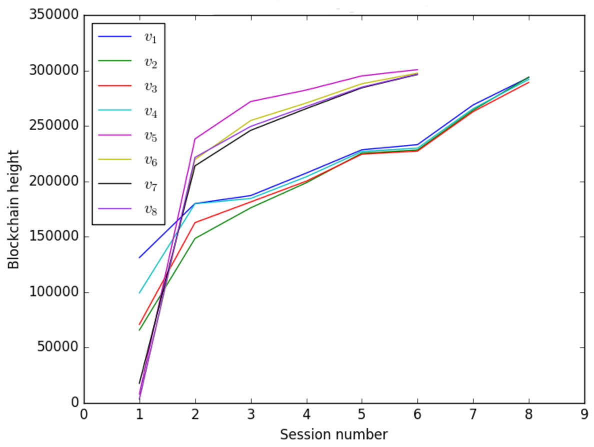

Fig. 3: This graph shows how the blockchain (best) height of a victim node (y-axis) varies with session number (x-axis). We ran nodes $\{v_1, v_2, v_3, v_4\}$ in Experiment 1 and nodes $\{v_5, v_6, v_7, v_8\}$ in Experiment 2.

1. We extract sequence of block-ids of the blocks requested in each session to compute $\mathcal{A}.SessionBid = \{\beta_1, \beta_2, \ldots, \beta_{32}\}$. (see Sect. 4.1).
2. We ran `consecutive`(β_i, β_j) for each $\beta_i, \beta_j \in \mathcal{A}.SessionBid$. If it returns *True*, then the inputs β_i and β_j are related to the consecutive sessions of a victim node. (see Sect. 4.2).
3. We construct a *Bitcoin session graph* $G_1^* = (\mathcal{S}_1^*, \mathcal{E}_1^*)$ using the output we got by running `consecutive` procedure, see Figure 4.
   (a) For each β_i in $\mathcal{A}.SessionBid$, we have a vertex in $\mathcal{S}_1^*$; thus, $\mathcal{S}_1^* = \{\beta_1, \beta_2, \ldots, \beta_{32}\}$. (see Sect. 4.3).
   (b) The edges $\{\beta_i, \beta_j\} \in \mathcal{E}_1^*$, if `consecutive`$(\beta_i, \beta_j) = $ *True*.

We got a *precision* of 0.90 and *recall* of 0.71. The high *precision* value shows that if an edge is present in the *Bitcoin session graph*, then it has a good chance of being a correct edge; however, the *recall* value shows that we have missed some edges.

Experiment 2 The main objective of the second experiment is to measure the performance when victim nodes connect to Bitcoin network, with and without Tor, in different sessions.

We ran six sessions for each victim in $\mathcal{V}_2 = \{v_5, v_6, v_7, v_8\}$ (total 24 sessions); in the first three sessions they are connected to Bitcoin netowrk over Tor, and, in the next three sessions, they are connected directly to the Bitcoin network. We have executed all the steps mentioned in Experiment 1 to compute the *Bitcoin session graph* $G_2^* = (\mathcal{S}_2^*, \mathcal{E}_2^*)$, see Figure 5. We found that the `consecutive` links the sessions with a *precision* of 1.0 and *recall* of 0.75.

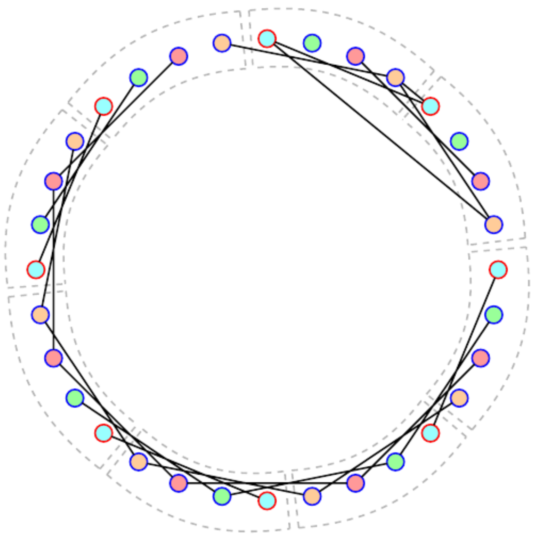

Fig. 4: This figure shows the *Bitcoin session graph* $G_1^* = (\mathcal{S}_1^*, \mathcal{E}_1^*)$. There are 32 vertices representing 32 sessions we ran. The vertices related to an identical victim node have the same color.

To get more insight of the graph $G_2^* = (\mathcal{S}_2^*, \mathcal{E}_2^*)$, we have partitioned $\mathcal{S}_2^*$ according to the six sessions of each victim, $\mathcal{S}_2^* = g_1|g_2|\ldots|g_6$ (shown by dotted regions in the figure), where g_i contains the sequence of block-ids requested by the victims in i^{th} session. For example, g_1 contains the sequences of block-ids requested by victims in their first session.

The edges between the vertices contained in $g_1 \bigcup g_2 \bigcup g_3$ show that we could link the Bitcoin over Tor sessions, and the edges between the vertices contained in $g_4 \bigcup g_5 \bigcup g_6$ show that we could link the sessions when victims connect to Bitcoin network directly.

An edge between a vertex from g_3 to a vertex in g_4 is important, because g_3 contains the sequences of block-ids requested when the victims connect to the Bitcoin network over Tor, and g_4 contains the sequences of block-ids requested when they connect to Bitcoin network directly. An edge between a vertex from g_3 and a vertex from g_4 confirms that we can link the sessions of victims when they ran Bitcoin over Tor and Bitcoin without Tor (direct connection).

Discussion We got a high *precision*, because initially the nodes request blocks at a faster rate; thus, in a short period of time, the differences of *best-heights* of the nodes become significant, allowing the attacker to distinguish between the sessions of different nodes. The *recall* value is also high because the heights of the consecutive sessions achieved at the end of the first session and at the beginning of the other session are close.

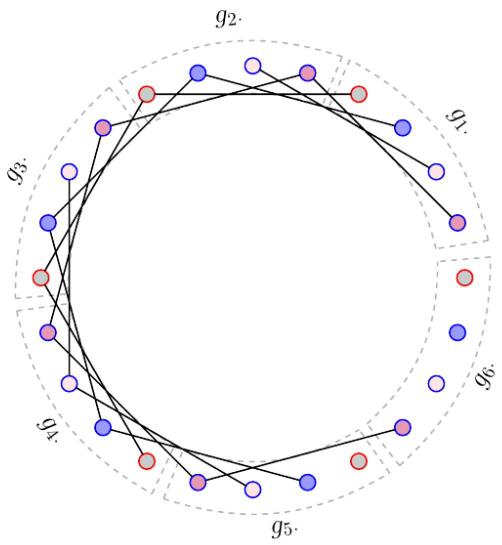

Fig. 5: This figure shows the *Bitcoin session graph* $G_2^* = (\mathcal{S}_2^*, \mathcal{E}_2^*)$. There are 24 vertices representing 24 sessions we ran. The vertices related to an identical victim node have the same color. (The set of vertices is partitioned into subsets $g_1|g_2|\ldots|g_6$, see Exepriment 2 for more details.)

Another interesting observation is that when nodes are running Bitcoin over Tor then Algorithm 1 performed better. We found the rate of blockchain update is comparatively slower for proxied connections than for direct connections, it is because of the additional number of proxy nodes (onion routers) between the sender and the receiver. We believe that a slower rate of blockchain update could be one of the reasons why nodes did not catch up heights close to each other.

One might ask how *precision* and *recall* change when we increase the numbers of nodes and sessions. Theoretically, increasing the numbers of victim nodes and sessions could result in the following situation: suppose s_1 and s_2 are the sessions of two distinct nodes, such that the blockchain height at the end of session s_1 is *close* (i.e., the height difference is less than the threshold th described inC) to the starting blockchain height of the session s_2. In this scenario, our algorithm will incorrectly link s_1 and s_2, thereby, will decrease the experimentally obtained *precision*. However, we emphasize that such events will be infrequent, even in large-scale experiments. For concreteness, such event occurred only once in our 56 sessions conducted across two experiments. Moreover, it is worth noting that a node requests for blocks from his peers in the increasing order of blockchain height, and such a pattern does not change even in large-scale experiments. Therefore, our algorithm that crucially relies on the aforementioned pattern will still be able to correctly link them; as a result, the *recall* is unlikely to be significantly affected. We leave it as a future work as to how to appropriately design a large-scale experimental set-up to test Algorithm 1 that also takes into consideration the *ethical issues*.

6 Countermeasure

If the victim nodes request blocks in an unordered fashion, then it will not be possible for an adversary to estimate their Blockchain height, and, therefore, he cannot link sessions. Below we describe it in more detail.

Nodes exchange information on their Blockchains using `VERSION`, `GETBLOCKS`, and `GETHEADERS` messages (see Sect. 2). Using this information, the peers compare their Blockchain heights. A peer with higher height sends `INV` or `HEADERS` to the one with lower height.[12] Instead of sending the actual information of Blockchain, if a node sends height-values chosen uniformly from 1 to *best-height*, and if the block-id is chosen uniformly, then the other peers will not be able to deduce the Blockchain height. Similarly, if a node chooses a point uniformly from the entire Blockchain, and starts requesting the `GETDATA` from there in every session, then it will result in requesting blocks which are already updated into the Blockchain. Therefore, the number of common block-requests of consecutive sessions might not be bonded by a threshold th. As a result, adversary would not be able to get a fixed value th to determine the consecutive sessions, ruling out the attack. However, due to requesting blocks already updated into the Blockchain, the node's Blockchain growth will become slow.

7 Conclusion

In this paper, we have shown that, in the Bitcoin main network, linking of the *consecutive sessions* is possible by analysing the block-requests the victim makes. Our approach relies on the observation that a node requests blocks in the increasing order of height; this observation leads to linking consecutive sessions of the node. Once consecutive sessions are linked, all others could be linked as well. We were able to link (consecutive) sessions with a high success rate in three settings: (1) when nodes connect directly; (2) when nodes connect using the Tor; and (3) when nodes connect with Tor and then without Tor. We have also suggested countermeasures against our attacks.

Acknowledgments The authors are grateful to the reviewers of CANS 2017 for their constructive comments. We also thank Sherman Chow for his insightful remarks regarding the experimental setup used in our work; the paper has benefitted immensely from them.

[12] The Blocks-First Sync and Headers-First Sync methods are two ways to update the Blockchain as described in Sect. 2.

References

1. Bitcoin wiki (2017), `https://en.bitcoin.it/wiki/`
2. Docker project code (2017), `https://github.com/docker/docker`
3. v0.13.2, bitcoin code project (2017), `https://github.com/bitcoin/bitcoin`
4. Biryukov, A., Khovratovich, D., Pustogarov, I.: Deanonymisation of clients in bitcoin p2p network. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM (2014)
5. Biryukov, A., Pustogarov, I.: Bitcoin over tor isn't a good idea. In: 2015 IEEE Symposium on Security and Privacy. pp. 122–134. IEEE (2015)
6. Dingledine, R., Mathewson, N., Syverson, P.: Tor: The second-generation onion router. Tech. rep., DTIC Document (2004)
7. Koshy, P., Koshy, D., McDaniel, P.: An analysis of anonymity in bitcoin using p2p network traffic. In: International Conference on Financial Cryptography and Data Security. pp. 469–485. Springer (2014)
8. Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G.M., Savage, S.: A fistful of bitcoins: characterizing payments among men with no names. In: Proceedings of the 2013 conference on Internet measurement conference. pp. 127–140. ACM (2013)
9. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
10. Reid, F., Harrigan, M.: An analysis of anonymity in the bitcoin system. In: Security and privacy in social networks, pp. 197–223. Springer (2013)
11. Ron, D., Shamir, A.: Quantitative analysis of the full bitcoin transaction graph. In: International Conference on Financial Cryptography and Data Security. pp. 6–24. Springer (2013)

A Peer Representation and Session

Suppose a victim node v_i is connected to adversarial nodes $\{a_j\}_{j\in[k]}$. The node v_i gets assigned multiple peer-ids p_{ij} outside the NAT making it difficult for a_j's to determine if they are connected to a single or multiple victim(s). The attacker nodes can check the *time of disconnect* of v_i to relate all the peer-ids assigned to v_i, and get a single representation (a, t), where a is the public address of v_i and t is the time of disconnect.[13]

The representation (a, t) is achieved as follows: the *ping* and *pong* protocol messages are exchanged periodically between peers to get the status of the connection; after node v_i goes offline at time t, it is not able to respond to the *ping* messages sent by a_j's; as a result, all the a_j's will detect at the same time t that *pong* messages are not coming from peer p_{ij} (a_j continues to identify v_i by p_{ij} even when it is disconnected); therefore, they conclude a victim is disconnected at time t enabling them to associate $p_{i1}, p_{i2}, \ldots, p_{ik}$ with a victim node, and get the *peer-representation* (a, t). Note that (a, t) is a peer-representation of the victim v_i outside the NAT.

After getting a single peer-representation (a, t) of v_i, the adversary can combine the Bitcoin protocol messages exchanged with the peers in $\{p_{i1}, p_{i2}, \ldots, p_{ik}\}$ to compute the session $S(a, t)$.

[13] The public address can be either public IP address or the onion address.

B Correctness of Algorithm 1

B.1 Background

Let $\beta_x, \beta_y \in \mathcal{A}.SessionBid$. The pair β_x and β_y can be related to three types: (1) consecutive sessions of an identical node, (2) non-consecutive sessions of an identical node, and (3) two different nodes. Below we describe the three cases in detail.

Related to consecutive sessions of an identical node A Bitcoin node continuously sends `GETDATA` messages to update its Blockchain. When the node disconnects, it misses response (`BLOCK` messages) of some `GETDATA` messages. Then in the new session, the node starts from the Blockchain height achieved at the end of the previous session, and again sends the `GETDATA` for the blocks it has missed in the previous session. Thus, there are repeated block-requests in the consecutive sessions of an identical node. This is an important observation for our attack, because it helps in defining a threshold th, below which if two sessions have the common block-requests, then they could correspond to consecutive sessions of an identical node.[14] More formally, if $|\beta_x \bigcap \beta_y| \leq th$, then they may correspond to an identical node.

Related to non-consecutive sessions of an identical node A node requests and receives blocks in each session; thus, the sessions between the two non-consecutive sessions update blocks into the Blockchain. Therefore, if we combine the block-ids requested in two non-consecutive sessions, the output will not have ids of blocks that lie between the heights achieved at the end of first session and at the start of the other session; these are the blocks updated between the two non-consecutive sessions. Since a node does not request the blocks after it is updated into the Blockchain, it requests disjoint sets of blocks in the non-consecutive sessions. More formally, if $|\beta_x \bigcap \beta_y| = 0$, then they may correspond to non-consecutive sessions.

Related to two different nodes Two nodes can request common blocks or disjoint sets of blocks depending upon their Blockchain height. Therefore, if β_x and β_y are from two different nodes, then we could get the following cases:

1. $|\beta_x \bigcap \beta_y| = 0$. This happens when the sessions related to β_x and β_y contain block-requests for disjoint sets of blocks.
2. $0 < |\beta_x \bigcap \beta_y| \leq th$. When the sessions related to β_x and β_y have Blockchain heights similar to consecutive sessions, then the numbers of common block-requests contained in them could be bounded by the threshold th.
3. $|\beta_x \bigcap \beta_y| > th$: The sessions related to β_x and β_y can contain the numbers of common block-requests greater than the threshold th. It happens when the first node achieves the height of the other in some session and then requests common blocks greater than th.

[14] In our experiments, we take the maximum number of repeated block-requests in the consecutive sessions to be the threshold th (see Sect. C for more details).

To determine if two sessions are of identical node or two different nodes, the case 3 as described above could be useful. This is explained using the following example. Suppose, two nodes v_i and v_j, where height of v_j is significantly large (greater than th) before they start the session. In the new session, the node v_i will make block requests for the blocks which are already requested and received by v_j, whereas v_j will not make repeated block requests for the blocks it has. Therefore, the previous session of v_j and the current session of v_i could have common block-requests much greater than th, whereas two sessions of v_j have common block-requests bounded by a threshold th. This shows that, if two sessions have common block-requests more than th then they are of different nodes, otherwise of an identical node. More formally, if $|\beta_x \bigcap \beta_y| \geq th$, then β_x and β_y are of two different nodes.

B.2 On the correctness of Algorithm 1

We note that Algorithm `consecutive` requires two checks (two *if conditions*) to conclude whether the inputs are of consecutive sessions of an identical node. Below we describe the two *conditions*.

1. We compare the intersection of β_x and β_y with a value th, where th is the threshold of the number of common block-ids a node requests in the consecutive sessions (see B.1). If we have number of common block-ids in β_x and β_y greater than than the threshold th, then they are related to different nodes and the algorithm return *False* (see B.1).

2. Let us denote the sessions related to β_x and β_y by S_x and S_y. As we can see, the second *if* condition makes use of parameters h_x^s, h_x^e, h_y^s, and h_y^e. Since the blocks are requested in the increasing order of height, h_x^s is close to the Blockchain height at the *start* of the session S_x and h_x^e is close to the height at *end* of S_x, same holds for h_y^s and h_y^e. Following are the reasons why the condition is true for consecutive sessions.
• Without loss of generality, assume that session S_x happens before S_y. If S_x and S_y are consecutive, then we have $|h_y^e - h_x^s|$ as the output of *max*. Since the consecutive sessions have common block-requests from the same blocks (see B.1), we get $|\beta_x| + |\beta_y|$ greater than $|h_y^e - h_x^s|$. Thus, the algorithm returns *True*.
• Without loss of generality, assume that session S_x happens before S_y. If S_x and S_y are non-consecutive, then we have $|h_y^e - h_x^s|$ as the output of *max*. The sessions between S_x and S_y update blocks into Blockchain (see B.1); thus, there are blocks between heights h_x^e and h_y^s whose block-ids are not contained in $\beta_x \bigcup \beta_y$. Thus, $|h_y^e - h_x^s|$ is greater then the value $|\beta_x| + |\beta_y|$ and the algorithm returns *False*.
• The third case is when S_x and S_y are of different nodes but have common block-requests less than th. This could be further divided into two cases $|\beta_x \bigcap \beta_y| = 0$ and $0 < |\beta_x \bigcap \beta_y| \leq th$ (see B.1). When intersection is empty then correctness is proved as follows: without loss of generality assume that the output of *max* is $|h_y^e - h_x^s|$; the output of *max* is greater than $|\beta_x| + |\beta_y|$ because of missing requests for the blocks between the heights h_y^e and h_x^s; thus, the algorithm returns

False. (When intersection is non-empty and less than th, the correctness holds because it is unlikely that two different nodes have heights similar to consecutive sessions.)

C Determining Threshold *th*

The value th is the threshold for the number of common block-requests sent in the consecutive sessions. It is used in the Algorithm 1, namely, `consecutive`, to determine: if two sets of block-ids correspond to consecutive sessions.

We have set th at 200 in our experiments, because we found the value 200 to be close to the number of common block-ids exchanged in the consecutive sessions. We now describe how the th is related to `consecutive` procedure.

Difficulty in setting *th* very high Let us define a random variable $X_{ij} = |\beta_i \bigcap \beta_j|$ (that denote the number of common block-requests). Note that X_{ij} is uniformly distributed over all integers between zero and the current *best-height* of Blockchain, because we *assume* that the block-requests issued by a node inside the NAT is independent of the block-requests made by others. Using Markov inequality we get.

$$Pr(X_{ij} \geq th) \leq \frac{\mathbb{E}(X_{ij})}{th}. \tag{5}$$

Let us define a random variable $\texttt{consecutive(}\beta_i\texttt{,}\beta_j\texttt{)}$ as follows:

$$\texttt{consecutive(}\beta_i, \beta_j\texttt{)} = \begin{cases} 1 & \textit{if } X_{ij} < th. \\ 0 & \textit{if } X_{ij} \geq th. \end{cases}$$

From the definition above, we get

$$Pr\big(\texttt{consecutive(}\beta_i, \beta_j\texttt{)} = 0\big) = Pr(X_{ij} \geq th).$$

By putting values from Eq. 5, we get

$$Pr(\texttt{consecutive(}\beta_i, \beta_j\texttt{)} = 0) \leq \frac{\mathbb{E}(X_{ij})}{th}.$$

Let p be the probability that $\texttt{consecutive(}\beta_i, \beta_j\texttt{)}$ *returns* "1", we get

$$\begin{aligned} p &= 1 - Pr(\texttt{consecutive(}\beta_i, \beta_j\texttt{)} = 0) \\ &\geq 1 - \frac{\mathbb{E}(X_{ij})}{th} \end{aligned}$$

The expected number of common block requests $\mathbb{E}(X_{ij})$ is a constant. Therefore, if the value of th is set to be high, then the probability that $\texttt{consecutive(}\beta_i, \beta_j\texttt{)}$ returns "1" increases; thus, the attack might end up linking the sessions which correspond to different nodes (wrong linking).