2019 IEEE European Symposium on Security and Privacy Workshops

Eclipsing Ethereum Peers with False Friends

Sebastian Henningsen Daniel Teunis Martin Florian Björn Scheuermann

Weizenbaum-Institute for the Networked Society
Humboldt-Universität zu Berlin
Berlin, Germany

***Abstract*—Ethereum is a decentralized Blockchain system that supports the execution of Turing-complete smart contracts. Although the security of the Ethereum ecosystem has been studied in the past, the network layer has been mostly neglected. We show that Go Ethereum (Geth), the most widely used Ethereum implementation, is currently vulnerable to eclipse attacks, effectively circumventing recently introduced security enhancements[1]. Our false friends attack exploits the Kademlia-inspired peer discovery logic used by Geth and enables a low-resource eclipsing of long-running, remote victim nodes. An adversary only needs two hosts in distinct /24 subnets to launch the eclipse, which can then be leveraged to filter the victim's view of the Blockchain. We discuss fundamental properties of Geth's node discovery logic that enable the false friends attack, and propose countermeasures.**

I. Introduction

A dependable and secure network layer is vital for blockchain systems, since they build on the assumption of equal information at every peer [1], [2]. This assumption is violated if *eclipse attacks* are possible. In an eclipse attack, an adversary monopolizes the connections of a victim, effectively filtering the victim's view of the blockchain. Eclipse attacks enable a variety of follow-up attacks such as double spending and stubborn mining [3].

Despite its important role, the network layer has so far received surprisingly little attention in systems like Bitcoin [2], [4] or Ethereum [5], leading to vulnerabilities.

In [6], a low-cost eclipse attack has been proposed that exploits the Kademlia-inspired [7] peer discovery logic of *Go Ethereum*, the official reference implementation of Ethereum[2]. The attack is mounted after a victim node has been restarted and is based on flooding the node's discovery table with Sybil [9] nodes. Generating a new node ID, and henceforth a new Sybil node, involves only an ECDSA key pair generation, which makes the attack lightweight. As an answer to the discovered attack vector, Geth $\geq$ v1.8.0 introduces several countermeasures to increase the difficulty and necessary resources to flood the complete discovery table.

However, as we show in this paper, eclipse attacks on Geth are still possible with very limited effort. We propose the *false friends* attack that enables the eclipsing of current Ethereum nodes. Despite the subnet restrictions implemented in Geth v1.8.0, we only need two IP addresses from distinct /24 subnets for a successful attack. Additionally, and in contrast to [6], we do not necessarily require a restart of the victim node since peer churn is high and existing connections will eventually be dropped. Instead of overwriting the complete discovery table with Sybil nodes, we subtly insert adversarial nodes with carefully selected node IDs, exploiting the interplay between peer discovery and connection management.

Geth chooses new peers either by directly selecting nodes from its discovery table or by starting a Kademlia-style lookup to a random target, which yields new node contact information. We compromise both mechanisms, in slightly different ways. We insert a limited number of Sybil nodes into the victim's discovery table, with an activity pattern that favors these Sybils when new connections are set up. For new contacts resulting from lookup operations, we pre-compute a large number of node IDs and present tailored choices when queried during a lookup, effectively offering "better" (albeit false) peers than all honest nodes visible to the victim.

Existing connections have to be terminated before the available slots can be filled with adversarial nodes. However, as we noticed through measurements, peer connections are terminated regularly without additional intervention: $95\,\%$ of connections longer than $60\,\mathrm{s}$ were shorter than $5.5\,\mathrm{d}$ We were, in effect, able to successfully eclipse a live node by actively waiting and incrementally injecting adversary nodes into the victim's peer lists[3]. Our measurements indicate that a targeted false friends attack on a specific Ethereum node can be successfully completed within a few days.

In summary, the contributions of our paper are:

- The discovery, description and evaluation of the *false friends* attack, an eclipse attack on current Geth versions that exploits fundamental properties of Geth's peer discovery logic
- A description and theoretical analysis of Ethereum's network layer management algorithms, based on an analysis of the Geth codebase; previously available information is scarce.
- The discussion of possible countermeasures—both easy fixes to prevent the presented attack and ideas for tackling the fundamental challenge of securing Ethereum's overlay network.

The rest of this paper is structured as follows. Section II describes the high-level network architecture of Geth and

[1]We responsibly disclosed the vulnerability to core Ethereum developers.

[2]Other clients might also be vulnerable. We focus on Geth as it is estimated to be used in roughly $76\,\%$ of clients [8].

[3]Without attacking nodes operated by other network participants, and without risking any harm for the Ethereum network, of course.

DOI 10.1109/EuroSPW.2019.00040

Section III gives a detailed explanation of the Kademlia-inspired peer discovery. In Section IV we then present the actual false friends attack, i.e. how we exploit Geth's peer discovery logic. We present our analytical evaluation and measurements in Sections V and VI. Section VII discusses possible countermeasures against the presented attack. We conclude the paper with a summary of related work (Section VIII) and concluding remarks (Section IX).

II. Background: the Ethereum Network Stack

In the following, we introduce the overall architecture of Ethereum's Peer-to-Peer network as implemented in Go Ethereum. Unlike similar descriptions in related works [6], [8], the information presented here uses a naming of high-level components that is more strongly aligned with the official Ethereum terminology.

The overall network architecture of Ethereum is summarized in Figure 1. Ethereum's network layer consists of four major components, namely: *discv4* for node discovery; *RLPx* as a secure transport layer; *DEVp2p* for session management on top of RLPx and the actual *Ethereum protocol* (*eth*) which runs on top of DEVp2p. The Whisper protocol (for decentralized applications) and the *Swarm* protocol (for decentralized file storage) are other subprotocols on top of DEVp2p.

DEVp2p not only provides the foundation for the Ethereum protocol and other application protocols, it also manages connections to other peers which form the overlay on which blocks and transactions are distributed. Geth by default has a total of 25 TCP connections to other peers speaking the Ethereum protocol. Of these 25 slots, 17 are reserved for inbound connections (initiated by other peers), whereas the remaining 8 are allocated for outbound connections. In this case, inbound means that a remote peer sent a `SYN`-packet to start a TCP connection with the local peer. No further restrictions apply to inbound connections; if an inbound slot is available Geth simply accepts any connecting peer that supports the Ethereum protocol and operates on the same network (main, testing, etc.). The 8 outbound slots are therefore especially important, as they are the most difficult ones to get under control for an attacker mounting an eclipse attack.

In contrast to DEVp2p, the discv4 node discovery stores information about *all* node types in the overlay. This includes nodes without support for the Ethereum protocol (which is a perfectly valid case in the design logic of Ethereum's protocol stack). The discv4 node discovery is inspired by the Kademlia DHT [7]. Information about known overlay nodes is stored in a table separated into so-called k-buckets (or simply *buckets*, in the following).

The outbound connections are established to nodes that are returned from the discovery table. Every time not all outbound slots are occupied, the DEVp2p peer management requests the discovery table in two distinct fashions depicted in Figure 2.

First, half of the *currently empty* slots (rounded down) are filled with a direct request to the discovery table via the function `ReadRandomNodes`. Second, the remaining slots are filled from the lookup-buffer, which holds the result of a

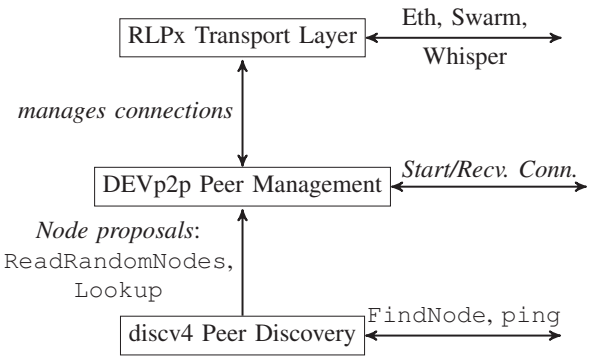

Figure 1. Overview of the Ethereum Network Stack.

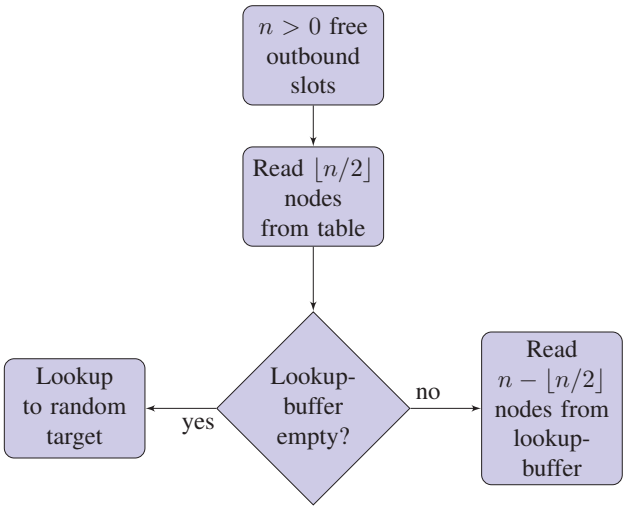

Figure 2. How outbound connections are established.

Kademlia-like lookup to a random target ID. Note that this procedure is repeated *every time an outbound slot becomes available*. Therefore, Geth fills half of the *currently available* slots with each mechanism. Depending on the situation, this skews the distribution of outbound connections towards either mechanism. If only one slot becomes available at a time, the lookup-buffer is favored (`ReadRandomNodes` gets $\lfloor 0.5 \rfloor = 0$ slots). Otherwise, if two lookup-buffer slots become available repeatedly, `ReadRandomNodes` is favored in comparison to the lookup-bufer.

Our false friends attack exploits these two interfaces between node discovery and peer management. The discovery table therefore constitutes a particularly important component of Ethereum for our purposes, and deserves a closer look.

III. Node Discovery and Selection

Ethereum's node discovery table largely resembles a *Kademlia* [7] routing table. Unlike in Kademlia, however, its sole purpose is to manage a set of known nodes which serves as a basis for establishing connections in DEVp2p. In [10]

it is conjectured that Kademlia was chosen as a basis due to future plans to shard the blockchain, i.e., to partition it over the network.

A. Kademlia in a Nutshell

Kademlia is a UDP-based, peer-to-peer distributed hash table (DHT) that is used to locate decentrally stored data efficiently [7]. Each node is uniquely identified by its randomly generated 160-bit node ID. A data item stored in the DHT is found by its key, which is simply a 160-bit hash of the data itself. Hence, node IDs and keys share the same representation; in the following we use the term ID for both. Kademlia leverages this by storing data at nodes whose node ID is "close" to the data's key. Closeness is defined as the bitwise XOR between two IDs, taken as an integer value, i.e., $d(x, y) = x \oplus y$. A node stores its known neighbors in so-called *k-buckets* which partition the known network based on the local node's ID. Every k-bucket (or simply bucket) stores up to k neighbors. Intuitively, node IDs are treated as the leaves of a binary tree and each bucket stores a distinct branch of the tree. Bucket i stores nodes whose distance is in $[2^i, 2^{i+1})$, which effectively corresponds to the length of the common prefix between two node IDs.

Items in the DHT and nodes on the network are located by so-called *lookups*. A lookup successively queries nodes that are closest to the desired target ID (key or node ID).

B. Node IDs

As in Kademlia, node IDs in Ethereum serve as public identifiers for each node in the Ethereum network. A node ID in Ethereum is a marshaled 512-bit ECDSA public key. However, distance computations only operate on Keccak256-hashes [11] of node IDs, effectively yielding node IDs with 256-bit length. When referring to node IDs in the following we implicitly mean hashed ECDSA public keys. Node IDs are supposed to be static, as stated in the official Ethereum documentation:

> *Each node is expected to maintain a static private key which is saved and restored between sessions. It is recommended that the private key can only be reset manually [...].*[4]

It is easy to generate and use many different identities by creating ECDSA key pairs.

C. Buckets and Log-Distance-Metric

The buckets of Geth's discovery table hold up to $k = 16$ nodes each. Exactly as in Kademlia, the nodes in each bucket share a common property: the distance, according to some metric, between their node ID and the local node's ID is the same. [6], [8] state that Ethereum uses the so-called *log-distance* metric. We argue that this metric is identical to the distance metric used to determine buckets in Kademlia. The log-distance between two hashes can be defined as $\lfloor \log_2(\bar{N_1} \oplus \bar{N_2}) \rfloor$, or equivalently, 255 - the length of their common prefix – which is exactly how buckets are organized in Kademlia. Due to the uniqueness assumption of node IDs, this yields $|\{0, ..., 255\}| = 256$ possible distances.

In response to the eclipse attack by [6], Geth $\geq$ 1.8.0 restricts the number of buckets to 17, starting from the furthest distance of 255 to the minimum possible log-distance of 239.

The log-distance metric leads to a skewed distribution of nodes between buckets: most of the lower buckets are empty, since the probability to fall into a specific bucket decays exponentially with the associated distance [7].

D. Entering a Bucket

A local node learns of neighboring nodes either by receiving an unsolicited `ping` packet or in the course of a lookup operation. The lookup process also initiates a `ping`/`pong` exchange, which then triggers the node to be added to the discovery table. In any case, before a node enters the discovery table, a number of checks are performed which are depicted in Figure 3. Assume that the local node receives either a `ping` packet or a `pong` reply to a previously sent `ping`. Two cases are to be distinguished: first, the node may already be in its respective bucket; in this case it is simply moved to the first position. This induces a "least recently active" sorting of the nodes within a bucket [7], where activity simply means sending (responding to) a `ping`-packet. Second, in case the node is not already in a bucket, it is added if the bucket is not full. If the bucket is already full, candidate nodes are stored in a so-called *replacement list* that stores up to ten nodes.

Every 5 s (on average), the last node of a random bucket is pinged and replaced with a random node from the respective replacement list if it fails to respond. In contrast to buckets, the replacement list is a simple FIFO queue that evicts the last entry every time a previously unknown node is added to the list. Last but not least, a node is only added to its respective bucket (or replacement list) if it meets certain IP address restrictions: Geth restricts the number of IP addresses coming from the same /24 subnet to two per bucket, and to ten in the whole discovery table.

E. FindNode-Requests

Lookups in Ethereum are used to discover new peers.[5]. These lookups are performed iteratively by sending so-called *FindNode requests*, to which the recipient answers with a `neighbors` packet containing information about nodes from its discovery table.

The most important use of lookups in our scenario is to populate the so-called *lookup-buffer*. As already outlined, the lookup-buffer is one of two methods by which the DEVp2p subsystem finds new nodes to connect to. When the lookup-buffer is empty, Geth populates it by starting a lookup to a random target. That is, it sends a FindNode request to peers that are "close" to the random target. For lookups, Ethereum does *not* use the log-distance metric for sorting the buckets (cf. Section III-C) but instead uses the plain xor-metric. To

[4]https://github.com/ethereum/devp2p/blob/6504d410bc4b8dda2b43941e1cb48c804b90cf22/rlpx.md, accessed 15.04.19

[5]FindNode requests are also used to populate the discovery table and to resolve node IDs to IP addresses [6], which is outside the scope of this paper.

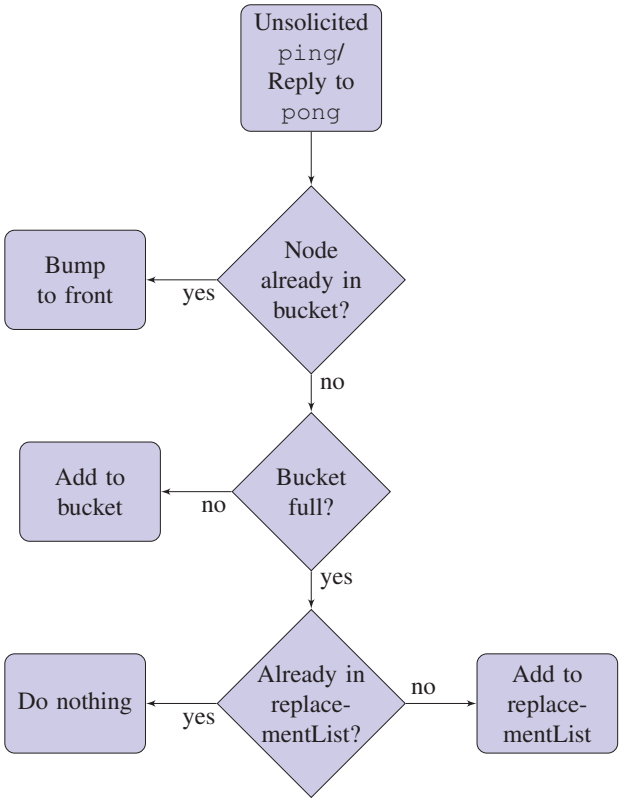

Figure 3. How nodes enter buckets.

illustrate, let $\bar{N}_1, \bar{N}_2$ be two node IDs and t a (random) target ID. For Ethereum (and likewise Kademlia), $\bar{N}_1$ is closer to t than $\bar{N}_2$ iff $\bar{N}_1 \oplus t < \bar{N}_2 \oplus t$, where $\oplus$ denotes the bitwise xor-operation and the result is taken as the binary representation of an unsigned integer. To ease notation, we define the abbreviation $<_t$ as $\bar{N}_1 <_t \bar{N}_2 :\Leftrightarrow (\bar{N}_1 \oplus t < \bar{N}_2 \oplus t)$.

The iterative lookup procedure to populate the lookup-buffer is visualized in Algorithm 1. First, a random target ID t is chosen. Subsequently, all known peers from the discovery table are sorted according to $<_t$, effectively yielding the 16 peers that are closest to the random target t. In a next step, a FindNode request is sent to each of these 16 peers, asking them for their respective neighbors that are closest to t. If successful, each queried peer will answer with a `neighbors` packet containing up to 12 peers (1280 byte). All received neighbors are combined, sorted by $<_t$, and again restricted to the 16 peers closest to the random target t. This yields the result set of the first round. This process is iterated until the result set eventually stabilizes. If a result set contains the same peers as the result set of the previous iteration, the procedure terminates.

IV. The False Friends Attack

After having established the necessary background in the previous sections, we now describe the details of our *false friends* attack, with an in-depth analysis of the attack following in Section V.

Algorithm 1 Populate Lookup-buffer

$t \leftarrow$ random node ID
$N_0 \leftarrow \{16$ closest known peers to $t\}$
loop
  for $o_i \in N_0$ **do**
    $F_i \leftarrow \{$closest peers of o_i to t, as returned by $o_i\}$
  $N \leftarrow N_0 \cup \bigcup_{i=0}^{15} F_i$

  $N_1 \leftarrow \text{sort}(N, t)[0:15]$ // 16 closest to t
  if $N_0 = N_1$ **then return** N_0
  else
    $N_0 \leftarrow N_1$

To eclipse a victim, its 8 slots for outbound connections as well as the 17 slots for inbound ones have to be filled with adversarial nodes. The inbound connections slots can easily be filled since Geth does not impose any restrictions on inbound connections. Hence, it suffices to start multiple Geth instances on different ports and configure them to repeatedly connect to the victim. Due to the lack of restrictions, *one* host with one IP address is enough to fill the inbound slots. Note that these Geth instances do *not* need to actively participate in block and transaction distribution.

To fill the outbound connection slots, we have to make sure that only adversarial nodes are proposed to the DEVp2p peer management via the two mechanisms (cf. Section II). Whereas [6] fills the whole discovery table with Sybil nodes to ensure that only these are proposed to the peer management, we leverage properties of the peer selection mechanism to achieve the same result with only one Sybil node per neighbor table bucket. It suffices to have one Sybil node in each bucket of the neighbour table to make sure that only adversarial nodes are returned to the peer management. This effectively circumvents the implemented countermeasures and still needs very little resources (two IP addresses in distinct /24 subnets).

In contrast to [1], [6] we do not necessarily require a restart of the victim node for our attack to be successful, though it speeds up the attack. In both cases, an adversary has to wait until existing connections are terminated for other reasons, such as timeouts. Our measurements (discussed in Section VI) indicate that connections on the Ethereum network are rather short-lived and a successful attack against a non-restarting victim node is therefore possible in a matter of days.

Our attack is facilitated by the fact that countermeasures 2 and 3 from [6] were not implemented in Geth. Countermeasure 2, a fixed mapping between the IP address and ECDSA key would raise the requirements for a false friend attack to 25 unique IP addresses (one for each connection slot). While moderately increasing the necessary resources for an adversary, the impediment for the network is significant, since multiple Geth instances behind a NAT would be impossible. Countermeasure 3, making the mapping of IDs to buckets in

the neighbour table secret, is a theoretically viable mechanism to drastically raise the bar for an attacker. However, one would lose the benefits of Kademlia in case routing becomes relevant; this might be the reason why the developers chose not to implement countermeasure 3.

A. Taking Over `ReadRandomNodes`

The function `ReadRandomNodes` returns (per default) at most 4 nodes from the discovery table, which are then used by the peer management to establish outbound connections. Most importantly, `ReadRandomNodes` only returns the *head* of randomly chosen buckets. Presumably, this design choice is due to the implicit sorting by activity within a bucket (cf. Section III): peers in the front of a bucket are more active and/or have a better latency than the others and are therefore favorable to connect to. This behavior can easily exploited by an adversary, since the sorting by activity merely requires the adversary to regularly send a `ping`-packet to stay ahead of the other peers. Therefore, it is sufficient for an attacker to populate each bucket with *one* node instead of the whole discovery table. To this end, an adversary repeatedly generates new ECDSA key pairs, computes the node ID and checks whether this particular ID is mapped to the desired bucket.

Current versions of Geth maintain 17 buckets and implement an IP-based restrictions such that at most 2 nodes from the same /24 subnet can be included in the same bucket and at most 10 nodes from the same /24 subnet can be in the whole discovery table. With these current properties of Geth, only *two* IPs from distinct /24 subnets are necessary for successfully compromising `ReadRandomNodes`.

B. Exploiting the lookup-buffer

The lookup-buffer is the second source used by DEVp2p to get potential peers to connect to. It is populated with a Kademlia-like iterative lookup of a random target ID. To this end, the local node sends FindNode-Requests with a random target to those nodes from the discovery table that are closest to that target (cf. Section III-E). From the received node set, the 16 closest nodes are used to populate the lookup-buffer, sorted by their distance to the target. DEVp2p then partially fills the open outbound connections slots by going through the lookup-buffer from the top (i.e. minimum distance).

To fill the lookup-buffer with adversarial nodes two steps are necessary: First, an adversarial node must be queried during the lookup-process. Second, the node IDs returned by the adversary must be smaller than all other node IDs returned during the lookup. The first step is *always* given when there is an adversarial node in each bucket: the xor-distance is mainly influenced by the length of the common prefix and each bucket stores node IDs with a specific common prefix length. Therefore, an adversarial node in each bucket ensures that the attacker is always queried during a lookup (cf. Section V-B for a detailed analysis).

The second step can easily be solved by generating sufficiently many node IDs. Since node IDs are hashed ECDSA keys, they are uniformly distributed over the ID-space; hence, the more node IDs we generate, the higher the chances to be smaller than the rest of the returned IDs. In the end, by choosing the number of pre-computed keys high enough, it is very likely that all of our 16 closest IDs are closer to the target than any ID naturally occurring in the Ethereum network.

V. Analysis of the False Friends Attack

In the following we analyze the mechanics of our false friends attack. We compute the expected number of necessary key pair generations and for entering every bucket and to exploit the lookup-buffer. Furthermore, we study FindNode-Requests in detail.

A. Entering a Bucket

Our false friend attack requires one adversarial node in each bucket. The question arises how many key pairs we have to generate and how much time it takes to do so.

Since SHA256 is a cryptographic hash function, we assume node IDs to be uniformly distributed [12], i.e., each bit has a probability of $\frac{1}{2}$ of being 0 or 1. Therefore, each ECDSA key pair generation with subsequent hashing corresponds to fair coin tosses repeated independently of each other. Let $\bar{N}$ be the hashed node ID of the victim node, i.e., $\bar{N}$ is fixed. Then, for some generated hashed node ID, say $\bar{h}$, the probability that the first bit of $\bar{h}$ is equal to the first bit of $\bar{N}$ is $\frac{1}{2}$. Recall that the log-distance metric measures the length of the common prefix between $\bar{N}$ and $\bar{h}$ (cf. Section III-C). Hence, with probability $\frac{1}{2}$, the two hashes differ at the first bit, which corresponds to a log-distance of 255. For subsequent buckets the concept is similar: the probability to have a log-distance of 254 is $\frac{1}{4}$ since we have to be equal in the first *and* second bit, i.e., $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$. In summary, the probability to have a specific log-distance to a given target hash is

$$p := \mathbb{P}[\text{log-distance}(h_1, h_2) = i] = 2^{i-256}. \tag{1}$$

Changing perspective, we can now calculate the expected number of necessary key pair generations to fall into a specific bucket. Finding a key pair for a desired bucket, or equivalently, a desired log-distance, can be modeled as a series of independent Bernoulli trials until the first success. Each key pair generation is a Bernoulli trial with success probability p (from Equation (1)). Repeatedly performing Bernoulli trials and stopping at the first success yields a geometric distribution. Therefore, generating key pairs for a specific bucket can be modeled as a geometric distribution, with expectation

$$\mathbb{E}[\#\text{ key pair generations for log-distance } i] = \frac{1}{p} = 2^{256-i}. \tag{2}$$

For example, to generate an ID with the (in Ethereum) lowest possible log-distance of 239 one would, on average, need $2^{17} = 131072$ key pair generations and hash operations. Generating a node ID for every bucket requires an average number of operations of

$$\sum_{i=239}^{255} 2^{256-i} = \sum_{i=1}^{17} = 2^i = 262142. \tag{3}$$

Note that these node ID generations need to be performed only once per victim node.

B. Computing the Probability to Receive a FindNode-Request

In the following we analyze how probable it is to be asked during a FindNode-Request round. We distinguish two cases; first, the current implementation in Geth and second, when the buckets would hold more than $k = 16$ nodes.

1) Implementation in Geth: We can assume that node IDs are uniformly distributed in $\{0, 1, \ldots, 2^{256} - 1\}$ since they are hashed public keys with 256 bit length. As in Kademlia, each ID can be viewed as the leaf of a binary tree, thus, each bucket stores node IDs from a specific sub-tree. For the lookup-process, the victim generates a random target ID, say D, and computes its 16 closest neighbors to D. Note, that the number of closest neighbors that Geth computes coincides with maximum possible number of nodes in a bucket.

"Close" is defined in terms of the simple xor-metric; for two IDs a, b the distance is defined as $d(a, b) := a \oplus b$, taken as integer. Under the xor-metric, node IDs that have a longer common prefix D are thus considered closer than ones with a shorter common prefix. Each bucket partitions the binary tree of node IDs into branches by their common prefix. Therefore, the closest neighbors to D are the ones in D's bucket.

Recall from the previous section that we insert an adversarial node ID into each bucket for our false friends attack. Hence, searching for the 16 closest neighbors will *always* return at least one attacker-controlled node ID.

2) Situation with Larger Buckets: A simple countermeasure to the current situation in Geth is to simply increase the size of the buckets. In the following we analyze the probability for an attacker to receive a FindNode-Request in different scenarios.

Without loss of generality assume node IDs to be within $[0, 1]$, as a simplification assume them to be continuously distributed on said interval. Let $D \in [0, 1]$ be the uniformly random target ID chosen by the lookup-process and $Y_1, \ldots, Y_N$ be the IDs of honest nodes stored in the bucket of D, say b. We consider the case $N > 16$.

Assume for the moment that the adversary has exactly *one* node ID in b, with ID Z. Although the $Y_1, \ldots, Y_N$ share a common prefix, their suffix is distributed uniformly at random because node IDs are hashes. For an attacker to receive a FindNode-Request it suffices to be smaller (w.r.t. D) than the 17-th closest ID, i.e., the 17-th *order statistic*. We denote the ordering of IDs Y_i with respect to D as $Y_{(1)} <_D Y_{(2)} <_D \ldots <_D Y_{(N)}$, where $Y_{(1)}$ is the node with minimum distance. It now remains to compute the following probability:

$$\mathbb{P}[Z < Y_{(17)}]. \tag{4}$$

The density of Z is $f_Z(z) = 1$, due to its uniformity. Similarly, let $f_Y(y)$ denote the density of some Y. By definition of probability and the expectation we have for any independent Y, Z with $Z \sim U[0, 1]$:

$$\mathbb{P}[Z < Y] = \int_{y=0}^{1} \underbrace{\int_{z=0}^{y} f_Z(z)\mathrm{d}z}_{=y} f_Y(y)\mathrm{d}y \tag{5}$$

$$= \int_{y=0}^{1} y \cdot f_Y(y)\mathrm{d}y \tag{6}$$

$$= \mathbb{E}[Y]. \tag{7}$$

It is well-known that the order statistics of uniform variables are Beta-distributed [13], i.e., $Y_{(l)} \sim Beta(l, N + 1 - l)$. Inserting that into Equation 7 we get

$$\mathbb{P}[Z < Y_{(l)}] \overset{(7)}{=} \mathbb{E}[Y_{(l)}] \overset{\text{Beta distr.}}{=} \frac{l}{N+1}. \tag{8}$$

In general, the attacker can have multiple, say $a \in \mathbb{N}$ nodes in the bucket b. To get queried, *at least one attacker ID* has to be within the closest nodes. Denote the adversarial IDs by $Z_1, \ldots, Z_a \sim U[0, 1]$. Then we obtain:

$$\mathbb{P}[\text{At least one attacker ID within } k \text{ closest}] \tag{9}$$

$$= 1 - \mathbb{P}[Z_1 > Y_{(l)} \wedge Z_2 > Y_{(l)} \wedge \ldots Z_a > Y_{(l)}] \tag{10}$$

$$\overset{i.i.d.}{=} 1 - \mathbb{P}[Z_1 > Y_{(l)}] \cdot \ldots \cdot \mathbb{P}[Z_a > Y_{(l)}] \tag{11}$$

$$\overset{(8)}{=} 1 - \left[1 - \left(\frac{l}{N+1}\right)\right]^a. \tag{12}$$

Intuitively, the more adversarial nodes there are in bucket b, the more unlikely it becomes *not* to get queried during the lookup-process. Figure 4 shows the result of Equation (12). The probability for the adversary to receive a FindNode-Request is plotted over the number of adversarial nodes in bucket b. For the number of nodes per bucket we consider three cases:

1) $N = 32$, double the size of current buckets.
2) $N = 136$, i.e., the size of a bucket corresponds to half of the current size of the complete discovery table.
3) $N = 272$ which corresponds to the maximum size of the current discovery table (17 buckets à 16 nodes each).

C. Filling the Lookup-buffer with Pre-Computed Node IDs

Recall, that in order to take over the lookup-buffer, we identified two necessary steps: First, an adversarial node must be queried during the lookup-process. Although this is purely a matter of chance, we saw that the probability to get queried is relatively high even with a small number of nodes. In a second step, the node IDs returned by the adversary must be smaller (with respect to the random target) than all other node IDs returned during the lookup. Intuitively, this can be ensured by pre-computing a large number of node IDs, since each ID generation corresponds to a draw from the uniform distribution. Every draw has the same probability to be smaller than any other node ID on the Ethereum network. Therefore, the more node IDs we generate, the more likely this event becomes. The question that remains is the following: how many ECDSA key pairs should an adversary generate in advance in order to almost always return the smallest node ID?

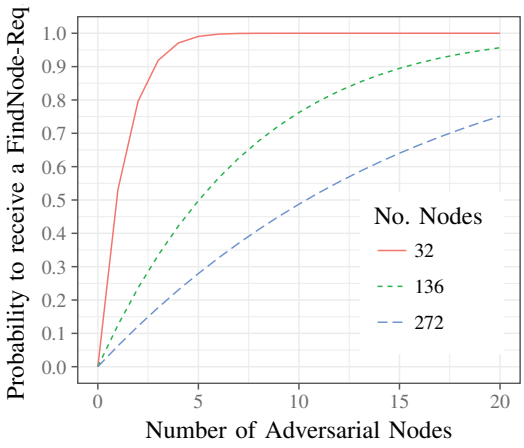

Figure 4. Probability that the adversary gets queried with a FindNode-Request for a given number of adversarial nodes in the victims discovery table.

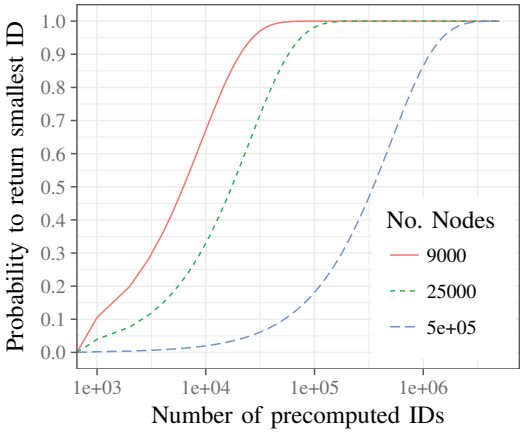

Figure 5. Probability that the lowest node ID is returned, depending on the number of honest nodes in the network.

To model this scenario, assume all other nodes already replied to the FindNode-Request with target D, which yields node IDs (sorted w.l.o.g) $0 \leq x_1, < x_2 \ldots < x_m \leq 1$. [6] Each node ID induces two intervals around itself on the space of possible node IDs:

$$[0, x_1), (x_1, x_2), (x_2, x_3), \ldots, (x_m, 1]. \tag{13}$$

For a new node ID to be the minimum, it has to fall into the first interval and only the first interval. Each interval is equally likely to occur, because the likelihood of falling into an interval only depends on its size when sampling from a uniform distribution. Since the existing node IDs $x_1, \ldots, x_m$ are uniformly distributed themselves, the intervals have, on average, the same size.

Let $Y \sim U[0, 1]$ represent a node ID generation. There are $m + 1$ intervals, the chances of falling into the first interval (i.e., having the minimum node ID) by uniform sampling is

$$\mathbb{P}[Y <_D \min\{X_1, \ldots, X_m\}] = \frac{1}{m+1}. \tag{14}$$

In other words, every generated node ID has a chance of $p := \frac{1}{m+1}$ of being the minimum node ID. Therefore, repeating the process of generating node IDs again yields a Bernoulli trial with success probability p.

In reality, we do not know the other node IDs before we start generating our own, meaning that whether a draw was successful cannot be determined. Instead, an adversary pre-computes a large number of IDs and simply returns the smallest ones with respect to the random target, if she receives a FindNode-Request. Still, we can bound the probability that *at least one* of our draws is smaller than the minimum returned by the honest nodes. Let there be m node IDs in the Ethereum network and n pre-computed node IDs by the adversary. For convenience, we define $Y_{\min} := \min\{Y_1, \ldots, Y_n\}$ and $X_{\min} := \min\{X_1, \ldots, X_m\}$.

$$\mathbb{P}[Y_{\min} <_D X_{\min}] = 1 - (1 - \frac{1}{m+1})^n \tag{15}$$

Naturally, this is a lower bound on the probability, since a lookup normally does not yield the minimum node ID of the complete Ethereum network. The resulting probability is depicted in Figure 5 for different choices of n, the number of node IDs in the network. It can be seen that the more honest nodes there are, the smaller the probability for an adversary to have the minimal node ID becomes.

Note that we have to distinguish between node IDs in the discovery table and actual nodes on the main Ethereum network. The Ethereum protocol is running concurrently with other protocols on the same communication channels and packet structures; therefore the number of node IDs is ten times larger than the number of Ethereum nodes at roughly $3 \cdot 10^6$ node IDs [8]. In case of returning the smallest ID for a FindNode-Request, this behavior slightly raises the bar for an attacker.

The red and green lines depict a situation where every node ID would correspond to an actual node on the Ethereum network. The red line shows the probability for $n = 9000$ nodes, as reported by Ethernode[7]. The green line corresponds to $n = 25000$ nodes in the network, the sum of all approaches discussed in [8]. The blue line corresponds to an upper bound on the number of node IDs of $n = 5 \cdot 10^5$ nodes. In all cases, however, $5 \cdot 10^6$ pre-computed ECDSA key-pairs are enough to return the minimal node ID almost certainly.

VI. Evaluation

We evaluated the previously described concepts using a victim node deployed specifically for this task. The victim

[6] Technically, the sorting is w.r.t. to $<_D$. The xor-operation only induces a permutation of the x_i, preserving the uniform distribution. We therefore purposely omit the xor-operation for the ease of understanding.

[7] https://www.ethernodes.org/network/1

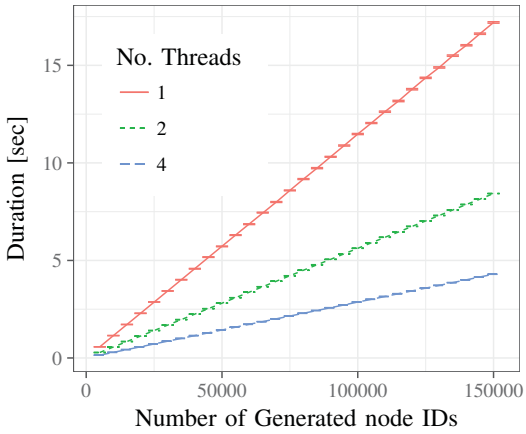

Figure 6. Mean duration of node ID generation with different numbers of parallel threads. The (very small) error bars show the 95% confidence interval for 100 runs.

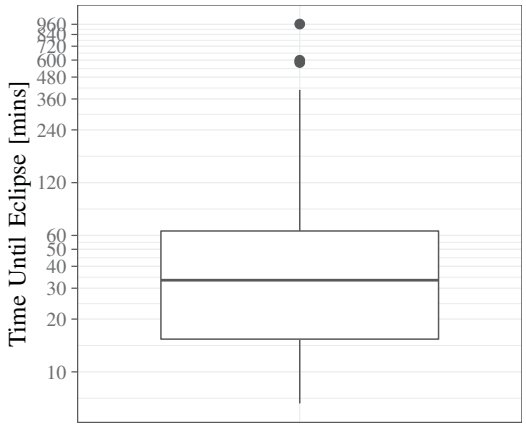

Figure 7. Log-scale box plot of attack durations when attacking a recently started victim. The plot depicts the median duration as well as the upper and lower quartile and all outliers outside 1.5 times the interquartile range.

had the latest Geth version from github (v1.8.20) and was connected to the Ethereum main network.

A. Pre-Computing Node IDs

The generation of node IDs is essential for placing a node in each bucket and to ensure that the lowest ID is returned during the lookup-process. Therefore, we measured the calculation time for new ECDSA key pairs and the corresponding hashes. Our measurements were conducted on a system with an Intel® Core™ i5-6600K processor with four logical cores. The results depicted in Figure 6 show that, on average, 35000 ECDSA keys and corresponding hashes can be generated per second when using four parallel threads. Generating a node ID to enter bucket number 239 (the smallest bucket) would therefore only take $7.5\,\mathrm{s}$ on average. For the real-world implementation of the attack, we pre-computed $5{\cdot}10^6$ node IDs to hijack the lookup-buffer. Computing this many IDs takes roughly $2.5\,\mathrm{min}$ using four parallel threads and $11\,\mathrm{min}$ with a single thread. Note that even when attacking different victim nodes this computation has to be performed only once, since the target of a lookup is random and does not depend on any victim-specific information.

B. Attack Implementation

First, to compare the performance of the attack to [6], we repeatedly attacked a recently restarted victim. Recall that we do not necessarily require that the victim is restarted, since our attack relies on high peer churn. The reliance on churn implies a delay, as previously established connections must be terminated before their slots can be occupied by an adversary. However, the waiting time is small for recently restarted nodes.

The attack was started immediately after the victim node became online. No connection slot was yet occupied, but the neighbor table always contained benign nodes, due to countermeasures introduced in [6]. We measured the time until every slot was filled with an attacker-controlled node, with a cutoff timeout of $24\,\mathrm{h}$ after which the experiment was restarted. One could argue that also in unsuccessful attempts, the victim would eventually have been eclipsed. The attack was repeated 50 times, out of which 45 times were successful within the cutoff timeout, whereas 5 attempts did not complete in that time. Figure 7 shows the results in a log-scale box plot. The box depicts the upper and lower quartile of measured durations, the median is indicated as a solid line inside the box. Measurements outside 1.5 times the interquartile range are considered as outliers, plotted as dots.

It can be seen that out of the 45 successful attacks $75\,\%$ completed in just over $60\,\mathrm{min}$, indicating that an adversary can eclipse recently restarted nodes within a reasonable time span. This finding implies that peer churn in the neighbor table and in the peer management is high in recently restarted nodes.

C. Distribution of Connection Durations

Since recently restarted nodes exhibit low-duration connections and high churn, the question arises if the same behavior is true for long-running nodes, i.e., how connection durations are distributed. To this end, we ran a unaltered Geth node for $18.76\,\mathrm{d}$ (450 hours) and logged the duration of every connection. The results are depicted in Figure 8, showing the cumulative distribution of durations in the trace. To improve the readability of the Figure, we excluded connections with a duration shorter than $60\,\mathrm{s}$, which were $90.26\,\%$ of all connections, yielding a total of 361 connections[8].

[8]We suspect the heterogeneity of the discovery table as the major cause for the abundance of connections shorter than $60\,\mathrm{s}$ [8]. Ethereum is embedded in a family of protocols (Section II), all of which use the same ports and message structures. Kim *et al.* [8] report that only $10\,\%$ of node IDs in the discovery table correspond to peers speaking the Ethereum protocol, out of which only $50\,\%$ operate on the Ethereum main network. Hence, in the majority of situations DEVp2p tries to establish connections to peers which are not useful and immediately discards them again.

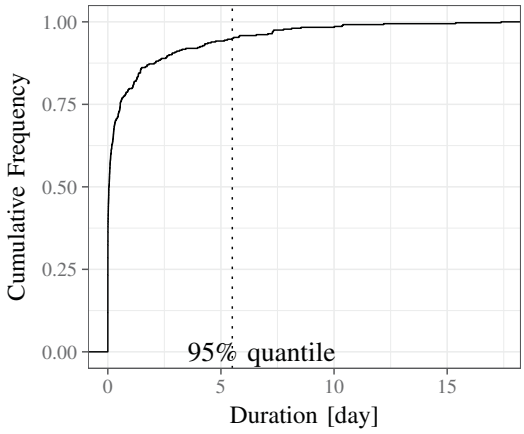

Figure 8. Cumulative distribution of durations in the trace. 95 % of the considered durations where shorter than 5.5 d.

It can be seen in Figure 8 that the longest connection duration was 17.38 d, but the majority of connections was much shorter lived. The quantile shows that 95 % of the considered connections were shorter than 5.5 d; only 18 connections where longer than this duration. Though the peer churn is lower than in a restarted node, it is still surprisingly high for which we have several conjectures: Geth's development cycle is fast and requires frequent updates, either due to security fixes or protocol changes. Additionally, read (write) timeouts of 20 s (30 s) on the TCP-level are relatively small compared to other networks like Bitcoin. On the UDP-level, timeouts are currently set at 500 ms, while [14] report average inter-node latencies of roughly 180 ms, with 10 % of peers having a latency higher than 276 ms. Consequently, buckets in the discovery table experience a high level of churn, making it easy to enter even fully filled ones.

Given that most connections on the Ethereum network are rather short-lived, we conducted a proof-of-concept attack without restarting the victim. We let the victim node run without any attack activity for 72 hours to populate the discovery table, mimicking a more realistic network state. In our experiment the false friends attack was successful after 4.765 d (114.4 hours). Repeated measurements on long-running nodes is subject to future work.[9]

VII. Countermeasures

Several quickly realizable modifications to Geth are conceivable that will immediately increase the costs of a successful false friends eclipse. One possibility are more stringent *IP subnet restrictions* in the discovery table and on the DEVp2p connection layer. While increasing the bucket size is not promising (cf. Section V-B), enforcing *any* subnet restriction on the replies of FindNode-Requests would increase the number of unique IP adresses necessary for a successful attack. Another low-invasive countermeasure is to consider all known nodes in `ReadRandomNodes`, instead of only the heads of each bucket. Furthermore, we are only able to perform our attack without requiring a restart of the victim node because peering relationships in Ethereum are currently very short-lived. Increasing timeouts on both the TCP- and UDP-level could decrease this volatility.

On a more fundamental level, we argue that the *complexity* of Geth's current node selection logic is a major enabler for attacks such as [6] and our false friends attack. New peers are chosen based on their node ID, which arguably does not make any sense if the goal of the resulting overlay is flooding identical information to all nodes (in contrast to ID-based routing). Node IDs are, however, trivially manipulatable by adversaries to optimize the placement of adversary nodes in peer discovery tables. The complexities of ID-based peer selection are therefore not only unnecessary, but also detrimental to security. For a sustainable, long-term fix we strongly suggest to ignore node IDs for all aspects of peer discovery for the Ethereum protocol. Instead, peering decisions should be weighted by more expensive-to-manipulate node characteristics, such as IP addresses or, perhaps, publicly locked Ether stake linked to individual nodes.

Decades of research on Sybil-attacks in peer-to-peer-networks [9] suggest that in a completely trustless setting it is only viable to make the creation of a multitude of adversarial nodes expensive, not impossible. The implications of this are twofold. First, in the typical blockchain scenario one honest node is sufficient to prevent an eclipse attack. Therefore, the probability of filling the whole peer list with adversarial nodes must be minimized by means that are robust to a potentially substantial population of Sybil nodes. We suggest that a promising approach for achieving this is a combination of maintaining a large peer set and using a node selection process which as closely as possible resembles a uniform draw from the set of all network nodes. For example, Geth's complex Kademlia-inspired bucket structure and discovery logic can be replaced with a single data structure holding a large number of nodes (peer candidates) from which peers are drawn uniformly at random. Second, nodes that are profitable targets for Eclipse attacks (high-profile merchants, miners) should not rely on a purely trustless node selection logic. Instead, these nodes should statically include known and trusted nodes into their peer list, as seems to be practice in the Bitcoin network [1]. In other words, potentially attractive targets might want to invest manual effort to *choose their friends wisely*.

VIII. Related Work

The literature on security considerations in peer-to-peer networks in general and Kademlia-based networks in particular is vast; in the following we focus on research most closely related to our attack instead of providing a broad overview.

A. Attacks and Countermeasures in Overlay Networks

Singh *et al.* [15], Castro *et al.* [16], and Sit *et al.* [17] survey eclipse attacks and countermeasures on peer-to-peer overlay

[9] In two subsequent experiments, the victim was left with only one benign connection after 4.875 d and 9.5 d, respectively (i.e., the node was only one connection away from being fully eclipsed).

networks. Similar to our reasoning, [15], [16] conjecture that Sybil attacks cannot be solved in a purely peer-to-peer fashion. As a remedy, they propose a central trusted certificate authority to bind node IDs to identities which subsequently enables the implementation of countermeasures not possible in purely trustless systems. A decentralized mitigation is proposed and thoroughly analyzed in [18], which is based on purposely letting lookups diverge, so that an adversary cannot easily eclipse a target by poisoning its logical proximity.

B. Attacks on Kademlia-based networks

The security Kademlia [7] and its inspired implementations have been studied extensively [10], [19]–[21]. Steiner *et al.* [19] explore the space of possible attacks and implications whereas subsequent works focus on optimizations of these attacks [10], [21] and circumventing implemented countermeasures [20]. Most approaches require the ability to arbitrarily choose node IDs. Similar to our false friend attack where we insert carefully selected node IDs into the victim's discovery table, [21] present a low-resource approach to poison routing entries in the KAD network. Given multiple attacking nodes, the ID space is partitioned and routing entries hijacked by spoofing messages. In Ethereum, message spoofing and arbitrary node ID choice are impossible, making our attack conceptually different, though closely related to previous attacks on the KAD network. Most notably, [10] also conjecture that a purely trustless countermeasure cannot exist, due to the fundamental problem of Sybil identities [9].

C. Eclipse Attacks in Blockchain Systems

Heilman *et al.* [1] were the first to study eclipse attacks on peer-to-peer Blockchain-systems, in particular Bitcoin. Despite the introduced countermeasures, eclipse attacks are still possible when exploiting BGP [22]. For Ethereum, [23] describe an attack on the block synchronization mechanism. When an Ethereum peer misses a block, it will start a synchronization with exactly one neighboring peer. An adversary can leverage this behavior to indefinitely stall the synchronization or inject an adversarial chain of blocks.

As noted throughout this paper, several eclipse attacks on Ethereum are described in [6]. Our approach differs since we do not fill the complete table with adversarial nodes instead insert node IDs with specific properties.

IX. Conclusion

We presented the *false friends* attack, an eclipse attack applicable to current versions of Geth, the by far most popular Ethereum node software. Our attack requires only 2 IPs from distinct /24 subnets to be successful. Moreover, and in contrast to previous attacks, it can be successfully mounted without assuming that the victim node reboots at some point. Empirical measurements of our attack in the live Ethereum mainnet indicate that even without a restart of the victim node, a false friends eclipse can be completed in a matter of days. Our discovery is even more striking when considering that countermeasures against similar attacks were only recently introduced to the Geth codebase. We argue that the ongoing vulnerability of Geth is at least partly due to a fundamentally unsuited node discovery approach. While we propose both short- and long-term countermeasures to the false friends attack, existing literature hints that in a completely trustless setting, eclipse attacks can only be made expensive, not impossible. Potentially attractive targets might wish to invest manual effort towards choosing their friends wisely.

Acknowledgements

We would like to thank to Holger Döbler, Roman Naumann, Elias Rohrer, Samuel Brack, Till Neudecker and Florian Tschorsch for the highly valuable discussions on the topic.

References

[1] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on bitcoin's peer-to-peer network," in *Proc. of USENIX Security Symposium*, 2015.

[2] F. Tschorsch and B. Scheuermann, "Bitcoin and beyond: A technical survey on decentralized digital currencies," *IEEE Communications Surveys and Tutorials*, vol. 18, no. 3, 2016.

[3] K. Nayak, S. Kumar, A. Miller, and E. Shi, "Stubborn mining: Generalizing selfish mining and combining with an eclipse attack," in *Proc. of EuroS&P*, IEEE, 2016.

[4] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[5] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014.

[6] Y. Marcus, E. Heilman, and S. Goldberg, "Low-resource eclipse attacks on ethereum's peer-to-peer network," *IACR*, vol. 236, 2018.

[7] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the XOR metric," in *Proc. of IPTPS*, Springer, 2002.

[8] S. K. Kim, Z. Ma, S. Murali, *et al.*, "Measuring ethereum network peers," in *Proc. of IMC*, ACM, 2018.

[9] J. R. Douceur, "The sybil attack," in *Peer-to-peer Systems*, Springer, 2002.

[10] T. Locher, D. Mysicka, S. Schmid, and R. Wattenhofer, "Poisoning the kad network," in *Proc. of ICDCN*, ACM, 2010.

[11] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak," in *Annual international conference on the theory and applications of cryptographic techniques*, Springer, 2013.

[12] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering - Design Principles and Practical Applications*. Wiley, 2010.

[13] J. E. Gentle, *Computational statistics*. Springer, 2009, vol. 308.

[14] A. E. Gencer, S. Basu, I. Eyal, R. van Renesse, and E. G. Sirer, "Decentralization in bitcoin and ethereum networks," *CoRR*, vol. abs/1801.03998, 2018. eprint: 1801.03998.

[15] A. Singh, T. Ngan, P. Druschel, and D. S. Wallach, "Eclipse attacks on overlay networks: Threats and defenses," in *Proc. of INFOCOM*, IEEE, 2006.

[16] M. Castro, P. Druschel, A. J. Ganesh, A. I. T. Rowstron, and D. S. Wallach, "Secure routing for structured peer-to-peer overlay networks," in *Proc. of OSDI*, USENIX Association, 2002.

[17] E. Sit and R. T. Morris, "Security considerations for peer-to-peer distributed hash tables," in *Prof. of IPTPS*, Springer, 2002.

[18] D. Germanus, S. Roos, T. Strufe, and N. Suri, "Mitigating eclipse attacks in peer-to-peer networks," in *Proc. of CNS*, IEEE, 2014.

[19] M. Steiner, T. En-Najjary, and E. W. Biersack, "Exploiting KAD: possible uses and misuses," *CCR*, vol. 37, no. 5, 2007.

[20] M. Kohnen, M. Leske, and E. P. Rathgeb, "Conducting and optimizing eclipse attacks in the kad peer-to-peer network," in *Proc. of IFIP NETWORKING*, Springer, 2009.

[21] P. Wang, J. Tyra, E. Chan-Tin, *et al.*, "Attacking the kad network - real world evaluation and high fidelity simulation using DVN," *Security and Communication Networks*, vol. 6, no. 12, 2013.

[22] M. Apostolaki, A. Zohar, and L. Vanbever, "Hijacking bitcoin: Routing attacks on cryptocurrencies," in *Proc. of S&P*, IEEE, 2017.

[23] A. Gervais and K. Wüst, "Ethereum eclipse attacks," ETH Zurich, 2016.