Low-Resource Eclipse Attacks on Ethereum's Peer-to-Peer Network

Yuval Marcus*† Ethan Heilman* Sharon Goldberg*
* Boston University
†University of Pittsburgh

***Abstract*—We present eclipse attacks on Ethereum nodes that exploit the peer-to-peer network used for neighbor discovery. Our attacks can be launched using only two hosts, each with a single IP address. Our eclipse attacker monopolizes all of the victim's incoming and outgoing connections, thus isolating the victim from the rest of its peers in the network. The attacker can then filter the victim's view of the blockchain, or co-opt the victim's computing power as part of more sophisticated attacks. We argue that these eclipse-attack vulnerabilities result from Ethereum's adoption of the Kademlia peer-to-peer protocol, and present countermeasures that both harden the network against eclipse attacks and cause it to behave differently from the traditional Kademlia protocol. Several of our countermeasures have been incorporated in the Ethereum geth 1.8 client released on February 14, 2018.**

I. Introduction

The Ethereum cryptocurrency is currently second only to Bitcoin in terms of market capitalization.Ethereum has also emerged as a leading platform for raising capital through the sale of tokens [9] hosted by Ethereum smart contracts; at the end of 2017, there were over 20K such tokens, with a total market capitalization of several billion USD [11]. From a technical perspective, Ethereum is interesting because it is so different from Bitcoin. Both Bitcoin and Ethereum use proof-of-work algorithms to drive consensus, but Bitcoin's consensus algorithm uses the simple longest-chain rule [35], while Ethereum's consensus algorithm is based on the more elaborate GHOST protocol [42] and may eventually move to a proof-of-stake consensus [20]. Both Bitcoin and Ethereum support smart contracts, but Bitcoin smart contracts must be written in a highly-restrictive assembly-like language [7], while Ethereum's language is Turing complete [14], [18]. Both Bitcoin and Ethereum uses a peer-to-peer network to communicate the state of their blockchains, but Bitcoin's network is meant to emulate a unstructured random graph [23], while Ethereum's [10] is meant to emulate a structured graph based on the Kademlia DHT [33], [16].

The paper was disclosed via Ethereum's bug bounty program on January 9, 2018, and then lightly updated to reflect the patching applied to the Ethereum geth 1.8 (iceberg) client on February 18-27, 2018. This paper was first posted online on March 1, 2018 and last updated on March 1, 2018.

Despite the growing body of research on the security of Ethereum's consensus algorithm [21], [25], [26], [42] and scripting language [18], [30], the properties of Ethereum's peer-to-peer network have been largely unexplored. Nevertheless, a recent line of work [21], [37], [23], [22] has made it clear that the security properties of a proof-of-work blockchain rests on the security of its underlying peer-to-peer network. Simply put, if the peer-to-peer network partitions, causing different nodes see different views of the blockchain, then how can these nodes actually come to a consensus about what the blockchain actually is?

In [23], Heilman *et al.* highlighted these risks by demonstrating the first *eclipse attacks* on Bitcoin's peer-to-peer network. In an eclipse attack, the attacker completely controls its victim's access to information, and thus filter the victim's view of the blockchain, or co-opt the victim's computing power as part of more sophisticated attacks [37], [21]. In this paper, we consider eclipse attacks where the attacker monopolizes all of the victim's incoming and outgoing connections, isolating the victim from the rest of her peers in the network.

A. Why Ethereum is vulnerable to eclipse attacks

To conventional wisdom (*e.g.,* [36]) suggests that Ethereum's peer-to-peer network is more resilient to eclipse attacks that than that of Bitcoin. After all, Bitcoin nodes make only eight outgoing connections by default, while Ethereum nodes make thirteen outgoing connections by default. Also, Ethereum's peer-to-peer network has cryptographically authenticated messages, while Bitcoin nodes do not authenticate peer-to-peer network messages; this means that the Bitcoin peer-to-peer network is vulnerable to attacks on integrity by man-in-the-middle attackers [23], [17] and via manipulations of BGP, the Internet's routing protocol [17].

We demonstrate that the conventional wisdom is false. We present new eclipse attacks showing that, prior to the disclosure of this work in January 2018, Ethereum's peer-to-peer network was significantly less secure than that of Bitcoin. Our eclipse attackers need only control two machines, each with only a single IP address. The attacks are *off-path*—the attacker controls endhosts only, and does not occupy a privileged position between the victim and the rest of the Ethereum network. By contrast, the best known *off-path* eclipse attacks on Bitcoin [23] require the attacker to control hundreds of host machines, each with a distinct IP address.[1] For most Internet users, it is far from trivial to obtain hundreds (or thousands) of IP addresses. This is why the Bitcoin eclipse attacker envisioned by [23] was a full-fledged botnet or Internet Service Provider, while the BGP-hijacker Bitcoin eclipse attacker envisioned by [17] needed access to a BGP-speaking core Internet router. By contrast, our attacks can be run by any kid with a machine and a script.

The key issue we exploit is that nodes in the Ethereum network are identified by their cryptographic ECDSA public key. Remarkably, Ethereum versions prior to geth v1.8 allow one to run an unlimited number of Ethereum nodes, each with a different ECDSA public key, from the same single machine with the same single IP address. Because generating a new ECDSA public key is trivial—one need only run the ECDSA key generation algorithm—our attacker can trivially create thousands of Ethereum node IDs in seconds, without expending significant compute resources. Our attacker therefore generates a set of Ethereum node IDs, and then uses a coordinated strategy to cheaply launch eclipse attacks from two host machines, (each) with just a single IP address. Worse yet, Ethereum nodes form connections to peers in biased fashion (*i.e.,* some node IDs are more likely to become peers than others) that is easily predicted by the attacker. Therefore, our attacker carefully selects his node IDs so that the victim is more likely to connect to attacker node IDs, rather than legitimate ones.

We disclosed our attacks and their countermeasures to the Ethereum bug bounty program on January 9, 2018. Some of our countermeasures were adopted in the geth 1.8 client; see Section V.

B. Our results

The Ethereum developers [10] state that the Ethereum peer-to-peer network protocol is based on the Kademlia DHT [33]. However, the design goals of the two are dramatically different. Kademlia provides an efficient means for storing and finding content in a distributed peer-to-peer network. Each item of content (*e.g.,* a video) is stored at small subset of peers in the network. Kademlia ensures that each item of content can be discovered by querying no more than a logarithmic number of nodes in the network. By contrast, the Ethereum protocol has just one item of content that all nodes wish to discover: the Ethereum blockchain. The full Ethereum blockchain is stored at each Ethereum node. As such, Ethereum's peer-to-peer network is not needed for content discovery; it is only used to discover new peers. This means that Ethereum inherits most of the complicated artifacts of the Kademlia protocol, even though it rarely uses the key property for which Kademlia was designed .[2]

Exposition. Our first contribution is a detailed exposition of Ethereum's peer-to-peer network and its relationship to the Kademlia protocol. Since the network is largely undocumented (apart from some short discussion in [16], [3], [4]) we present a new exposition developed by reverse engineering the code of the popular Ethereum geth client (Section II).

Attacks. Our second contribution is two off-path eclipse attacks and one attack by manipulating time:

Eclipse by connection monopolization (Section III). We exploit the fact that the network connections to Ethereum client may *all* be *incoming*, *i.e.,* initiated by other nodes. Thus, our attacker waits until the victim reboots (or deliberately forces the victim to reboot by sending a packet-of-death to the Ethereum client [15] or the host OS [1], [2]), and then immediately initiates incoming connections to victim from each of its attacker nodes. The victim is eclipsed once all connection slots are occupied by the attacker.

Eclipse by owning the table (Section IV). Our connection-monopolization attack can be trivially eliminated by forcing Ethereum clients to make connections that are both *incoming* (initiated by other nodes) and *outgoing* (initiated by the client). But even if such countermeasure was adopted, we show that Ethereum is still vulnerable to low-resource eclipse attacks. To that end, we present an eclipse attacker that uses a carefully-crafted set of node identifiers to repeatedly ping the victim. When the victim restarts, the victim forms all thirteen of her outgoing connections to the attacker with high probability. To complete the eclipse, the attacker monopolizes the remaining connection slots

[1]Also, since the release of [23] in 2015, Bitcoin has put in countermeasures that have significantly raised the bar for eclipse attacks.

[2]In 2014, the Ethereum developers [3] wrote: "Kademlia-style network well-formedness guarantees a low maximum hop distance to any peer in the network and its group." However, Ethereum rarely requires some node a to "find" some specific other node b, so this "well-formedness" appears to be unnecessary. The only time this feature is used is when a one node wishes to resolve any given node ID (*i.e.,* ECDSA public key) to its IP address, and this seems only to be used when the user of a node supplies statically-configured nodes to the node's peer-to-peer network, or if a node ID's corresponding IP address is missing; see Section II-G. There have been some rumblings about "sharding", where each Ethereum node need only store a small fraction of the blockchain. ("... at the P2P level, ... a broadcast model is not scalable since it requires every node to download and re-broadcast O(n) data (every transaction that is being sent), whereas our decentralization criterion assumes that every node only has access to O(c) resources of all kinds." [13], [12].) It is possible that Kademlia was chosen for the peer-to-peer network in order to support sharding, even though sharding has not been deployed.

with unsolicited incoming connections from attacker node identifiers.

Attack by manipulating time (Section VI. We also present an attack that can cause a node to be eclipsed if it local clock is more than 20 seconds ahead of the other nodes in the Ethereum network. Such an attack can be accomplished *e.g.,* by manipulating the network time protocol (NTP) used by the host running the Ethereum node [31], [32].

Countermeasures (Section V). Our third contribution is set of countermeasures that can be used to prevent these attacks. Our most important recommendation is that Ethereum stop using ECDSA public keys as the sole node identifier; a combination of IP address and public key should be used instead. Beyond this, we show how to harden Ethereum by via design decisions different from those that are traditionally part of the Kademlia protocol. Several of our countermeasures have been adopted in geth v1.8, released February 14, 2018.

C. Implications of Eclipse Attacks

Given the increasing importance of Ethereum to the global blockchain ecosystem, we think its imperative that countermeasures preventing them be adopted as soon as possible. Ethereum node operators should immediately upgrade to geth v1.8. We briefly discuss the implications of leaving these vulnerabilities unpatched.

Attacks on consensus. Eclipse attacks can be used to co-opt a victim's mining power and use it to attack the blockchain's consensus algorithm [23], [37], [21]. [21] showed how eclipse attacks can be used as part of optimal adversarial strategies for double spending and selfish mining.

Attacks on blockchain layer-two protocols. In a blockchain layer-two protocol (*e.g.,* Bitcoin's Lightning Network [38], Ethereum's Radian network [8]), pairs of users post transactions on the blockchain in order to establish a payment channel between the two users. The users can then pay each other using transactions that are *not* posted to the blockchain; these off-blockchain payments are fast, because they are not subject to blockchain-related performance bottlenecks. Finally, the payment channel is closed by posting on-blockchain transactions that reflect the new balance of coin between the two users. Importantly, the security of these protocols requires that no off-blockchain payments are sent after the payment channel is closed. Thus, an eclipse attacker can trick his victim into the thinking the payment channel is still open, even while the non-eclipsed part of the network sees that payment channel is closed. If the victim is *e.g.,* a merchant that releases goods in exchange for off-blockchain payments, then the attacker can once again obtain the goods without paying.

Attacks on smart contracts. Ethereum's smart contracts have several unique properties. For instance, smart contracts can contain variables that have hold state that can changed by transactions that are posted to the Ethereum blockchain. [18] points out that Ethereum smart contracts may be attackable if users see inconsistent views of the blockchain; we point out that an eclipse attack can be used to inject these inconsistencies.

As a simple example, consider an Ethereum smart contract that is used to auction off a digital cat. The contract has variable x that counts the number of bids made on the cat. Alice might only be willing to expend Ether to send a transaction bidding on the cat iff $x < 5$. An eclipse attacker could show Alice a view of the blockchain where $x < 5$, causing Alice to sign a bid transaction T. The attacker then sends T to the non-eclipsed portion of the network, tricking Alice into bidding on the cat even though $x > 5$.

II. Ethereum's Peer-to-Peer Network

This section provides details about Ethereum's peer-to-peer network based on the geth client. Geth is the most popular Ethereum client.[3] Our description is of geth's main neighbor discovery protocol, known as the RLPx Node Discovery Protocol v4 [6], which, prior to the disclosure of this paper and the release of geth v1.8.0 on February 14, 2018, has been largely unmodified from that released with the very first geth client in 2015 (when Ethereum first came online). A newer v5 version exists, but is still considered experimental, and is currently only used by Ethereum 'light node' clients. This paper uses geth version 1.6.6, released on June 23, 2017.

A. Kademlia similarities and differences

While Ethereum's peer-to-peer network is based [10] on the Kademlia DHT [33], its purpose is quite different. Kademlia is designed to be an efficient means for storing and finding content in a distributed peer-to-peer network. Ethereum's peer-to-peer network is only used to discover new peers.

In the Kademlia network, each item of content is associated with a key (a b-bit value), and is stored only at those peers whose node ID (a b-bit value) that is "close" to its associated key. Kademlia's notion of "closeness" is given by the XOR metric, so that that the distance between b-bit strings t and t' is given by their bitwise exclusive or (XOR) $t \bigoplus t'$, interpreted as an integer. Each Kademlia node has a datastructure consisting of b distinct *buckets*, where bucket i stores network information about k peers at distance i (from her node ID per to the XOR metric). To look up a target item of content associated with key t, a Kademlia node looks

[3]As of January 4, 2017, more the 70% of Ethereum nodes run geth, according to Ethernodes.org.

in her buckets to find the node IDs that are "closest" to t, and asks them to either (a) return the content associated with t, or (b) to return some node IDs that are even "closer" to t. This lookup process proceeds iteratively until the key is found. The number of nodes that queried in a lookup is logarithmic in the number of nodes in the network; this is the key property for which the Kademlia protocol was designed.

Ethereum uses the same XOR metric and the same bucket data structure. However, Ethereum nodes have no need to identify which peers store a target item of content (since there is only 'item of content'—the Ethereum blockchain—that is stored by all peers). As such, lookup is mostly used to discover new peers. (There is a small and rare exception to this, when resolving a node ID to its IP address. See Section II-G.) To do so, the Ethereum node chooses a random target t, looks in her buckets to find $k = 16$ node IDs closest to the target t, and asks them each to return k node IDs from their buckets that "closer" to the target t, resulting in up to $k \times k$ newly discovered node IDs. From these $k \times k$ newly-discovered node ID, the k nodes closest to the target t are then asked to return k nodes that are even closer to t. This process continues iteratively until no new nodes are found. In other words, Ethereum lookup is mostly a fancy way to populate buckets with randomly-chosen node IDs.

We now proceed with a detailed exposition of Ethereum's peer-to-peer network.

B. Node IDs.

Peers in the Ethereum network are identified by their node IDs. A node ID is a $b = 512$ bit (64 byte) cryptographic ECDSA public key. Multiple Ethereum nodes, each with a different node ID, can be run on a single machine that has a single IP address. It is easy to generate an ECDSA key—one need only run the ECDSA key generation algorithm. There is no mechanism that checks that unique node IDs correspond to unique network addresses; thus, it is possible to run an unlimited number of nodes from the same machine with the same IP address. This is the main vector exploited in our attacks.

C. Network connections.

UDP connections. UDP connections are used only to *exchange information about the peer-to-peer network*. There is no limit on the number of UDP connections, except that at most 16 UDP connections can be made concurrently.

There are four types of UDP messages. A *ping* message solicits a *pong* message in return. This pair of messages is used to determine whether a neighboring node is responsive. A *findnode* message solicits a *neighbor* messages that contains a list of 16 nodes that have been seen by the responding node. (A node will only respond to a *findnode* request if the querying node is already in his `db`, see Section II-D.)

All UDP messages are timestamped and cryptographically authenticated under the sender's ECDSA key (*aka,* the sender's node ID). To limit replay attacks, the client drops any UDP message whose timestamp is more than 20 seconds older than the client's local time. To prevent *pong* messages from being sent from spoofed IP addresses, the *pong* also contains the hash of the *ping* to which it is responding.[4]

TCP connections. Meanwhile, *all blockchain information* is exchanged via encrypted and authenticated TCP connections. The total number of TCP connections at any given time is `maxpeers`, which is set to 25 by default. To eclipse a node, the attacker must continuously occupy all `maxpeers` of the target's TCP connections.

A TCP connection is *outgoing* if it was initiated by the client (*i.e.,* the client sent the TCP SYN packet) and *incoming* otherwise. A client can initiate up to $\lfloor \frac{1}{2}(1 + \texttt{maxpeers}) \rfloor$ (13, by default) outgoing TCP connections with other nodes. By contrast, prior to geth v1.8.0, there was no limit on the number of unsolicited incoming TCP connections, other than `maxpeers`. This means that a client could have *all* `maxpeers` of its TCP connections be unsolicited incoming connections, a fact we exploit in our brute-force eclipse attack of Section III.

D. Storing network information

A client stores information about other nodes in two data structures. The first is a long-term database, called `db`, which is stored on disk and persists across client reboots. The second is a short-term database, called `table`, which contains Kademlia-like *buckets* [33] that are always empty when the client reboots. The `db` is used for long-term storage of network information, while the `table` is used to select peers (*i.e.,* outgoing TCP connections), as described in Section II-G.

The `db`*.* The `db` is stored on disk, and contains information about each node that the client has seen. (A node has been *seen* if it responds to a *ping* message sent by the client with a valid *pong* response.) There is no limit to the size of the `db`.

Each `db` entry consists of a node ID, IP address, TCP port, UDP port, time of last *ping* sent to the node, time of last *pong* received from the node, and number of times the node failed to respond to a *findnode* message. A node's age is the time elapsed since the time of last *pong* received from the node. Each hour (starting from time of the first successful bonding, see Section II-E),

[4] Unfortunately, however, prior to geth v1.8.0, geth v4 nodes did not check this hash value. (This seems to be an implementation flaw, since geth v5 nodes do check the hash value.)

the client runs an eviction process that removes nodes from the `db` that are older than 1 day.

`table`. The `table` is always empty when the client reboots. The `table` consists of 256 *buckets*, each of which can hold up to $k = 16$ entries. Each entry records information about another Ethereum node—specifically, its node ID, IP address, TCP port, UDP port. Entries in each bucket are sorted in the order in which that were added to the bucket. When the client discovers a new node that maps to a bucket that is already full, the client pings the last node (*i.e.,* oldest) in the bucket. Prior to geth v1.8.0, if this old node fails to respond with a *pong*, the new node is added to the bucket and the old node is pushed out; otherwise, the new node is not added to the bucket. Because *ping* messages are sent via UDP, the client can still send them even if all of its available TCP connections are in use.Nodes are also removed from the `table` when they fail to respond to the client's *findnode* request more than four times. All of the above is similar to how buckets are maintained in the Kademlia protocol [33].

Nodes are mapped to buckets according to the `logdist` function, Ethereum's modification of the XOR metric used in the Kademlia protocol [33]. The `logdist` function measures the distance between two node IDs as follows. First, each node ID is hashed with SHA3 to a 256-bit value. If the r most significant bits of the two hash values are the same, but the $r + 1$st bit of the two hash values is different, then the `logdist` is r. A node ID that has `logdist` r from client's node ID is mapped to bucket $256 - r$ of the client's `table`. Crucially, the mapping from node IDs to buckets in the `table` is public (so that its trivial for an adversary to predict which node ID will map to which bucket in a victim's table). We exploit the public nature of this mapping (inherited from the Kademlia protocol) in our eclipse attack of Section IV. Indeed, in Section V we explain why Ethereum should not make this mapping public.

Another crucial fact affecting our eclipse attacks, is that `logdist` results in a highly-skewed mapping of nodes to buckets. The probability[5] that a node will map to bucket $256 - r$ is

$$p_r = \tfrac{1}{2^{r+1}} \qquad (1)$$

Most nodes therefore map into the last few buckets, and the vast majority of the other buckets remain empty. Specifically, we expect half the nodes to map to Bucket 256, a quarter of nodes to Bucket 255, ..., and only a vanishingly-small $\frac{1}{2^{256}}$-fraction of nodes map to Bucket

[5]This follows because SHA3 maps each node ID to a random 256-bit string. The probability that the second string has its first r bits identical to the first string, and its $r + 1$st bit different, is as in equation (1).

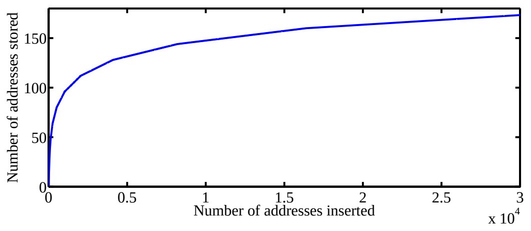

Fig. 1. Expected number of nodes actually stored in the `table`, versus the number of nodes inserted into the `table`.

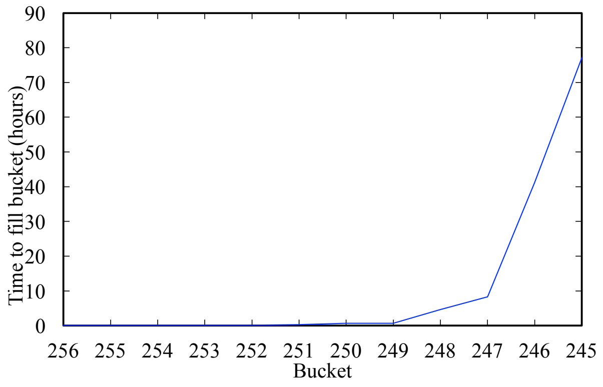

Fig. 2. Time (min) it takes for each bucket to fill to its max capacity of 16 nodes. Results shown for a 27-day old node run out of a data center in New York from July 14 to August 10 2017.

1. A client that has inserted N nodes into its `table` can expect the table to store only

$$E[\text{nodes stored in } \texttt{table}] = \sum_{i=1}^{256} \max(16, N\tfrac{1}{2^{i+1}}) \quad (2)$$

nodes. To put this is perspective, the `table` has a capacity of $256 \times 16 = 4096$ nodes, but a client that has inserted $N = 25,000$ nodes into the `table` (*i.e.,* essentially the entire Ethereum network as of August 2017) can expect the `table` to store only 168 nodes. Figure 1 expands upon this point by plotting the expected number of nodes stored in the `table` versus the number of nodes the client attempted to insert into the `table`.[6]

Our experiments confirm this behavior. Figure 2 presents the time it takes to fill each bucket to its maximum capacity, for a node we ran for 27 days out of a datacenter in New York. We can see that bucket 245 was full after about 3.2 days while bucket 246 only managed to completely fill for a short 2.5-day period out of the entire 27-day lifetime of the node.

E. Populating the data structures

There are several ways to populate the `db` and `table`.

[6]In recognition of the fact that most of the lower-number buckets are empty, geth v1.8.0 reduced the total number of buckets from 256 to only 17.

Boostrap nodes. When a client that is booted up for the first time, it has an empty db, and only knows about six hardcoded *bootstrap nodes*.

Bonding. The *bonding* process is used to populate both the db and table, as follows. When the client bonds with a node, the clients first checks if

1) the node exists in his db,
2) the db records zero failed responses to *findnode* requests, and
3) the db records that the node has responded with a *pong* within the last 24 hours.

If so, the client immediately tries to add the node to its table. (The node is actually added to the table only if there is space, see Section II-D). Otherwise, the client send a *ping* to the node. If the node responds with a *pong*, we say that bonding is *successful*. If bonding is successful, the client adds/updates the node's entry in its db, and also tries to add the node to it's table.[7]

Unsolicited pings. The client receives an unsolicited *ping* from another node, the client responds with a *pong* and then bonds to the node.

Lookup. The client can also discover nodes using the iterative lookup (t) method. (This is similar to Kademlia's lookup; see Section II-A.)

The lookup (t) method relies on the following notion of "closeness" to a target t, where t is a 256-bit string. Let a and b be two node IDs, and let

$$d_A = \text{SHA3}(a) \oplus t$$
$$d_B = \text{SHA3}(b) \oplus t$$

where $\oplus$ is the bitwise XOR. Then a is "closer" to t if $d_A < d_B$; otherwise, b is "closer" to t. (This is Kademlia's XOR metric!)

The lookup function uses this notion of closeness to discover nodes as follows. First, the 16 nodes in the table that are closest t are selected. Second, the client queries each of these 16 nodes with a *findnode* message. The *findnode* message contains the target t. Upon receiving a *findnode* message from the client, the other node identifies the 16 nodes "closest" to t that are in its own table data structure, and returns these 16 nodes to the client in a *neighbor* message. The client obtains information about up to 16 new nodes from each of the 16 nodes that it queried. The client now has information about up to $16 \times 16 = 256$ new nodes, and bonds with each of these 256 new nodes.

Finally, the client combines (1) the set of 16 nodes that were queried with a *findnode* message, and (2) the set of (up to 256) new nodes to which the client bonded successfully. From this combined set of nodes, it identifies the 16 nodes that are closest to the target t. It then repeats the process with these 16 closest nodes (querying each node (that has not been queried in a previous iteration) with a *findnode* message containing the target t, learning about up to 16 new nodes from each node, and then identifying the 16 nodes closest to t from the new set of $16 \times 16 + 16$ nodes. This continues iteratively until the set of 16 closest nodes stabilizes (*i.e.,* remains unchanged after an iteration). The 16 closest nodes are then added to the lookup_buffer data structure, which is a FIFO queue.

If a node fails to respond to the client's *findnode* request five times in a row (as recorded in the db), then lookup evicts that node from the client's table.

F. Seeding

An Ethereum client also has a seeding process. Prior to geth v1.8.0, the seeding process could trigger in three ways: (1) when the node reboots, (2) every hour, and (3) if lookup () is called on an empty table.

The seeding process first checks if the table is non-empty. If so, nothing happens. Otherwise, the client (1) bonds to each of the six bootstrap nodes, and (2) bonds to $\min(30, \ell)$ seed nodes that are randomly-selected nodes from the db and are no more than 5 days old (where ℓ is the number of nodes in the db that are no more than 5 days old). When this bonding is complete, the client runs lookup (self), where self is the SHA3 hash of the client's own node ID. (The use of lookup (self) when seeding is inherited directly from Kademlia [33]. It is intended to populate other nodes' buckets with the client's newly-online node ID.)

Our eclipse attack in Section IV exploits the fact that the seeding process does nothing when the table is non-empty.

*G. Selecting peers (*i.e.*, outgoing TCP connections)*

Very roughly speaking, an Ethereum client selects half of its outgoing TCP connections from its lookup_buffer, and half from its table. More precisely, outgoing TCP connections are established as follows.

When an Ethereum client boots up, a task runner is started and runs continuously. The task runner populates the client's db and table and creates up to $\lfloor \frac{1}{2}(1 + \texttt{maxpeers}) \rfloor$ (by default 13) outgoing TCP connections to other nodes on the Ethereum network.

Task runner. The task runner has a max of 16 concurrent tasks, along with a queue of tasks that should be run. Whenever there are less than 16 concurrently running tasks, the task runner runs each task in the queue, until the maximum of 16 is reached. If the there are

[7] geth 1.8.0 modifies this behavior so that a node must be in the table for at least 5 minutes before it is added to db. However, this does not have a significant effect on our eclipse attacks, simply because it's easy for the attacker to stay online for at least 5 minutes.

still fewer than 16 concurrently running tasks, then new tasks are created per the task-creation algorithm below and then run. Any new tasks that have not been run (because the maximum of 16 concurrently-running tasks has been reached) are pushed to the task-runner queue. There are two types of tasks: `dial_task` and `discover_task`.

A `discover_task` is a call of `lookup` (t) where t is a random 256-bit target string. (See Section II-E.)

A `dial_task` is an attempt to make a new TCP connection, or *dial*, to another node. Before creating a `dial_task` for a node, the task runner first runs the following five checks: checking that the node is

1) not currently being dialed,
2) not already a connected peer,
3) not itself,
4) not blacklisted, and
5) not recently dialed.

Resolving an unknown IP. A `dial_task` is called on a node ID. Typically, the client knows the IP address associated with a node ID, and so the TCP connection can easily be initiated. However, there are two rare cases where the client does not know the IP address associated with a node ID n: (1) when the node ID was statically configured by a user (without its IP address), or (2) if the IP address field is empty. In these rare cases, the client uses the traditional Kademlia iterative content resolution process to resolve the node ID n to its IP address, *i.e.,* calling `lookup` (n). To the best of our knowledge, this is the *only* place the Ethereum peer-to-peer network uses the iterative content lookups that Kademlia was designed to enable.

Task-creation algorithm. Initialize x as $\kappa = \lfloor \frac{1}{2}(1 + \texttt{maxpeers})\rfloor$ (by default 13).

1) Decrement x for each outgoing peer connection that is either (a) currently connected, or (b) currently being dialed (by a `dial_task`).
2) If the client has no peers, and more than 20 seconds have elapsed since the client restarted, selects a bootstrap node (*i.e.,* one of the six nodes that are hardcoded in the geth code). If the five checks listed above pass, decrement x and create a `dial_task` to the selected bootstrap node.
3) Fill `random_buffer`, which has a capacity of $\lfloor \frac{1}{2}(\lfloor \frac{1}{2}(1 + \texttt{maxpeers})\rfloor)\rfloor$ nodes (by default 6), with nodes selected uniformly at random from the `table`. If the `random_buffer` is non-empty, overwrite any nodes in the buffer. Select the first $\lfloor \frac{1}{2}(x)\rfloor$ nodes in the buffer. Decrement x and create a `dial_task` for each selected node that passes the five checks above.
4) While $x > 0$, remove the first node in the `lookup_buffer`, and if that node passes the five checks above, decrement x and create a `dial_task` to that node. If the size of the `lookup_buffer` is less than x, and `lookup` function is not running, create a new `discover_task`.

III. Eclipse by Monopolizing Connections.

Our first attack is simple. The client can be eclipsed if the attacker establish `maxpeers` incoming TCP connections (to its own adversarial nodes) before the client has a chance to establish any outgoing TCP connections.

A. The attack.

Our attack exploits several facts. First, we use the fact that while an Ethereum client can have at most `maxpeers` TCP connections, all of these connections can be incoming (*i.e.,* initiated by other nodes). Second, we use the fact that when a client reboots, it has no incoming or outgoing connections. Third, we use the fact that when a client reboots, it quickly starts its TCP and UDP listeners, but then takes a long time to establish outgoing connections.[8]

Therefore, our attacker creates $N \gg \texttt{maxpeers}$ attacker node IDs (by randomly-generating N ECDSA keys), waits until the victim reboots, and then immediately starts cycling through the N attacker node IDs, making incoming TCP connections to the victim. The victim is eclipsed if all its `maxpeers` incoming TCP connections are occupied by the attacker before the victim has a chance to make any outgoing connections.

Vulnerability: Reboot. As in the eclipse attacks of [23], our attack requires the victim to reboot. There are several reasons why a victim might reboot, including outages, power failures, or attacks on the host OS. Software updates to the geth Ethereum client also present predictable reason to reboot. Reboots can also be elicited using DDoS attacks that exhaust memory or bandwidth, or as a result of a "packet of death" that crashes the host [1], [2]) or the Ethereum client (see*e.g.,* [15]). As noted by [23], "the bottom line is that the security of the peer-to-peer network should not rely on 100% node uptime".

B. Experiments.

We tested our attack on live Ethereum victim nodes. Each victim node was instrumented and modified to send out no node information in its *neighbor* messages (to avoid polluting the production Ethereum network with our attacker node IDs), but was otherwise unchanged from the regular geth v1.6.6 client. This node had the default value $\texttt{maxpeers} = 25$.

[8]For a DigitalOcean node running with 2GB RAM, 2 vCPU, it took 2 seconds for the TCP listener to start and then 8 more seconds before `lookup (self)` was called (see Section II-F).

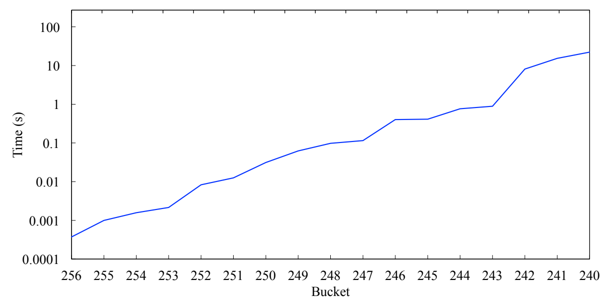

Fig. 3. Time (in seconds) required to craft an attacker node that maps to a specific bucket in the victim's `table`. We report an average of 5 runs, computed on a MacBook Pro with Intel Core i5: 2.9 GHz and 16 GB RAM.

In one experiment, we had a victim in New York, $N = 1000$ attacker node IDs on two attacker machines. One attacker was in New York (upload speed: 60Mbps) and one was in Boston (upload speed: 22Mbps). Our victim had been online for a day before the attack began. We ran 50 consecutive tests. In each test we returned the victim to its state prior to the attack, rebooted the victim, and then launched the attack. We managed to eclipse the victim in 49 of 50 tests. In the one test where our eclipse attack failed, our victim made an outgoing connection to a legitimate node.

Our next experiment tested the impact of network latency. Our victim was in Singapore. We used two attacker machines in New York (upload speeds ranging from 140 kbps - 1 Mbps) and $N = 1000$ attacker node IDs. Using the same methodology, we did 53 consecutive tests, and managed to eclipse the victim in 43 of 53 tests.

C. Countermeasure

This attack requires that all of client's connection slots can be occupied by unsolicited incoming connections. So, we have the following simple countermeasure:

Countermeasure 1. To stop this attack, Ethereum should enforce an upper limit on the number of incoming TCP connections, forcing clients to make a mix of incoming and outgoing TCP connections.

Status: Countermeasure 1 is live as of geth 1.8.0. There is now a configurable upper bound on incoming connections, which defaults to $\lfloor \frac{1}{3} \texttt{maxpeers} \rfloor = 8$.

IV. Eclipse by table poisoning

Ethereum nodes are still vulnerable to eclipse attacks even if Countermeasure 1 is adopted. The following eclipse attack succeeds with very high probability and can be launched using only two machines, each with a single IP address.

A. The attack.

Step 1: Craft attacker node IDs. This attack exploits the fact that the mapping from node IDs to buckets is public, so that a crafted attacker node ID will map to bucket $256 - r$ with probability 1, while an honest node ID is maps to bucket i with probability $\frac{1}{2^{r+1}}$ (see Section II-D). We therefore *craft* a set of attacker node IDs (*i.e.,* ECDSA keys) designed to fill all r last buckets in the victim's `table`. We use rejection sampling, as follows, to create $n \times 16$ node IDs: Let n be the number of buckets we want to fill in the victim's `table`. For $r = 0$, we randomly generate a node ID (ECDSA key), and keep it if it maps to bucket $256 - r$ (*i.e.,* if it is `logdist` r from the victim's node ID, see Section II-D). We continue until we find 16 nodes mapping to bucket $256 - r$. We then increment r and repeat until $r = n - 1$.

Notice that it gets harder to craft node IDs as n increases (*i.e.,* finding 16 nodes that map to bucket $256 - r$ takes expected time $16 \cdot 2^{r+1}$). Fortunately for the attacker, even a victim that is not under attack and has been online for days, is likely to have a `table` that contains only a few hundred nodes (Section II-D)). Thus, our attacker only needs to craft a few hundred node IDs. In our experiments we let $n = 17$ to craft a total of 272 node IDs. We can generate these 272 crafted node IDs in under 15 minutes on a laptop (MacBook Pro i5 at 2.9 GHz with 16 GB of RAM). See also Figure 3, a logarithmic plot of the time required to craft a single attacker node that maps to a specific bucket.

Step 2: Insert attacker nodes into `db`. Now we need to insert our crafted node IDs into the victim's `db`. We do this from a single attack machine. For each crafted attacker node, our attack machine send a *ping* to the victim. The victim will respond by bonding with attacker node. Recall (Section II-E) that bonding to a new node involves sending it a *ping*, waiting for its *pong*, and then inserting the node into the `db` and `table` (if there is space in the `table`). To ensure that the attacker nodes stored in the victim's `db` are not evicted because they became too stale, our attack machine *ping*s the victim at least once every 24 hours. We also ensure that our attacker machine responds to all of the victim's *ping* requests (with a *pong*) and *findnode* requests (with an empty *neighbor* message).

Step 3: Reboot and eclipse the victim. As in [23] and our attack from Section III, the final vulnerability we leverage is reboot. A client's `table` is empty upon reboot. We then aggressively ping the victim using our crafted attacker nodes IDs in order to fill the victim's `table` with attacker node IDs, causing the victim to makes all κ of its outgoing TCP connections to attacker nodes. The attacker also owns the remaining connections slots simply by making `maxpeers` $- \kappa$ incoming connections to the victim immediately upon reboot. To do this, we proceed as follows.

First, immediately upon reboot, our attacker machine *ping*s the victim for each of our crafted attacker node IDs. This causes the victim to *quickly* bond with our attacker node IDs, thus insert them into its `table`. Why does bonding occur quickly? This follows because, during bonding, the victim inserts the attacker node directly into its `table` without pinging it first. This happens because Step 2 of the attack ensured that (1) the attacker node ID is in the victim's `db`, and that the `db` records (2) zero failed responses to *findnode* requests, and (3) a *pong* response within the last 24 hours.

Next, we leverage the fact that an Ethereum client first starts up its UDP listener (and thus, it accepts incoming *ping*s that trigger bonding, as just described) and then later starts up its seeding process (Section II-F).[9] However, by the time the seeding process starts, its likely that the `table` already contains nodes (all of which belong to the attacker), and so the seeding process does nothing! Thus, none of the nodes stored in the `db` are moved in the `table`, and our attacker node IDs do *not* need to compete for space in the `table` with honest nodes IDs in stored in the `db`. The only competition is from honest nodes in the Ethereum network, that *ping* our victim and thus cause the victim to bond to them.

The victim is eclipsed because we use a second attacker machine to make least `maxpeers`$-\kappa$ incoming TCP connections to the victim immediately upon reboot.

B. Experimental results

We present the results of several experiments. In sum, using one attack machine in Boston and one attack machine in New York, we managed to eclipse a node located in New York, that had been online for 33 days, in 34/51 tests. Using the same two attack machines, we managed to eclipse a node located in Singapore, that had been online for 1 hour, in 44/50 tests.

33-day node experiment. We tested our attack on a victim node (controlled by us) located in New York. This node had been online on the Ethereum network for 33 days before we began our attack on August 16, 2017. Our victim node was instrumented (to record stats about the `db` size, when TCP connections were added or dropped, *etc.*) and modified to send out no node information in its neighbor network (to avoid polluting the production Ethereum network with our attacker node IDs), but was otherwise unchanged from the regular geth v1.6.6 client. This node had the default value `maxpeers` $= 25$.

[9]For a node running on a DigitalOcean node with 2GB RAM, 2 vCPU, this gap in time is about 1 second.

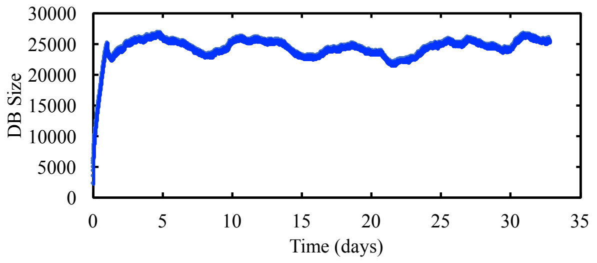

Fig. 4. `db` size across the 33-day lifetime (ending August 16, 2017) of the victim node in New York.

Figure 4 shows the size of the victim's `db` before we began our attack. The `db` stores 25,580 nodes, *i.e.,* almost all of the nodes in the Ethereum network. (According to ethernodes.org, on August 16, 2017, the Ethereum network contained a total of about 24K nodes.)

We start the attack (see Section IV-A) with Step 1 and Step 2, after which we repeatedly reboot the node and attack it with Step 3.[10]We used two attack machines. One attack machine, in Boston, had a 22 mbps upload speed, and was charged with pinging our victims using the 272 attacker node IDs crafted in Step 1 of the attack. The other attack machine, in New York, had a 42 Mbps upload speed, and was charged with making incoming connections to our victim (for each reboot that runs Step 3 of the attack) using 1000 different attacker IDs.

We eclipsed the node (*i.e.,* owned all `maxpeers` $=$ 25 of its TCP connections) in 34 of 51 reboots (66%). The victim did not add *any* seed nodes to its `table` (because we prevented seeding from executing because we inserted attacker node into the `table` before the victim started up its seeding process) in 19 of 51 reboots (37%); 18 of these (94%) resulted in a successful eclipse attack. Even when our victim did manage to seed its table,[11]we still manage to own all of its outgoing connections almost all of the time, *i.e.,* in 49 of 51 tests (96%). In other words, in 88% of our tests, we do indeed manage to achieve our goal of owning the victim's table, and thus also owning *all* of its outgoing connection.

Why, then, do we sometimes fail to eclipse the victim? This mostly occurs because honest nodes in the network know about our victim node, and thus make incoming connections to it (*i.e.,* in 15 of 49 reboots) . The reason

[10] Due to time/money constraints, we only had a few nodes that we kept online for many days before launching our attacks, and so we had to perform multiple reboots on the same node. This slightly skews our results, because while a node is rebooted it might fail to respond to *ping* and *findnode* requests, causing it to become less attractive in the eyes of other nodes. During first reboot, the victim is more attractive to other nodes in the network than during the last reboot.

[11]Even if the attacker does not manage to prevent seeding from executing, there is still a high probability that the eclipse attack will succeed. This is because our $16n$ attacker node IDs need only compete for space in the `table` with 30 seed nodes + 6 bootstrap nodes that are added to the `table` by the seeding process.

for this is as follows. Recall that there is no upper limit on the number of incoming connections that can be made to a node; this is what gives rise to our connection-monopolization eclipse attack of Section III. Meanwhile, it can take time for a node to establish its maximum of κ outgoing connections to nodes from its `table`, which is what are trying own with this attack. In our experiments, we want to distinguish between this "owning the `table`" eclipse attack and the connection-monopolization eclipse attack of Section III. As such, our attacker does not aggressively fill all $\texttt{maxpeers} = 25$ of the victim's connection slots with incoming connections (because this would just be the connection-monopolization attack), and instead leaves some slots open to allow the victim to make outgoing connections to attacker nodes. Sometimes, while we are waiting for outgoing connections to be made, an honest node will sneak in with an incoming connection, thus causing our eclipse attack to fail. Ironically, if Ethereum put a hard limit on the number of incoming connections (Countermeasure 1 that stops the connection-monopolization attack), we could design our attacker to fill in exactly *all* of the incoming connection slots, stopping honest nodes from sneaking in their incoming connections, and ensuring that our eclipse attack would succeed.

Young node experiment. With this experiment, we test whether high network latency harms the success rates of our attacks. We tested our attack on a victim node (controlled by us) located in Signapore, that had been online for 1 hour with a `db` size of 7000 nodes. This victim was instrumented and configured in the usual way with $\texttt{maxpeers} = 25$. As before, we used one attack machine in Boston (with a 500kb-1mb/sec upload speed) and one attack machine in New York (with a 200kb-1mbps upload speed). As before, we ran Step 1 and Step 2 of the attack, and then repeatedly rebooted the victim and ran Step 3 each time. We managed to eclipse the node (own all 25 of its TCP connections) in 44/50 reboots (88%). There was only 1 reboot for which we failed to eclipse the victim because it made an *outgoing* connection to an honest node. As before, the remainder of our failed eclipse attempts (5 of 50) resulted from honest nodes sneaking in incoming connections (while we are waiting for the victim to make outgoing connections to attacker nodes).

C. Scaling the attack

The attack we described is tailor made for a particular victim. Specifically, for each victim, Step 1 of the attack creates an attacker table that contains $16n$ node IDs that are crafted to fill the particular victim's last n buckets. This step takes some time (minutes), making it difficult to scale this attack to run against multiple victims in the Ethereum network.[12] To scale the attack to many victims, we could instead use a lookup table, which contains crafted node IDs that can be used to launch the attack of Section IV-A on *any* victim in the Ethereum network.

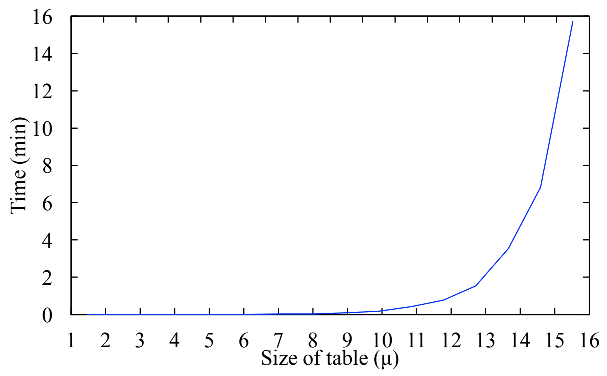

Fig. 5. Time (min) it takes to create a lookup table with size μ. Computed on a MacBook Pro with specs: Intel Core i5: 2.9 GHz; 16 GB RAM

The lookup table. The lookup table consists of 2^{μ} buckets, each of which can store 16 nodes IDs. To populate the lookup table, randomly choose a node ID (*i.e.,* by generating a random ECDSA key pair), and record the node ID in the bucket that corresponds to the μ most significant bits of the SHA3 hash of this node ID. Stops choosing random node IDs when each of the 2^{μ} buckets is full.

To fill a lookup table with 2^{μ} buckets, the expected number of random ECDSA keys that must be chosen by the attacker is[13]

$$2^{\mu+4} \sum_{i=1}^{2^{\mu+4}} \frac{1}{i} \approx ((\mu+4)\ln 2 + 1) \cdot 2^{\mu+4}$$

For hex-encoded ECDSA private keys, each of which is 64 bytes, the size of the lookup table is $2^{\mu} \cdot 16 \cdot 64 = 2^{(\mu+10)}$ bytes. For $\mu = 16$, the storage requirement for the lookup table is about 67MB. Figure 5 shows the time required to create a lookup table with a size of μ.

We can use the lookup table to find 16 nodes that map to bucket $256 - r$ of any target victim's `table` as follows. Let t be the SHA3 hash of the target victim's node ID. Let $\bar{t}$ be an $r+1$-bit string whose r most significant bits are the r most most significant bits of t, and whose $r+1$st bit is the inverse of the $r+1$th bit of t. Return the 16 nodes stored at bucket $\bar{t}$.

[12]Indeed, the Ethereum developers are banking on this difficulty. The Ethereum Wiki [5] states that "Overwriting the entire peer table of a node requires significant computational effort even with relatively low k" (where $k = 16$ is the number of nodes that can fit in each bucket).

[13]This follows because we have a coupon collector problem with $16 \cdot 2^{\mu} = 2^{\mu+4}$ distinct coupons. See [34, page 32] for the expected number of coupons required.

V. Countermeasures

We now present several countermeasures.

A. Rethink Node IDs.

Our first set of countermeasures makes it more difficult for the attacker to craft attacker nodes, and run them all from a single machine with a single IP address.

Countermeasure 2. One-to-one mapping between IP and ECDSA key. We have exploited the fact that an unlimited number of node IDs can be created without expending significant resources, and then run a single machine with a single IP address.[14] To remedy this, we suggest that clients apply a strict one-to-one mapping between ECDSA public key and IP addresses. That is, each IP address should be associated with a unique ECDSA public key, and vice versa. This strict mapping should apply to nodes stored in the `db`, nodes stored in the `table`, nodes learned from *neighbor* messages.

One way to achieve this is to have a separate data structure that stores the one-to-one mapping from ECDSA key to IP addresses. To deal with situations where a node's IP address changes while its ECDSA key remains the same, the client should be able to update the IP address associated with a given ECDSA key in the `bijectionTable`. To deal with situations where a different ECDSA key is used with a previously-seen IP address, the client should also be able to update the ECDSA address associated with a given IP address in the `bijectionTable`.

Whenever a node is added/updated in the `table` or `db`, the client should first update the `bijectionTable`. If that node's IP or ECDSA key is already present in the `bijectionTable`, then instead of creating a new entry in the `table` or `db`, the existing `table` or `db` entry corresponding to that IP or ECDSA key should be updated.

To prevent attackers from gaming the update mechanisms, the client should only perform an update once the IP address have "proved" that the IP address holds the corresponding secret ECDSA key. To do this, the client could send a *ping* request and wait for *pong* response, which should be digitally-signed by the corresponding ECDSA key.

Status: Countermeasure 2 is not implemented as described above in geth v1.8.0, which still supports running multiple nodes on single IP. Instead, geth v1.8.0 limits the number of nodes in the same /24 subnet to be two per bucket and 10 for the whole `table`.

Countermeasure 3. Non-public mapping from node ID to bucket. A second key property exploited in Section IV is that the mapping of node IDs to buckets in the `table` is public (see Section II-D). This property was inherited from the Kademlia protocol. In Kademlia, the "distance" between node a and node b is reflected in mapping of node a to a bucket in node b's `table`; iterative lookups of content become possible because all nodes in the network use the same distance metric, and progressively get "closer" to the content they look up (Section II-A). Ethereum, however, has little use for iterative lookups (Sections II-A,II-G), so there little use for a universal "distance" between nodes.

Thus, instead of computing the `logdist` between the candidate's node ID n and the client's node ID (per Section II-D), the mapping could be

`logdist` $(s_1, \text{SHA3}(n, s_2))$

where s_1 and s_2 are long-lived 256-bit secrets, randomly-chosen when the node boots up for the first time (as is done in Bitcoin [23]). Because the adversary does not know the local secrets, the adversary cannot predict what bucket its crafted node ID will land in. Also, "salting" the SHA3 hash prevents attacks that infer s_1 (*e.g.,* by crafting a node IDs n, attempting to insert them into the `table`, and inferring whether insertion was successful, in order to learn the first few bits of s_1). This countermeasure levels the playing field between attacker nodes ID and legitimate nodes ID, making them equally likely to map to a given bucket in the client's `table`.

Countermeasure 4. Make `lookup (self)` non-public. Along the same line as Countermeasure 3, we recommend that calls to `lookup (self)`, where `self` is the (publicly-known) SHA3 hash of the client's own node ID, be replaced with calls to `lookup` (s_3), where s_3 is a secret that is chosen uniformly at random each time `lookup (self)` is required.[15]

Status: Countermeasures 3 and 4 were *not* implemented in geth v1.8.0. Both Countermeasures 3 and 4 eliminate the public distance metric that Kademlia leverages in order to allow for logarithmic lookups of content. Currently, the Ethereum peer-to-peer network uses this feature only in the very rare case of resolving a node ID to its IP address (see Section II-G). Nevertheless, the Ethereum developers decided to preserve this feature so

[14]Note: We caution against a solution that makes the node ID a function of (IP address, ECDSA key) or (IP address, port, ECDSA key), since this still allows multiple nodes to have the same IP address.

[15]Interestingly, the use of `lookup (self)` upon reboot comes directly from the Kademlia protocol, where it enables a node to add itself into other node's `tables` when first coming online. Ethereum has no need for this, since a newly-rebooted node adds herself to other nodes' `table` using a completely different process. (Specifically, the newly-rebooted node sends an unsolicited *findnode* messages to a peer (as part of lookup) and then *bonds* with all the nodes learn from the *neighbor* response; bonding causes the newly-rebooted node to be added to the other node's `db` and `table` (Section II-E).

that it available for use in future versions of the Ethereum protocol.

Since geth v1.8.0 still allows the attacker to freely craft node IDs that land in specific buckets, the partial implementation of Countermeasure 2 in geth v1.8.0 means that for a given victim's `table`, there can be at most 10 attacker node IDs associated with each attacker IP address. While this improves on the situation prior to geth v1.8.0 (where we could eclipse our victim using just one or two IP addresses), it does not raise the bar for attackers quite as high as we had hoped.

B. Make seeding more aggressive

The following countermeasures are designed to prevent the attacker from disabling the seeding process, as in Section IV-A.

Countermeasure 5. Always run seeding. Run the seeding process (Section II-F) even if the `table` is not empty. This way, upon reboot, the victim will always insert node IDs from the `db` into its `table`, increasing the probability that the victim establishes outgoing connections with legitimate nodes IDs stored in its `db`.

Countermeasure 6. Eliminate the reboot exploitation window. Even with Countermeasure 5 in place, an attacker can still exploit that fact that when a client reboots, its UDP listener starts up several seconds before the seeding process concludes. This time window gives the attacker an opportunity to *ping* the victim from many attacker node IDs at high rate, causing the client's `table` to fill up with attacker node IDs, before the client has a chance to insert legitimate node IDs into the `table` by calling `lookup (self)`.

Thus, we recommend that Ethereum clients disable the bonding process to nodes learned via unsolicited *ping* messages, until after `lookup (self)` has finished adding nodes to the `lookup_buffer`. At this point, the client will have inserted into its `table` up to 35 nodes (from seeding) and as well as all the nodes learned from *neighbor* messages obtained during the `lookup (self)` process) for a total of L nodes.

Per equation (2), the expected number of nodes that will already be in the `table` is therefore $\sum_{i=1}^{256} \max(16, L\frac{1}{2^{i+1}})$. The attacker nodes ID can therefore only be inserted into the `table` only the after the nodes learned from the `db` and *neighbor* messages have already been inserted in the `table`.

Moreover, once the nodes learned from the `db` and *neighbor* messages have been inserted into the `table`, the task runner has already started making outgoing connections (see Section II-G). Thus, it is more likely that the victim will make at least one of its outgoing connection at least of one of these nodes. This makes eclipsing the victim, upon reboot, significantly more challenging. This follows because the attacker must own most of the victim's `db`, and control most of the nodes from which the victim solicits *neighbor* messages.

Status. Countermeasures 5 and 6 are live in geth v1.8.0. The v1.8.0 client waits until seeding is complete before bonding with an unsolicited ping. As an additional related countermeasure, the v1.8.0 client also runs `lookup` on three random targets during seeding in order to add more legitimate nodes from the `db` to the `table`. This prevents the attacker from inserting its own attacker nodes ID into a nearly-empty `table` during seeding (Section IV).

VI. Attack by manipulating time

We now show how manipulating the local clock at a victim node can turns a 'established' node with knowledge of the network (and is known to other geth nodes in the network) into one that knows nothing about the network (and is unknown to other geth nodes). Worse yet, the victim will refuse to accept network information learned from most honest nodes, while happily accepting information from the attacker.

A. 'Erase' the victim from the network

Recall from the Section II-C that Ethereum limits replay attacks by time-stamping UDP messages; a node will reject a UDP message if its timestamp is more than 20 seconds old. (Note, however, that this does not prevents replays sent within a 20 second time window.)

We exploit this timestamping as follows. We maliciously change a victim's clock so that it is more than 20 seconds in the future, by *e.g.,* attacking NTP [31], [32]. This means that the victim will reject any honest UDP message as expired (because, from the victim's point of view, the request is more 20 seconds old). This results in two things.

(1) The victim will forget about all other nodes. This follows because the node stop accepting *pong* and *neighbor* responses from honest nodes. After a few days of this, the victim's `db` will evict all honest nodes (because it thinks they have failed to respond with a valid *pong* within the last 24 hours, Section II-D). Moreover, the victim's `table` and `db` will start evicting honest nodes. This follows because each time the victim calls `lookup` and thus sends *findnode* requests to 16 nodes in its `table`, the queried nodes will respond with a *neighbor*'s request that the victim considers to be expired. When this happens five times in a row for

a given honest node, the victim evicts that honest node from it's `table` (Section II-E).[16]

(2) Other geth nodes will forget about the victim. This follows because the victim stops responding to *ping* and *findnode* requests from other honest nodes (because it thinks they have expired). Because the victim fails to respond to *ping* requests (with a *pong* response), after 24 hours an honest geth node `db` will evict the victim (if the honest node is running the peer-to-peer protocol as described in Section II-D). Also, once the victim fails to respond to the fifth *findnode* request from an honest node, the honest node's `table` will also evict the victim (if the honest node is running the protocol as described in Section II-E).

Note, however, that the above items (1) and (2) do *not* result in an eclipse of the victim, because they have no impact on any established (incoming or outgoing) TCP connections made by the victim. That said, the victim is now highly vulnerable to eclipse attacks.

B. Experiments

We began a timing attack on a geth node that had been running in a datacenter in New York since July 14, 2017. We began the attack 34 days later, on August 17, 2017, and stopped the attack on September 4, 2017. During our attack, we emulated an NTP attack [31], [32], [39] by maliciously changing the victim's local clock. We moved our victim forward into the future, first by 25 seconds, then 70 seconds, then 5 minutes, 7 minutes, 9 minutes and finally 13 minutes, as shown in Figure 6.[17]

Forgetting. From Figure 6, we can see that after 3 days the number of nodes in the victim's `db` drops dramatically, from several hundred thousand down to about a dozen. The number of nodes in the `table` follows the same trend. Thus, our victim has forgotten about all but a dozen nodes in the network, making her more vulnerable to eclipse attacks.

Being forgotten. Figure 6 shows the number of peers our victim had over the lifetime of our attacks. Interestingly, our victim still has a number of peers even after the timing attacks occurs. A deeper look at the data shows that the vast majority of those peers are not running the geth client; instead, they are other Ethereum node implementation, specifically parity and

[16]Even during the attack, the victim still insert the six bootstrap nodes into its `table` as part of the seeding process (Section II-F). However, any UDP messages sent by the bootstrap nodes will be rejected as expired, causing the victim to reject all information about other nodes in the network.

[17]Note that NTP has a "panic threshold" of about 16 minutes; if a host's local time quickly change by more than 16 mins, the NTP daemon will panic and restart [31]. For this reason we limited the time shifts used in our attacks to be less than 16 minutes.

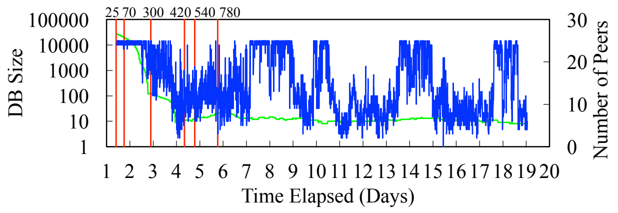

Fig. 6. DB size (green line) and number of peer connections (blue line) for a victim whose time has been pushed in the future. Red vertical lines represent when victim's time was pushed further into the future. Number on top of the line represents the total seconds pushed into the future. The victim came online on July 14, 2017, and the timing attack began on August 17 and ended on September 4 2017.

Ethereum(J). Specifically, for the last 11 days of the attack, a total of only 130 geth clients from unique (IP, port number) connected to our victim, as compared to 64374 non-geth clients from unique (IP, port numbers). Beyond this, Figure 6 there are several points in time when victim is highly vulnerable, because she only has one or two connected peers. Thus, our experiments indicate that the risk of being forgotten (due to a timing attack) is mitigated by the presence of Ethereum implementations other than geth (*i.e.,* (parity, Ethereum(J)).

C. Low-resource eclipse attack.

A victim of a timing attack that has (a) forgotten about most other nodes, and (b) been forgotten by most other nodes, is highly vulnerable to a low-cost eclipse attack that requires very few attacker node IDs. One could launch a version of the attack in Section IV, except that our attacker node IDs will have their clocks set to the same time as the victim, so that the victim accepts their UDP messages (while continuing to reject UDP messages from all honest nodes).

This sort of attack can circumvent most of the protections provided by our countermeasures. (1) If Countermeasure 2 was put in place, so that it became expensive for the attacker to create many unique node IDs, then this approach reduces the number of node IDs that the attacker has to create because the `table` is mostly empty. (2) Our experiments (Section IV-B) indicate that our eclipse attacks sometimes fail because honest nodes are making incoming connections to our victim node. However, if all honest geth nodes forget about our victim, and thus stop making connections to him, then the probability of a successful eclipse attack improves significantly. (3) The timing attack causes the victim to reject all *neighbor* messages from honest nodes, and to empty out its `db`. This essentially eliminates the protective value of Countermeasures 3, 4,5,6, whose role is to fill the `table` with nodes from the `db` and *neighbor* messages before the attacker has a chance to insert its own node IDs into the `table` upon reboot.

D. Countermeasures

This is a simple way to address this attack.

Countermeasure 7. Use nonces. Timestamps should be eliminated from UDP packets. Instead, each UDP query packet (*ping* and *findnode*) should include a fresh random nonce and the returned UDP response (*pong* and *neighbor*) should include this nonce. (This is a standard technique used in DNS [24], TCP [28] *etc.*) This would eliminate replays of old UDP responses and is not susceptible to time manipulation attacks, improving on the status quo which is vulnerable to (1) replays within the 20 second time window and (2) our time manipulation attack. We note that nonces can only prevent the replay of *response* packets (*pong* and *neighbor*), they do not prevent replays of *query* packets (*ping* and *findnode*). (This is because the nonce is response provides protection because it is matched against the query!)

Status. Countermeasure 7 has not been implemented in geth v1.8.0, because it is not backwards compatible with existing RLPx v4 packet formats.

VII. Related work

Peer-to-peer networks. There is a long line of work on eclipse attacks, starting with the work of [40], [19], [41]; see [44] for a survey.

Our work is most related to the prior eclipse attacks on the Kademlia protocol [43], [29], [27]. While our eclipse attackers have the goal of isolating a victim node from the rest of its peers in the network, [43], [29], [27]'s eclipse attackers wish isolate a target item of content. This changes the structure of our attacks: our victims are eclipsed when they reboot, while attackers in [43], [29], [27] poison the `table` with attacker node IDs, in hopes that the victim will contact the attacker when it looks up the target item of content. That said, both our attack of Section IV and [43], [29], [27] achieve this by crafting a set of node IDs and using them to launch attacks. (Interestingly, the idea of using crafted node IDs goes back to the very first eclipse attack paper [19].)

To defend against [43]'s attacks, implementors (eMule 0.49a) adopted a three countermeasures similar to our Countermeasure 2: (1) One IP address must not have more than one node ID, (2) A bucket must not contain more than two nodes from the same /24 IP address block, (3) the whole `table` must not contain more than ten nodes from the same /24 IP address block. (Countermeasures (2) and (3) but not (1) are now live in geth v1.8.0.) It is important to note that [27] circumvents these countermeasures by exploiting that Kademlia's content discovery (lookup) process. Because this process iteratively queries no more than a logarithmic number of node IDs, [27]'s attack only requires a logarithmic number of IP addresses (practically, about 10 node IDs). Our setting is different, for two reasons. First, Ethereum does not use iterative lookups to discover content; iterative lookups are used in the rare event that a node ID must be resolved to its IP address (Section II-G). Thus, [27]'s attack is less relevant. Second, while Kademlia must have a public mapping from node IDs to buckets (in order to enable iterative lookups of content), Ethereum only rarely makes use of this feature. As such, our Countermeasures 3,4 limits this vulnerability by making the mapping from node IDs to buckets non-public; unfortunately, however, these Countermeasures 3,4 have not been adopted as of geth v1.8.0.

Blockchain eclipse attacks. The first eclipse attack on a blockchain's peer-to-peer protocol (Bitcoin) was presented in [23], and its implications on consensus and double-spending have been explored in *e.g.,* [37], [21]. Meanwhile, [17] shows how Internet routing with BGP can be used to partition Bitcoin's peer-to-peer network; these attacks leverage the fact that Bitcoin peer-to-peer messages are not authenticated. Fortunately, Ethereum does not suffer from these attacks since all peer-to-peer messages as digitally signed. In a complementary work, [45] consider exploiting Ethereum's block propagation algorithm for eclipse attacks; their attacker sends the victim an bogus (forged) blockchain and then exploits a flaw in the block propagation algorithm to prevent the victim from connecting to any other peer. Our attacks are agnostic to its block propagation algorithm, and instead exploit Ethereum's peer-to-peer network.

VIII. Conclusion

Ethereum inherits most of the complicated artifacts of the Kademlia protocol, even though it has little use for the key property for which Kademlia was designed (*i.e.,* logarithmic content discovery). We have demonstrated that this creates serious vulnerabilities. Specifically, we presented eclipse attacks that can be launched by an attacker that controls only a two machines, each with a single IP address. To remedy this, we have suggested a set of countermeasures that eliminates some artifacts of the Kademlia protocol. Our countermeasures raise the bar for eclipse attackers, by forcing them to control thousands of IP addresses (rather than just two) in order to successfully launch attacks. Many of our countermeasures have been adopted in geth v1.8.0, hardening Ethereum against the eclipse attacks presented in this paper.

Acknowledgements

We thank Felix Lange and Martin Holst Swende from the Ethereum Foundation for their professional response to our bug-bounty disclosure and for productive collaboration on the development of patches.

This material is based upon work supported by the US National Science Foundation under Grant 1350733. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] Intel ethernet controller vulnerable to packet of death. http://www.zdnet.com/article/intel-ethernet-controller-vulnerable-to-packet-of-death/, February 7 2013.

[2] Microsoft issues fix for resurrected ping of death. https://gcn.com/Blogs/CyberEye/2013/08/Microsoft-patch-ping-of-death-IPv6.aspx, August 2013.

[3] libp2p whitepaper. https://github.com/ethereum/wiki/wiki/libp2p-Whitepaper, November 16 2014.

[4] ethereum: wiki: Devp2p wire protocol. https://github.com/ethereum/wiki/wiki/%C3%90%CE%9EVp2p-Wire-Protocol, October 5 2015.

[5] Ethereum: Wiki: Kademlia peer selection. https://github.com/ethereum/wiki/wiki/Kademlia-Peer-Selection, Oct 5, 2015.

[6] Rlpx: Cryptographic network & transport protocol. https://github.com/ethereum/devp2p/blob/master/rlpx.md, October 5 2015.

[7] Bitcoin wiki: Script. https://en.bitcoin.it/wiki/Script, December 6, 2017.

[8] The ethereum radian network. https://raiden.network/, 2017.

[9] The ethereum wiki: Erc20 token standard. https://theethereum.wiki/w/index.php/ERC20_Token_Standard, October 17, 2017.

[10] Ethereum wire protocol. https://github.com/ethereum/wiki/wiki/Ethereum-Wire-Protocol, June 23, 2017.

[11] Etherscan: Token tracker. https://etherscan.io/tokens, December 20 2017.

[12] ethersphere: swarm. https://github.com/ethersphere/swarm-guide/blob/master/contents/introduction.rst, December 7 2017.

[13] Sharding faq. https://github.com/ethereum/wiki/wiki/Sharding-FAQ, December 27 2017.

[14] Solidity. http://solidity.readthedocs.io/en/latest/, December 20, 2017.

[15] Talos-2017-0471: Multiple vulnerabilities in the cpp and parity ethereum client. https://www.talosintelligence.com/reports/TALOS-2017-0471, January 9 2018.

[16] Luke Anderson, Ralph Holz, Alexander Ponomarev, Paul Rimba, and Ingo Weber. New kids on the block: an analysis of modern blockchains. *arXiv preprint arXiv:1606.06530*, 2016.

[17] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking bitcoin: Routing attacks on cryptocurrencies. *arXiv preprint arXiv:1605.07524*, 2016.

[18] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, pages 164–186. Springer, 2017.

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

[20] Nikhilesh De. New code released for vlad zamfir's ethereum 'casper' upgrade. *CoinDesk*, November 21 2017.

[21] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 3–16. ACM, 2016.

[22] Arthur Gervais, Hubert Ritzdorf, Ghassan O Karame, and Srdjan Capkun. Tampering with the delivery of blocks and transactions in bitcoin. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 692–705. ACM, 2015.

[23] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *USENIX Security Symposium*, pages 129–144, 2015.

[24] A. Hubert and R. van Mook. *RFC5452: Measures for Making DNS More Resilient against Forged Answers*. https://tools.ietf.org/html/rfc5452, January 2009.

[25] Aggelos Kiayias and Giorgos Panagiotakos. Speed-security tradeoffs in blockchain protocols. *IACR Cryptology ePrint Archive*, 2015:1019, 2015.

[26] Aggelos Kiayias and Giorgos Panagiotakos. On trees, chains and fast transactions in the blockchain. *IACR Cryptology ePrint Archive*, 2016:545, 2016.

[27] Michael Kohnen, Mike Leske, and Erwin P Rathgeb. Conducting and optimizing eclipse attacks in the kad peer-to-peer network. In *International Conference on Research in Networking*, pages 104–116. Springer, 2009.

[28] M. Larsen and F. Gont. *RFC6056: Recommendations for Transport-Protocol Port Randomization*. https://tools.ietf.org/html/rfc6056, January 2011.

[29] Thomas Locher, David Mysicka, Stefan Schmid, and Roger Wattenhofer. Poisoning the kad network.

[30] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[31] Aanchal Malhotra, Isaac E Cohen, Erik Brakke, and Sharon Goldberg. Attacking the network time protocol. In *NDSS*, 2016.

[32] Aanchal Malhotra, Matthew Van Gundy, Mayank Varia, Haydn Kennedy, Jonathan Gardner, and Sharon Goldberg. The security of ntp's datagram protocol. *IACR Cryptology ePrint Archive*, 2016:1006, 2016.

[33] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[34] Michael Mitzenmacher and Eli Upfal. Probability and computing: Randomized algorithms and probabilistic analysis. 2005.

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

[36] Christopher Natoli and Vincent Gramoli. The balance attack against proof-of-work blockchains: The r3 testbed as an example. *arXiv preprint arXiv:1612.09426*, 2016.

[37] Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 305–320. IEEE, 2016.

[38] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network, 2015.

[39] Jose Selvi. Bypassing http strict transport security. *Black Hat Europe*, 2014.

[40] A. Singh, T.-W. Ngan, P. Druschel, and D.S. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM*. Citeseer, 2006.

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

[42] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[43] Moritz Steiner, Taoufik En-Najjary, and Ernst W Biersack. Exploiting kad: possible uses and misuses. *ACM SIGCOMM Computer Communication Review*, 37(5):65–70, 2007.

[44] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. A survey of dht security techniques. *ACM Computing Surveys (CSUR)*, 43(2):8, 2011.

[45] Karl Wüst and Arthur Gervais. Ethereum eclipse attacks. Technical report, ETH Zurich, 2016.