Attacking the Kad Network

Peng Wang, James Tyra, Eric Chan-Tin, Tyson Malchow, Denis Foo Kune,
Nicholas Hopper, Yongdae Kim
University of Minnesota - Twin Cities
200 Union Street SE
Minneapolis, MN 55455
{pwang,tyra,dchantin,malchow,foo,hopper,kyd}@cs.umn.edu

ABSTRACT

The Kad network, an implementation of the Kademlia DHT protocol, supports the popular eDonkey peer-to-peer file sharing network and has over 1 million concurrent nodes. We describe several attacks that exploit critical design weaknesses in Kad to allow an attacker with modest resources to cause a significant fraction of all searches to fail. We measure the cost and effectiveness of these attacks against a set of 16,000 nodes connected to the operational Kad network. We also measure the cost of previously proposed, generic DHT attacks against the Kad network and find that our attacks are much more cost effective. Finally, we introduce and evaluate simple mechanisms to significantly increase the cost of these attacks.

Categories and Subject Descriptors

C.2.0 [**Computer Networks**]: General—*Security and protection*

General Terms

Security

Keywords

P2P, Security, Attack, Kad

1. INTRODUCTION

The Kad network is a peer-to-peer distributed hash table (DHT) based on Kademlia [20]. It supports the growing user population of the eDonkey [10] [1] file sharing network by providing efficient distributed keyword indexing. The Kad DHT [2] is very popular, supporting several million concurrent users [31, 27], and as the largest deployed DHT, its dynamics has been the subject of several recent studies [32, 30, 29, 28].

DHT Security in general – the problem of ensuring efficient and correct peer discovery despite adversarial interference – is an important problem which has been addressed in a number of works [9, 26, 4, 17, 12, 23, 15, 24, 13]. However, the majority of these works assume a DHT with ring topology and recursive routing; Kademlia uses a fundamentally different, "multi-path" iterative routing algorithm as well as a different topology. To our knowledge, no specific, applicable analysis of the security properties of the Kademlia DHT or the deployed Kad network has appeared in the literature, despite the potential impact of an attack on this network.

In this paper, we describe an attack on the Kad network that would allow a few malicious nodes with only modest bandwidth to effectively deny service to nearly all of the Kad network. Our attack has two phases – the first phase is to "collect routing table entries", which we call *the preparation phase*, and the second phase is to attack queries on the Kad network, which we call *the execution phase*. Having collected routing table entries [3], it is not obvious how to use them to halt Kad lookups: since Kademlia is specifically designed to tolerate faulty routing-table entries by employing parallel lookup, the simple attacks discussed in the literature (such as dropping or misrouting queries [26, 4]) will not impede the majority of lookups: an attacker who owns 50% of all routing table entries would halt at most 34% of all Kad queries using these techniques.

We describe a new attack on the general Kademlia search algorithm that successfully prevents an intercepted query from completing, and show how to exploit design weaknesses in Kad to further reduce the cost of the attack. We experimentally evaluate the two phases of our attack by connecting roughly 16,000 victim nodes to the live Kad network and attacking them directly. Extrapolating from these results, we estimate that an attacker using a single workstation with a 100 Mbps link can collect 40% of the routing table entries in the Kad network in less than one hour, and prevent 75% of all Kad keyword lookups.

A secondary contribution of this paper is an experimental measurement of the cost of two generic DHT attacks against the Kad network. We find that the Sybil attack [9], which works by creating enough long-lived identities that the attacker owns a significant fraction of routing table entries, is significantly more expensive than our hijacking attack, both in terms of bandwidth and in terms of wall-clock time. We also evaluate the cost of index poisoning [16] against Kad to ensure that 75% of all search results are incorrect (notice that this is a weaker goal than ensuring that 75% of lookups fail). We find that the bandwidth cost of this attack is higher than the cost of our attack on Kademlia lookups.

Our attack is different from the Sybil attack because we do not introduce any new identities in the DHT. It is also different from the Eclipse attack [25] because we actively acquire entries rather than passively promoting compromised nodes.

[1] eDonkey is a server-based network where clients perform file searches. Kad is a decentralized P2P network. aMule/eMule are the two most popular clients which can connect to both the eDonkey and the Kad network.

[2] There are several Kademlia-based networks such as the Azureus BitTorrent DHT, but we will refer to the aMule/eMule DHT as Kad.

[3] obtaining the routing table of other nodes in the network

SecureComm 2008, September 22 – 25, 2008, Istanbul, Turkey

Finally, we present several potential mitigation mechanisms for increasing the cost of our attack on Kad lookup while keeping the design choices made by the designers of the Kad protocol. We evaluate these mechanisms in terms of their effectiveness and incremental deployability. We find that a very lightweight solution can effectively eliminate hijacking and greatly increase the cost of lookup attacks, while having minimal impact on the current users of Kad.

New versions of the two most popular Kad clients have recently been released – aMule 2.2.1 on June 11, 2008 and eMule 0.49a on May 11, 2008. We show that although they have new features intended to improve security, our attacks still work with the same resource requirements.

Contributions: The contributions of this paper are as follows:

- We are the first to show that a large-scale attack on a widely-deployed P2P network can easily be performed, with experimental measurements.
- Our attack is much more efficient and effective than previously known attacks – the costs to perform our preparation phase is less than the costs of launching a Sybil attack and the costs for our execution phase is less than the costs of launching an index poisoning attack – comparison is made in Section 6. Moreover, our execution phase can disrupt control plane operation (building and maintaining routing tables) instead of just attacking the data plane and thus, is stronger than an index poisoning attack.
- An attacker with moderately low resources can easily cripple the Kad network and we hope that this paper will help developers and users to fix the vulnerabilities in the eMule/aMule Kad.

The remainder of this paper is organized as follows. Section 2 gives an overview of the design and vulnerabilities of Kad. Section 3 gives further details of our primary attack on Kad, and Section 4 gives analytical and experimental results on the cost-effectiveness of this attack. Section 5 reports on a related attack with lower bandwidth costs in the second phase. Section 6 compares our attack to general DHT attacks, while Section 7 discusses mitigation strategies for Kad. Section 8 outlines the recent changes in the Kad clients and how they affect our attacks. Finally, Section 9 discusses related work on Kad and DHT security, and Section 10 presents our conclusions and directions for future work.

2. BACKGROUND

In this section, we first present some background on the Kademlia algorithm and Kad's design. We then highlight the primary design flaw in Kad that enables our attack. We finally discuss our attack model.

2.1 Overview of Kademlia and Kad

Kademlia. In Kademlia, every node has a unique ID uniformly distributed in the ID space. The distance between two nodes is the bitwise XOR of the two node IDs, the "XOR metric". Every data item (i.e., a [key, value] binding) stored by the Kademlia network has a key. Keys are also uniformly distributed in the same ID space as node IDs. Each data item is stored by several *replica roots* — nodes with IDs close to the key according to the XOR metric.

To route query messages, every node maintains a routing table with $O(\log(N))$ entries, called k-*buckets*, where N is the size of the network. Figure 1 (a) shows a Kademlia routing table. A k-bucket on level i contains the contact information of up to k nodes that share at least an i-bit prefix with the node ID of the owner. Kademlia biases routing tables toward long-lived contacts by placing a node in a k-bucket only if the bucket is not full or an existing contact is offline.

Kademlia nodes use these routing tables to route query messages in $O(\log(N))$ steps. When node Q queries key x, it consults its routing table and finds α contacts from the bucket closest to x. Q consults these contacts in parallel, which each return k of their contacts. Next, Q picks the α closest contacts from this set, repeating this procedure until it cannot find nodes closer to x than its k closest contacts, which become the replica roots.

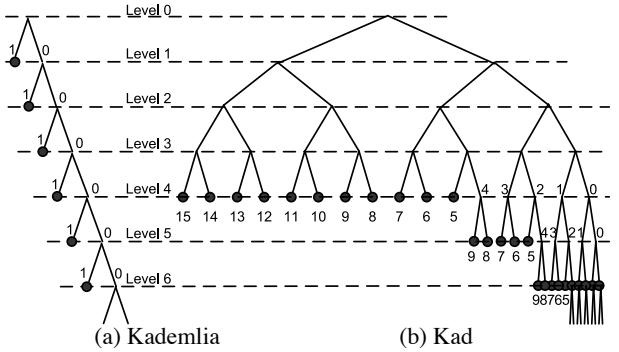


Figure 1: Routing Table Structures of Kademlia and Kad. Leaves depict k-buckets

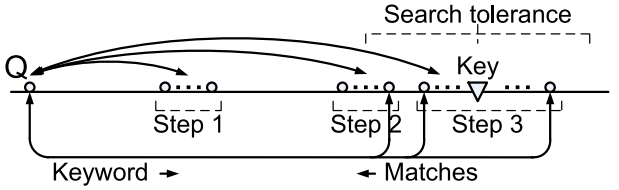


Figure 2: Kad keyword search

Kad. Kad uses random 128-bit IDs. Unlike some other DHT networks, in which nodes must generate their IDs by applying a cryptographic hash function to their IP and/or public key, Kad does not have any restriction on nodes' IDs. Unlike Kademlia, the Kad replica roots of a data item $\langle x, v\rangle$ are nodes with an ID r such that $r \oplus x < \delta$ where δ is a *search tolerance* hard-coded in the software; so different data items may have different numbers of replica roots.

The routing table structure of Kad, shown in Figure 1(b) is slightly different from Kademlia. Starting from level 4, k-buckets with an index $\in$ **[0, 4]** can be split if a new contact is inserted in a full k-bucket, whereas in Kademlia, only the k-buckets with index **0** can be split. Kad implementations use k-buckets of size k = **10**. The wide routing tables of Kad result in short routing paths. Stutzbach and Rejaie [31] show that the average routing path length is **2.7** assuming perfect routing tables, given the size of the current Kad network.

Suppose A and B are Kad nodes, where B is in a k-bucket at level i of A's routing table. Then we say that B is an *ith level contact of* A, and that A has an *ith level back-pointer to* B. In Kad, any node can be a contact of another node. Due to the symmetry of the XOR metric, if both A and B are in the other's routing table then they are most likely at the same level. Also, from the routing table owner's point of view, a k-bucket on the ith level covers a $\frac{1}{2^i}$ fraction of the ID space. For example, the 11 k-buckets on the **4th** level cover $\frac{11}{16}$ of the ID space. Hence, on average, $\frac{11}{16}$ of the owner's queries will use contacts in these k-buckets as the first hop.

A Kad node learns about new nodes either by asking nodes it already knows while searching, or by receiving messages from nodes. New nodes are inserted into its routing table if the corresponding k-bucket is not full or can be split. A node tests the liveness of its contacts opportunistically while searching, or (if necessary) periodically with HELLO_REQ messages to check if they are still alive. The testing period for a contact is typically 2 hours.

A Kad node Q looking for a particular keyword first computes the MD4 hash of that keyword as the key and starts a keyword search following steps shown in Figure 2. Starting from its rout-

ing table, at each step Q picks its three contacts closest to the key and sends them a KADEMLIA_REQ message; these contacts send KADEMLIA_RES messages with additional contacts, and the process repeats until a replica root is located. While this query procedure is similar to that of Kademlia, the major difference is the termination condition. After finding a live replica root, Q sends a SEARCH_REQ message including the keyword to the replica root, which returns many "matches" to the keyword. Q stops sending both KADEMLIA_REQ (for finding more replica roots) and SEARCH_REQ (for finding more matches) messages when it receives more than 300 matches, even if all of the matches are returned by a single replica root.

If all three nodes that Q contacts in a given step are offline or simply slow, Q attempts to recover the search as follows. For each keyword query, Q maintains a long list of backup contacts, consisting of 50 contacts from Q's routing table plus unused contacts returned by intermediate hops. Until a query terminates, Q will wake up once every second and check whether the query has received any new replies in the last three seconds; if not, it picks the closest backup node, removes it from the list, and sends it a KADEMLIA_REQ message. After 25 seconds, Q prepares to stop and will not send more requests to intermediate hops. For example, if all nodes in the list are offline, then Q sends 22 ($\mathbf{25 - 3 = 22}$) messages to backup contacts, before it eventually times out.

2.2 Design Vulnerabilities in Kad

Our attacks are all primarily enabled by Kad's weak notion of node identity and authentication. Since, as in most file sharing networks, there is no admission control, nor any cost of creating an identity, the Sybil attack is straightforward to implement, although we will show that by itself this is a somewhat ineffective attack. Of more concern is that, while IDs are persistent, there is no verifiable binding between a host and its ID. The design decision to support persistent IDs allows a user to significantly reduce her startup time – recall that a node's routing table depends on its ID. The wall-clock time to construct a reasonably complete routing table is well above the median Kad session time of 7 minutes reported in [32], and keeping a persistent ID and routing table for each node makes it possible to avoid this penalty. This design also avoids complication from NAT traversal. Furthermore, it seems that the designers chose to avoid tying a node's ID to its IP address to support node mobility, e.g. users who move from wired to wireless connections or connect via a modem pool with (consequently) varying IP addresses. A further optimization with this approach is that a node that goes offline at one (IP, port) location and comes online at another can essentially "repair" the routing table entries it affects by doing so. Unfortunately, the decision to create no verifiable binding between a node and its ID make it possible for anyone to exploit the "repair" operation and collect more routing table entries. In essence, the ID of a node serves as its authentication as well; since node IDs are public information, this predictably leads to several attacks.

2.3 Attack Model

Our attack is designed under the assumption that the attacker controls only end-systems and does not require corruption or misrouting of IP-layer packets between honest nodes. We describe our attack under the assumption that the attacker's goal is to degrade the service of the Kad network, by causing a significant fraction of all keyword searches to fail. However, the same techniques can be applied with little modification to cause failure of a significant fraction of searches either for a specific set of keywords or initiated by a specific set of nodes.

We also assume an attacker's primary cost is in bandwidth, and the attacker has enough computational and storage resources to process messages and store states. This is a realistic assumption since, as shown in Section 3, processing Kad messages does not involve expensive computations and the total amount of state in the network is under 20GB.

3. ATTACKING THE KAD NETWORK

Since we assume an attacker does not corrupt IP communication between honest nodes, to effectively attack keyword queries the attacker must first cause honest nodes to send keyword queries to its malicious nodes. Then it must make these queries fail. Thus, conceptually, our attack has a *preparation phase*, where the attacker poisons as many routing table entries as it can manage, and an *execution phase*, where the attacker causes queries routed through its malicious nodes to fail. In practice, however, the execution phase can begin in parallel with the preparation phase.

3.1 Preparation Phase

Crawling. Suppose an attacker controls n hosts with index $i, i \in [\mathbf{0}, n-\mathbf{1}]$. For simplicity, we assume each host has an equal amount of bandwidth. The attacker creates a table with tuples $\langle i, IP_i, port_i \rangle$. This table is distributed to the n hosts. Then a malicious node is started on each computer. Each node generates an ID $M_i = \frac{2^{128} \times i}{n}$ so that the n IDs partition the ID space into n pieces. Next they join the Kad network and find their neighbors in the ID space. Starting from its neighbors, each M_i discovers nodes with IDs in the range $[M_i, M_{i+1})$, by picking a previously discovered node, and "polling" its routing table by making appropriate KADEMLIA_REQ queries. This process continues until M_i either fails to discover additional nodes or finds its available bandwidth exhausted.

Back-pointer hijacking. In addition to polling the nodes that it discovers, after M_i learns the routing table of node A, it also *hijacks* a certain fraction of the pointers in A's routing table as follows. Suppose A has honest node B in its routing table. By sending a HELLO_REQ message to A claiming to be from ID_B, M_i can hijack this back-pointer. This hijacking is attributable to three factors. First, Kad does not have ID authentication and allows nodes to pick their own IDs. Second, Kad node IDs are not specific to a node's network location; a node that changes its IP address will retain its ID and update its address with HELLO_REQ messages. Third, when receiving such a HELLO_REQ, A does not verify whether B is still running at the current IP address and port.

After creating a false contact by hijacking a back-pointer, it is possible that the false contact could later be corrected by one of three methods: [4]

1. If A is also in B's routing table, and B sends a KADEMLIA_REQ or HELLO_REQ to A, A will update the pointer. To prevent this, M_i will also hijack B's pointer to A.
2. If node C is one of A's contacts, and has B as a contact, C could include B in a KADEMLIA_RES message. This can be prevented by hijacking C's pointer to B as well.
3. If node C is not one of A's contacts, but has B as a contact, there is a small probability that when C is discovered as an intermediate hop, it returns B in a KADEMLIA_RES message. This scenario is unlikely, since A already has a pointer to B's ID, and the intermediate hops of a keyword search increase the prefix match length unless a timeout occurs.

In our attack, M_i attempts to prevent cases (1) and (2) above. Our experiments produced no instances of case (3).

3.2 Execution Phase

The execution phase of our attack exploits weaknesses in Kad's routing algorithm to cause queries to fail when a malicious node is used as a contact. In other DHTs, malicious nodes can fail queries

[4] In eMule, only the first scenario will result in correction of the back-pointer.

by *query dropping*, *misrouting queries*, and/or *replica root impersonation*. The Kademlia parallel routing algorithm is designed to resist dropping, and in particular it would be counterproductive for an attacker to fail to respond to a KADEMLIA_REQ, because this would cause the querier to drop the malicious node from its routing table. We note, however, that Kad inherits a generic weakness from Kademlia: at each intermediate step, the *closest* contacts are used to discover the next hops, so that an attacker who knows or can impersonate arbitrary nodes in the ID space can "hijack" the query by returning at least α nodes that are closer to the key than those returned by other intermediate hops. The details of how to fail a query after this point depend on the termination conditions of the DHT. We tested two methods of failing a Kad query using this idea.

Fake Matches. This attack exploits the fact that a keyword query terminates when the querier Q receives more than 300 keyword matches in response to SEARCH_REQ messages. Thus, when a malicious node receives a SEARCH_REQ for a keyword, it can send a list of 300 bogus matches in response. Since the response list is long enough, the querier will stop sending KADEMLIA_REQ or SEARCH_REQ messages even though it hasn't reached a live honest replica root yet, causing the query to fail.

We found that this attack works with aMule and early versions of eMule clients [5]. However, eMule clients version 0.47a and later will not halt unless the matches all correspond to the specific keyword the user used to generate the query. Thus, to defeat this client, the attacker must be able to "reverse" the hashed key and find the corresponding keyword. For many popular searches, this can be done in advance by dictionary search; however, we did not attempt to measure the dictionary size necessary to ensure a high probability of success with this approach.

In either case, this attack depends on malicious nodes receiving SEARCH_REQ requests before honest replica roots can respond to a search. Our attack achieves this goal as follows. Each KADEMLIA_REQ for a keyword query carries the key. Node N is a replica root for key K if $ID_{\mathbf{N}} \oplus K < \delta$ where δ is the *search tolerance*. Thus for each KADEMLIA_REQ received, a malicious node can generate a contact whose ID is a replica root. The IP and port fields are set to point to the malicious node $M_{\mathbf{i}}$, where $i = K \mod n$. Upon receiving this reply, the querier will send a KADEMLIA_REQ to the malicious colluder $M_{\mathbf{i}}$ to find more replica roots and to confirm that it is alive. The colluder $M_{\mathbf{i}}$ receives the KADEMLIA_REQ and finds $i = K \mod n$, i.e., it is responsible for sending false matches to the keyword. Hence it replies to show it is alive without introducing other colluders. Receiving this reply, the querier sends a SEARCH_REQ message to $M_{\mathbf{i}}$, who proceeds as described above.

"Stale" contacts. A more efficient attack that works with all clients we tested exploits Kad's timeout conditions. Recall that if all three of the closest nodes at a given step timeout, a Kad client will find its closest backup contact, and try to contact that node; this process repeats every second until more live contacts are found or 25 seconds have elapsed. Thus, when M receives a KADEMLIA_REQ, it generates a KADEMLIA_RES with 30 contacts. For the ith contact, the ID is set as $key - i$, and the IP and port can be set to anything not running a Kad node. For example, they can be set to an unroutable address or a machine targeted for a DDoS attack. Receiving the reply from M, with high probability Q inserts the contacts at the beginning of its list of possible contacts since these contacts are very close to the key. Three of them will be tried by Q immediately. Since they don't reply, after three seconds, Q will try one more every second. Finally, after another 22 seconds, Q will stop trying more contacts. The attack may fail if Q finds an honest replica root before it receives the reply from M.

[5] At the time of writing, we used aMule 2.1.3 and eMule 0.48a

This attack is simple, works with high probability against any keyword, and has a very low bandwidth overhead - it takes one KADEMLIA_RES to attack one keyword query. After compressing, the message contains about 128 bytes of data. Thus our attacker simply attacks every keyword query it sees in this manner.

4. ATTACK EVALUATION

To evaluate the effectiveness and bandwidth cost of our attack, we launched the attack on a large number of simulated victim nodes connected to the Kad network. The victim nodes use a modified aMule client to save resources. We also validated the attack techniques at a smaller scale using the latest eMule release at the time of writing (0.48a).

4.1 Validation of Attack Techniques

We validate the effectiveness of our attack techniques against eMule with the following experiment. We used one victim node Q – running version 0.48a of the eMule client – and one malicious node M. In one run of the experiment, Q joins the Kad network and populates its routing table. After an hour, we start the malicious node, which tries to hijack fraction p of Q's routing table. [6] Figure 3 (a) shows the experiment result where p is set to **10%**, **20%**, and **30%**. The measured percentage is computed as $f = \frac{\mathbf{h}}{\mathbf{c}}$, where h is the number of contacts hijacked by M and c is the number of contacts polled by M. The measured percentage is larger than the planned percentage because the hijack code was configured to hijack a routing table with 860 contacts. At the time of hijacking, however, Q has only about 750 contacts and some of the contacts are stale, so they are neither returned by Q nor used in keyword queries.

To test the effectiveness of our attack on keyword queries, we measured the percentage of failed keyword queries given different percentages of contacts hijacked. With f fraction of contacts hijacked, with probability at least $\mathbf{1} - f^{\mathbf{3}}$, at least one hijacked contact should be used in a query. In the experiment, we input a list of 50 keywords [7] to Q and count the number of failed queries. Figure 3 (b) shows that the result is close to our expectation.

4.2 Bandwidth Usage

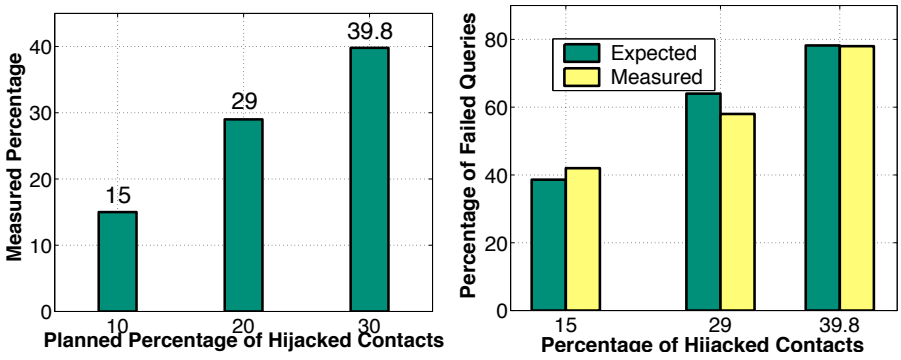
In our attack, bandwidth is used for three tasks: hijacking backpointers, maintaining hijacked back-pointers, and attacking keyword queries. Assuming the worst case for the attacker, every node is stable and its routing table is fully populated. The Kad network has approximately one million nodes, so a fully populated routing table has 86 k-buckets – 11 k-buckets on the 4th level and 5 k-buckets for each of the $\mathbf{\log(1,000,000) - 5 \approx 15}$ additional levels.

Hijacking Back-Pointers. Suppose an attacker wants to stop fraction g of the queries of a victim, then it should hijack $p = \sqrt[3]{\mathbf{1} - g}$ of the victim's routing table. The attacker can send one KADEMLIA_REQ message to poll a k-bucket, so it takes 86 KADEMLIA_REQ messages to poll a routing table. Then the attacker sends one HELLO_REQ message per hijacked back-pointer. So it takes $\mathbf{86 \times 10} \times p = \mathbf{860} \times p$ HELLO_REQ messages to hijack p fraction of backpointers in a routing table. [8] Therefore, in the preparation

[6] To simplify the discussion, p fraction of contacts in each of Q's k-buckets are hijacked.

[7] The list includes popular movies, songs, singers, softwares, filename extensions, etc.

[8] To simplify the discussion, we assume the attacker hijacks the same percent of contacts in every k-bucket of a victim. To optimize the attack, an attacker should prefer to hijack high level backpointers, since high level contacts are used more often in queries. As a special example, on average, $\frac{\mathbf{11}}{\mathbf{16}} = \mathbf{68.75\%}$ of a node's queries use the top (4th) level contacts. In this case, the number of messages (of all four types) and bandwidth costs are less.



(a) Hijacking back-pointers (b) Attacking keyword queries

Figure 3: Attack Technique Validation

phase, the number of messages and the bandwidth cost to attack g fraction of queries sent by n Kad nodes are:

$$\text{Number of messages} = 86 \times n + 860 \times \sqrt[3]{1-g} \times n \quad (1)$$

$$\text{Bytes in} = 86 \times n \times 322 + 860 \times \sqrt[3]{1-g} \times n \times 55 \quad (2)$$

$$\text{Bytes out} = 86 \times n \times 63 + 860 \times \sqrt[3]{1-g} \times n \times 55 \quad (3)$$

Maintaining Hijacked Back-Pointers. Kad nodes ping their contacts periodically. To maintain hijacked back-pointers, malicious nodes must reply to these HELLO_REQ messages. The period of pinging a contact increases and will be fixed at two hours if the contact is in the routing table for more than two hours. For maintenance, every hour, a node also sends a KADEMLIA_REQ message to fix a k-bucket, but only if the k-bucket has eight or more empty slots. We ignore the cost of handling these KADEMLIA_REQ messages since they are less frequent. It is very unlikely that a high level k-bucket has eight or more empty slots, especially when an attacker hijacks high level back-pointers. Hence the number of messages and the bandwidth cost are:

$$\text{Number of messages per second} = \frac{860 \times \sqrt[3]{1-g} \times n}{2 \times 3600} \quad (4)$$

$$\text{Bytes in (out) per second} = \frac{860 \times \sqrt[3]{1-g} \times n \times 55}{2 \times 3600} \quad (5)$$

Attacking Keyword Queries. The uplink cost to attack one keyword query is a single 128-byte KADEMLIA_RES message, while the downlink cost is a single 63-byte KADEMLIA_REQ message. Suppose the users of the Kad network issue w keyword queries per second, on average. The total bandwidth cost of attacking g fraction of keyword queries is $w \times g$ KADEMLIA_RES messages per second. Hence we estimate that the number of messages and the bandwidth cost to attack g fraction of queries sent by n Kad nodes are:

$$\text{Number of messages per second} = w \times g \times n \quad (6)$$

$$\text{Bytes in per second} = w \times g \times n \times 63 \quad (7)$$

$$\text{Bytes out per second} = w \times g \times n \times 128 \quad (8)$$

To estimate w, we joined 216 nodes with random IDs to the Kad network, each through a different bootstrapping node scattered throughout the Kad network. Every node counted the number of keyword-search KADEMLIA_REQ messages it received in each one-hour period and the average was computed. This experiment ran for 24 hours. The one-hour period with the highest average number of queries resulted in 405 queries per host. [9] Hence we estimate that, to attack all keyword queries of the whole Kad network, the download bandwidth required is 7.09 megabytes per second (MBps), and upload bandwidth required is about 14.4 MBps.

[9] Although the average number of query messages was measured during a short period, we believe this is sufficient to show the order of magnitude of the bandwidth required for our attack.

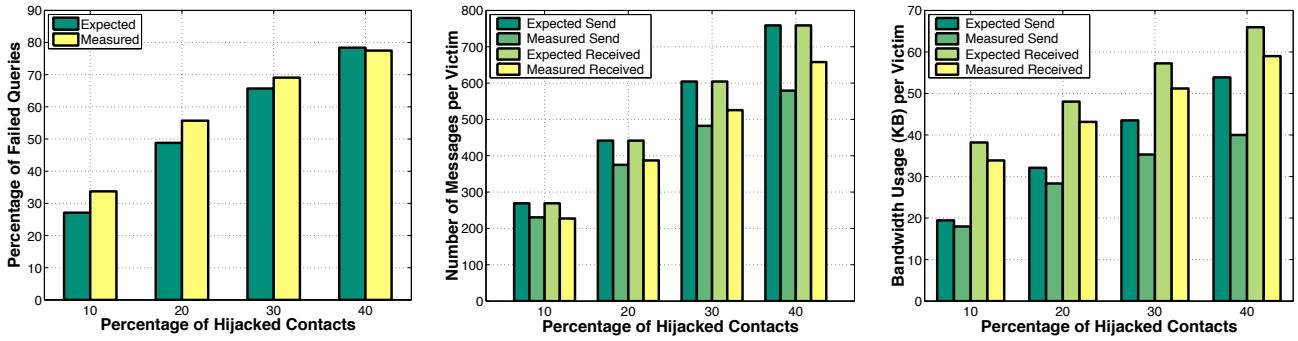
4.3 Large Scale Experiment

In this experiment, we use about 500 PlanetLab [5] machines to run a large number of Kad nodes as victims, and a server in our lab to run the attackers. The victim nodes for this experiment ran a slightly modified aMule client: as with eMule and aMule, the victim client has two layers – the DHT layer provides lookup services (for keyword search, for example) to the application layer, which handles functions like file publishing and retrieval. The DHT layer was largely unmodified. It follows the same protocols for maintaining routing tables and parallel iterative routing as eMule and aMule, and uses the same system parameters, e.g, time interval between HELLO_REQ messages. In the application layer, the modified client issues random keyword queries periodically. To save bandwidth and storage of the PlanetLab nodes, however, it does not support PUBLISH_REQ and SEARCH_REQ messages from other Kad nodes. In other words, the victims provided routing service to the Kad network, but not binding services.

During the experiment, about 25,000 victim nodes bootstrapped from 2000 different normal Kad nodes. If it fails to bootstrap, a victim node exits without issuing any keyword queries. In our experiments, $11,303 - 16,105$ nodes bootstrapped successfully. After a successful bootstrap, each node sends a message to the attacker registering as a victim. In the next two hours, the victims build their routing tables and help other normal Kad nodes route KADEMLIA_REQ messages. After that, each victim sends 200 queries, one every 9 seconds, and exits five minutes after sending the last query. The attacker starts at the same time as the victims. It listens for registration messages, and starts to hijack the routing tables of victims after 1.5 hours, then attacks every keyword query. The attack run for one hour (half hour for hijacking, half hour for attacking queries). To avoid attacking normal Kad nodes, the victims do not provide the attacker as a contact to normal Kad nodes.

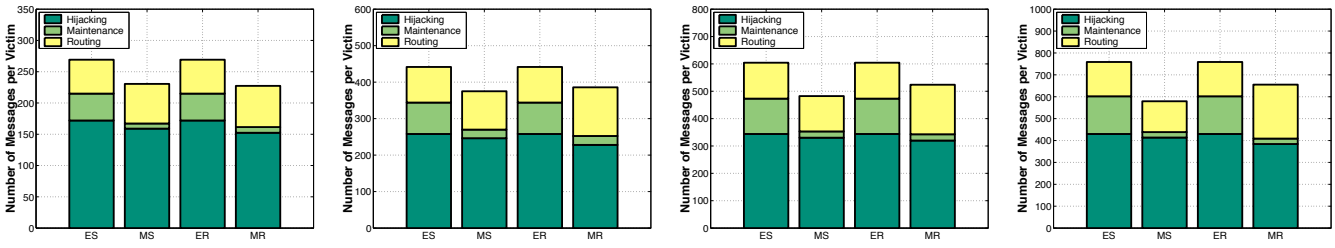
Figure 4 (a) shows the comparison between expected and measured keyword query failures, where we say a query fails if the victim does not find any normal Kad nodes within the search tolerance of the target ID. In the **10%**, **20%**, and **30%** cases, the measured frequency is higher than the expected number. However, the difference between the measured numbers and expected numbers decreases as the percentage of hijacked contacts increases. In the **40%** case, the measured frequency is slightly lower than the expected figure.

Figure 4 (b) and (c) show the attacker's message and bandwidth costs. The attack cost was slightly less than expected. To find the reason, the messages collected are categorized into three categories: (i) hijacking, (ii) maintenance, and (iii) routing attack, as



(a) Query Fail Rate (b) Number of Messages (c) Bandwidth

Figure 4: Large Scale Attack Simulation: $11,303$ $16,105$ victims and 50 attackers. In (b) and (c) the numbers of messages and bandwidth costs are normalized based on the number of victims in each experiment.



(a) 10% (b) 20% (c) 30% (d) 40%

Figure 5: Number of Messages in Detail: ES, MS, ER, MR stand for expected sent, measured sent, expected received, and measured received respectively. The numbers of messages are normalized based on the number of victims in each experiment.

shown in Figure 5. The number of messages used for hijacking (i) is close to the expectation. The difference is mainly due to messages lost at the victim side: one lost KADEMLIA_RES results in several fewer HELLO_REQ messages. The attackers received many fewer maintenance messages (ii) than the expectation. This is due to the short period of the attacks: most victims finished before maintaining their hijacked contacts. In a longer term attack, the number of messages for maintaining hijacked back-pointers should be close to the expectation. The attackers receive more routing messages (keyword queries) (iii) than expected. We analyzed the logs of the attackers and found that a large number of keyword queries are received more than once. A victim sends multiple copies of a keyword query to an attacker if several hijacked contacts are used in the query. The fact that some keyword queries are received multiple times and others are not received (hence cannot be attacked) suggests that the hijacking algorithm can be improved. One way to improve is to first analyze the polled routing table, then selectively hijack contacts according to the distance between the contacts. The number of routing messages sent is close to the expectation because repeated queries received in a short period are dropped.

5. REFLECTION ATTACK

The major disadvantage of the the proposed attack is that it has an ongoing cost of around 100 Mbps. However, a slight twist on this attack involves hijacking a node's *entire* routing table so that the entries in the routing table point to the victim itself rather than to the attacker - we call this the reflection attack [10]. This greatly reduces the ongoing cost of the attack, while leaving the victim unable to contact any other Kad nodes. Since a node does not perform any check on an IP address and port to determine whether it is its own, a hijacked node will continue to send messages to itself and reply to itself, so that most of the routing table remains hijacked indefinitely.

Although the attack will render the network nearly inoperable at the time it is perpetrated, we expect that the Kad network would slowly recover over time, for a number of reasons. First, there will be some nodes offline at the time of the attack, who are in the routing tables of online nodes. When these nodes rejoin the network and send HELLO_REQ messages to their contacts, their routing table entries will be restored. Second, there will be a few contacts in a node's routing table that cannot be hijacked: each node classifies contacts into one of five types, 0-4. Nodes with type 0-2 (which we will call in aggregate "Type 2") have successfully responded to multiple KADEMLIA_REQ or HELLO_REQ messages; those with type 3 are "new contacts" that have not yet replied to a request; and nodes with type 4 have failed to reply to a recent request. When responding to the requests of others, a Kad node will only send a "Type 2" contact. Thus we can only hijack the "Type 2" contacts; but a few type 3 or 4 contacts may later reply to the node and be promoted. Thus it may be necessary to repeat the process periodically to limit the network's recovery.

We deployed and tested a small scale evaluation of this type of attack and found it to be highly successful. The experiment was set up with 48 victim nodes deployed across 3 machines, each victim node bootstrapping from a different node in the real Kad network. Once bootstrapping is complete and after waiting for 5 minutes, each victim will send a HELLO_REQ to the attacker node. After waiting 2 hours (to allow the victim nodes to stabilize their routing tables) the attacker starts the hijacking attack. It will poll the routing table of each victim and hijack all received contacts. To track the rate of recovery, the victim nodes print their routing tables every 10 minutes. Since the victim nodes are connected to the real Kad network, and we did not hijack backpointers to the victims,

[10]It can be argued that a simple check can be performed by every node so that their entries are not themselves, but this is a proof of concept and UDP spoofing can easily be performed by the attacker to have two nodes A and B's routing table entries point to each other

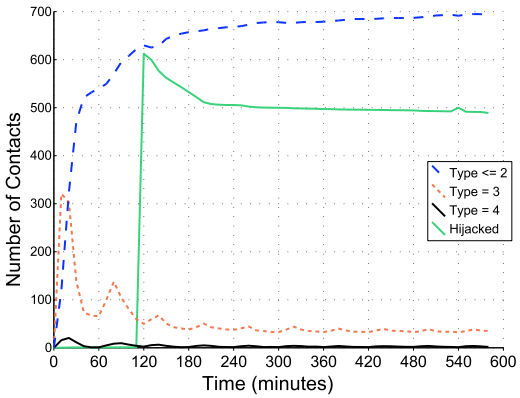


Figure 6: Average hijacked and total contacts over time

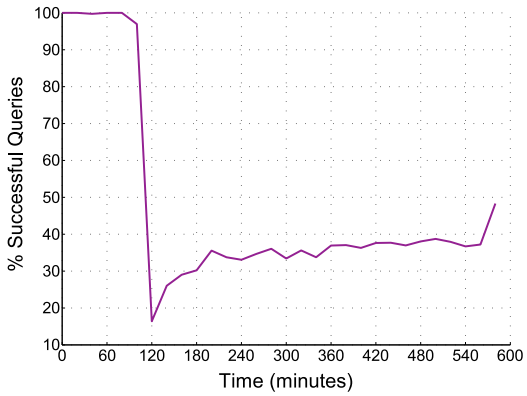


Figure 7: % Successful queries, over 20-minute windows

it should be the case that our experiment overestimates the rate of recovery.

Figure 6 shows the average number of contacts in the routing table for the 48 victim nodes. The number of contacts are further divided by type and whether they were hijacked. At the beginning of the experiment, the number of type 3 contacts is high since all these contacts have just been discovered. As time progresses, the number of type 3 contacts decreases, and the number of type 2 contacts increases. After 2 hours, the attacker starts the hijacking attack. The number of hijacked contacts increases rapidly and then decreases as the victims recover slowly. The number of type 2 contacts includes the number of hijacked contacts. We can see that even after 8 hours, roughly 70% of the victim's contacts still point back to itself. These results suggest that at the full network scale, a second round of hijacking may be sufficient to fully disconnect Kad.

We also measured how the query success rate of the victims changed in the course of the attack. Starting 6 minutes after bootstrapping, each victim sent a query to a randomly chosen key once every 3 minutes, and recorded whether it successfully located a replica root for the key. Figure 7 shows the results of this measurement. We can see that the fraction of successful queries is essentially equal to the fraction of non-hijacked "Type 2" contacts. In the full attack, the contacts of these nodes would also be useless, so this experiment understates the impact of the attack.

Finally, we recorded the cost, in bytes sent and received, of the attack. The total number of bytes per victim sent by the attacker was 52,718, and the total number of bytes per victim received by the attacker was 74,992. Thus an attacker with 166 Mbps of downlink capacity and 117 Mbps uplink capacity could complete the reflection attack on the entire Kad network in one hour, with very little subsequent bandwidth usage.

6. COMPARISON TO OTHER ATTACKS

In this section, we discuss and evaluate several alternative attacks on Kad that rely on similar weaknesses, and present techniques to mitigate these attacks.

6.1 Sybil Attack

Because P2P file sharing systems lack any form of admission control, they are always vulnerable to some form of Sybil attack. A Sybil attack on a P2P routing protocol is used to collect back-pointers, which are used to attract query messages. Therefore, the effectiveness of a Sybil attack can be computed from the set of back-pointers collected by the Sybil nodes. In a measurement study, we joined 28 Sybil nodes to the Kad network. These Sybil nodes were modified to record information about their back-pointers, while maintaining their routing tables and responding to KADEMLIA_REQ messages normally. We identified back-pointers to a Sybil node S as follows. Normal nodes find out if their contacts are alive or not by sending HELLO_REQ or KADEMLIA_REQ messages before their expiration time. Since the longest expiration time of a contact is two hours, S keeps a list of the nodes that have sent it a KADEMLIA_REQ or HELLO_REQ message in the past two hours. At the same time, periodically, S sends a KADEMLIA_REQ message to every node B on this list with its nodeID (S) as the target key. If B's KADEMLIA_RES includes S, then it knows that it is on B's routing table.

In Figure 8 (a), we see that, on average, a Sybil node collects about 500 back-pointers after 24 hours, and about 1400 back-pointers after one week (168 hours). The fraction of queries a Sybil node receives from a back-pointer depend on the common prefix length (*CPL*) between the Sybil node's ID and the back-pointer's ID, because the *CPL* determines the Sybil node's contact level on the back-pointer's routing table.

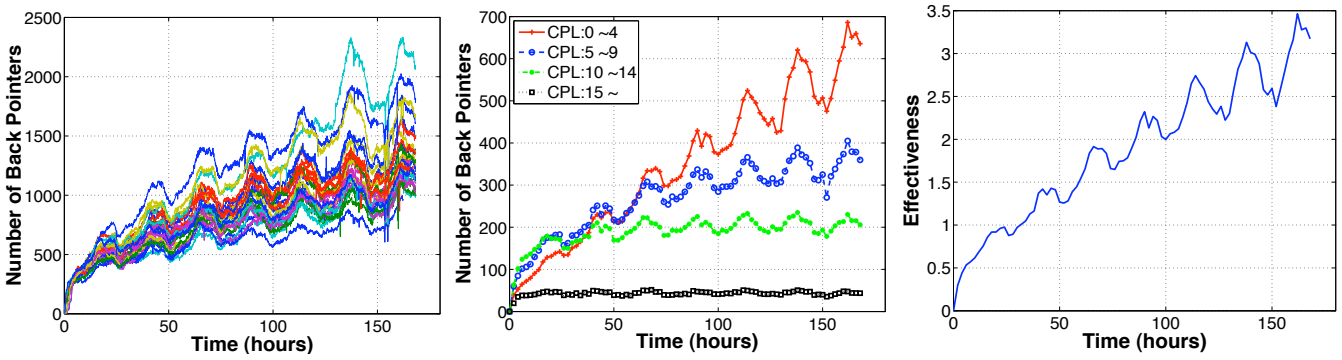
Figure 8 (b) shows that, the number of back-pointers with *CPL* $>=$ **15** quickly becomes stable at approximately 50. After 40 hours, the number of back-pointers with $CPL \in$ **[10, 14]** is stable at approximately 200. Assuming node IDs are uniformly random, on average, there are approximately 1000 ($\frac{1}{2^{10}} \times$ **1,000,000**) nodes with *CPL* $>=$ **10**. The Sybil nodes are on $\frac{1}{4}$ of these 1000 nodes' routing tables. The number of back-pointers with shorter *CPL* keeps increasing since there are more potential candidates. The early hours of Figure 8 (b) also show that, initially, the number of back-pointers with *CPL* $<=$ **4** increases slower than others. This is because nodes' high level k-buckets are usually full, so it takes more time for Sybil nodes to become high level contacts.

We consider a Sybil node to be *completely* part of the Kad network if it attracts as many queries as a stable, honest node [11]. Thus, both Sybil and normal nodes should have the same number of ith level back-pointers, where $i \in$ **[0, log(N))** (Note that higher level contacts are used more frequently than lower-level ones). Since on average, the number of contacts and the number of back-pointers of a node are the same, we say a Sybil node has successfully joined the Kad network if it has approximately **11** $\times$ **10** 4th level back-pointers and **5** $\times$ **10** ith level back-pointers where $i \in$ **[5, log(N))**. Following this argument, we compute the *effectiveness* of a Sybil node (how many stable nodes it is equivalent to) as follows, assuming it has m back-pointers with $CPL_i, i \in$ **[1**, m**]**:

$$\text{effectiveness} = \sum_{i=1}^{m} \alpha_i, \tag{9}$$

$$\text{where } \alpha_i = \begin{cases} \frac{1}{160} & \text{if } CPL_i = 0 \\ \frac{1}{160} \times 0.8 \times \frac{1}{2^{CPL_i - 1}} & \text{else} \end{cases}$$

[11] New nodes that just joined the network are not included.



(a)Total (b) Average, clustered by *CPL* (c) Effectiveness

Figure 8: Sybil Attack Measurement: 28 Sybil nodes run for one week. (a) shows the total number of back-pointers. One line represents one node. (b) shows the average number of back-pointers clustered by the common prefix length (*CPL*) between the Sybil node's ID and the back-pointer's ID. (c) shows the average of Sybil nodes' effectiveness computed with formula (9)

Figure 8 (b) shows that the effectiveness of a Sybil node reaches 1 after approximately 24 hours. Then, the effectiveness increases linearly and reaches 3.5 after 162 hours (almost a week). A linear regression with intercept 0 gives the slope of this line as **0.02** effective nodes per Sybil node-hour, with p-value **0.014** and mean squared error **0.12**. Thus we estimate that, to control 40% of the backpointers in Kad, a naïve Sybil attack will require roughly $400,000/0.02 = 20$ million Sybil node-hours. Clearly backpointer hijacking dramatically reduces the wall-clock time and bandwidth expenditure necessary to attack Kad.

6.2 Index Poisoning Attack

In the index poisoning attack [16], an adversary inserts massive numbers of bogus bindings between targeted keywords and nonexistent files. The goal is that when a user searches for a file, she will find as many or more bogus bindings as bindings to actual files. For instance, if every legitimate binding is matched with a bogus binding, then 50% of her search results are useless; if there are three bogus bindings for every legitimate binding, then 75% of her search results are useless.

This attack can also be applied to deny access to the keyword search service provided by Kad, by targeting all existing (keyword, file) pairs. As with our attack, this attack would involve two phases: a preparation phase in which the attacker infiltrates the network to learn all possible (keyword, file) pairs and an execution phase to insert three bogus (keyword, file') pairs for every pair in the network. Thus the bandwidth complexity of the attack depends on the number of bindings currently in the network and the rate at which bindings must be refreshed.

To estimate the number of (keyword, file) bindings in the Kad network, we joined 256 nodes with uniformly distributed IDs to the live Kad network, and recorded all "publish" messages received by each node for one 24-hour period. Each publish messages is a binding between a (hashed) keyword, a (hashed) file, and some meta-information such as the file name and size. To be conservative, we ignored the meta-information and counted only the number of unique (keyword, file) hash pairs seen by each node. The total number of such unique pairs seen by our 256 node sample was 2,000,000. Since the average size of publish message seen by our sample was 163 bytes, we estimate that publishing enough strings to cause 50% of all Kad bindings to be bogus would require 14.74 MBps; to get to 75% the required bandwidth is 44.22 MBps. Due to the fact that bindings are removed after 24 hours, this cost is incurred continuously throughout the attack.

Note that this attack has a cost roughly three times the cost of our attack, and is also much weaker: on average, a determined user can simply try four of the bindings returned by a poisoned keyword search and one will be a legitimate entry. Furthermore, index poisoning does not interfere at all with the underlying routing mechanism, so DHT lookups related to joins, leaves, and routing table maintenance proceed without disruption. Attacks based on our method affect all DHT lookups equally.

7. MITIGATION

Our attacks rely on two weaknesses in Kad: weak identity authentication coupled with persistent IDs allow pointer hijacking, so that we can intercept many queries; while overaggressive routing (always contacting the three closest contacts) allows us to hijack a query once it has been intercepted. We will discuss measures to mitigate each of these weaknesses, as well as the extent to which they are incrementally deployable.

Identity authentication. Recall that the proposed attack is successful because the malicious node M can hijack an arbitrary entry in A's routing table (say, pointing to B) by sending a HELLO_REQ to A with the fields $\langle ID_B, IP_M, port_M \rangle$. The attack can be mitigated through a number of means. The simplest is to simply disregard these messages when they would change the IP address and/or the port of a pointer: if a node goes offline and comes back with a different IP address and/or port, it will be dropped from any routing tables it is on, but can retain its own routing table.

Another lightweight mitigation technique is to "trust but verify:" When A receives a HELLO_REQ to update B's IP and port, it sends a HELLO_REQ message to $\langle IP_B, port_B \rangle$ to see if B is still running with the previous IP and port. If B (or some node) replies to the HELLO_REQ, then A will not update its routing table. This solution allows nodes to retain their routing tables across invocations, and to stay on the routing tables of others after changing IP addresses. On the other hand, it does not completely eliminate hijacking: since Kad nodes have high churn rates, it is likely that many entries on A's routing table will be offline, and M can effectively hijack these entries. However, the cost of the attack now increases as M will expend time and bandwidth looking for offline contacts. Both this technique and the previous one are fully incrementally deployable in that a client using these algorithms can fully interoperate with current Kad nodes, and will be protected against having its own routing table hijacked. However, these techniques do not protect against hijacked intermediate contacts that might be returned by older clients during a query, or against Sybil attacks that claim an ID close to an expired routing table entry.

Limited protection from Sybil attacks can be obtained using a semi-certified identity, for example Node B could use $hash(IP_B)$ as its node ID.[12] Here every ID is tied with the corresponding IP

[12]Several alternatives are possible: the 64 MSBs can be derived

Table 1: Comparison of identity authentication methods

Method	Secure	Persistent ID	Incremental deployable
Drop Hello with new IP/Port	Yes	No	Yes
Verify liveness of old IP	No	Yes	Yes
ID=hash(IP)	Yes	No	No
ID=hash(Public Key)	Yes	Yes	No

address; clients should refuse to use contacts that do not have the proper relationship between ID and IP address. This approach prevents routing table hijacking, and limits the set of IDs an attacker can choose in a targeted attack. However, it is not incrementally deployable, and does not support mobility: if a node changes IP addresses, it will need to rebuild its routing table and will be dropped from the routing table of others.

Another alternative is that node B uses $hash(PK_B)$ as its ID, where PK_B is a public key. B can then either sign its HELLO_REQ when it changes its IP and/or port, or extra rounds (with new opcodes) can be added to allow newer clients to authenticate node IDs, while older clients continue to ignore the existence of this binding. In eMule, every node already generates its own public/private key pair, used for an incentive mechanism similar to that of BitTorrent. This solution allows all clients to retain their existing routing tables. Newer clients will have only authenticated contacts on their routing tables, while older clients will have both types of contacts. If intermediate contacts are also authenticated, this solution protects new clients from hijacked intermediate contacts, but requires a critical mass of peers running authenticated clients. It does not prevent chosen-ID attacks, although such attacks will carry higher computational costs due to the need to generate public keys that hash to a chosen ID prefix.

Table 7 summarizes the methods discussed above. Since a mitigation method must be secure and incrementally deployable, "Drop HELLO_REQ with new IP" becomes the winner. In addition, this method does not change the behavior of the Kad network. To support this argument, we conducted an experiment recording the frequency of HELLO_REQ messages with a new IP address and/or port. We joined 214 nodes to the Kad network and recorded every HELLO_REQ with new IP and/or port. After 4.5 days, on average, each node had 5284 different contacts, of which only 171 contacts (3.23%) were updated with a new IP and/or port.

Routing Corruption. Without some defense against Sybil attacks, routing attacks are still possible even with the above mitigation mechanisms. Recall that routing attacks work in Kad because although every node performs three parallel lookups, those lookups are not independent. If node A wants to perform a search, it will send out three KADEMLIA_REQ to the closest nodes ((B, C, and malicious node M) to the target T (in the XOR metric) that A knows about. M can "hijack" all three search threads by replying to A with at least three contacts that are close to T. This can be mitigated by keeping the strands of a search separate: at each stage of the search, A should send a KADEMLIA_REQ to the closest contact it has not yet used in each strand. Note that it is possible that a thread of the lookup might "dead-end." In this case, A should restart the thread from the earliest unused contact in another thread. A should not terminate a search until it has received a reply to a SEARCH_REQ or timed out in each thread.

This routing algorithm mitigates, but does not eliminate, the effects of routing attacks. Suppose that an attacker controls 40% of all of the backpointers in the current Kad network; then he should be able to prevent roughly 98% of all queries from succeeding, under the current routing algorithm – he has a 78% chance of stopping the query at each hop – but could prevent only 45% of queries made with the "independent thread" routing algorithm. At 10% of backpointers, these figures become 59.5% and 1.7%, respectively. We thus conclude that this technique is easy to incrementally deploy (and will immediately improve attack resistance for any client that upgrades), and that it is critically important to implement mitigation techniques for both weaknesses.

from $hash(IP_B)$ and the 64 LSBs from $hash(IP_B \| port_B)$ to support NAT; if subnet-level attackers are a concern the 64 MSBs can be derived from $hash(IP_B/24)$; etc.

8. RECENT CHANGES

New versions of both the aMule and eMule clients have recently been released – aMule 2.2.1 on June 11, 2008 and eMule 0.49a on May 11, 2008. Both clients use the same updated version of the Kad (which we will call Kad2) algorithm.[13] The main changes which affect our attacks are described here.

Kad2 implements a flooding protection mechanism that limits the number of messages processed from each IP address, for example, a node can receive at most 1 KADEMLIA_REQ per IP address every 6 seconds. While this mechanism increases the time required to poll a single routing table, it does not increase the time required to poll the entire network, since an attacker can contact many nodes in parallel while not exceeding the rate of 1 request per 6 seconds at any individual node.

Each Kad node limits entries in its routing table by IP address and /24 subnet. Clearly, this change prevents the reflection attack presented in Section 5. However, if backpointer hijacking is still possible, an attacker who can spoof UDP packets can still effectively partition the network into disjoint subsets of size 900 by pointing all of the routing table entries of each partition to the other members of the partition.

Finally, Kad2 includes code that may be used to prevent hijacking. The new code contains a boolean variable which indicates whether entries in the routing table can be updated (change in IP address). This variable can be set to false so that the entries are never updated and this will prevent a hijacking attack (This is our first proposed mitigation method in Section 7 – "Drop Hello with new IP/Port"). Since this variable is set to true currently (to reduce the number of dead contacts and to enable long-lived nodes to continuously contribute to the network, although our measurements indicate that such behavior is uncommon) it does not prevent hijacking attacks; we have empirically confirmed this by running the new client and successfully hijacking a single backpointer.

The lastest eMule and aMule clients implement *Protocol Obfuscation* [22] by encrypting packets. A node sends different encryption keys to different contacts in plaintext when the contacts are inserted into its routing table, and it stores these keys in the routing table along with the contacts' protocol versions. In future protocol versions, these encryption keys *could* also be used to serve as authentication tokens to prevent hijacking attacks; note that an attacker cannot utilize clients' backward compatibility to bypass the authentication step because the contacts' protocol versions are recorded in the routing table. In this case, although it is still possible, the hijacking attack is much harder to launch since an attacker needs to intercept the communication between honest nodes.

In summary, the latest Kad clients implement several features which could be used in future versions to mitigate our attack. However, the current version only slightly increases the cost of our attack. We still need only 1 IP address with the same network and storage resources to crawl the whole Kad network and collect the routing tables of all nodes. To hijack backpointers, our attack now requires 1 IP address per hijacked contact. For example, to hijack **30%** of the top level buckets (**3** out of **10** contacts in each bucket) in each routing table (see Footnote 8) – stopping more than **60%** of queries – now requires $3*11$ (top-level buckets) $=$ **33** IP addresses. Note that the same **33** IP addresses can be used for all of the hijacked backpointers since IP filtering is done locally for each node.

[13]Both clients still support the old Kad protocol for backward compatibility.

9. RELATED WORK

Since Kademlia [20] was introduced in 2001, several variations have been implemented, including the discontinued Overnet and eDonkey2000 projects, and also the separate eMule [11], aMule [1] and MLDonkey projects. Kademlia is in use by several popular BitTorrent clients as a distributed tracker [2, 18, 3]. Because Kad seems to be the largest deployed DHT, several studies have measured various properties of the network. Steiner *et al*. [27] crawl the Kad network and report that most clients only stay for a short period and only a small percentage stay for multiple weeks; while Stutzbach and Rejaie measured the lookup performance [31] and churn characteristics [32] of the deployed Kad network. None of these works address the security of Kad.

Sit and Morris [26] present a taxonomy of attacks on DHTs and applications built on them. They further provide design principles to prevent them. Lynch *et al*. [17] propose to use a Byzantine Fault Tolerance replication algorithm to maintain state information for correct DHT routing. The Sybil attack has been studied by several groups [14, 9]. Two Sybil-resistant schemes based on social links were recently proposed in [19, 7]. Castro *et al*. [4] design a framework for secure DHT routing which consists of secure ID generation, secure routing table maintenance, and secure message forwarding. Fiat and Saia [12, 23] give a protocol for a "content-addressable" network that is robust to node removal. Kubiatowicz [15] make Pastry and Tapestry robust using *wide paths*, where they add redundancy to the routing tables and use multiple nodes for each hop. Fiat *et al*. [13] define a *Byzantine join* attack model where an adversary can join Byzantine nodes to a DHT and put them at chosen places. Singh *et al*. [25] observe that a malicious node launching an eclipse attack has a higher in-degree than honest nodes. They propose a method of preventing this attack by enforcing in-degree bounds through periodic anonymous distributed auditing. Condie *et al*. [6] induce churn to mitigate eclipse attacks. Liang et al. [16] report that substitution of "fake content" in place of the desired values on the KaZaA P2P network is prevalent but also detectable. Naoumov and Ross [21] proposed to exploit Overnet as a DDoS engine with index poisoning and the generic routing table poisoning. El Defrawy *et al*. [8] proposed to misuse BitTorrent to launch DDoS attacks. While several of these works report on DHT routing attacks, none address Kad or Kademlia specifically, and none are tested on a widely-deployed DHT.

10. CONCLUSION

We have demonstrated that it is possible for a small number of attackers, using approximately 100 Mbps of bandwidth, to deny service to a large portion of the Kad network. By contrast, direct DDoS to the same number of hosts would require roughly 1 Tbps of bandwidth, assuming an average downstream capacity of 1 Mbps per Kad node. Moreover, we showed that our attacks are more efficient than currently known attacks (Sybil and Index Poisoning). These attacks highlight critical design weaknesses in Kad, which can be partially mitigated.

Even with the recent security updates to Kad, we have shown that our attack still works using nearly the same resources. However, an easy change to the code can prevent hijacking attacks.

Acknowledgements. We are grateful to Hendrik Breitkreuz for helpful discussions about the latest versions of the Kad client. This work was funded by the NSF under grant CNS-0716025.

11. REFERENCES

[1] aMule network. `http://www.amule.org`.
[2] Azureus. http://azureus.sourceforge.net.
[3] BitComet. http://www.bitcomet.com.
[4] CASTRO, M., DRUSCHEL, P., GANESH, A., ROWSTRON, A., AND WALLACH, D. S. Secure routing for structured peer-to-peer overlay networks. In *OSDI* (2002).
[5] CHUN, B., CULLER, D., ROSCOE, T., BAVIER, A., PETERSON, L., WAWRZONIAK, M., AND BOWMAN, M. Planetlab: an overlay testbed for broad-coverage services. *Sigcomm Comput. Commun. Rev.* (2003).
[6] CONDIE, T., KACHOLIA, V., SANKARARAMAN, S., HELLERSTEIN, J., AND MANIATIS, P. Induced churn as shelter from routing table poisoning. In *NDSS* (2006).
[7] DANEZIS, G., LESNIEWSKI-LAAS, C., KAASHOEK, M. F., AND ANDERSON, R. Sybil resistant DHT routing. In *ESORICS* (2005).
[8] DEFRAWY, K. E., GJOKA, M., AND MARKOPOULOU, A. Bottorrent: Misusing BitTorrent to Launch DDoS Attacks. In *Usenix SRUTI* (June 2007).
[9] DOUCEUR, J. R. The sybil attack. In *Proc. of the IPTPS02* (2002).
[10] eDonkey network. `http://www.edonkey2000.com`.
[11] eMule network. `http://www.emule-project.net`.
[12] FIAT, A., AND SAIA, J. Censorship resistant peer-to-peer content addressable networks. In *SODA* (2002).
[13] FIAT, A., SAIA, J., AND YOUNG, M. Making chord robust to byzantine attacks. In *ESA* (2005).
[14] FRIEDMAN, E., AND RESNICK, P. The Social Cost of Cheap Pseudonyms. *J. of Economics and Management Strategy* (2001).
[15] HILDRUM, K., AND KUBIATOWICZ, J. Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In *DISC* (2003).
[16] LIANG, J., KUMAR, R., XI, Y., AND ROSS, K. W. Pollution in P2P file sharing systems. In *INFOCOM 05* (2005).
[17] LYNCH, N., MALKHI, D., AND RATAJCZAK, D. Atomic data access in content addressable networks. In *IPTPS* (2002).
[18] Mainline. http://www.bittorrent.com.
[19] MARTI, S., GANESAN, P., AND GARCIA-MOLINA, H. DHT Routing Using Social Links. In *P2PDB* (2004).
[20] MAYMOUNKOV, P., AND MAZÍERES, D. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS* (2001).
[21] NAOUMOV, N., AND ROSS, K. Exploiting P2P systems for DDoS attacks. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems* (2006).
[22] Protocol Obfuscation. `http://www.emule-project.net/home/perl/help.cgi?l=1&rm=show_topic&topic_id=848`.
[23] SAIA, J., FIAT, A., GRIBBLE, S., KARLIN, A., AND SAROIU, S. Dynamically fault-tolerant content addressable networks. In *IPTPS* (2002).
[24] SINGH, A., CASTRO, M., DRUSCHEL, P., AND ROWSTRON, A. Defending against eclipse attacks on overlay networks. In *EW11* (2004).
[25] SINGH, A., NGAN, T.-W. J., , DRUSCHEL, P., AND WALLACH, D. S. Eclipse attacks on overlay networks: Threats and defenses. In *Infocom* (2006).
[26] SIT, E., AND MORRIS, R. Security Considerations for Peer-to-Peer Distributed Hash Tables. In *IPTPS* (2002).
[27] STEINER, M., BIERSACK, E. W., AND EN-NAJJARY, T. Actively Monitoring Peers in KAD. In *IPTPS 07* (2007).
[28] STEINER, M., EFFELSBERG, W., EN NAJJARY, T., AND BIERSACK, E. W. Load reduction in the KAD peer-to-peer system. In *DBISP2P 2007, 5th International Workshop on Databases, Information Systems and Peer-to-Peer Computing, September, 24, 2007, Vienna, Austria* (Sep 2007).
[29] STEINER, M., EN NAJJARY, T., AND BIERSACK, E. W. Analyzing peer behavior in KAD. Tech. Rep. EURECOM+2358, Institut Eurecom, France, Oct 2007.
[30] STEINER, M., EN-NAJJARY, T., AND BIERSACK, E. W. A global view of kad. In *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement* (New York, NY, USA, 2007), ACM, pp. 117–122.
[31] STUTZBACH, D., AND REJAIE, R. Improving lookup performance over a widely-deployed DHT. In *Infocom 06* (2006).
[32] STUTZBACH, D., AND REJAIE, R. Understanding churn in peer-to-peer networks. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement* (2006).