SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks. (2009)
Published online in Wiley InterScience
(www.interscience.wiley.com) DOI: 10.1002/sec.172

Attacking the Kad network—real world evaluation and high fidelity simulation using DVN ‡

Peng Wang*,†, James Tyra, Eric Chan-Tin, Tyson Malchow, Denis Foo Kune, Nicholas Hopper and Yongdae Kim

Department of computer science, University of Minnesota, Twin Cities, MN, U.S.A.

Summary

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. Using our large-scale simulator, DVN, we successfully scaled up to a 200 000 node experiment. 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. Copyright © 2009 John Wiley & Sons, Ltd.

KEY WORDS: P2P; Kad; simulation; attack; security

1. Introduction

The Kad network is a peer-to-peer distributed hash table (DHT) based on Kademlia [1]. It supports the growing user population of the eDonkey [2][§] file sharing network by providing efficient distributed keyword indexing. The Kad DHT[‖] is very popular, supporting several million concurrent users [3,4], and as the largest deployed DHT, its dynamics has been the subject of several recent studies [5–8].

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 [9–17]. 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

*Correspondence to: Peng Wang, Department of computer science, University of Minnesota, Twin Cities, MN, U.S.A.
†E-mail: pwang@cs.umn.edu
‡A previous version of this paper has appeared at SecureComm 2008.
§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.
‖There are several Kademlia-based networks such as the Azureus BitTorrent DHT, but we will refer to the aMule/eMule DHT as 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,[¶] 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 [10,11]) 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 emphasize that our attack is new and is different from other similar attacks such as Sybil and Eclipse attacks. However, it does rely on basic vulnerabilities such as lack of authentication. 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 1 h, and prevent 75% of all keyword lookups. We also experimented with a variant of the attack. Instead of attacking keyword lookups, we attacked the control plane, stopping 95% of routing requests. Attacking the control plane is more powerful than attacking the data plane since the latter depends on the control plane for successful keyword search queries.

Our evaluation methods include high fidelity simulations of the Kad network on our event-based simulator DVN (Distributed Virtual Network) of which our largest simulation consisted of 200 000 nodes. The simulation results allowed us to quantify the effect of the attacks on a larger scale without affecting real users in the Kad network while running with the actual protocol stack code ported as a library to our simulator. Thus, we validated our new attack on the real Kad network and for larger scale simulations, we used DVN. Both the proposed attack and proposed simulator are important to demonstrate the vulnerability of the Kad network.

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 [18] 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 [19] because we actively acquire entries rather than passively promoting compromised nodes.

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 been released—aMule 2.2.1 on 11 June 2008 and eMule 0.49a on 11 May 2008. We show that although they have new features intended to improve security, our attacks still work with the same resource requirements. We worked with the developers of eMule to provide a simple fix for our attacks and as of eMule 0.49b [20], our attacks were mitigated.

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. Our simulator DVN is explained in Section 4. Section 5 gives analytical, experimental, and simulation results on the cost-effectiveness of our attack. Section 6 reports on a related attack with lower bandwidth costs in the second phase. Section 7 compares our attack to general DHT attacks, while Section 8 discusses mitigation strategies for Kad. Section 9 outlines the recent changes in the Kad clients and how they affect our attacks. Finally, Section 10 discusses related work on Kad and DHT security, and Section 11 presents our conclusions and directions for future work.

[¶]obtaining the routing table of other nodes in the network.

DOI: 10.1002/sec

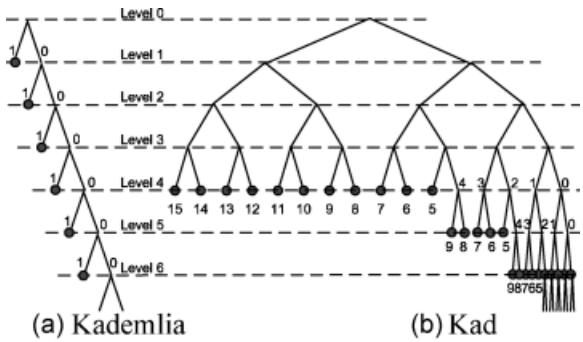

Fig. 1. Routing table structures of (a) Kademlia and (b) Kad. Leaves depict *k*-buckets.

2. Background

2.1. Overview of Kademlia and Kad

2.1.1. 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.

2.1.2. 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 [3] 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 h.

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 routing table, at each step Q picks its three contacts closest to the key and

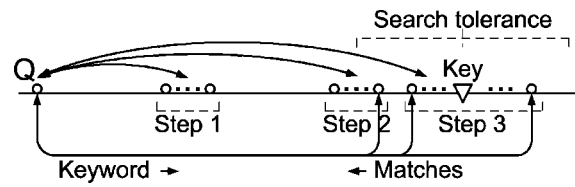

Fig. 2. Kad keyword search.

DOI: 10.1002/sec

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 3 s; if not, it picks the closest backup node, removes it from the list, and sends it a KADEMLIA_REQ message. After 25 s, 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 ($25 - 3 = 22$) 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 min reported in Reference [5], 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 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 as well as node searches to fail. 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 20 GB.

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

3.1.1. Crawling

Suppose an attacker controls n hosts with index $i, i \in [0, n-1]$. For simplicity, we assume each host has an equal amount of bandwidth. The attacker creates a table with tuples $\langle i, \mathrm{IP}_i, \mathrm{port}_i \rangle$. This table is distributed to the n hosts. Then a malicious node is started on each computer. Each node generates an ID

DOI: 10.1002/sec

$M_i = \frac{2^{128} \times i}{n}$ 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.

3.1.2. 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—this is a lack of entity authentication where the actual node is not authenticated. 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. The last two factors are due to a lack of message authentication, that is, the messages sent by the Kad nodes are not verified. Our attack relies on both types of authentication failures to succeed.

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:[#]

(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 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.

3.2.1. 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.[**] 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

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

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

DOI: 10.1002/sec

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 $\mathrm{ID}_N \oplus K < \delta$ 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_i, where $i = K \bmod n$. Upon receiving this reply, the querier will send a KADEMLIA_REQ to the malicious colluder M_i to find more replica roots and to confirm that it is alive. The colluder M_i receives the KADEMLIA_REQ and finds $i = K \bmod n$, 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_i, who proceeds as described above.

3.2.2. `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 s 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 3 s, Q will try one more every second. Finally, after another 22 s, Q will stop trying more contacts. The attack may fail if Q finds an honest replica root before it receives the reply from M.

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. The Distributed Virtual Network (DVN) Simulator

Due to their distributed nature, accurate analytical modeling of peer-to-peer systems such as the Kad network has proven to be challenging. As a result, simulation has frequently been employed, but general purpose simulation has presented several challenges, leading many groups to develop their own ad-hoc simulators. Typical simulation strategies fall into two broad groups—high-level and low-level simulators. High-level simulators can scale up to hundreds of thousands of nodes, but typically test only a high-level description of the design. Low-level simulations may test working code by running several instances on a single machine; these simulations produce results that are faithful to the implementation but are typically limited to a thousand or fewer nodes per machine. Thus the typical choices involve a trade-off between the scalability and fidelity of a simulation. While DVN was developed with simulation of the Kad Network in mind, it was designed as a general purpose packet-based simulator providing the fidelity of low-level simulators while allowing large simulations on the order afforded by high-level simulators.

In the evaluation of the attacks on the Kad network, it was important to validate our attacks without disrupting the actual network. We wanted to use the real C and C++ Kad code in our simulation and scale to large networks. To this end, we used our high fidelity distributed DVN simulator to simultaneously run multiple modules implementing different protocol stacks including code from the real Kad implementation and the modified Kad attacker nodes.

We also compared DVN with WiDS [21], the state-of-art high-fidelity network simulator that shares numerous design characteristics with DVN. Our simulator DVN surpasses WiDS in terms of simulation time and memory usage.

4.1. Design Requirements

A simulation platform should be able to provide the following

4.1.1. Scalability

The simulator must allow experiments consisting of a large number of nodes and messages. There should

DOI: 10.1002/sec

not be any hard limits imposed, instead, the simulator should only be constrained by resource limits. Specifically, as long as there is memory available, the simulator should be able to instantiate new nodes and model packets flows in the network. To support the goal of simulating a million-node deployment, the simulator would benefit from distributed computations in order to take advantage of multiple machines when the resources required exceed the capacity of a single one.

4.1.2. Fidelity

The architecture should be able to run code that is very close to the actual implementation to minimize risks of mutation therefore bug introduction when the actual code is implemented. Moreover, the simulator should allow code from real implementations to be ported from current active projects to run on it, thereby allowing accurate modeling of the actual network. The porting effort should be significantly smaller than the re-implementation effort. The code designed for the simulator should also be easily 'exported' so that it can be used on a real implementation. Additionally, the simulator should provide a means to support the following secondary goals:

Meaningful Network Model. The architecture should provide support for realistic network conditions, encountered by large deployments such as non-transitive connectivity and network partitions.

Event scheduling. In order to produce replicable experiments, the simulator should support scheduling of a series of events—such as node addition, deletion, network merge or partitioning—at predetermined times.

4.1.3. Node diversity

The architecture should support nodes running multiple versions, or with different settings, to model situations like incrementally deployed upgrades, networks with 'super peers', or the effects of malicious nodes on a network simultaneously.

4.1.4. Portability

The simulation platform should be flexible enough to run on multiple OSes, thus allowing developers and researchers a choice of platform better suited to their needs.

4.2. DVN Architecture

DVN is a discrete event network simulator that offers a scalable model for large networks with easily portable modules for high fidelity and a scripting language supporting heterogeneous nodes to run repeatable experiments while facilitating the isolation of specific variables affecting a network by tweaking individual parameters. It can run in distributed mode, dividing the virtual nodes evenly on different processes or even separate machines. Fidelity is at the heart of the design of DVN. It supports C/C++ libraries that can be built from real implementations. Communication protocol implementations such as eMule for Kad, need to be ported into library modules compliant to its Simple Network Routing Interface, SNRI. From there, cross compiling a DVN module library into a production application is done by using the SNRI bridge to a real UDP/IP stack. A protocol implementation written natively using SNRI can be crossed compiled into real applications in a single step as well. An illustration of the architecture is presented in Figure 3(a). The various components of DVN are described next.

4.2.1. Core engine

The core of DVN is the event scheduler. It has a tunable time slot granularity, typically set to one millisecond, thus all events are approximated to the nearest millisecond. Inserting an event is done by finding the appropriate time slot using a hash table. If there are multiple time slots in a single hash table bucket, a minimum heap is used to identify the correct one. Finally, a simple dynamic array is used to keep track of all concurrent events. DVN uses another minimum heap to keep track of all the time slots, allowing for retrieval of the next time slot in the order of $O(\log n)$ for n time slots. The overall structure is shown in Figure 3(b). For large simulations, DVN can operate in distributed mode. The DVN master of a distributed simulation is responsible for assigning virtual nodes to actual DVN worker instances, each with their own event scheduler. It then distributes the events for relevant virtual nodes to the assigned DVN workers through the Simulation Distributor. Each worker keeps track of local events and communicate events that affect nodes on remote workers using the underlying UDP/IP network. The master is responsible for orchestrating the beat by sending the start of each simulated time slot and waiting for all workers to complete before starting the next time slot. Each DVN worker reports the next closest time slot that the master should propagate to the other workers. Each DVN worker instance can load modules that contain the implementation of the protocol in the form of a module library. Multiple versions of the implementation can exist for the simulation of a heterogeneous network.

DOI: 10.1002/sec

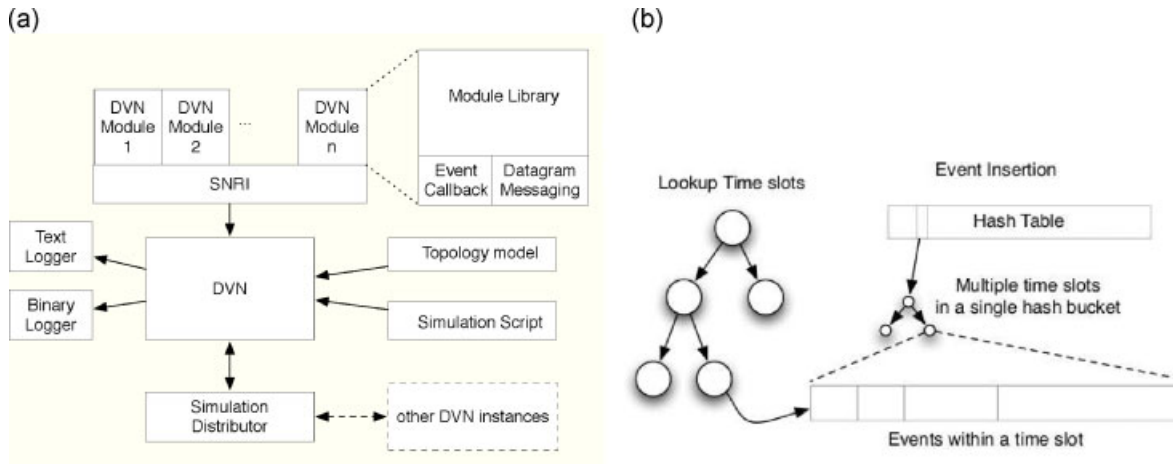

Fig. 3. (a) Architecture of DVN and (b) DVN's event scheduler.

4.2.2. DVN modules

Creating a module from the ground up is simple. A protocol implementor can use the UDP-like datagram messaging system to send packets to remote nodes. To implement timeouts, the callback system is available. A packet arriving for a given node will trigger the incoming datagram callback function to allow the node to process it. Global variables within a module can be safely used as long as they are registered *via* SNRI, which will then allow DVN to track those variables as part of the current node's state and swap those in prior to running the node's logic. Because DVN can load more than one module in a given simulation, it allows a user to create a heterogeneous network with multiple variants of a protocol, or completely different protocols for that matter. This feature is an important tool for research in P2P security. For example, it allows a user to measure the impact of an attack on a network that has stabilized, as well as measure the effectiveness of countermeasures to the attack.

4.2.3. Simple network routing interface (SNRI)

SNRI is the interface we created that defines a simple set of operations both inbound and outbound on any module that is to be used with DVN. It allows the protocol to set callback events or send datagrams to a remote host. When an event is triggered or a message received, that layer makes the appropriate calls into the module library. SNRI was designed to be simple to enable rapid development and testing of a protocol implementation. Using a lightweight wrapper, the protocol code could then be deployed against a real network stack. This allows for the identical code that would otherwise run on a real network to be tested within the virtual network offered by DVN.

4.2.4. Simulation description language (DSIM)

To aid in the simulation setup, DVN provides a scripting language called DSIM, which is based around Flex [22]. DSIM is a simple event-based language used to model simulations within DVN. It describes all components involved in a DVN simulation: network topology, node instantiation, network events, and simulation timing. A simple language construct offering both variable assignment and a miniature yet robust functional set allows DVN simulations to be made complex while their expression and description remain simple. The DSIM options are parsed by Flex and interpreted by DVN. These configuration options are the primary way that users interact with DVN. The commands available range from starting nodes to specifying network topologies and characteristics. Scripting commands are available to allocate and introduce nodes into the network by dynamically loading modules described in the porting process above. All calls are time dependent and the simulation can therefore alter network topology at any time to simulate large-scale network events.

4.2.5. Logging

DVN provides a central logging mechanism that can save events in ASCII text format or binary format. Individual modules can log events based on their

DOI: 10.1002/sec

internal states, or DVN can log events from the simulation itself. The statistics collection methods are left to the module developers in order to not limit the type, frequency, and amount of data meaningful to the analysis of the simulated network.

4.2.6. Network topology

DVN only allows modeling of the underlying network by specifying the network delay, reliability, and variability. Separated networks can be created at any time during the simulation and those networks can be connected at anytime as well. The interface layer between the module and DVN can be used to model arbitrary network behavior. Since DVN models overlay networks which typically reside above the OSI transport layer, the topology model supported by the DSIM language is fairly simple. It provides the ability to build subnets with independent characteristics, and allows interconnections of those subnets with links whose characteristics can be independently specified as well. This model provides an adequate abstraction to avoid burdening the user with specifying all possible links in the system.

4.3. Porting Kad on DVN

We used aMule v2.1.3 and isolated the Kad protocol implementation by removing the graphical user interface and replacing system calls made to a UDP stack with DVN functions to send and receive packets. Since both of these network functions operate on basic data structures, the porting was relatively simple. All references to system time were also changed to references to virtual DVN time. The memory management of Kad was done using static and global variables. This was changed to reside within a single structure that is allocated at initialization time.

4.4. Evaluation

4.4.1. Scalability

Our test platform included a DELL PowerEgde 6950 with four dual-core AMD 8216 (2.4 GHz) CPUs and 16 GB of RAM. We ran DVN with the Kad module, logging output serially to a file. We generated DVN simulation (DSIM) files for 1000, 5000, 10 000, and 20 000 nodes. Each of the simulations was on a single simulated network with 250 ms latency for message delivery, and no dropped messages. The bootstrap set consisted of 10 interconnected nodes. Then the rest of the nodes were added in batches of 300 nodes at a time, using any of the nodes in the aforementioned set as a bootstrap node. We were able to successfully perform simulations of 200 000 Kad nodes on this machine. Maximum memory usage during this simulation was 10 GB or 50 KB per node. We were able to simulate 14 DVN hours in 77.4 h—a slowdown of 5.5 times. While this is slower than realtime, it is expected for such a large simulation.

The output logs were divided into chunks of 100 DVN seconds. The number of messages sent in each 100-s window was recorded and plotted as the simulation progressed, as shown in Figure 4 (left figure). These messages include Hello, Kademlia and Bootstrap messages and are used for discovering the network. The initial spike is due to the bootstrapping nodes discovering the network. The periodic waves are due to various routing table maintenance messages. We also measured the amount of time required to process each of those 100 DVN second time slots and

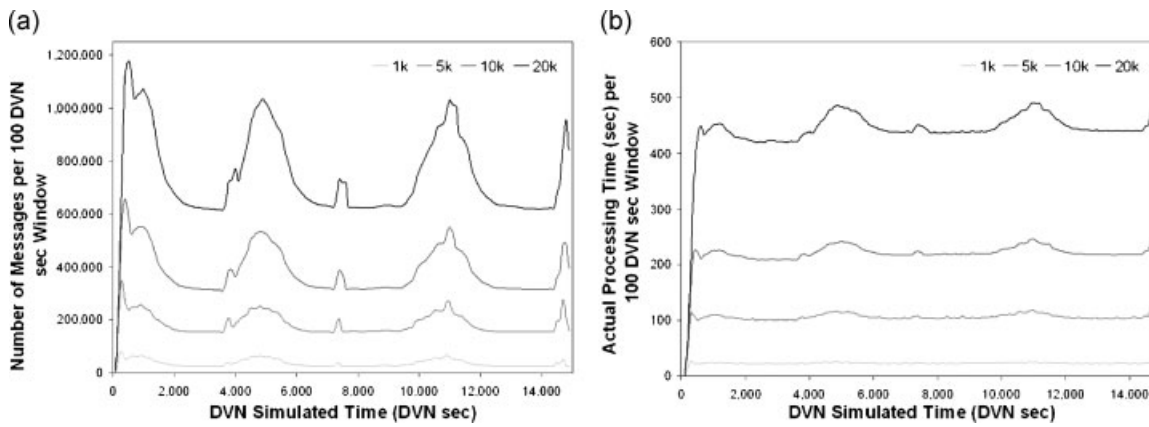

Fig. 4. DVN performance for 1, 5, 10, and 20 thousand Kad nodes.

DOI: 10.1002/sec

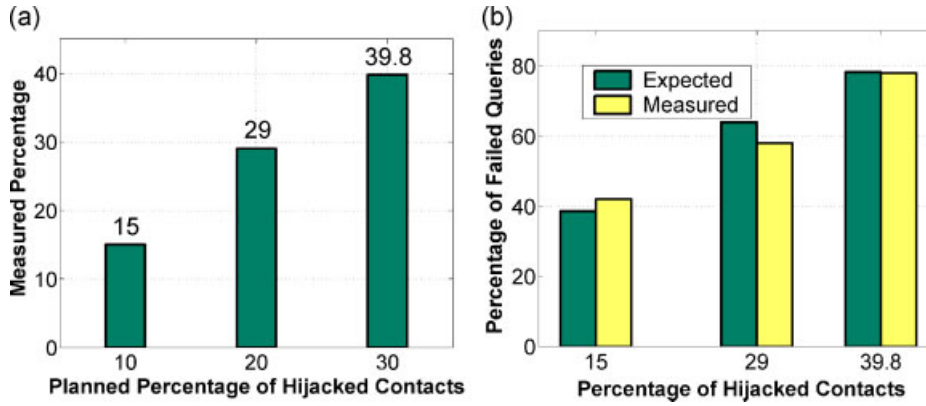

Fig. 5. Attack technique validation: (a) hijacking back-pointers and (b) attacking keyword queries.

plotted them as the simulation progresses as shown on the graph on the right. It is clear that the processing time is directly proportional to the number of messages being transmitted within the system. The number of messages is directly proportional to the number of nodes being simulated. Message transmission events are visible only when large traffic spikes are observed and those become the dominant factor affecting network performance.

The overhead incurred by the DVN simulator is relatively small compared to the resource consumption of the actual Kad nodes. The actual DVN architecture requires less than 30 MB. The rest of DVN's memory consumption depends on the network traffic, since the scheduler will need to hold on to the packets until delivery. Therefore, the memory requirement is dependent on the number of packets and the size of those packets. In our overhead estimation experiments, we ran between 1000 and 1500 nodes at 100 nodes increase per experiment. The memory consumption on the heap was noted for each experiment. We were able to estimate that the memory consumption for each Kad node on a small simulation was about 165 KB. The complete resulting overhead of DVN including memory used to hold all messages during transit was found to be less than 30%.

4.4.2. aMule on DVN

To verify the correctness of DVN, we compared the network behavior of the Kad nodes running on DVN and the ones running in a real network. In this experiment, we ran 3000 virtual nodes for 2 h on DVN and in parallel, 3000 Kad nodes on a separate testbed (call it itlabs), consisting of 14 actual machines. Figure 5 shows the number of lookup messages[††] sent in 1 min window. The DVN Kad nodes and itlabs Kad nodes exhibit the same behavior.

We also compare the memory footprint between running a Kad node on a normal machine and running it on DVN. 1000 nodes are created in DVN and run for 30 DVN minutes. At the peak of the traffic (Figure 4) at time 200 s, the virtual memory usage for DVN was 43.1 MB. The same experiment was performed, deploying 1000 Kad nodes on a single machine and letting them run for 30 min. At the peak of the traffic, each Kad node was using about 1.5 MB. Thus, 1000 Kad nodes deployed on a single machine will use 1.5 GB of virtual memory.

4.4.3. Symmetric links

A symmetric link is where node *A* has node *B* in its routing table and node *B* also has *A* in its routing table. We want to know the percentage of symmetric links in a Kad node's routing table. This allows us to determine whether the Kad network is symmetric or asymmetric. We expect that the Kad network, due to its design, would be mostly symmetric, because each node would be added in the same bucket at the appropriate level of the routing table.

Finding the number of symmetric links in a routing table involved having access to both a node, its routing table, and the nodes in the routing table. This would be hard to evaluate on the real network as we did not control all the nodes; our real deployment is limited to 16 000 nodes. Crawling the whole Kad network

[††]These lookup messages are used to maintain nodes routing tables.

DOI: 10.1002/sec

and polling every node's routing table is possible but would involve a substantial bandwidth cost and would not result in an instantaneous snapshot of the network, since it would take on the order of 1 h at 100 Mbps to poll every node's routing table. By that time, the earlier nodes that were polled would have had their routing tables changed due to churn. Moreover, deriving the percentage of symmetric links analytically is difficult due to its dependence on network dynamics, such as churn. Thus, simulation was the most feasible route. We found that the percentage of symmetric links in our simulated environment is 53%. Thus, for a particular node A, half of its routing table entries also have A in their routing table.

5. Attack Evaluation

We evaluated the effectiveness and bandwidth cost of our attack by launching 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.

5.1. 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.[‡‡] Figure 5(a) shows the experiment result where p is set to 10, 20, and 30%. The measured percentage is computed as $f = \frac{h}{c}$, 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 $1 - f^3$, at least one hijacked contact should be used in a query. In the experiment, we input a list of 50 keywords[§§] to Q and count the number of failed queries. Figure 5(b) shows that the result is close to our expectation.

5.2. Bandwidth Usage

In our attack, bandwidth is used for three tasks: hijacking back-pointers, 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 $\log(1\,000\,000) - 5 \approx 15$ additional levels.

5.2.1. Hijacking back-pointers

Suppose an attacker wants to stop fraction g of the queries of a victim, then it should hijack $p = \sqrt[3]{1-g}$ 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 $86 \times 10 \times p = 860 \times p$ HELLO_REQ messages to hijack p fraction of backpointers in a routing table.[‖‖] Therefore, in the preparation phase, the number of messages and the bandwidth cost to attack g fraction of queries sent by n Kad nodes are

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

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

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

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

[§§]The list includes popular movies, songs, singers, software, filename extensions, etc.

[‖‖]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 back-pointers, since high level contacts are used more often in queries. As a special example, on average, $\frac{11}{16} = 68.75\%$ 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.

DOI: 10.1002/sec

5.2.2. 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 2 h if the contact is in the routing table for more than 2 h. 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

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

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

5.2.3. 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

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

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

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

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 1-h period and the average was computed. This experiment ran for 24 h. The 1-h period with the highest average number of queries resulted in 405 queries per host.[¶¶] 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.

5.3. Large Scale PlanetLab Experiment

In this experiment, we use about 500 PlanetLab [23] 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.

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 2 h, 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 s, and exits 5 min 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 h, then attacks every keyword query. The attack run for 1 h (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 6(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

[¶¶] 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.

DOI: 10.1002/sec

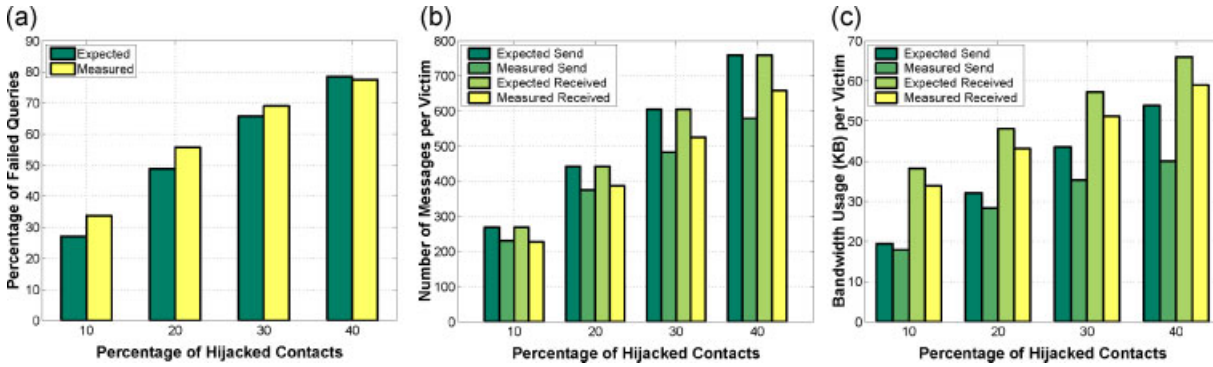

Fig. 6. Large scale attack simulation: 11 303–16 105 victims and 50 attackers. In (b) and (c) the numbers of messages and bandwidth costs are normalized based on the number of victims in each experiment: (a) query fail rate, (b) number of messages and (c) bandwidth.

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 6(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 shown in Figure 7. 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 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.

5.4. Large Scale Simulation

We performed large-scale simulations on DVN of a variant of our attack with 50 000 nodes. Instead of attacking the keyword search lookup process, we attacked the routing process. Each of our nodes performed a node lookup for a random Kad ID

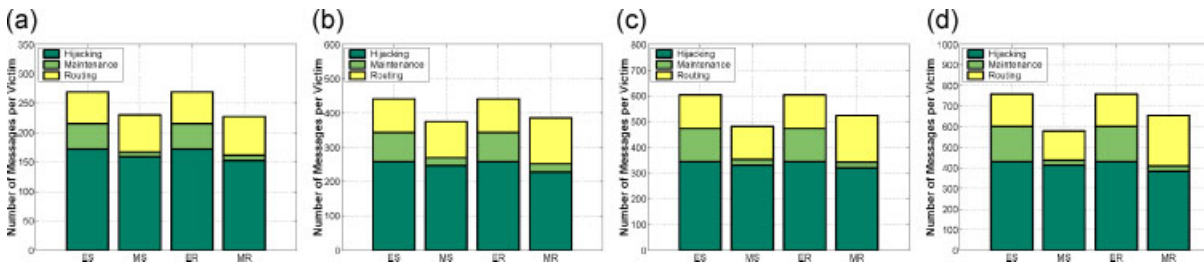

Fig. 7. Number of messages in detail: ES, MS, ER, and MR stand for expected sent, measured sent, expected received, and measured received, respectively. The numbers of messages are normalized based on the number of victims in each experiment: (a) 10%; (b) 20%; (c) 30%; (d) 40%.

DOI: 10.1002/sec

every 10 min. We then performed an attack similar to the keyword search attack, but focusing instead on the control plane. If our attacker node receives a KADEMLIA_REQ, it replies back with 10 Kad IDs that are very close to the target ID (target+1,target+2,. . . ,target+10). These 10 results all have the attacker's IP address.

We did not perform this attack on the real Kad network since we did not want to hijack real Kad nodes. Moreover, an experiment on PlanetLab would be limited to a network size of 16 000 nodes. We were able to scale to a much bigger network using DVN. We were also able to show the full capability of our attack, which focuses on the control plane, rather than the keyword search attack which is on the data plane. Attacking the control plane (routing requests) is more potent since the keyword search (data plane) is dependent on successful routing. An attacker with modest resources can bring down the whole Kad network using this routing attack. Finally, we were able to simulate churn. The churn model is based on Reference [5]. We believe that this churn model is accurate since the measurements were performed on the real Kad network.

In our simulation, we join 50 000 nodes and 50 attacker nodes. After 15 simulated hours, the attacker nodes start to attack the whole network by polling every node's routing table and hijacking 15% of it. We evaluated the percentage of routing requests that failed. We also evaluated the percentage of each node's routing table which has been hijacked. Figure 8 shows the result. As expected, the failure rate increases when the attack starts. With 15% of the routing table hijacked, we expect the probability of choosing a non-hijacked contact from our routing table for the first hop to be 0.85^3. With an average of three hops for each node ID lookup, the probability of contacting at least one attacker node is $(1-(0.85^3)^3) = 77\%$. However, each victim node has a 15% chance of returning at least one attacker node. This increases the overall probability of a victim node contacting an attacker node to more than 90%. This is reflected in our result. Over time, contacts from hijacked search results will begin to populate the victims' routing tables. This has the net effect of increasing the overall hijacked percentage. Our attacker nodes stay online indefinitely, thus responding to all requests. The initial hijacking spreads to other nodes because the hijacked victim nodes will occasionally reply to KADEMLIA_REQ with the attacker's IP address.

With only 0.1% of the network being malicious, and hijacking only 15% of every node's routing table, almost every routing request is hijacked. This effectively brings the whole Kad network down as without successful routing, a P2P network is not able to function. We want to stress that the crawling, polling, and hijacking can be performed in parallel and over time, that is, every node's routing table does not have to be hijacked at the same time. This reduces the load on the attacker nodes. Moreover, the spread of the initial hijacking spreads over time. Thus, an even smaller percentage of initial hijacking can be performed and as long as the malicious nodes stay online, other nodes will get 'hijacked' as well. We only show simulation results with 15% of initial routing table hijacking, but the same result was obtained with varying hijacking percentages. We were able to only perform simulations with 50 000 nodes due to the simulation running six times slower than real time.

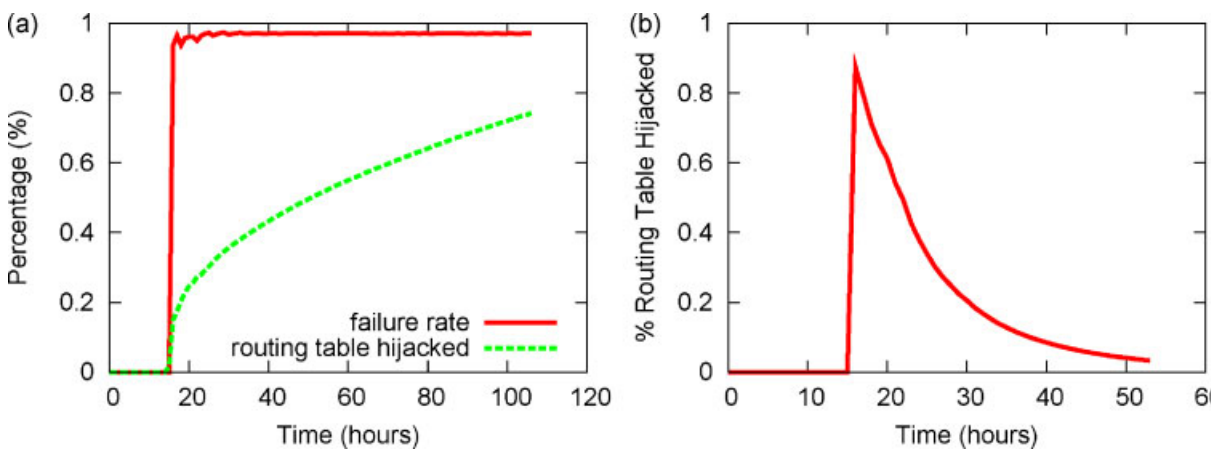

Fig. 8. (a) Control-plane attack with 50 000 nodes and initial routing table hijacking of 15% and (b) reflection attack with 50 000 nodes indicating % routing table hijacked.

DOI: 10.1002/sec

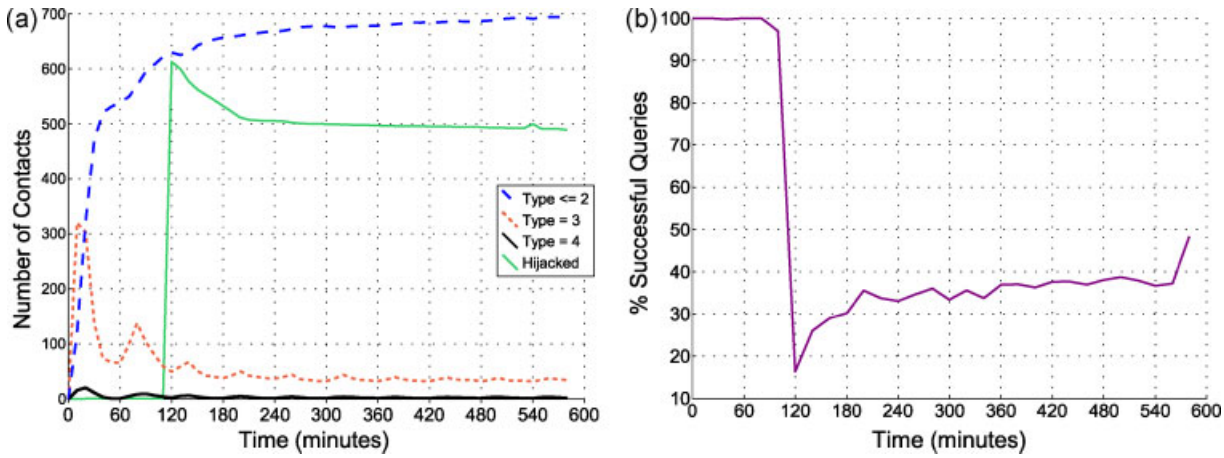

Fig. 9. (a) Average hijacked and total contacts over time and (b) % successful queries, over 20-min windows.

6. Reflection Attack

The major disadvantage of 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.[***] 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 will 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.

6.1. Reflection Experiment

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 three machines, each victim node bootstrapping from a different node in the real Kad network. Once bootstrapping is complete and after waiting for 5 min, each victim will send a HELLO_REQ to the attacker node. After waiting 2 h (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 min. Since the victim nodes are connected to the real Kad network we did not hijack backpointers to the victims. It should be the case that our experiment overestimates the rate of recovery.

Figure 9(a) 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 h,

[***] 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.

DOI: 10.1002/sec

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 h, 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 over the course of the attack. Starting 6 min after bootstrapping, each victim sent a query to a randomly chosen key once every 3 min, and recorded whether it successfully located a replica root for the key. Figure 9(b) shows the results of this experiment. 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 1 h, with very little subsequent bandwidth usage.

6.2. Reflection Simulation

Using DVN, we ran a simulation of our reflection attack with 50 000 nodes. At time 15 h all nodes which are online (about 50% of the network—the rest of the network has churned out) have their routing tables completely hijacked, such that all entries contain their own IP address. As the non-attacked nodes churn back in, they will fix these entries since the attacker nodes leave the network after the initial hijacking. The routing table rate of recovery for those attacked nodes is shown in Figure 8(b).

The rate of recovery is faster than our experiments because at the time the attack took place, only 25 000 nodes (out of 50 000 nodes) were online and were attacked. The rest of the network was not attacked. When those nodes come back online, they will start sending a HELLO_REQ to the nodes in its routing table, so that it can determine the liveness of the entries in its routing table. Thus, the routing table entries will get fixed. In our experiments from Section 6.1, the real Kad nodes are most likely still alive and will not send a HELLO_REQ to the victim nodes, thus the routing table takes longer to fix. On average, each node will send a HELLO_REQ to an entry in its routing table every 4 h. Since 50% of the routing tables in the Kad network are symmetric (Section 4.4.3, churn helps the network to recover. This is why the rate of recovery in our simulation is faster than for our experiments. Moreover, the routing tables of those nodes that were originally attacked but churned out, are still hijacked and will take much longer to be repaired, since those nodes are no longer in other peers' routing tables.

7. 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.

7.1. 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 2 h, *S* keeps a list of the nodes that have sent it a KADEMLIA_REQ or HELLO_REQ message in the past 2 h. 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 10(a), we see that, on average, a Sybil node collects about 500 back-pointers after 24 h, and about 1400 back-pointers after 1 week (168 h). 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.

DOI: 10.1002/sec

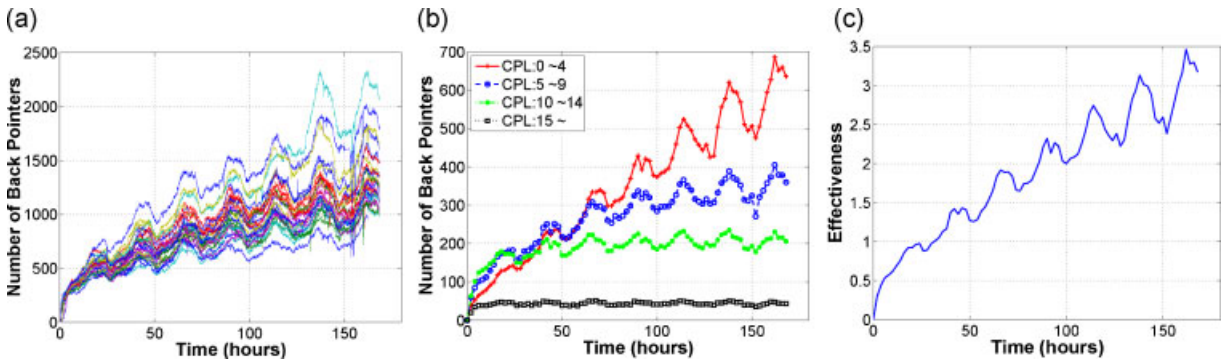

Fig. 10. Sybil attack measurement: 28 Sybil nodes run for 1 week. (a) Total: it shows the total number of back-pointers. One line represents one node. (b) Average, clustered by CPL: it shows the average number of back-pointers clustered by the common prefix length (CPL) between the Sybil node's ID and the back-pointer's ID. (c) Effectiveness: it shows the average of Sybil nodes' effectiveness computed with formula (9).

Figure 10(b) shows that, the number of back-pointers with CPL ≥ 15 quickly becomes stable at approximately 50. After 40 h, 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 10(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.[†††] 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 11×10 4th level back-pointers and 5×10 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, $i \in [1, m]$:

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

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

Figure 10(b) shows that the effectiveness of a Sybil node reaches 1 after approximately 24 h. Then, the effectiveness increases linearly and reaches 3.5 after 162 h (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.

7.2. Index Poisoning Attack

In the index poisoning attack [18], 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 there is a bogus finding for every legitimate 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.

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

DOI: 10.1002/sec

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-h 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 h, 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.

8. 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.

8.1. 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 $\langle \mathrm{ID}_B, \mathrm{IP}_M, \mathrm{port}_M \rangle$. 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 $\langle \mathrm{IP}_B, \mathrm{port}_B \rangle$ 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 $\mathrm{hash}(\mathrm{IP}_B)$ as its node ID.[‡‡‡] Here every ID is tied with the corresponding IP 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

[‡‡‡]Several alternatives are possible: the 64 MSBs can be derived from $\mathrm{hash}(\mathrm{IP}_B)$ and the 64 LSBs from $\mathrm{hash}(\mathrm{IP}_B \| \mathrm{port}_B)$ to support NAT; if subnet-level attackers are a concern the 64 MSBs can be derived from $\mathrm{hash}(\mathrm{IP}_B/24)$; etc.

DOI: 10.1002/sec

Table I. 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

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 I 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.

8.2. 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.

9. Recent Changes in Kad

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

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

DOI: 10.1002/sec

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 one KADEMLIA_REQ per IP address every 6 s. 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 one request per 6 s at any individual node. Also, 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 6. 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 8—'Drop Hello with new IP/Port'). Since this variable is currently set to true (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 Kad2 client and successfully hijacking a single backpointer. The clients also implement *Protocol Obfuscation* [24] 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 Kad clients implement several features which could be used in future versions to mitigate our attack. However, that version only slightly increases the cost of our attack. We still need only one 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 one 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 9)—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. However, the latest version of the eMule clients (version 0.49b and 0.49c) do implement a mitigation for our attacks, after some discussions with the developers of eMule.

10. Related Work

Since Kademlia [1] was introduced in 2001, several variations have been implemented, including the discontinued Overnet and eDonkey2000 projects, and also the separate eMule [25], aMule [26], and MLDonkey projects. Kademlia is in use by several popular BitTorrent clients as a distributed tracker [27,28]. Because Kad seems to be the largest deployed DHT, several studies have measured various properties of the network. Steiner *et al*. [4] 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 [3] and churn characteristics [5] of the deployed Kad network. None of these works address the security of Kad.

Sit and Morris [10] present a taxonomy of attacks on DHTs and applications built on them. They further provide design principles to prevent them. The Sybil attack has been studied by several groups [9,29]. Three Sybil-resistant schemes based on social links were recently proposed in References [30–32]. Castro *et al*. [11] design a framework for secure DHT routing which consists of secure ID generation, secure routing table maintenance, and secure message forwarding. Fiat and Saia [14] 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*. [17] 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*. [19] 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

DOI: 10.1002/sec

periodic anonymous distributed auditing. Condie *et al*. [33] induce churn to mitigate eclipse attacks. Naoumov *et al*. [34] propose to exploit Overnet as a DDoS engine with index poisoning and the generic routing table poisoning. El Defrawy *et al*. [35] propose 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.

Parallel and distributed simulators based on discrete events have been developed previously [36–38] and Lin *et al*. [39] observe that the core of the aforementioned simulators is divided into two categories: conservative and optimistic. They report that in the conservative engines, the logical time is advanced in a coordinated fashion, which is the same technique used by DVN to guarantee the chronological order of events. DVN uses a master-worker relationship similar to WiDS [39] physical network with synchronization messages [40]. WiDS has shown valuable support for developing a distributed protocol implementation and move it to the real world. DVN was built to support the reverse path where code released in the real world was ported to a DVN module while minimizing the porting effort. In the WiDS toolkit [21], the authors introduce Slow Message Relaxation (SMR) as a tradeoff between performance and accuracy. Given DVN's performance on a single machine, there was no need for such a tradeoff and although tunable, the time granularity was fixed at the millisecond level.

We wanted to minimize the porting effort to the simulation platform to minimize the possibility of bug introduction and maximizing fidelity. In that respect, the porting effort and scalability issues in running the Kad node on NS-2 [41] would not have been practical. Using PDNS—Parallel/Distributed NS [42] for scalability would have been possible, but it was not clear that the performance would have been acceptable for very large simulations in distributed mode. NS-3 [43] pays close attention to realism where each node is a computer's outer shell and hosts a complete communication stack. In our experiments, we only needed the top layers of the stack to support the application layer where the overlay routing takes place. Therefore, we optimized DVN by trimming out the layers under the IP network layer. ModelNet [44] allows researchers to run unmodified software prototypes and supports a finer granularity of the network topology than the description supported by DVN's dsim language but doesn't scale to the sizes we needed. The SSFNet simulator [45] is an infrastructure built in Java that can support nodes written in Java and C++ compliant to their API bindings. The largest documented simulation containted up to 384 000 nodes [46]. However, the authors mention that their model used approximations of the worm infection patterns to generalize the simulation at a coarse level, while only simulating parts of the network in detail. In our simulation, we wanted to run a full protocol implementation for the entire network with hundreds of thousands of nodes for a higher fidelity.

MACE [47] is a compiler that outputs C++ source code from protocol specifications. The output generated could then be compiled into a DVN module once the event callbacks and datagram network interface are put in place. Haeberlen *et al*. [48] suggests that current simulation and experiments are considering single points in the entire possibility space. Naicken *et al*. [49] analyze six simulators [50–58] along the some of the criteria mentioned in the introduction. DVN differs from by stripping the communication stack as much as possible to reduce the overhead and focus on the DHT overlay at the application laye for high fidelity, while presenting a realistic network layer.

11. 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 security updates to Kad, we have shown that our attack still works using nearly the same resources. However, an easy change to the code prevents hijacking attacks in the latest version. We have also introduced a novel large scale efficient and high fidelity simulator, DVN, which allowed us to simulate 200 000 nodes and other attacks which would have been harder to measure using the real network (without impacting the whole network), such as the control-plane routing attack.

Acknowledgements

We are grateful to Hendrik Breitkreuz for helpful discussions about the latest versions of the Kad client.

DOI: 10.1002/sec

This work was partly funded by the NSF under grant CNS-0716025, and KISA (Korea Information Security Agency).

References

1. Maymounkov P, Mazíeres D. Kademlia: a peer-to-peer information system based on the XOR metric. In *International Workshop on Peer-to-Peer Systems*, 2001.
2. eDonkey Network. Available at: http://www.edonkey2000.com
3. Stutzbach D, Rejaie R. Improving lookup performance over a widely-deployed DHT. In *INFOCOM*, 2006.
4. Steiner M, Biersack EW, En-Najjary T. Actively monitoring peers in KAD. In *International Workshop on Peer-to-Peer Systems*, 2007.
5. Stutzbach D, Rejaie R. Understanding churn in peer-to-peer networks. In *ACM SIGCOMM Conference on Internet measurement*, 2006.
6. Steiner M, En-Najjary T, Biersack EW. A global view of kad. In *ACM SIGCOMM Conference on Internet measurement*, New York, NY, USA, 2007; 117–122 (ACM).
7. Steiner M, En Najjary T, Biersack EW. Analyzing peer behavior in KAD. *Technical Report EURECOM+2358*, Institut Eurecom, France, October 2007.
8. Steiner M, Effelsberg W, En Najjary T, Biersack EW. Load reduction in the KAD peer-to-peer system. In *DBISP2P 2007, 5th International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, Vienna, Austria, 24 September 2007.
9. Douceur JR. The Sybil attack. In *International Workshop on Peer-to-Peer Systems*, 2002.
10. Sit E, Morris R. Security considerations for peer-to-peer distributed hash tables. In *International Workshop on Peer-to-Peer Systems*, 2002.
11. Castro M, Druschel P, Ganesh A, Rowstron A, Wallach DS. Secure routing for structured peer-to-peer overlay networks. In *Operating Systems Design and Implementation*, 2002.
12. Lynch N, Malkhi D, Ratajczak D. Atomic data access in content addressable networks. In *International Workshop on Peer-to-Peer Systems*, 2002.
13. Fiat A, Saia J. Censorship resistant peer-to-peer content addressable networks. In *ACM-SIAM Symposium on Discrete Algorithms*, 2002.
14. Saia J, Fiat A, Gribble S, Karlin A, Saroiu S. Dynamically fault-tolerant content addressable networks. In *International Workshop on Peer-to-Peer Systems*, 2002.
15. Hildrum K, Kubiatowicz J. Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In *International Symposium on Distributed Computing*, 2003.
16. Singh A, Castro M, Druschel P, Rowstron A. Defending against eclipse attacks on overlay networks. In *ACM SIGOPS European Workshop*, 2004.
17. Fiat A, Saia J, Young M. Making chord robust to byzantine attacks. In *European Symposium on Algorithms*, 2005.
18. Liang J, Kumar R, Xi Y, Ross KW. Pollution in P2P file sharing systems. In *INFOCOM*, 2005.
19. Singh A, Ngan T-W , Druschel P, Wallach DS. Eclipse attacks on overlay networks: threats and defenses. In *INFOCOM*, 2006.
20. eMule ChangeLog. Available at: http://www.emule-project.net/home/perl/news.cgi?l=1
21. Lin S, Pan A, Zhang Z, Guo R, Guo Z. WiDS: an integrated toolkit for distributed system development. In *Hot Topics in Operating Systems*, 2005.
22. Flex. Available at: http://flex.sourceforge.net/
23. Chun B, Culler D, Roscoe T, *et al*. PlanetLab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review* 2003; 3–12.
24. Protocol Obfuscation. Available at: http://www.emule-project.net/home/perl/help.cgi?l=1&rm=show_topic&topic_id=848
25. eMule Network. Available at: http://www.emule-project.net
26. aMule Network. Available at: http://www.amule.org
27. Azureus. Available at: http://azureus.sourceforge.net
28. Mainline. Available at: http://www.bittorrent.com
29. Friedman E, Resnick P. The social cost of cheap pseudonyms. *Journal of Economics and Management Strategy*, 2001.
30. Marti S, Ganesan P, Garcia-Molina H. DHT routing using social links. In *International Workshop on Peer-to-Peer Systems*, 2004.
31. Danezis G, Lesniewski-Laas C, Kaashoek MF, Anderson R. Sybil resistant DHT routing. In *European Symposium on Research in Computer Security*, 2005.
32. Yu H, Gibbons PB, Kaminsky M, Xiao F. SybilLimit: a near-optimal social network defense against Sybil attacks. In *2008 IEEE Symposium on Security and Privacy*, 2008.
33. Condie T, Kacholia V, Sankararaman S, Hellerstein J, Maniatis P. Induced churn as shelter from routing table poisoning. In *Network and Distributed System Security Symposium*, 2006.
34. Naoumov N, Ross K. Exploiting P2P systems for DDoS attacks. In *International Conference on Scalable information systems (InfoScale*, 2006.
35. El Defrawy K, Gjoka M, Markopoulou A. Bottorrent: misusing BitTorrent to launch DDoS attacks. In *Usenix SRUTI*, June 2007.
36. Riley G, Fujimoto RM, Ammar M. A generic framework for parallelization of network simulations. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 1999.
37. Cowie J, Liu H, Liu J, Nicol D, Ogielski A. Towards realistic million-node internet simulations. In *International Conference on Parallel and Distributed Processing Tech- niques and Applications (PDPTA)*, 1999.
38. Fujimoto RM, Perumalla K, Park A, Wu H, Ammar MH, Riley GF. Large-scale network simulation: how big? how fast? In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2003.
39. Lin S, Pan A, Guo R, Zhang Z. Simulating large-scale p2p systems with the wids toolkit. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2005.
40. Riley GF, Fujimoto RM, Ammar MH. Network aware time management and event distribution. In *Workshop on Parallel and Distributed Simulation (PADS)*, 2000.
41. The Network Simulator—ns-2. Available at: http://www.isi.edu/nsnam/ns/
42. PDNS—Parallel/Distributed NS. Available at: http://www.cc.gatech.edu/computing/compass/pdns/index.html
43. The ns-3 Network Simulator. Available at: http://www.nsnam.org/
44. Vahdat A, Yocum K, Walsh K, *et al*. Scalability and accuracy in a large-scale network emulator. *SIGOPS Operating Systems Review* 2002; 271–284.
45. Scalable Simulation Framework, SSFNet. Available at: http://www.cc.gatech.edu/computing/compass/pdns/index.html
46. Liljenstam M, Nicol DM, Berk VH, Gray RS. Simulating realistic network worm traffic for worm warning system design and testing. In *WORM '03: Proceedings of the 2003 ACM Workshop on Rapid Malcode*, New York, NY, USA, 2003; 24–33 (ACM).
47. Killian CE, Anderson JW, Braud R, Jhala R, Vahdat AM. Mace: language support for building distributed systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.
48. Haeberlen A, Mislove A, Post A, Druschel P. Fallacies in evaluating decentralized systems. In *International Workshop on Peer-to-Peer Computing*, 2006.

DOI: 10.1002/sec

49. Naicken S, Basu A, Livingston B, Rodhetbhai S, Wakeman I. Towards yet another peer-to-peer simulator. In *International Working Conference on Performance Modelling and Evaluation of Heterogeneous Networks*, 2006.
50. P2PSim. Available at: http://pdos.csail.mit.edu/p2psim
51. PeerSim P2P Simulator. Available at: http://peersim.sourceforge.net
52. Schlosser MT, Kamvar SD. Simulating a file sharing p2p network. *Technical Report*, Stanford University, 2002.
53. Narses Network Simulator. Available at: http://sourceforge.net/projects/narses
54. NeuroGrid. Available at: http://www.neurogrid.net
55. Yang W, Abu-Ghazaleh N. GPS: a general peer-to-peer simulator and its use for modeling BitTorrent. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2005.
56. OverlayWeaver. Available at: http://overlayweaver.sourceforge.net
57. DHTSim. Available at: http://www.informatics.sussex.ac.uk/users/ianw/teach/dist-sys
58. PlanetSim: An Overlay Network Simulation Framework. Available at: http://planet.urv.es/planetsim

DOI: 10.1002/sec