Fighting Spam with the NeighborhoodWatch DHT

Adam Bender*, Rob Sherwood†, Derek Monner*, Nate Goergen‡, Neil Spring*, and Bobby Bhattacharjee*
* Department of Computer Science, University of Maryland
Email: {bender, dmonner, nspring, bobby}@cs.umd.edu
† Deutsche Telekom Inc, R&D Labs
Email: robert.sherwood@telekom.de
‡ Department of Electrical and Computer Engineering, University of Maryland
Email: goergen@umd.edu

***Abstract*—In this paper, we present DHTBL, an anti-spam blacklist built upon a novel secure distributed hash table (DHT). We show how DHTBL can be used to replace existing DNS-based blacklists (DNSBLs) of IP addresses of mail relays that forward spam. Implementing a blacklist on a DHT improves resilience to DoS attacks and secures message delivery, when compared to DNSBLs. However, due to the sensitive nature of the blacklist, storing the data in a peer-to-peer DHT would invite attackers to infiltrate the system. Typical DHTs can withstand fail-stop failures, but malicious nodes may provide incorrect routing information, refuse to return published items, or simply ignore certain queries. The NeighborhoodWatch DHT is resilient to malicious nodes and maintains the $O(\log N)$ bounds on routing table size and expected lookup time. NeighborhoodWatch depends on two assumptions in order to make these guarantees: (1) the existence of an on-line trusted authority that periodically contacts and issues signed certificates to each node, and (2) for every sequence of $k+1$ consecutive nodes in the ID space, at least one is alive and non-malicious. We show how NeighborhoodWatch maintains many of its security properties even when the second assumption is violated. Honest nodes in NeighborhoodWatch can detect malicious behavior and expel the responsible nodes from the DHT.**

I. Introduction

Tracking the IP addresses of known spam sources is a useful tactic in combating the increasing volume of spam email. Several databases, or *blacklists*, of spam-sending IP addresses—usually of a server running a mail transfer agent (MTA)—exist to help users and programs classify incoming email as legitimate or spam. Unlike the email address of the sender, the actual IP address from which an incoming email message was received is hard to spoof [1]. The address of any MTA that relays the message is prepended to the message's header. Blacklist operators collect the IP addresses of MTAs known to source spam and make these collections queriable via the Internet. Blacklists commonly allow users to access their database via DNS; such a database is called a DNSBL.

DNSBLs have several drawbacks. First, data delivery is not secured. DNS responses are weakly authenticated and are thus susceptible to misdirection, spoofing [2], and cache poisoning attacks [3]. The need for a secure DNS service is well-documented, yet the reluctance to adopt such measures as DNSSEC [4] indicates that this problem is likely to persist. Because DNSBLs are available only via DNS, DNSBL queries are vulnerable to the same attacks as a regular DNS query.

Second, spammers frequently target DNSBLs and other under-provisioned anti-spam efforts with denial of service (DoS) attacks [5], [6], [7], [8], [9]. Blue Security [10], creators of the Blue Frog software that automatically sent opt-out messages from all of its users whenever one of them received a spam email, was the target of DoS attacks in May 2006. The severity of the attacks forced Blue Security to shut down its service.

A natural response to DoS attacks is to distribute the service among many peers. For instance, the Okopipi project was an attempt (now defunct) to create a distributed implementation of Blue Frog. Not only would the blacklist service itself be able to distribute its lookup service, but ordinary users who wanted to contribute could do so by volunteering their resources. Storing a blacklist in a distributed hash table (DHT) such as Chord [11] would provide resilience to DoS attacks and is a natural choice because the simple interactions with a DNSBL exactly match the put/get interface of a DHT.

However, the sensitive nature of the data stored in such a DHT would invite attacks on the DHT itself. Not only would spammers attack individual nodes, they would attempt to join the DHT in order to subvert it. Most DHTs are not resilient to such Byzantine adversaries. The few DHTs designed in such an attack model are inefficient and complex, in that they occasionally rely on flooding or Byzantine agreement among a subset of the nodes. The DHT of Castro et al. [12] relies on *redundant routing*, which floods messages along multiple paths, while S-Chord [13] requires occasional flooding of $O(\log^2 N)$ messages.

We propose DHTBL, a secure, distributed implementation of a blacklist. DHTBL consists of two components: the NeighborhoodWatch secure DHT which stores the blacklist and a client that queries the DHT about individual IP addresses that forward mail. NeighborhoodWatch allows a small set of trusted nodes to administer a large set of untrusted nodes that provide secure content delivery. This design greatly increases the availability and capacity of the service without requiring that every node be trusted. We take advantage of the fact that blacklists are already centrally managed; we use this same authority to manage the nodes in a DHT. We make

[1]This work was partially supported by NSF awards CNS 0626629, ANI 0092806, and CNS 0435065, and NIH award NS35460. N.G. was supported by a DoD SMART Fellowship.

one additional assumption: in the flat identifier-space of the DHT, there is no sequence of $k+1$ consecutive nodes that are malicious, where k is a system parameter chosen by the blacklist administrator. That is, at least one in $k+1$ consecutive nodes is alive and honest. A larger value of k means that this assumption is more likely to hold, but also that load on the trusted nodes increases (Section V-A). NeighborhoodWatch maintains the logarithmic complexity of original DHTs and is secure as long as the assumptions are maintained.

Using a relatively small trusted resource to secure a scalable infrastructure is an important theme in distributed systems that makes several designs practical. For instance, in cryptography, a public key infrastructure (PKI) can be established by a single trusted keypair. Maheshwari et al. [14] present a system, TDB, which uses a small amounted of trusted storage to build a trusted database on untrusted hosts. Our system employs a similar concept, in which few trusted hosts enforce the correctness of a DHT consisting of many untrusted hosts. The ratio of trusted hosts to untrusted hosts is only limited by the bandwidth and processing power of the trusted hosts. The end result is a potential increase, by several orders of magnitude, in the number of hosts responsible for content (in this case, blacklist) distribution.

We do not address some of the general concerns of DNSBLs, such as their response time [15] and whether they are effective against Zipf-like distributions of spammers' IP addresses [16]. Studies [16] have shown that 80% of spam sources are listed in some DNSBL; we believe this, combined with their popularity, are indicators of their utility.

The rest of this paper is organized as follows. In Section II, we present the security model for NeighborhoodWatch, the underlying system on which DHTBL is built. We show the design of NeighborhoodWatch in Section III and of DHTBL in Section IV. We analyze some aspects of the DHT in Section V and present an evaluation in Section VI. Section VII presents related work, and Section VIII concludes.

II. Security Model and Assumptions

Previous designs of secure DHTs [12], [13] are able to guarantee security when a $\frac{1}{4}$ fraction of nodes are corrupt. We adopt a similar model, in that we allow for some fraction f of the nodes to be malicious, but we do not place hard bounds on f. Instead, we assume that for every sequence of $k+1$ consecutive nodes in the flat ID space of the DHT, at least one is alive and honest, where k is a system parameter. We call this the *fundamental assumption*. The insight is that if nodes cannot choose where they are placed in the DHT (an assumption we justify momentarily), malicious nodes would have to corrupt a large fraction of the N nodes in the DHT in order to obtain a long consecutive sequence of corrupted nodes. By storing sequences of nodes in routing tables, honest nodes are guaranteed to know of at least one other honest node that is "near" a given point in the DHT. For a given value of f, a corresponding k can be chosen so that the fundamental assumption holds with high probability. In Section V-A, we analyze how likely it is that the fundamental assumption holds for given values of f, k, and N.

As the fraction of malicious nodes in the DHT increases, the likelihood that the fundamental assumption holds decreases. If, however, malicious nodes could be discovered and removed from the DHT, then the number of malicious nodes would be kept at a manageable level. In order to remove nodes from the DHT, we assume the existence of an on-line trusted authority, potentially distributed, that periodically issues signed certificates to nodes. These certificates are called *neighborhood certificates*, or nCerts, for reasons which are explained in Section III. nCerts have a relatively short expiration time compared to the average lifetime of a node. Nodes need a current, valid nCert in order to participate in the system. In order to remove a malicious node, the authority simply refuses to sign a fresh nCert for that node.

Maintaining security is then a matter of detecting malicious nodes. We introduce several mechanisms by which misbehavior can be detected and proven. When a node is known (or strongly suspected) to be malicious, it is expelled from the system. Thus, all nodes have incentive to behave properly.

We assume that the adversary cannot place a corrupt node anywhere it wishes in the DHT. We enforce this condition by requiring that nodes obtain a signed public key before they are admitted into the DHT. This certificate is distinct from a neighborhood certificate and may even be issued by a separate, off-line authority. We call this authority the *CA*. Nodes use these certificates to sign responses to certain DHT messages. In addition, a node's ID is taken to be the hash of its public key. This prevents nodes from choosing their location in the DHT. By making certificates expensive to acquire, as do Castro et al. [12], we combat the Sybil attack [17].

Public key certificates serve another purpose as well: when a node's key certificate expires, it must obtain a fresh one. As a consequence, its ID will change, and the node will be relocated to a new portion of the DHT. Although this is a potentially expensive operation, it also limits the lifetime of a sequence of $k+1$ corrupt nodes, as eventually they will be redistributed.

Additionally, NeighborhoodWatch requires all nodes to have loose clock synchronization. If a node's clock differs from the system clock, it may impair its own ability to participate effectively in DHT operations, but it will not harm the operation of other nodes.

III. The NeighborhoodWatch DHT

In this section, we present the NeighborhoodWatch DHT. The DHT supports three operations, all of which are reliable when the fundamental assumption holds:

1) $lookup(id)$, which takes an ID and returns a reference (nCert) to the node responsible for storing items with the given ID
2) $publish(id, item)$, which stores data item $item$ in the DHT under ID id at the node responsible for id as well as its k successors

3) $retrieve(id)$, which returns either a data item published under the given ID or a statement, signed by the node responsible for the ID, that no item was published with that key.

While NeighborhoodWatch will still operate correctly in most cases when the fundamental assumption is violated, security is not guaranteed. Malicious nodes can undetectably hide published items, prevent new nodes from joining, and cause routing failures. This motivates the reliance on a trusted authority: when malicious nodes misbehave, their behavior can be *proven*; upon witnessing proof of misbehavior, the trusted authority can remove malicious nodes from the DHT. We designed NeighborhoodWatch so that any attempt by a node to lie about whether or not an item was published will implicate that node as faulty, and it will be expunged from the system. Therefore, a corrupt node's only course of action is to be maliciously non-responsive. We further show how to remove such nodes from the DHT, leaving only correct peers.

NeighborhoodWatch is based on Chord [11], which routes queries using $O(\log N)$ messages while requiring each node to store only $O(\log N)$ links to other nodes. These links are called a node's *finger table*. The ID space of Chord is the integers between 0 and $2^m - 1$ from some integer m. Chord orders IDs onto a ring modulo 2^m. A node with ID x stores fingers to nodes with ID $x + 2^i \mod 2^m$ for integers $0 \leq i < m$. The *successor* of n is the node whose ID is immediately greater than n's ID modulo 2^m. Likewise, the *predecessor* of n is the node whose ID is immediately less than n's.

A. Overview

Unlike S-Chord, which partitions nodes into disjoint neighborhoods, NeighborhoodWatch assigns a *neighborhood* to each node. This neighborhood consists of the node itself, its k successors, and k predecessors. A single node will therefore appear in $2k+1$ neighborhoods. NeighborhoodWatch requires $2k + 1$ nodes to bootstrap.

NeighborhoodWatch employs an on-line trusted authority, dubbed the Neighborhood Certification Authority, or NCA, to sign certificates attesting to the constituents of neighborhoods. The NCA has a globally-known public key, $NCA.pk$, and corresponding private key $NCA.sk$. The NCA may be replicated, and the state shared between NCA replicas is limited to a private key, a list of malicious nodes, and a list of complaints of non-responsive nodes.

The NCA creates, signs (using a secure digital signature algorithm) and distributes neighborhood certificates, or nCerts, to each node. Nodes renew their nCerts on a regular basis by contacting the NCA. Similarly, joining nodes receive an initial nCert from the NCA. nCerts list the current membership of a neighborhood, accounting for any recent changes in membership that may have occurred. Using signed nCerts, NeighborhoodWatch is able to verify the set of nodes that are responsible for storing an item with ID x.

Nodes maintain and update their finger tables as in Chord. For each of n's successors, predecessors, and finger table entries, node n stores a full nCert (instead of only the node ID and IP address as in Chord). When queried as part of a lookup operation, nodes return nCerts rather than information about a single node.

B. Neighborhood Certificates

A node n has ID $n.id$, IP address $n.ip$, port $n.port$, public key $n.pk$ (along with a signed certificate from the CA), and private key $n.sk$. Let the *info* of node n be defined as $\widehat{n} = \{n.id, n.ip, n.port, n.pk\}$. The predecessor of n is $p(n)$, the i'th predecessor of $p(n)$ is $p^i(n)$, and likewise n's immediate successors are $s(n)$, $s^2(n)$, etc. The *range* of n is the integer interval $(p(n).id, n.id]$. Node n is said to be $owner(x)$ for any ID x in the range of n.

Malicious nodes may try to subvert lookups by lying about their range. By including $p(n)$ in nCert$_n$, NeighborhoodWatch allows any node to determine the range of n given (a fresh copy of) nCert$_n$. As new nCerts are issued periodically, it is possible for a node to hold several nCerts at once. When queried, a malicious node might present an old nCert in an attempt to hide a newly-joined node. Therefore NeighborhoodWatch includes the entire neighborhood of n in nCert$_n$ to serve as witnesses to the freshness of nCert$_n$. Anyone can determine the accuracy of nCert$_n$ by querying each member of the neighborhood and comparing the returned nCerts. If at least one honest neighbor exists, its nCert will reveal any hidden nodes and implicate malicious ones.

Epochs nCerts cannot be explicitly revoked—once a certificate is distributed, it cannot be "called back", since using certificate revocation lists requires a publish and lookup infrastructure very similar to the one we are trying to build. Therefore, to prevent malicious nodes from persisting in the DHT, nCerts must expire periodically. To facilitate this, we divide time into sequentially-numbered epochs, and have each nCert specify the last epoch for which it is valid.

There are two kinds of epochs, *join* and *renew*, which alternate. New nodes may join only in a join epoch; existing nodes may renew their nCerts only in a renew epoch. The length of each is a system parameter, though typical values would be on the order of tens of minutes. The epoch length is a trade-off between the frequency of overhead incurred by the recertification process and the length of time that a proven malicious node can remain in the DHT, since a node can only be expelled by the NCA refusing its renew request.

The current epoch is denoted by an integer which monotonically increases over time. nCerts issued in a join epoch e_j are valid through epoch $e_j + 1$, i.e., the next renew epoch. nCerts issued in renew epoch e_r are valid through epoch e_r+2, i.e., the next renew epoch. A node who fails to renew its nCert before it expires has effectively left the system, as other nodes simply ignore expired nCerts.

The reason for separating the periods in which a node can join from the periods where nodes renew their nCerts is to prevent the following scenario: an honest node n requests that an NCA replica renew nCert$_n$. While this is in progress, a new node j joins, perhaps through a different NCA replica, and an nCert$_n$ containing j is issued to n. Afterward, the initial renew

operation completes, and n receives a new nCert without j. This would produce inconsistencies among the nodes in n's neighborhood, and could potentially lead to n being implicated as malicious. By separating join and renew epochs. only nodes who are affected by a joining node receive new certificates in a join epoch, and neighborhoods are stable (barring removal of unresponsive nodes) throughout a renew epoch.

nCert Format Let $\mathsf{Sign}_K(msg) = (msg, \sigma_K(msg))$ denote the application of a secure digital signature algorithm to msg, where $\sigma_K(msg)$ is the secure digital signature of msg with key K. The format of nCert$_n$ is:

$$\widehat{n} = \{n.id, n.ip, n.port, n.pk\}$$
$$nodes = \widehat{p^k(n)}, ..., \widehat{p(n)}, \widehat{n}, \widehat{s(n)}, ..., \widehat{s^k(n)}$$
$$nCert_n = \mathsf{Sign}_{NCA.sk}(nodes, e)$$

where e is the last epoch in which nCert$_n$ is valid.

C. Routing

NeighborhoodWatch uses iterative routing, meaning that a querier q searching for $owner(id)$ will contact each hop on the path to $owner(id)$, rather than passing the query off for another node to route. This allows q to recover from routing failures. By using iterative routing, q can ensure that each step of the routing protocol makes progress towards $owner(id)$.

To execute $lookup(id)$, a querier q that is a DHT node examines its finger table to find the nCert of the closest known predecessor of id; call this node p. If q is not a DHT node, it requests the nCert of any DHT node it knows about; in this case, that node is called p. In either case, q requests that p provide the nCert of its closest known predecessor of id. Let the nCert that p returns be nCert$_{next}$. q examines nCert$_{next}$ and determines if it is valid. Several criteria must be met for nCert$_{next}$ to be valid: clearly, it must be signed by the NCA and it must not have expired. Also, $next.id$ must be at least halfway between p and id, which will be the case if p's finger table is correct. If $next$ does not allow q to progress at least half of the distance to id, which may be the case if $next$'s finger table has not stabilized, then q has the option of either querying a different node in any nCert it has or continuing with a sub-optimal hop. If nCert$_{next}$ is valid, q replaces p with $next$ and repeats the process, stopping when it receives a valid nCert$_{owner(x)}$.

If p responds with an invalid nCert, or simply doesn't respond, q queries one of the other nodes listed in nCert$_p$. If any of these nodes is correct, q is able to make progress towards id while querying at most $2k + 1$ nodes. Each successful query halves the remaining distance between p and id, resulting in at most $O((2k+1)\log N)$ message per query.

A malicious node m that was previously $owner(id)$, but has since relinquished that range to a newly joined node j, may present an old but valid nCert$_m$ which shows m as $owner(id)$ instead of j. Note that this can only occur during the epoch that j joins or the following recertify epoch; after that time, the nCert$_m$ showing m as $owner(id)$ will be expired. In this case, m cannot suppress the keys for which j is now responsible, as will be shown in Section III-G.

D. Join and Renew

To renew its nCert, a node n presents nCert$_n$ to an NCA replica. The NCA obtains the nCert of each node in nCert$_n$ to check the validity of nCert$_n$ (i.e., to make sure n is not presenting an old-but-unexpired nCert that does not contain a newly-joined node). If nCert$_n$ matches the view of n's neighborhood that the other nCerts describe, the NCA issues a fresh nCert to n, containing the same neighborhood of nodes, but with an expiration time of the next renew epoch.

The process of joining the DHT is similar to that of renewing, but more nodes need to be queried since more nCerts need to be issued. When a node n wishes to join a NeighborhoodWatch instance, it finds nCert$_{owner(n.id)}$ and presents it to an NCA replica. First the NCA ensures that n has a valid private key certificate from the CA. Then the NCA uses the nCert to retrieve the nCerts of the k successors and k predecessors of $owner(n.id)$. The NCA then requests other nCerts as necessary to obtain a full view of the neighborhoods of each node in nCert$_{owner(n.id)}$, so as to verify the neighborhoods of each node that will receive a new nCert when n joins. The NCA then creates new nCerts for n and each of the $2k+1$ nodes in nCert$_{owner(n.id)}$ ($owner(n.id)$ becomes $s(n)$), setting the expiration epoch for each to the next (renew) epoch e, and sends each new nCert to the associated node. The join process is shown in Figure 1.

Once n has joined the DHT, it fills in its finger tables by querying its neighbors for the appropriate nCerts. It also stores the nCerts of all nodes in its neighborhood.

When a node n receives a new nCert, the NCA verifies n's neighborhood. The NCA contacts each node in nCert$_n$. If a contacted node provides a conflicting certificate, it is malicious; if it provides nothing, it is unresponsive. Such nodes are not included in the new nCert$_n$ and are replaced by the appropriate neighbor. To prevent inconsistencies in issued certificates, the NCA replicas must coordinate to maintain a shared list of malicious and unresponsive nodes and refuse to insert these nodes into any granted nCert.

E. Publishing

NeighborhoodWatch provides a $publish(id, item)$ operation, which stores $item$ in the DHT under id. Let $n = owner(id)$. When a node p wishes to publish $item$ to the DHT with key id, it first finds nCert$_n$. Recall that the nodes in nCert$_n$ include $n, s(n), s^2(n), ..., s^k(n)$. Let these nodes be the *publish nodes* of nCert$_n$. p will contact each of the publish nodes and request that the node store $item$. This models the replication procedure in Chord, where the k nodes succeeding id store copies of $item$.

Let d be a publish node that p contacts. If d decides to store the item, it returns a signed *receipt* that it has stored the item. The format of a receipt is:

$$R_{id,d} = \mathsf{Sign}_{d.sk}(id, Hash(item), d.id, e)$$

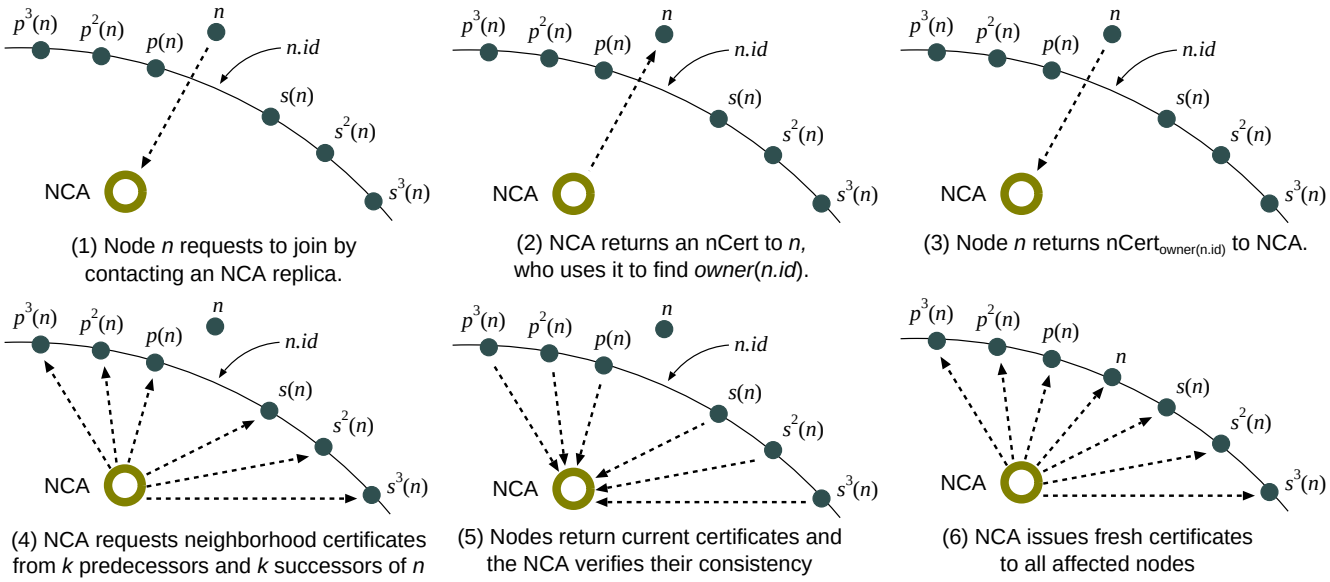

Fig. 1. The join process in the NeighborhoodWatch DHT. Here $k = 3$.

where e is the current epoch. This receipt is used to implicate d if it maliciously refuses to return the item when requested to do so.

If d does not respond, then p considers d to be *unresponsive* and informs the NCA. The consequences of this action are detailed further in Section III-H.

Stored items are self-certifying [18], meaning that given the pair $(id, item)$, there exists a way to verify that the object published under ID id is in fact $item$. One technique for self-certifying items is to set $id = hash(item)$, for some collision-resistant hash function. Another way to make items self-certifying is for the publisher to sign them, assuming anyone retrieving the item can locate the publisher's public key. By using self-certifying items, malicious nodes are prevented from returning "fake" items in response to a retrieve request.

F. Receipt Storage and Retrieval

When a node p publishes a data item to the DHT, it receives (up to) $k + 1$ receipts from the nodes that store the item. In order to use receipts to expose nodes that refuse to return published items, the receipts must be made available to anyone that might observe a dishonest response to a retrieve request. One option is for p to store the receipts for items it publishes and give copies to whomever requests such a receipt. However, this introduces two problems: first, if p goes offline or crashes, the receipts of all items p published become unavailable. Second, when looking for an item published in an ID, a node must know who published that item in order to find the receipt. This is not feasible in the general case where the querier may not know (or care) who published the item it seeks.

We propose a mechanism by which receipts are stored in the DHT itself. Of course, receipts must be stored at IDs independent of the item being stored, so that with high probability a corrupted neighborhood does not store all the receipts for items published to that neighborhood. Also, we want to avoid the problem of storage explosion: the publish of a single item could result in the storage of $k + 1$ receipts; if receipts of *those* publish operations result in receipts stored in the DHT, another $(k + 1)^2$ items would need to be stored, and so on. We define a limit on the *receipt factor*, RF, that dictates how many levels of receipts are published to the DHT. This paper considers the case where $RF = 1$, that is, receipts are stored in the DHT, but receipts-of-receipts are not stored.

When a publisher p receives a receipt $R_{id,d}$ from d for an item published under ID id, p publishes the receipt under the ID $R.id = Hash(id||d.id)$ using the normal $publish$ protocol. $R_{id,d}$ is then stored on the $k + 1$ nodes following $R.id$ in the DHT. p receives receipts from these nodes to verify that the publish is successful, but does not publish these receipts. A node requesting an item with ID $item.id$ from d can check if d has created a receipt R because that node knows both $item.id$ and $d.id$, and can thus determine $R.id$.

Assume d is malicious and returns negative responses to retrieve requests for item i. For d to remain undiscovered, not only must the $k+1$ nodes responsible for storing i be corrupt, but all $k + 1$ copies of receipts for i must be unavailable. For this last condition to be true, corrupt neighborhoods of $k + 1$ nodes must exist around *each* of the receipts. This occurs with probability close to $\left(p^{k+1}\right)^{k+1}$, which is extremely small even for large values of p and small values of k.

G. Retrieving

Like the $publish(id, item)$ operation, a node r executing $retrieve(id)$ also begins by using the $lookup()$ operation to find $\text{nCert}_{owner(id)}$. r chooses a publish node d from $\text{nCert}_{owner(id)}$ and requests $item$ from d by sending id. If d responds with $item$ (which r can verify because items are self-certifying), $retrieve(id)$ returns $item$ and terminates. If d responds saying that no such item exists, it may be the case that $item$ was never published, or it may be that d is

trying to subvert r's request. For this reason, we require that all negative responses be signed by the responding node. r stores d's response and requests $item$ from another publish node, until either $item$ is found or all publish nodes have returned a negative response.

If a node d that r contacts responded with a signed statement that it does not have $item$, yet one of d's neighbors did possess $item$, then r searches for $R_{id,d}$. If it is able to find such a receipt, it presents the receipt and the signed negative response to the NCA, who expels d from the DHT. Even if all nodes issue a negative response, if r suspects that $item$ was in fact published under id, then it may still search for receipts. Collusion-resistant receipt storage is discussed in Section III-F, and the process of expelling nodes is discussed in Section III-H.

A malicious node thus risks exposing itself if it issues a receipt when it intends to not return items, returns a negative response when an item was published to it, or returns an item that was not published. A malicious node can avoid exposure only by not responding to any requests.

H. Removing malicious and unresponsive nodes

Previously in this section, we have shown that any attempt by a malicious node to "lie" to an honest node, whether it be by refusing to return an item it stores or returning an unpublished item, will cause the node to be expelled from the DHT (as a result of the NCA not issuing a fresh nCert to the node). Thus it is in the best interest of malicious nodes to be maliciously unresponsive, that is, to communicate with the NCA to ensure that it receives fresh nCerts, yet not respond to any messages from peers. Here, we show how the system can evict such unresponsive nodes. This property comes at the cost of storing state at the NCA. Specifically, the NCA records the list of nodes that have "complained" about a given DHT node, as well as a list of expelled nodes to prevent them from joining.

Whenever a node d fails to respond to a message from node n, n sends a statement to the NCA to that effect. If the number of nodes that complain about d crosses some threshold, the NCA then determines whether d is being maliciously unresponsive. The NCA does this by finding random nodes in the DHT and asking them to submit requests to d. These nodes report to the NCA whether or not d responded. The NCA must do this so that d does not detect that it is being probed. Therefore the requests should be distributed over time and come from a randomly selected set of nodes.

The NCA requests m DHT nodes to make a request to d and observes the results. Let θ_{alive} and θ_{dead} be two parameters that the NCA uses in determining whether d should be expelled:

- If more than $\theta_{alive} \times m$ nodes report d as alive, the NCA takes no further action.
- If fewer than $\theta_{dead} \times m$ nodes report d as alive, the NCA expels d.
- If number of "alive" responses is between $\theta_{dead} \times m$ and $\theta_{alive} \times m$, the NCA re-runs the test at a later time.

θ_{alive} can be set fairly high, as even if d is not malicious but still fails to respond to many requests, it should be removed from the DHT for better performance. θ_{alive} should be selected so that the probability of selecting $\theta_{alive} \times m$ malicious nodes, when an f-fraction of the DHT is malicious, is low. This process is similar in spirit to the *send challenge* of PeerReview [19].

I. Protecting the NCA

In order for NeighborhoodWatch to function properly, the NCA must be continuously available. Distributing the NCA helps to ensure against operational errors and localized failures, but even widely-distributed systems are vulnerable to DoS attacks. In order to protect the small set of NCA replicas, we leverage the fact that only certain hosts (DHT nodes) have a need to contact the NCA. Note that this condition does not apply to current DNSBLs; every mail client must be able to query the DNSBL hosts.

In order to prevent large-scale DoS attacks, the administrators of the NCA can employ a system similar to Platypus [20]. Platypus provides authenticated source routing by using *network capabilities*. Network capabilities are (*waypoint*, *principal*) pairs that are inserted into packets, where *waypoint* is a router through which the packet is to be routed and *principal* is the entity responsible for sending the packet. The owners of capabilities can create delegated capabilities which other users can use.

To apply Platypus to securing an NCA replica, routers (waypoints) in an NCA replica's ISP filter all traffic to the NCA replica that does not have a valid capability. When a node wishes to join NeighborhoodWatch, it receives a delegated capability from an existing node. This allows it to contact the NCA to obtain its initial nCert. The NCA also includes a new capability that is unique to the newly-joined node and the waypoint through which the node will send requests to the NCA.

An attacker may try to DoS the NCA using compromised nodes that have obtained valid capabilities. However, Platypus allows capabilities to be revoked at waypoints; the attack can be mitigated by revoking the capabilities of any attacking nodes. In addition, routers can identify and drop all incoming packets that have delegated capabilities, if they are being used for attacks.

It is also possible to protect the NCA with architectures like SOS [21] and Mayday [22], which are designed to protect a vital service during emergencies. Due to space constraints, we will omit discussion of how to employ these architectures to protect the NCA.

IV. DHTBL: Building a blacklist on top of NeighborhoodWatch

Building a blacklist on top of NeighborhoodWatch is straightforward. An organization, Org, interested in maintaining a blacklist operates a set of NCA nodes. Org has a public/private keypair $(\mathsf{Org}.pk, \mathsf{Org}.sk)$.

Org publishes entries, consisting of IP addresses of hosts known to source spam, to the blacklist, akin to how DNSBLs add records now. Once Org determines that an address should be blacklisted, they sign the IP address with Org.sk. The signature is published to the DHT using the hash of the IP address as the key. When a client wishes to know if a certain IP address ip is blacklisted, it computes the hash of ip, queries a DHT node using the hash as the key, and receives a response. If the response is a signature, the client uses Org.pk to verify that it is a valid signature of ip.

By using digital signatures, DHTBL is able to provide assurances that regular DNSBLs cannot. DNS is vulnerable to many types of attacks, and responses are easy to forge [2]. Because DHTBL delivers signed data to the client, there is negligible probability that an adversary can alter a record that was published to the DHT, nor produce a signature implicating an innocent IP address.

DHTBL can also be extended beyond one organization. There are many DNSBLs in existence, and mail clients often contact more than one for each incoming email. DHTBL allows different organizations to publish their records in a single repository. Each organization may have its own private key, so that clients know which organization(s) has blacklisted a given IP address. Nodes would only accept items that have been signed by a key belonging to one of these known organizations. Combining existing blacklists into a single, trusted service saves bandwidth, and potentially time, at the client.

NeighborhoodWatch nodes can consist of additional hosts provided by Org or volunteers; customers of the blacklist could be required to participate as compensation for use of the service. We believe the most likely scenario of a DHTBL deployment would consist of a few nodes from each ISP that wishes to make use of DHTBL's service. These nodes would be similar to DNS servers: reliable (thus minimizing churn) and well-maintained (thus limiting the number of malicious nodes). Each node requires a public key certificate to join the DHT; requiring nodes to apply for certificates discourages spammers from obtaining too many while permitting ISPs to obtain several.

One problem of storing records in a DHT, as opposed to a hierarchical database, is that we lose the ability to aggregate blacklisted prefixes. To blacklist an entire /24 network, a DNSBL needs to store only one record, while 256 entries need to be created and published to the DHTBL. We do not believe this is a serious problem, namely because of the vast increase in storage that DHTBL can provide compared to a DNSBL run by a single organization. We are also looking into methods by which multiple records can be aggregated without affecting the performance of lookup operations.

V. Analysis

In this section, we analyze two aspects of NeighborhoodWatch: how likely the fundamental assumption (that each sequence of $k+1$ nodes contains at least one node which is honest and alive) will hold as a function of the probability that a random node is malicious, and the cost of storing an item and its receipts in the DHT. These components, along with the certification process (which we evaluate in Section VI), are the ways in which NeighborhoodWatch differs most from traditional DHTs. As other operations, i.e. lookup, are not dramatically different than those of Chord, we omit their analysis due to space constraints.

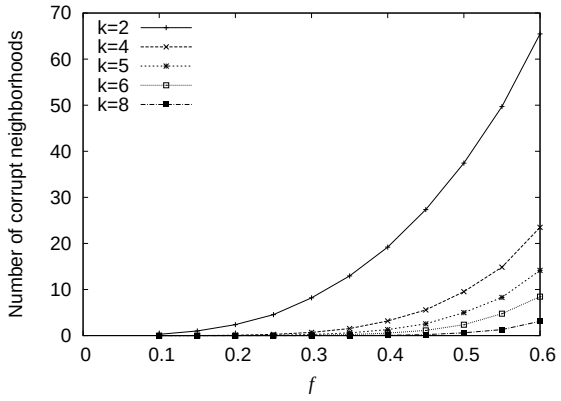

Fig. 2. The expected number of corrupt neighborhoods as function of the probability that a node is bad. Here, $N = 300$.

A. Validity of fundamental assumption

If the fundamental assumption is violated, then NeighborhoodWatch cannot guarantee that a $retrieve(id)$ operation will succeed, even if an item has been published to the DHT under id. Note that $lookup(id)$ operations will be successful as long as there is no sequence of $2k+1$ nodes that are all malicious. To examine the likelihood that the fundamental assumption holds, we simulated an instance of the DHT and counted the number of sequences of length $k+1$ that violate the assumption as a function of N (the number of nodes), k (the number of successors stored in a certificate), and f (the probability that a randomly-chosen node is malicious).

For a system with N nodes, there are N sequences of $k+1$ nodes. Let a node be bad with probability f. Each sequence individually violates the assumption with probability f^{k+1}. However, the sequences are not independent of each other; the expected number of bad sequence is not $N \times f^{k+1}$ (though this is a reasonable approximation for small f).

In Figure 2 we plot the expected number of bad sequences for several values of k as a function of f. Note that the greater the value of N, the higher chance that there is a corrupt sequence. However, even when $f = 0.5$, the number of expected bad sequences is small (below 1 when $k = 8$). This gives an improvement over Castro et al.'s system [12] and S-Chord [13], which are only secure when $f < 0.25$.

B. Cost of storing items in DHT

The NCA publishes signed records of spammer's IP address in the DHT. Each record consists of the signature of the hash of the IP address. We used 40-byte ECDSA signatures (described in Section VI) in our implementation; records are only 40 bytes.

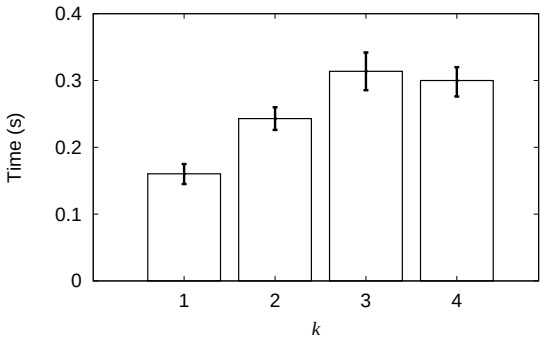

Fig. 3. Recertification times vs k. The certificate size is $2k + 1$. Vertical bars represent 95% confidence intervals.

In order to store a B-byte item in NeighborhoodWatch, $k+1$ copies of the item are stored. Thus, $B(k+1)$ bytes are required for storing copies of the item. In addition, $k + 1$ receipts are stored in the DHT. Each receipt consists of two IDs, one hash, one timestamp (4 bytes), and one signature. Assuming that IDs and hashes are 20 bytes (the length of a SHA-1 hash) and that signatures are 40 bytes, receipts consume 104 bytes.

Each publish results in $k + 1$ receipts that are stored in the DHT. Each receipt is stored by $k + 1$ nodes. Thus storing an item in the DHT incurs an additional cost of $104*(k+1)^2$ bytes to store receipts, for a total cost of $(k + 1)(B + 104(k + 1))$ bytes required to reliably store a B-byte item. Thus, storing a single IP address consumes $104k^2 + 248k + 144$ bytes of system-wide storage.

VI. Implementation and Evaluation

We developed and deployed an implementation of NeighborhoodWatch on approximately 70 PlanetLab [23] nodes. The implementation is coded in 2400 lines of Ruby. For digital signatures, we used the elliptic curve digital signature algorithm (ECDSA) provided by OpenSSL. The ECDSA code is written in C++, with a wrapper written in SWIG. The ECDSA keys in our implementation are 160 bits long, resulting in signatures that are 320 bits long.

PlanetLab nodes were selected to create a large diversity of geographic node locations, latencies, and response times. After deploying NeighborhoodWatch, we then collected statistics of our secure DHT in operation, to better understand the response times required by the routine mechanisms involved in building and maintaining our secure DHT.

In Figure 3, we show the average time in seconds for a recertification operation to complete. The average recertification time depends on the length of a timeout. When the NCA requests node certificates from nodes, it will wait for up to $timeout$ seconds before requesting the node's nCert again; if the node does not respond to a second request, the NCA considers the node dead and replaces it with another. If $timeout$ is too low, churn is unnecessarily introduced, whereas if $timeout$ is too high, unresponsive nodes will have a greater effect on the average recertification time. When running experiments on PlanetLab, we set $timeout$ to 5 seconds.

VII. Related Work

We discuss two categories of related work: those focused on fighting spam, including those that use DHTs to do so, and those focused on DHT security considerations and the design of secure DHTs.

Spam Several previous works have focused on analyzing the characteristics of spam and presented ways to combat it. Jung and Sit [16] performed a study in which they observed that DNSBL lookups from a single university department account for almost 500,000 DNS queries per day. This accounted for 14% of all DNS queries.

In addition to using blacklists [24], [25] that identify spam by the IP address of the sender's MTA, clients can identify spam based on the content of email messages. Employing Bayesian networks [26] as a classification technique is currently a common practice.

Other work has explored using a DHT to provide decentralized content-based spam filtering. Zhou et al. [27] build a distributed text similarity engine based on a relaxation of DHTs in which a published item does not have a unique ID, but is instead index by a *feature vector*. Upon receipt of an email, a client searches the DHT to see if a similar message has been marked as spam. If not, the email is delivered. If that email is later marked as spam, it publishes the feature vector of the message so that other clients will learn that similar message are spam.

HOLD [28] is a method to encourage the adoption of DHTs by leveraging DNS to provide a key-based routing service. The authors suggest that DNSBLs be hosted on a DHT to withstand DoS attacks. However, they do not address the security requirements of the DHT, and their entire system is based on legacy DNS, which we have already argued is an insecure method of transmitting information.

Damiani et al. [29] present a structured P2P network that allows mail clients to query other mail servers as to whether an email is spam. Mail servers track the reputation of other servers, to weigh the validity of their responses.

Trinity [30] is a system which uses a DHT to identify email sent from hosts that are likely to be part of a botnet. The DHT tracks email sources and the number of emails sent by a source in a short time span. While the system is able to track the number of emails a host sends in a space-efficient manner, it requires secret keys to be shared between each DHT node and updated at regular intervals, which is difficult to enforce in dynamic systems like DHTs.

DHTs and security Distributed hash tables (DHTs) were introduced as a method of organizing peer-to-peer nodes to provide decentralized storage. The intent of the original DHT protocols (such as Chord [11], CAN [31], and Pastry [32]) was to minimize both lookup time and the amount of routing state stored at each node. For instance, Chord requires nodes to store links to $O(\log N)$ other nodes and queries take $O(\log N)$ messages for a DHT with N total nodes.

Sit and Morris [33] describe several ways in which adversarial nodes may attempt to subvert a DHT. Malicious nodes might attempt to route requests to other incorrect nodes, provide incorrect routing updates, prevent new nodes from joining the system, and refuse to store or return items.

Castro et al. propose a system in which secure routing can be maintained even when up to $\frac{1}{4}$ of the nodes are malicious. Their system counters the Sybil attack [17], in which a single malicious node joins the DHT in multiple locations, by requiring each node to have a certificate that binds its ID to the node's public key. These certificates are provided by an off-line certificate authority, who limits the number of certificates issued to a single entity. When a node detects that one of its queries for $owner(x)$ has resulted in a node that is unlikely to be $owner(x)$, it floods its request along multiple paths, potentially requiring a number of messages that is polynomial in the number of nodes.

The concept of grouping consecutive nodes into neighborhoods has been a feature of several secure DHT designs. Fiat, Saia, and Young propose S-Chord, which is also resilient to a $\frac{1}{4}$ fraction of malicious nodes. S-Chord partitions consecutive nodes into *swarms*, which act as the basic functional unit of the DHT. Lookups in S-Chord take $O(\log^2 N)$ messages (compared to $O(\log N)$ in NeighborhoodWatch) and each node stores $O(\log^2 N)$ links (compared to $O(\log N)$). Myrmic [34] is a secure DHT that was developed independently that makes similar system assumptions as we do. However, Myrmic does not consider securing the publish operation and has no mechanism for removing malicious nodes from the DHT when they are discovered. Bhattacharjee et al. [35] present a lookup primitive which can be verified through the use of threshold cryptography.

VIII. Conclusion

In this paper, we have presented the NeighborhoodWatch DHT, which is secure against a large fraction of malicious nodes. The system depends on a centralized (though replicated), trusted authority which contacts each node on a regular basis, as well as the assumption that sequences of consecutive corrupt nodes are not arbitrarily long. NeighborhoodWatch consists of a small set of trusted hosts that manage a large set of untrusted hosts, thereby allowing security guarantees to scale by orders of magnitude. By using an innovative receipt-storing scheme and digital signatures, NeighborhoodWatch is able to detect and prove malicious behavior; a corrupt node's only course of action is to be maliciously non-responsive. The centralized authority can remove malicious and non-responsive nodes from the DHT, leaving only correct peers.

We build DHTBL, a blacklist containing IP addresses of known spammers, on top of NeighborhoodWatch. DHTBL provides a secure, resilient blacklist service that also allows for secure message delivery (unlike systems built on top of DNS).

References

[1] J. Goodman, "IP addresses in email clients," in *CEAS*, 2004.

[2] D. Kaminsky, "Black ops 2008 – its the end of the cache as we know it," http://www.doxpara.com/DMK_BO2K8.ppt.

[3] L. Yuan, K. Kant, P. Mohapatra, and C.-N. Chuah, "Dox: A peer-to-peer antidote for DNS cache poisoning attacks," in *IEEE International Conference on Communications*, 2006.

[4] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, "DNS Security Introduction and Requirements," RFC 4033 (Proposed Standard), Mar. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4033.txt

[5] D. Tynan, "Sobig may be working for spammers," http://www.pcworld.com/article/112261/sobig_may_be_working_for_spammers.html.

[6] J. Leyden, "Spamhaus repels DDoS attack," http://www.theregister.co.uk/2006/09/18/spamhaus_ddos_attack/.

[7] S. Linford, "Spammers release virus to attack spamhaus.org," http://www.spamhaus.org/news.lasso?article=13.

[8] Spamhaus, "Virus and ddos attacks on spamhaus," http://www.spamhaus.org/attacks/viruses.html.

[9] Help Net Security, "DDoS attack hits clickbank and spamcop.net," http://www.net-security.org/news.php?id=2966.

[10] R. Lemos, "Blue security folds under spammer's wrath," http://www.securityfocus.com/news/11392.

[11] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications," in *SIGCOMM*, 2001.

[12] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach, "Secure routing for structured peer-to-peer overlay networks," in *OSDI*, 2002.

[13] A. Fiat, J. Saia, and M. Young, "Making chord robust to Byzantine attacks," in *ESA*, 2005.

[14] U. Maheshwari, R. Vingralek, and W. Shapiro, "How to build a trusted database system on untrusted storage," in *OSDI*, 2000.

[15] A. Ramachandran, D. Dagon, and N. Feamster, "Can DNS-based blacklists keep up with bots?" in *CEAS*, 2006.

[16] J. Jung and E. Sit, "An empirical study of spam traffic and the use of DNS black lists," in *IMC*, 2004.

[17] J. Douceur, "The Sybil attack," in *IPTPS*, 2002.

[18] I. Clarke, S. Miller, T. Hong, O. Sandberg, and B. Wiley, "Protecting free expression online with Freenet," *IEEE Internet Computing*, 2002.

[19] A. Haeberlen, P. Kouznetsov, and P. Druschel, "PeerReview: Practical accountability for distributed systems," in *SOSP*, 2007.

[20] B. Raghavan and A. C. Snoeren, "A system for authenticated policy-compliant routing," in *SIGCOMM*, 2004.

[21] A. D. Keromytis, V. Misra, and D. Rubenstein, "SOS: secure overlay services," in *SIGCOMM*, 2002.

[22] D. G. Andersen, "Mayday: distributed filtering for Internet services," in *USITS*, 2003.

[23] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak, "Operating systems support for planetary-scale network services," in *NSDI*, 2004.

[24] SpamCop, http://www.spamcop.net/.

[25] Spamhaus, http://www.spamhaus.org/.

[26] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz, "A Bayesian approach to filtering junk E-mail," in *AAAI Workshop on Learning for Text Categorization*, 1998.

[27] F. Zhou, L. Zhuang, B. Y. Zhao, L. Huang, A. D. Joseph, and J. Kubiatowicz, "Approximate object location and spam filtering on peer-to-peer systems," in *Middleware*, 2003.

[28] J. Considine, M. Walfish, and D. G. Andersen, "A pragmatic approach to DHT adoption," Boston University, Tech. Rep., 2003.

[29] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati, "P2P-based collaborative spam detection and filtering," in *P2P*, 2004.

[30] A. Brodsky and D. Brodsky, "A distributed content independent method for spam detection," in *HotBots*, 2007.

[31] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *SIGCOMM*, 2001.

[32] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *Middleware*, 2001.

[33] E. Sit and R. Morris, "Security considerations for peer-to-peer distributed hash tables," in *IPTPS*, 2002.

[34] P. Wang, N. Hopper, I. Osipkov, and Y. Kim, "Myrmic: Secure and robust DHT routing," U. of Minnesota, Tech. Rep., 2006.

[35] B. Bhattacharjee, R. Rodrigues, and P. Kouznetsov, "Secure lookup without (constrained) flooding," in *WRAITS*, 2007.