The Dynamics of Salsa: A Robust Structured P2P System

Safwan Mahmud Khan

Department of Computer Science, Erik Jonsson School of Engineering & Computer Science, The University of Texas at Dallas, Richardson, TX 75080, USA

E-mail: sainankhan@yahoo.com

Nayantara Mallesh

Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76019, USA

E-mail: nayantara.mallesh@mavs.uta.edu

Arjun Nambiar

Yahoo! Inc., 701 First Avenue, Sunnyvale, CA 94089, USA

E-mail: arjunn@yahoo-inc.com

Matthew Wright

Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76019, USA

E-mail: mwright@cse.uta.edu

Received: September 27, 2010 Accepted: December 26, 2010 DOI: 10.5296/npa.v2i4.474

Abstract

Salsa is a structured peer-to-peer system that is designed to perform robust and reliable lookups. It uses a distributed hash table based on hashes of the nodes' IP addresses to organize the nodes into groups. With a virtual tree structure, limited knowledge of other nodes is enough to route lookups throughout the system. We use redundancy and bounds checking when performing lookups to prevent malicious nodes from returning false information without detection. We show that our scheme prevents attackers from biasing lookups, while incurring moderate overheads, as long as the fraction of malicious nodes is less than 20%. The number of groups can be used as a tunable parameter to trade-off performance versus security. Salsa is resilient to nodes joining and leaving the system while node lookups are ongoing. The message overhead for system operations in a dynamic network is minimal, with the highest measured message overhead of 0.04 messages per node per minute in simulation time.

Keywords: P2P, Distributed hash table, Anonymous communications, Security, Privacy

1. Introduction

Peer-to-peer (p2p) systems offer tremendous benefits to users in applications such as file sharing, Internet telephony, streaming media, and anonymity systems. It is challenging to design a p2p system that enables peers to find resources in the system without a centralized server (like the Napster p2p file-sharing system) or expensive broadcasting techniques (like the Gnutella p2p file-sharing system). To meet this challenge, researchers have proposed a number of *structured p2p systems* based on *distributed hash tables* (DHTs), such as Chord [1] and CAN [4]. These systems allow peers to quickly and efficiently find resources anywhere in the system. An attacker, however, can undermine these systems by inserting a number of malicious peers (e.g. nodes he controls as part of a botnet) and having them give false results during the lookup process. This allows him to perform a simple denial of service attack by giving connection information for peers that do not exist. Worse, he can redirect the request to one of his malicious nodes that then can answer the query by, for example, providing malware instead of the requested file.

To solve this problem, we propose Salsa, a new peer-to-peer system that can reliably perform lookups for users. As with other peer-to-peer systems that use DHTs, our system maps each IP address to a point on the *ID space* using consistent hashing. We further divide the ID space into groups, conceptually organized as a binary tree for purposes of node lookup. Each node has knowledge of all the nodes in its own group, as well as knowing a limited number of nodes in other groups. This knowledge is enough to effectively route lookups throughout the system. Nodes use redundancy – which is enhanced by the path diversity in Salsa – and probabilistic checking when performing lookups to prevent malicious nodes from returning false information without detection.

We previously showed that Salsa prevents attackers from manipulating lookups, while incurring moderate overheads, as long as the fraction of malicious nodes is less than 20% [10]. The number of groups can be used as a tunable parameter in the system, depending on the number of peers, that can be used to balance performance and security.

Contributions: In the present work, we extend our study of the security and performance of Salsa to a dynamic environment in which nodes join and leave the system with lookups proceeding simultaneously. We discuss how new nodes join Salsa with the help of trusted peer already present in the system. The assistance of a trusted friend greatly reduces the risk of being biased by malicious nodes when a new node attempts to join the system. We explain the recursive and redundant lookup procedures along with a deeper look at the bounds checking procedure, which is used to filter out lookup results manipulated by malicious peers. Our simulation results show that when redundant lookups and the bounds checking procedure are used together, the lookup success rate is close to 77% (80% is optimal) with up to 20% malicious nodes in the system.

In Section 2, we describe related work in structured peer-to-peer systems. Section 3 overviews the design of Salsa, focusing mainly on how to handle network dynamics. We describe the simulation methodology and our results in Section 4 and conclude with future directions in Section 5.

2. Background

In this section, we briefly look at other structured peer-to-peer overlays and security considerations that have been studied to date.

2.1 Security in P2P Systems

A number of works have considered security and privacy issues for structured peer-to-peer systems. Danezis et al. propose an alternative routing strategy, called *zig-zag routing* for DHT-based systems that helps defend against Sybil attacks [2]. Zig-zag routing aims to avoid Sybil groups by ensuring diversity while getting close to the target. Both Borisov and Ciaccio have proposed adding anonymity to structured peer-to-peer systems [12, 13]. Borisov proposes the use of random walks on de Bruijn networks to help provide anonymity with reasonably short paths. Ciaccio proposes the use of *imprecise routing* in which the construction of neighbor tables is done from a range rather than with a precise value (as in Chord). This makes it difficult for an attacker to work backwards to determine the source of a request.

These systems all have in common the idea of adding randomness or diversity to the structured overlay, while keeping the general approach of rapid reduction in the distance to the target. Salsa also shares this feature, but it uses a unique structure that is designed to satisfy the different requirements of providing a structured overlay for large-scale systems.

Current overlay systems are resilient, and can route messages correctly from node to node. However, a fraction of nodes that are malicious can easily disrupt the network by maliciously routing or dropping messages, and misinforming peer nodes. Castro et. al. [3] discuss a number of attacks on structured P2P networks and present techniques to defend against such attacks. These above attacks and defenses apply to overlay systems such as Content Addressable Networks [4], Chord [1], Tapestry [5] and Pastry [6]. In general, limited knowledge of the network can be a significant issue for secure paths, as the initiator may be deceived. This is a key motivation of our work.

In order to prevent attackers from taking over P2P systems using sybil attacks, a number of solutions have been proposed. [2] uses real-world trust relationships between peers in order bootstrap the P2P system. New nodes can join the system only through a social contact that knows the owner of the joining node in the real world. Peers produce lookups that ensure diversity over the graph created by the bootstrap process. The idea is promising, with robust lookups possible even when the number of attackers is several times the number of honest nodes. However, the performance is poor in more optimistic cases.

SybilGuard [7] and SybilLimit [8] help limit the influence of malicious attackers by not allowing attackers to introduce large numbers of malicious peers into the system. This approach is also quite promising, but it only limits the attacker to the extent that he can make social connections to other users. If many users can be *socially engineered* into accepting connections, then these approaches can still leave many malicious peers that can disrupt lookups and other operations.

2.2 Robust Distributed Systems

This paper describes algorithms for handling changes in the Salsa system when nodes enter and exit the system. Similar issues have been studied in distributed systems for decades, such as the early work of Lamport, Shostak and Pease on fault tolerance and byzantine agreement [9]. In this work, we focus on algorithms that handle situations specific to Salsa, particularly with its group structure and combination of local and global information. For example, the merging of two groups requires agreement between nodes. While this could be solved by the robust Oral Message algorithm [9], we do not have the same requirements for agreement. In particular, we have designed Salsa so that mismatches in information between nodes are acceptable; most processes will succeed and most nodes will eventually learn about the real system state. Moreover, the security goals of Salsa only require probabilistic guarantees of success, motivating us to design for faster adaptation with lower overhead.

3. Design of Salsa

In this section, we first briefly describe the Salsa system design and then describe how it can be extended to handle dynamic environments in which nodes leave and join the system. The base design of Salsa is explained in more detail in our prior work [10]. Note that while the original purpose of Salsa was for anonymous communications, we here apply Salsa to general P2P systems with a variety of applications.

3.1 The Basic Salsa Design

Salsa is a structured DHT-based overlay that employs redundant lookups to make sure that lookups are not biased. The key to the Salsa design is to maximize the value of each redundant request by ensuring diversity in the path of peers taken by each request, since each redundant request is less helpful when it goes through the same peers.

ID Space and Groups: The Salsa DHT has an ID space based on a consistent hash function, such as MD5, which gives a 128-bit ID space. As with Chord [1] and other DHT-based systems, we can map files, services, and nodes to the ID space by taking a hash of the relevant identifier, such as the filename, the service name or identifier, or the node's IP address. The IDs are used to route lookup requests in the system. A node's ID marks its place in the ID space, and the node *owns* the ID space between itself and the node preceding it.

In Salsa, the ID space is divided into groups that own a contiguous range of IDs. A group consists of the nodes in the group's ID space. Every node knows about all the other nodes in its group, and these are called *local contacts*.

Groups form the leaves of a *virtual balanced binary tree* used for inter-group communication. Figure 1 illustrates the tree. Since each group owns a contiguous ID space, we say that each virtual node owns the ID space of all the leaves (groups) under it. Any virtual node in the tree has two children, each of which owns half of the ID space of the parent. To enable communication and lookups between groups, each node knows about a set of nodes in other groups called *global contacts*. A node has one global contact for each virtual node in the path from the group to the root of the tree. The global contact for a virtual node at a given level is selected randomly from the ID space of the other child of that virtual node. For example, in Figure 1, node N has one global contact in group 2 for level 1, one in group 0 for level 2 and one in group 2 for level 3.

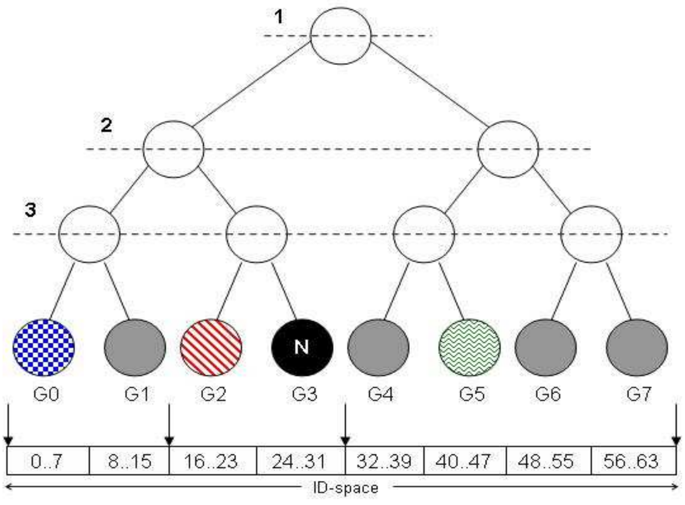


Figure 1. The binary tree structure of Salsa

Lookups: Given the Salsa structure and contact information, a lookup for information in the system is a simple recursive process. The requesting node first generates the desired ID (the *target ID*) by taking a hash of the relevant identifier (e.g. the file name). It then determines where the target ID belongs according to the virtual tree. It selects the global contact corresponding to the lowest (closest to the leaves) virtual node in the tree that owns the ID space in which the target ID can be found. For example, if the target ID is in the other half of the entire ID space, it will select the global contact corresponding to the root node in the tree. The requesting node asks the global contact to continue the lookup. This request is called a *lookup hop*.

The global contact then recursively continues the lookup. It identifies the global contact corresponding to the lowest virtual node that owns the ID space in which the target ID is located. Each lookup hop is at least one level lower in the tree than the previous lookup hop, thereby cutting the ID space in at least half each lookup hop (when lookup is conducted from level 1 to level 2, one of its sub-tree, left or right, is eliminated). The lookup process continues until a member of the target ID's group is contacted – all members of this group know which of their local contacts owns the target ID. This last node can then return the information directly back to the requesting node; alternatively, it can return the information back through the lookup path. Here to mention that, it is not possible by a single malicious node to occupy the root. It is totally a virtual tree and for each node the tree is different.

Redundancy: To provide greater assurance of lookup success, the requesting node actually asks multiple local contacts to redundantly perform the lookup. From among the results of the redundant lookups, the node whose ID is closest to the target ID is selected as the most likely owner of the target ID. Note that the target ID owner will be, by definition, closer to the target in the ID space than all other nodes. Thus, with only one honest lookup path from among the redundant lookups, the requesting node will find the real target ID. In other words, the attacker must gain control over all the redundant lookups to manipulate the requesting node's results.

As we showed in prior work [10], applying redundancy naively in Crowds [14] does not work, because the paths in crowds *converge* to traverse a small set of nodes. The attacker has a better chance to control at least one node on every lookup path by simply controlling a few of the nodes where the paths converge. The Salsa design, however, provides *path diversity*, in that the paths taken by each redundant request generally passes through a different set of nodes from all other requests. With path diversity, the chance that the attacker controls a given path is independent of the chance of controlling the other paths. Thus, the attacker's chance to control all the paths is minimized.

Bounds Check: For additional protection against manipulation of lookups, the requesting node can apply a simple *bounds check*. In the bounds check, we leverage the intuition that the target ID's owner should be close to the target ID – a returned ID that is relatively far away from the target ID in the ID space is likely to be an incorrect result. So we can set a threshold distance, based on the expected distribution of the distances of owners from their targets, and then reject result that's further away than the threshold. To provide flexibility for different node densities, we calculate the bound checking distance used by a node N as *bounds_checking_distance* = α * [*group_range(N)/number local contacts(N)*], where α is a system-dependent parameter. The bound checking distance is dependent on the group range and the number of local contacts of the query node. Therefore bounds checking will be different for different query nodes. The higher the value of α, the easier it is for a looked up node to pass the bounds test. Very high α values, however, increase the number of false negatives while very low α values increase the false positive rate. We compare the results for $\alpha = 1$ and $\alpha = 2$ in Section 4.

3.2 Initialization and Network Dynamics

Structured P2P systems like Salsa require significant coordination due to node churn. This includes nodes continuously joining and leaving the system over time. In this section, we discuss how a Salsa system could be built up securely and handle nodes entering and exiting the system. We present algorithms for distributed management, system operations, and methods for handling churn without giving advantages to attackers in a dynamic Salsa environment. We introduce more randomness into the lookup procedure than the previously proposed Salsa system [10], use flexible metrics for bounds checking, and discuss the implications of these changes with respect to lookup performance and efficiency.

3.2.1 Initialization

One issue for any peer-to-peer system is that an attacker who adds many nodes could dominate the system when it is relatively small, and control most of the system's functions. We see little hope in stopping this through the design of the system – Captcha-inspired Turing tests designed to ensure that a human is using each connection might be tried, but it is beyond the scope of this work [11].

3.2.2 Node Joining Procedure

We propose that nodes should, if possible, join through trusted friends that already have nodes in the network. This is also the best way of building a small network into a larger one. Algorithms 1, 2, and 3 (mentioned on next two pages) describe the join procedure. The new node N can have the friend F perform lookups to identify the nodes in N's group and its group ID range. F hashes N's IP address to get its ID, y. F then looks up y's current owner using redundant lookups. If the owner is honest it responds with the correct group ID range and the full list of its local contacts. F does not rely on the information sent solely by the owner, as it could be maliciously biased. Instead, it requests the same information from R_{join} different nodes around y's neighborhood. These redundant queries protect F against biasing by malicious nodes.

F uses ranges A, B, and C, near y from which to pick the R_{join} random IDs. Figure 2 shows these ranges, which are based on the size of F's group ID space (its *group range*).

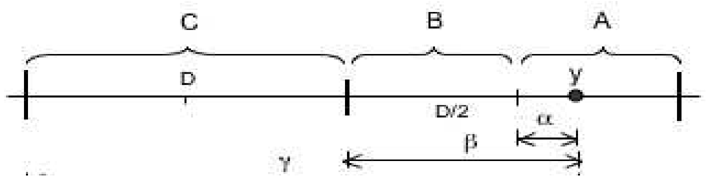


Figure 2. Ranges used to select random nodes to query during node joining.

Due to the uniform random nature of the hashing algorithm, similar-sized group ranges can be assumed system-wide. This is verified by experimental results presented in Section 4.4. Also, as groups split and merge over time, group ranges will change by a power of two. So, F estimates that the true size of N 's group range could be either half of, the same as, or double of its group range D. Range A contains y and is chosen in case the size of N 's group range is D/2. Range B is chosen in case the size of N's group range is D. Range C is chosen in case the size of N 's group range is 2D. F selects Rjoin IDs at random from each range and lookups the owners of these selected IDs using redundant lookups. F requests each node to send its group range and local contact list. The bounds checking procedure is not used during these lookups as we conservatively select information among the returned results later in the joining procedure.

Algorithm 1 *join(F)*: Node *N* joins the system with help from *F*

```
$G_N = help\_join(F, N)$
for all $L \in G_N$ do
    send $join\_message(N)$ to $L$ // $L$ adds $N$ to its local contacts and replies
    if NOT $recv\_join\_confirmation(L)$ then
5:      remove $L$ from $G$
    end if
end for
Generate global contacts
```

Upon receiving the Rjoin responses from all three ranges, F identifies N's group range as the largest group range that includes y. The largest group range is chosen in order to protect against malicious results that try to make the joining node's group size smaller than the actual size. While attackers could benefit from either larger or smaller group ranges, smaller group ranges are more likely to isolate the peer among attackers. Furthermore, the joining peer will be able to identify the correct group size later by using the group splitting procedure (Section 3.2.5). F sends to N the group range and information about all nodes with IDs in that range.

Now N has the information necessary for it to join the Salsa system. N announces to its group members that it has joined their group by sending a J OI N message to each node in its local contact list. The J OI N message contains N's local contact list. Each group member updates its local contact list and ID-space ownership information and responds to N with its updated local list. Hereafter, N builds its global contact list using the update global contact procedure explained in Section 3.2.4.

There is a tradeoff between performance and security that can be tuned by modifying redundancy Rjoin used by F. Honest responses would contain all nodes in the group. Malicious responses would contain a partial list of nodes consisting only of malicious group members. Any mismatches indicate the presence of an attacker in the group. To successfully bias F, the attacker must be sure that F has not received even a single honest response. In the absence of this knowledge attacker nodes are forced to send the full list of contacts.

```
Algorithm 2  $help\_join(F, N)$ Node $F$ helps node $N$ join the system
  $y \leftarrow Hash(IP_N)$; $D \leftarrow group\_size(F)$
  $Y \leftarrow redundant\_lookup(y)$
  $G_N = get\_group(Y)$ // asks $Y$ for its group info.

  // Define ID ranges based on possible group sizes
  // $N$'s group can be half or twice the size of $F$'s, or the same
  $\alpha = y \pmod{D/2}$; $\beta = y \pmod{D}$; $\gamma = y \pmod{2D}$

  // Find the range of $N$'s minimum-sized group
  $A\_min = y - \alpha$; $A\_max = y - \alpha + D/2$

  // Find the range of the other half of $N$'s group
  if $\alpha = \beta$ then
    // The other half is to the right
    $B\_min = y - \beta + D/2$; $B\_max = y - \beta + D$
  else
    // The other half is to the left
    $B\_min = y - \beta$; $B\_max = y - \beta + D/2$
  end if
  // Find the range of $N$'s group's sibling in the tree
  if $\beta = \gamma$ then
    // The sibling is to the right
    $C\_min = y - \gamma + D$; $C\_max = y - \gamma + 2D$
  else
    // The sibling is to the left
    $C\_min = y - \gamma$; $C\_min = y - \gamma + D$
  end if

  // Lookup nodes for each possible group size
  for $i = 0$ to $R_{join}$ do
    $G_N = group\_info(G_N, y, A\_min, A\_max)$
    $G_N = group\_info(G_N, y, B\_min, B\_max)$
    $G_N = group\_info(G_N, y, C\_min, C\_max)$
  end for
  return $G_N$
```

In case y neither belongs in the group range returned by y's current owners nor in the group ranges of any of the R_{join} other responses, F requests the owner of ID y − (D/2) to send its group information. If y does not belong in the group range of this response either, F repeats the joining procedure. If F fails to get N's group information it gives up. N may ask another of its friends or may ask a different advertised node to help it join the system. N may also try to join the system at a later time.

```
Algorithm 3  $group\_info(G, y, min, max)$: Extend group $G$ with information
from a node in the range $(min, max)$
  $t = select\_id\_from\_range(min, max)$
  $T = redundant\_lookup(t)$
  $G_T = get\_group(T)$ // asks $T$ for its group info.
  if $in\_group\_range(G_T, y)$ then
    // $merge\_groups(G_1, G_2)$ uses the larger ID space and the union of the
    nodes from the two groups.
    $G = merge\_groups(G, G_T)$
  end if
  return $G$
```

If N has no friends currently using the system, we propose that a subset of nodes be advertised on, e.g., a website or a bulletin board. An advertised node may post its own IP and those of its global contacts. N can then pick a subset of nodes from the advertised set and for each of them follow the steps in Algorithm 1 as in the case of the trusted friend. Since the global contacts of a node are random and verifiable by hashing the IP with the level of the tree, N has some assurance that they are not an attacker's handpicked selection. For g = 256 groups, n = 10000 nodes, and c = 1000 attacker nodes, the chance that a specific node's global contacts are all corrupt is approximately $(c/n)^{\log_2 g} = 10^{-8}$. The chance that any of the 1000 attackers has a full set of attacker global contacts is given by equation (1).

$$1-(1-(c/n)^{\log_2 g})^c = 10^{-5} \quad (1)$$

The user must be sure that all of the contacts connect correctly, lest an attacker node have multiple attacker contacts and provide fake addresses for the rest. Once N connects to the local contacts, she can redundantly request multiple addresses within her new group. The security of redundant requests applies as with normal lookups, and the joining procedure follows as in the trusted friend case.

3.2.3 Node Leaving Procedure

When a node leaves the system it should, ideally, notify the other group members and its global contacts before leaving. This should be sufficient to create a smooth transition, as nodes can update their global contacts prior to the next round of requests. The LEAVE message must include a brief handshake to prevent spoofed leave messages from becoming a denial of service attack or a way to redirect traffic to corrupt nodes. If a node abruptly disconnects, the nodes that had used it as a global contact will find out in the next round of requests and must issue a set of requests to determine the identity of the new global contact.

3.2.4 Updating the Global Contact List

Nodes do lookups through their global contacts. It is necessary for them to periodically update their list of global contacts. If a node finds that any of its global contacts is no longer active, it must lookup the new owner of the global contact ID. A node chooses the global contact for a given segment of the address space by concatenating the prefix bits of IDs in that segment, and the hash of its IP address. The number of prefix bits is determined by the level of the sub-tree in which the global contact is sought. Once the global ID is determined, the node does a redundant lookup for the owner of the ID to find its global contact.

3.2.5 Splitting a Group

Groups are split if the group population exceeds the configured high threshold, τ_{high}. When a new node joins, all nodes check to see if the number of nodes in their group is more than τ_{high}. If so, the group splitting procedure described in Algorithm 4 is invoked to split the group into half. The splitting node first determines the number of nodes in each prospective half. The population in each of the halves must be greater than the configured low threshold, τ_{low}. If both halves have the minimum population, the group is split. The splitting node updates its local and global contact lists according to the new group structure. It first determines its global contact ID in the new sibling group and identifies the owner of that ID as a new global contact. The splitting node then can simply discard IDs belonging to the new sibling group from its list of local contacts.

Algorithm 4 $group\_split(N, G)$: Node N splits its group G if $group\_size(N) > \tau_{high}$

```
   // Get size of left and right subgroups if G is split
   $g_{min} = group\_min(G); g_{max} = group\_max(G)$
   $g_{half} = g_{min} + (g_{max} - g_{min})/2$
   $G_1 = create\_group(g_{min}, g_{half}); G_2 = create\_group(g_{half}, g_{max})$
5:
   // if the group range for both subgroups > $\tau_{low}$, update N
   if $group\_size(G_1) > \tau_{low}$ AND $group\_size(G_2) > \tau_{low}$ then
      // N adds a global contact from the other subgroup
      // N keeps local contacts of its subgroup and discards the rest
10:   if $Hash(IP_N) < g_{half}$ then
         $add\_globalcontact(N, g_{half}, g_{max})$
         $remove\_localcontacts(N, g_{half}, g_{max})$
      else
         $add\_globalcontact(N, g_{min}, g_{half})$
15:      $remove\_localcontacts(N, g_{min}, g_{half})$
      end if
   end if
```

Each node splits its group individually and does not coordinate with other nodes in the group. As nodes become aware the population is greater than τ_{high}, they split the group and eventually the view of the group becomes consistent across all members. The local splitting procedure has the benefit that malicious nodes cannot influence the group structure of other nodes to the attacker's advantage. The system designer can configure τ_{high} and τ_{low} to balance the costs of managing large groups with the size of the Salsa tree.

3.2.6 Merging Two Groups

After joining, or during a lookup, a node N could find the group population to be less than τ_{low}. In this case, it invokes the merging procedure described in Algorithm 5. The merging node, N, broadcasts its local contact list to each of its group members with the MERGE message. Each member node matches its own local contact lists with the received one. If mismatches are found, the member node contacts each mismatched node. If the mismatched node is alive, it broadcasts its presence to all members of the group. This process also automatically updates the local contact lists of all group members. If a majority of the group members agree with N's local list, the process continues. N sends a MERGE message to its global contact in the sibling group. The global contact replies with a full list of its local contacts. If the new group population is less than τ_{high}, the sibling group's members are added to N's local list. The global contact is removed from the global list and merging is completed.

Algorithm 5 $group\_merge(N, G)$: Node N merges its group G with its sibling group $G_{sibling}$

```
  // broadcast local contact list to all group members
  for all $L_i \in G$ do
    $send\_merge\_message(localcontacts_N)$ to $L_i$ // $L_i$ calls $recv\_merge\_local()$
  end for
5: // if majority of group members reply with YES, continue merging
  if $merge\_consensus(L_i)$ then
    $GC = globalcontact(N, G_{sibling})$
    $send\_message(localcontacts_N)$ to $GC$ // GC calls $recv\_merge()$
    if $recv\_reply(GC, localcontacts_{GC}) == YES$ then
10:     for all $L_i \in G$ do
          $send\_message(localcontacts_{GC})$ to $L_i$
        end for
    end if
  end if
```

4. Simulation and Results

In this section, we first describe the simulation environment in which we tested the security and performance properties of Salsa. We then present the results of our study.

4.1 Simulation Setup

We performed a number of experiments aimed at testing the security properties of the Salsa system. For our tests, we used a 30-bit hash space and considered systems with 10000 nodes. We used G = 128, 256, 512 groups. For each test, we simulated 1000 separate systems and made 1000 lookups per system. We set the high and low thresholds, τ_{high} and τ_{low}, for the number of nodes in a group to 80 and 20 nodes, respectively. We built the simulator in Java, based on our static Salsa simulator [10]. We do not simulate in detail the underlying network or encryption operations. Rather, the experiments focus on the security of the system under varying degrees of attack with lookups taking place in dynamic conditions. Simulations are conducted in a dynamic environment, with lookups proceeding as nodes join and leave the system. In the presence of node churn, group population rise and fall, with groups requiring be splitting or merging over time, as the case may be. We measure the performance of the system in terms of the overhead messages and path lengths needed to perform successful lookups, using different levels of redundancy. Specifically, we demonstrate the effectiveness of redundant node selection and bounds checking in limiting the attacker's ability to bias node selection. We tested with the percentage of malicious nodes ranging from 0% to 20%. Beyond 20% malicious nodes, the attacker already controls a major fraction of system activity, even if lookups were guaranteed to be correct.

To perform a lookup we choose one node at random from the set of all nodes (assuming that it is honest for simplicity). Let us call this the *source node*. We then choose an ID at random from the entire ID-space, and we call this the *target ID*. The source node then performs a lookup for the target ID. The chance that the target ID is owned by a malicious node is approximately given by the fraction of malicious nodes in the system. As nothing can be done about the selection of these malicious nodes, we call these *abandoned lookups*. We aim to test the case where the target ID is owned by an honest node, as this is the chance for the attacker to bias the node selection to an attacker-controlled node. If we choose an honest target ID and the lookup result has been biased in this way, we say call it a *failed lookup*. Correspondingly, we define *successful lookups* as those in which the target ID was owned by an honest node and that node was returned. We checked the percentage of the total lookups that were successful, and compared the results with the number of lookups not abandoned. The difference–the percentage of failed lookups–shows the resiliency of the system to attackers randomly distributed over the ID-space.

We use redundancy to minimize the chance of being biased by a malicious node while looking up a chosen target ID. The level of redundancy, defined by the number of redundant lookup requests per lookup, is a tunable parameter to balance security with performance. The more redundant lookup requests the initiator uses, the greater the chance of getting a successful lookup. However, more lookups mean greater overhead in terms of numbers of messages. We show results for systems using between four and nine redundant lookups.

To study the use of bounds checking, we evaluate the system using different bounds, with results for both false positives and false negatives. When a lookup result falls outside the bound, the source must perform a lookup for a new ID. This increases the overhead of the system, but provides greater security, and we study this tradeoff as well.

4.2 Effect of Bounds Checking

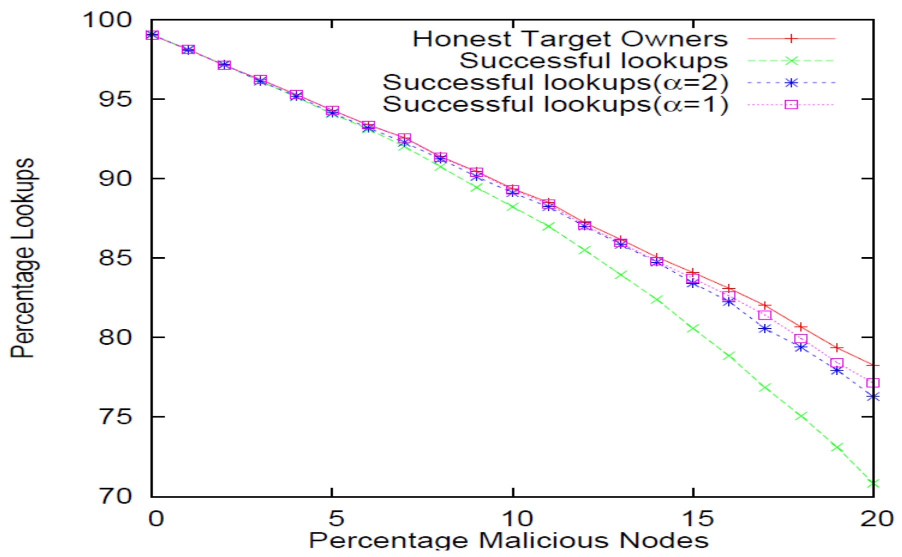


Figure 3. Effect of bounds checking: 10000 nodes, 256 groups with Redundancy = 5

After a node receives the results of a redundant lookup from its local contacts, it picks the result whose ID is closest to the target ID being looked up. Picking the closest ID increases the chance of picking the true owner, since an ID's owner is closer to it than any other node in the system. The closest result is then subjected to bounds checking as discussed earlier in Section 3.1. The effectiveness of the bounds checking procedure is evident from the results shown in Figure 3. The simulation is run on a dynamic system, with 10000 nodes, 256 groups, and a redundancy level of 5. At 20% malicious nodes, the number of successful lookups when bounds checking is not used is 70.8% with 78.2% honest target owners. When bounds checking is used, the success rate increases to 77.2% when a bounds checking factor of α = 1 is used. This means a reduction of nearly 85% in the number of biased lookups because the bounds checking procedure eliminates a large majority of biased results. As the α value decreases the bounds checking distance becomes smaller and results in a better success rate.

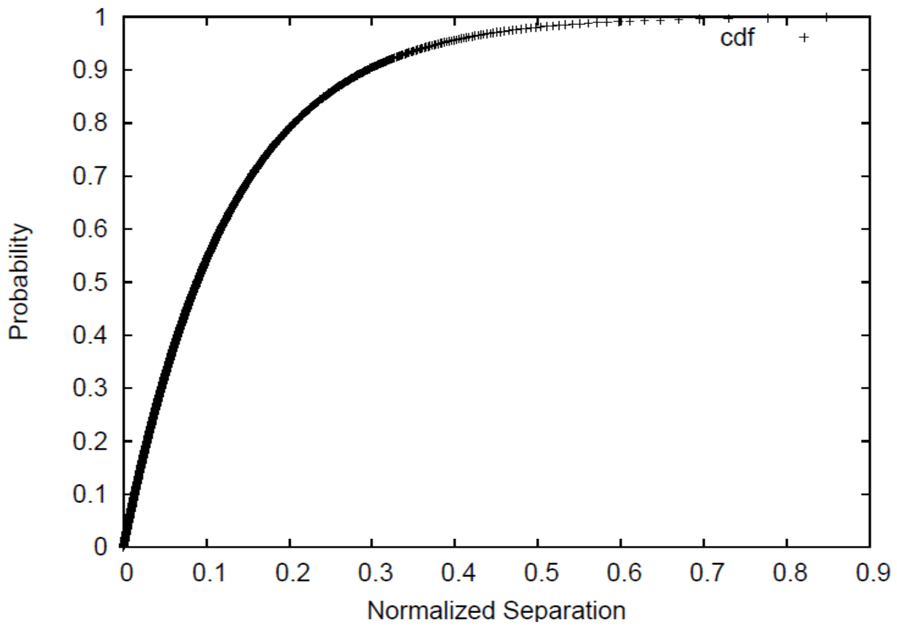


Figure 4. Distribution of IDs and corresponding owners

Figure 4 shows how IDs and their corresponding owners are distributed for n = 1000 nodes and g = 128 groups. The values are normalized by the group ranges. The group range is measured as the difference between the highest ID in the group (end of group) and the lowest ID in the group (beginning of group). In Figure 4 we see that almost 95% of IDs are less than a third of the group range away from their owners. More generally, this means that we can identify a bounds checking range for which most IDs are within the bound for legitimate owners. We explore this more fully in the following subsection.

4.3 False and True Positives and Negatives

The bounds checking procedure may result in any one of four cases. A *False Positive* (FP) result means that an honest lookup result was considered out-of-bounds and failed the bounds check. In case the returned node is really a malicious node and is rejected, we call it a *True Positive* (TP) result. A *False Negative (FN)* result means that a lookup result that was biased by the attacker was within bounds and passed the check. If a result passes the bounds check and is the true owner, we define the case as a *True Negative* (TN).

Table 1, shows the number of false and true negatives and positives for a bounds checking distance calculated with α=2 for different redundancy levels, number of nodes, and number of groups. For the same number of groups, say 256, we see that FNs decrease with increase in redundancy. Lower FN rates mean that fewer malicious results fall within the bounds and thereby go undetected. For example, as seen earlier in Figure 3, at 20% malicious nodes, when no bounds checking is done, the success rate is 70.8% when the number of honest owners is 78.2%. So, 7.4% lookups return malicious owners even though the target owner is honest.

Table 1. False and True Positives and Negatives for 20% malicious nodes with α =2

Redundancy	No. of nodes	No. of groups	%FP	%TP	%FN	%TN
4	10000	256	10.10	14.06	19.65	56.19
5	10000	256	10.89	10.45	18.65	60.01
6	10000	256	10.99	8.29	18.35	62.37
7	1000	32	18.32	5.70	15.33	60.66
		64	18.31	5.81	15.67	60.21
	10000	128	11.38	5.09	17.33	66.21
		256	11.53	6.50	17.86	64.11
		512	11.60	7.80	18.00	62.61
8	10000	256	11.69	5.56	17.58	65.17
9	10000	256	11.89	4.85	17.45	65.81

When bounds checking is used to detect malicious results, Table 1 shows, at redundancy 5, we have a FN rate of 18.65%. This means, 18.65% of those 7.4% returned biased results and falsely passed the bounds check. In terms of the total number of lookups, only 1.3% of the lookups that returned malicious nodes went undetected. With the bound checking distance set using α = 2 there is a non-negligible false negative rate, but we still get a substantial decrease in the attacker's ability to bias the lookup results. Table 2 shows that if we set the bounds distance more strictly, e.g. using α = 1, we greatly reduce the FN rate. However, this comes at the cost of more lookups due to an increase in the number of false positives. In Table 2, 5, 10, 15 and 20 are redundancy levels.

Table 2. False positives and negatives: comparison of α= 1 and α= 2

α value	False Positives	False Negatives			
		5	10	15	20
$\alpha = 1$	21.5%	3.5%	6.4%	9.6%	12.8%
$\alpha = 2$	10.8%	4.9%	9.1%	13.5%	18.6%

It is important to keep the FP rate small because it means that fewer good results are rejected due to failing the bounds test. In Table 1, for α = 2, we observe a small increase of about 1.93% in FPs from redundancy level 4 to 9, which is minimal and acceptable. If the FP rate is high, the requesting node will need to check multiple random IDs until a lookup result passes the bounds check. If the FP rate is given by f_p, the user would need to test $1/f_p$ IDs on average and will succeed with probability $1 - f_p^k$ after testing k IDs. For example, with a 50% false positive rate, the user must test two IDs on average and no more than ten with 99.9% probability. We measured the number of attempts that a node needs to successfully lookup a target ID, and the results are discussed in Section 4.6.

TPs detect results that have been biased maliciously. An interesting observation is that TPs increase with the number of groups. For example, for redundancy level 7 and 10000 nodes, we see that if the number of groups increases from 128 to 512, the percentage of TPs increases too. For the same redundancy level and number of nodes, as the number of groups increases, the number of lookup hops increases. This gives the attacker a higher chance to be on a lookup path and therefore a better opportunity to bias lookups. However, the bounds checking procedure is able to keep up with the attacker and detect a higher number of malicious results, as seen by the increased TP rate.

In all cases we see TNs are much higher than the other statistics, which means that a large majority of successful lookups get accepted. All the above results show the effectiveness of Salsa's bounds checking mechanism.

4.4 Group population distribution

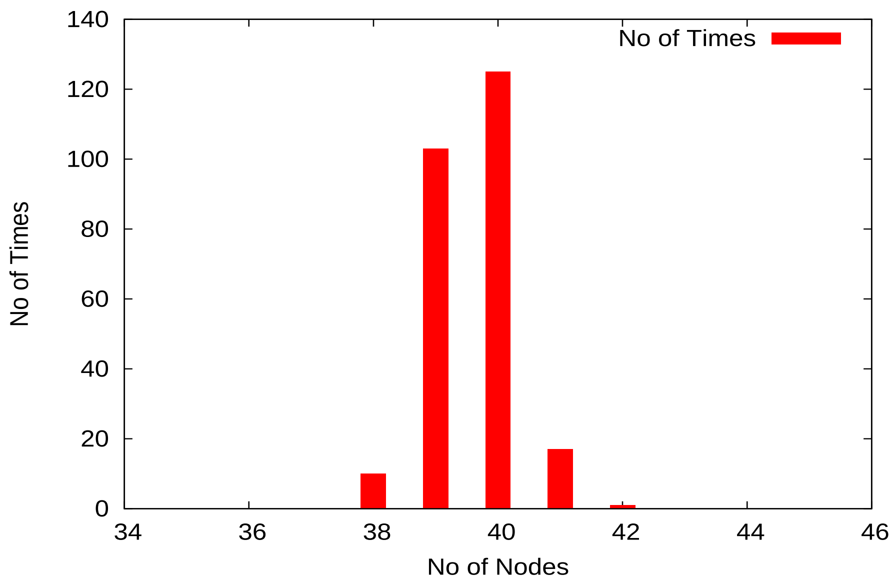


Figure 5. Distribution of group population for 256 groups and 10,000 nodes

The next result set shows the group population distribution statistics. Figure 5 shows for 1000 trials, with 10000 nodes and 256 groups, on average, about 49% of groups contain 40 nodes each, with all group populations falling within a small range of 38 to 42 nodes per group. This shows that group populations do not vary significantly and can be assumed to be similar over the Salsa system. This is important for the joining procedure as well as for the group splitting and group merging procedures as discussed in Sections 3.2.5 and 3.2.6.

4.5 Message Statistics

We now investigate the efficiency of the system, in terms of number of overhead messages required in order to maintain the system and obtain the success rates presented earlier. Figure 6 shows the message overhead per node per minute. Overhead messages include all the messages sent during node joining, node leaving and group merging procedures. The results show the extent to which redundancy affects the message overhead. When redundancy increases, message overhead grows. However the increase in message overhead due to redundancy is minimal per node. For example at 512 groups, it increases by just 0.17 messages per node per minute when redundancy levels increase from 4 to 9. We see a small, acceptable increase in message overhead for other number of groups, as well, as redundancy increases. The message overhead also varies between different numbers of groups. Systems with 128 groups have a higher message overhead compared to ones with 256 and 512 groups. This is expected because group populations are higher when the total number of groups is smaller. In the case of node joining and leaving, for example, the joining or leaving node would have to announce its joining a larger number of group members. The message overhead would thus be higher for 128 groups than for larger group numbers.

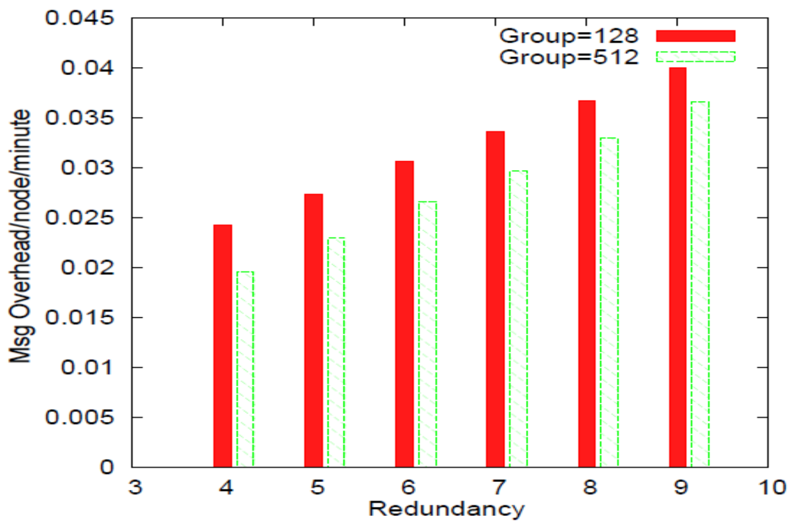


Figure 6. Message overhead: 10000 nodes, 20% attackers, α=2 at multiple redundancy levels

Figure 7 presents the messages per lookup in a system with 10000 nodes for different numbers of groups. The results show that the number of messages per lookup increases with increased redundancy. We notice an increase in overhead messages as the number of groups grows. For example there is a difference of about 6 messages per lookup between 128 groups and 512 groups at redundancy level 4. The correspondence between messages per lookup and number of groups is due to the fact that each lookup has a longer lookup path when the number of groups is higher. Since group populations are smaller when the number of groups is higher, a lookup would take longer to converge, resulting in more recursive lookups until the correct owner is found.

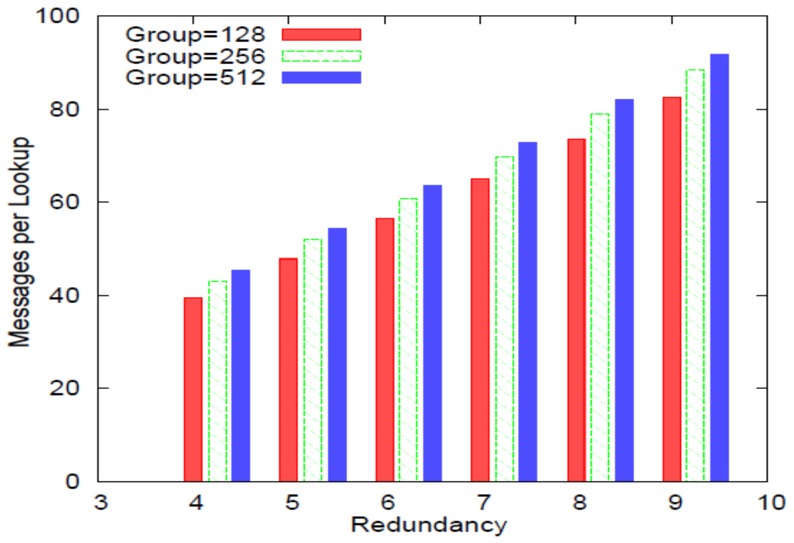


Figure 7. Messages per lookup: 10000 nodes, 20% malicious nodes and α=2 with different redundancy levels

In Figure 8, we show that the messages per join is greater for 128 groups than for 256 and 512 groups at a given redundancy level. When a node joins a group, it needs to inform all other group members of its arrival. With 10000 nodes and 128 groups the number of nodes per group averages around 78 nodes per group as compared to 39 and 19 nodes/group for 256 and 512 groups. Since the group population for 128 groups is much higher than for 256 and 512 groups, we see that the messages per join are also proportionately higher. The message overhead for 512 groups is close to the overhead for 256 groups because, average group populations at 512 groups is close to τ_{low}. This results in many groups merging with one another. The resulting rise in group population leads to an increase in overhead messages during subsequent join operations.

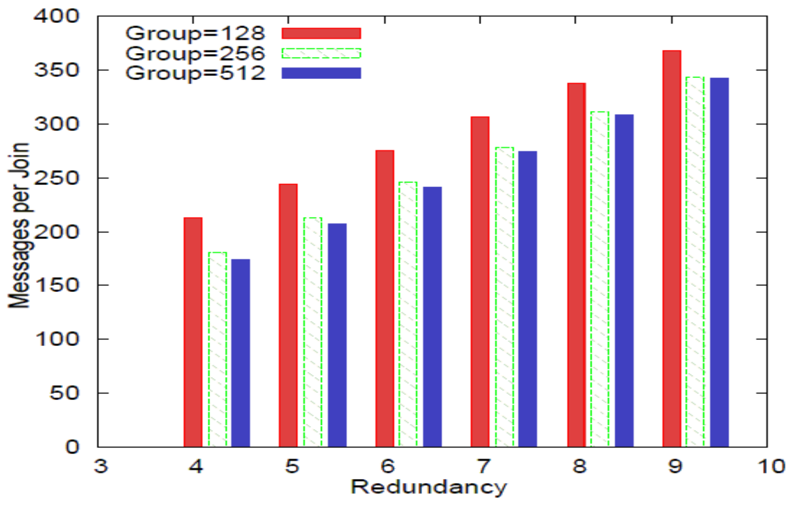


Figure 8. Message per join: 10000 nodes, 20% malicious nodes and α =2 with different redundancy levels

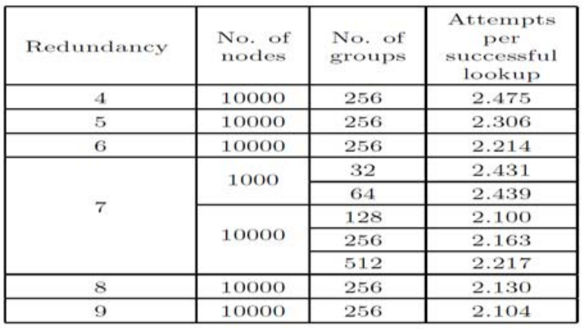
Simulation results showed no message overhead during group merging in some scenarios. For example, with 10000 nodes and 128 groups there were zero overhead messages for merging. This can be attributed to the high average group population of about 78 nodes, in such a case. Since nor a single group had a population below the lower population threshold of τ_{low}, no merge operations were required. We observed that 256 groups needed more messages than 512 groups during the merge operation. A consensus is required during the merge procedure in order to complete the operation. Due to this many messages are exchanged among group members in the current and sibling group. Since 256 groups has almost twice the group population of 512 groups, the number of messages for merging is greater for 256 than for 512 groups.

Updating global contacts requires one redundant lookup. There is no message overhead for splitting groups, since it is done locally by individual nodes.

4.6 Attempts per successful lookup

We present some results that help decide the α value for computing the bounds checking distance in a Salsa system. In Table 3, for α=1, the attempts per successful lookup ranges from 2.10 to 2.47. When we use α = 2, and thereby increase the range for bounds, the attempts per successful lookup goes down, ranging from 1.51 to 1.78. This is a clear improvement over α=1.

Table 3. Attempts per successful lookup for 20% malicious nodes and α =1 for different redundancy levels



Redundancy	No. of nodes	No. of groups	Attempts per successful lookup
4	10000	256	2.475
5	10000	256	2.306
6	10000	256	2.214
7	1000	32	2.431
		64	2.439
	10000	128	2.100
		256	2.163
		512	2.217
8	10000	256	2.130
9	10000	256	2.104

These results are seen in Table 4. However, the α value also affects the number of true and false positives and negatives. Therefore a decision on the value of α should be taken considering what trade-offs are acceptable for a particular system. Both Tables 3 and 4 show that if the redundancy level is increased, the attempts per successful lookup go down. Thus, the redundancy level is another factor that should be used while deciding the value of α.

Table 4. Attempts per successful lookup for 20% malicious nodes and α =2 for different redundancy levels

Redundancy	No. of nodes	No. of groups	Attempts per successful lookup
4	10000	256	1.780
5	10000	256	1.666
6	10000	256	1.603
7	1000	32	1.649
		64	1.661
	10000	128	1.510
		256	1.560
		512	1.597
8	10000	256	1.535
9	10000	256	1.520

5. Conclusions

In this paper we studied the performance of Salsa, a structured overlay architecture based on a DHT for robust lookups in the face of attacks. We have shown that the system organization helps its resilience to attack, even when nodes leave and join the system. The system uses redundant lookups to mitigate the risk of results biased by malicious nodes. The redundancy is a tunable parameter to balance performance and security. A 79% reduction in the number of biased lookups is observed when the redundancy factor increases from 4 to 9. The increased redundancy comes with a small increase in overhead of only 0.17 messages per node per minute in our dynamic simulations. Results show that Salsa is resilient to node churn and the number of messages due to lookups, nodes joining and nodes leaving is negligible on a per node basis per minute. Salsa uses bounds checking to find malicious nodes returned by lookups, which results in a 85% decrease in the lookup failure rate. We demonstrate the effectiveness of bounds checking in eliminating a large percentage of biased results, with only 1.3% of malicious results falsely passing the bounds checking procedure.

6. Acknowledgments

Thanks to Dawn Song for initial discussions about this project and Roger Dingledine for his input about Tor. Thanks to Nikita Borisov, Prateek Mittal, and Yongdae Kim for their analysis and helpful insights on circuit building using Salsa. This work was supported in part by the National Science Foundation under award number CNS-0549998 and CAREER award number CNS-0954133. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation.

References

[1] I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan, "Chord: A scalable Peer-To-Peer lookup service for Internet applications", in: Proceedings of the 2001

ACM SIGCOMM Conference, 2001.
[2] G. Danezis, C. Lesniewski-Laas, M. F. Kaashoek, R. Anderson, "Sybil-resistant DHT routing", in: Proc. ESORICS, 2005.
[3] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, D. S. Wallach, "Secure routing for structured peer-to-peer overlay networks", SIGOPS Oper. Syst. Rev. 36 (SI) (2002) 299–314.
[4] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Schenker, "A scalable content-addressable network", in: SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, ACM, New York, NY, USA, 2001, pp. 161–172.
[5] B. Y. Zhao, J. D. Kubiatowicz, A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and Routing", Tech. rep., Berkeley, CA, USA (2001).
[6] A. Rowstron, P. Druschel, Pastry: " Scalable, decentralized object location, and routing for large-scale peer-to-peer systems", in: Lecture Notes in Computer Science, 2001, pp. 329–350.
[7] H. Yu, M. Kaminsky, P. B. Gibbons, A. Flaxman, " SybilGuard: defending against sybil attacks via social networks", SIGCOMM Comput. Commun. Rev. 36 (4) (2006) 267–278.
[8] H. Yu, P. B. Gibbons, M. Kaminsky, F. Xiao, " SybilLimit: A near-optimal social network defense against sybil attacks", in: Security and Privacy, IEEE Symposium on, 2008.
[9] L. Lamport, R. Shostak, M. Pease, " The Byzantine generals problem", ACM Transactions on Programming Languages and Systems 4 (1982) 382–401.
[10] A. Nambiar, M. Wright, "Salsa: a structured approach to large-scale anonymity", in: Proc. ACM Conf. on Computer and Communications Security (ACM CCS), 2006.
[11] L. von Ahn, M. Blum, N. J. Hopper, J. Langford, "CAPTCHA: Using hard AI problems for security", in: Proc. Eurocrypt, 2003.
[12] N. Borisov, " Anonymous Routing in Structured Peer-to-Peer Overlays", University of California, Berkeley, CA, 2005, ph.D Thesis.
[13] G. Ciaccio, "Improving sender anonymity in a structured overlay with imprecise routing", in: Proc. Privacy Enhancing Technologies Workshop (PET), 2006.
[14] M. K. Reiter, A. D. Rubin, Crowds: "Anonymity for Web Transactions", ACM Transactions on Information and System Security 1 (1) (1998) 66–92.

Copyright Disclaimer