Pong-Cache Poisoning in GUESS [*]

Neil Daswani and Hector Garcia-Molina
Computer Science Department
Stanford University
Stanford, CA 94305-9045
{daswani,hector}@db.stanford.edu

ABSTRACT

This paper studies the problem of resource discovery in unstructured peer-to-peer (P2P) systems. We propose simple policies that make the discovery of resources resilient to coordinated attacks by malicious nodes. We focus on a novel P2P protocol called GUESS [8] that uses a *pong cache*, a set of currently known nodes, to discover new ones. We describe how to limit pong cache poisoning, a condition in which the ids of malicious nodes appear in the pong caches of good nodes. We propose an *ID smearing algorithm (IDSA)* and a *dynamic network partitioning (DNP)* scheme that can be used together to reduce the impact of malicious nodes. We also propose adding an *introduction protocol (IP)* as a basic mechanism to GUESS to ensure *liveness.* We suggest using a *most-recently-used (MRU)* cache replacement policy to slow down the rate of poisoning. Finally, we determine the marginal utility of using a malicious node detector (MND) to further limit poisoning, and the level of accuracy required of the detector.

Categories and Subject Descriptors

C.2.0 [**Computers-Communication Networks**]: General—*Security and protection*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems, Information networks*; I.6.3 [**Simulation and Modeling**]: Applications

General Terms

Security, Algorithms, Measurement, Experimentation

Keywords

peer-to-peer, security, denial-of-service

[*]This work was partially supported by NSF Grant CCR-0208683.

1. INTRODUCTION

In this paper we study the problem of resource discovery in unstructured peer-to-peer (P2P) systems, and we propose simple policies that make the discovery of resources (or nodes) resilient to coordinated attacks by malicious nodes. For concreteness, we focus on a novel P2P protocol called GUESS (Gnutella UDP Extension for Scalable Searches) [8] that is under consideration by the Gnutella Development Forum (GDF) [10], a grouping of independent Gnutella software vendors, for use in the Gnutella P2P network. The Gnutella file-sharing network is one of the most widely used P2P networks with over 100,000 concurrent users on the network at any one time offering 5 to 10 terabytes of files. In addition, over 10 distinct vendors have deployed Gnutella application software and over 35 million copies of these applications have been downloaded by users. We also note that other extremely popular P2P protocols such as FastTrack and eDonkey already use protocols similar to GUESS for resource discovery, and hence could also benefit from the techniques we develop in this paper.

As opposed to traditional Gnutella networks in which flooding is used to propagate queries through a software overlay topology, nodes in a GUESS network explicitly query other nodes one at a time. The protocol design was partially motivated by results on random walks in unstructured P2P networks [13]. In GUESS, each node keeps a cache of other nodes that are available to accept queries, and sends its queries to one of the nodes in its cache at random. Nodes are required to manage the cache by deleting nodes that are no longer available, and adding new nodes that are available. Nodes exchange information about which nodes are available by exchanging "ping" and "pong" messages, and, as a result, the caches that nodes use to store node ids of other nodes are called "pong caches."

Initial evaluations [1] suggest that GUESS can be a significantly more efficient search mechanism, compared to basic Gnutella, essentially because flooding is replaced by directed querying that can be throttled by the originating node. Furthermore, queries are not routed through untrusted third parties, so GUESS nodes are less susceptible to attacks by such intermediaries.

However, GUESS does have a new potential security weakness: the pong caches. For reasons that we describe in Section 3, malicious nodes will be interested in having their node ids appear in the pong caches of good nodes as a pre-cursor to many forms of attack. When ids of malicious nodes appear in the pong cache of a good node, we say that the good node's pong cache has been *poisoned*. And once a cache is poisoned, it becomes very hard for a good node to find other good nodes that may provide useful results. In this paper we focus on this pong-cache poisoning problem, and suggest techniques for making nodes more resilient to this attack.

The techniques that we propose are *complementary* to existing security mechanisms, and help us implement defense-in-depth within the context of a GUESS network. In particular, there are three rough categories of defenses that are required:

- *Prevention.* These defenses make it hard for malicious nodes to participate in the system. For instance, we can require that nodes obtain certified node ids, as proposed in reference [3], to limit the absolute number of malicious nodes, and/or require nodes to solve crypto-puzzles to obtain node ids with the intent of slowing down the rate at which malicious nodes can join.
- *Detection.* These defenses identify malicious nodes that have joined and attempt to remove them from the network and/or revoke their privileges. For instance, one can use a collaborative trust system to identify nodes that are providing bad results or misbehaving in other ways.
- *Containment.* Finally, these defenses attempt to minimize damage from nodes that have entered the system and have not yet been detected. Our techniques to deal with pong-cache poisoning are in this category.

We believe that for best results all three types of defenses must be employed in GUESS, or in any large scale distributed system. Since prevention and detection cannot be expected to work perfectly in practice, it is safer to design the GUESS cache management policies under the assumption that a few malicious nodes have gotten through and may poison caches. Indeed, as we will see in Section 5, even just a few malicious nodes can poison a large fraction of all the caches in the network, if good cache management policies are not used. Thus, it is critical to have strong containment policies in place. Of course, our policies will only reduce pong-cache poisoning instead of eliminating it altogether. As such, the policies we propose and evaluate provide practical security; they do not necessarily provide "provable" security.

Note that we are conducting this work early in the evolution of GUESS instead of waiting after its initial deployment, such that the protocol will not have to be retrofitted with security features only after the fact. Some of the decisions that nodes need to make are currently unspecified in the existing protocol specification, and we fill in these gaps with recommendations on how nodes should make these decisions to deal with attacks.

Also note that while we focus on the ping-pong resource discovery protocol used in GUESS, the need to maintain caches of available peers also arises in other contexts, such as wireless ad-hoc networks, grid computing, and autonomic computing. Thus, we believe that our work can be generalized in a straightforward fashion to be applied in these other areas.

Our specific contributions in this paper are:

- We define a model that captures how ping and pong messages affect pong cache behavior, and we describe the key decisions that nodes must make as they interact with other nodes (Section 2).
- We define expected behaviors for good and malicious nodes, and we outline the key research goals that need to be addressed to deal with pong-cache poisoning in GUESS (Section 3).
- We propose new mechanisms and improvements to existing mechanisms in the protocol to mitigate attacks by malicious nodes. These mechanisms control how nodes should initialize, insert, delete, and replace entries in their caches (Section 4).
- We evaluate the impact that malicious nodes have by poisoning pong caches of good nodes, and how the various policies that we propose mitigate poisoning (Section 5).

In closing this introduction, we make one final observation. Protocols like GUESS and Gnutella are called "unstructured" because each peer decides on its own what content to store and make available for searching. On the other hand, in a structured network (like a distributed hash table [15, 18]), data is algorithmically placed at nodes. Unstructured networks are more common in practice, and are used everyday to handle huge numbers of users and massive content. Indeed, out of the ten or so major file-sharing P2P protocols most widely used on the Internet (FastTrack, eDonkey, Gnutella, etc.), all except one (OverNet) are unstructured. Thus, we feel it is very important to study security issues in such systems, as we do in this paper. This is not to say that DHTs may be more or less secure, or more or less efficient. We believe that both approaches are worthy of investigation, but in this particular paper, we only have space to study one of the approaches.

2. BASIC MODEL

Our model considers a set of n nodes in a peer-to-peer network. Initially, the set of nodes participating in the network is $N = \{N_1, N_2, ..., N_n\}$. Gnutella networks are made up of leaf nodes and supernodes, where supernodes have more processing power and capacity than leaf nodes. We only choose to model supernodes (also known as Ultrapeers) in the network, as leaf nodes only send pings to supernodes, and do not respond to pings themselves. Since our model focuses on supernodes, the term node will be used to refer to a supernode in the remainder of the paper.

Some subset of the nodes in N are *good*, while others are *malicious*. Good nodes, in general, follow the protocols we describe, while malicious nodes may or may not. We assume that G denotes the set of good nodes, and that M denotes the set of malicious nodes. Of course, at any instant $G \cup M = N$ and $G \cap M = \emptyset$. We describe our threat model and the expected behaviors of malicious nodes in detail in Section 3.

Not all of the nodes in the system are expected to be available at the same time. Indeed, we expect some percentage of the nodes to be unavailable most of the time, although we expect the specific set of nodes that are unavailable at any given time to vary. Some typical reasons for unavailability could be node failure, overload due to a denial-of-service attack, or (if the node is a mobile device) it may be simply out of range of some of the other nodes in the network. We say that a node that becomes unavailable has "died." This "death" may not be permanent, but from the standpoint of resource discovery, the node will need to be re-discovered by other nodes once it becomes available again.

At some instant in time, a set of nodes $\Delta \subseteq G$ may die. At the same time that the set of nodes in Δ die, a set of nodes, B, can be born. The new set of nodes in the system is $N' = (N - \Delta) \cup B$.

We assume that each node N_i keeps a list of nodes that it typically relies on. In a GUESS network, this list is called a *pong cache*. Each node N_i has a fixed-size pong cache, $P(N_i)$, associated with it where $P(N_i)$ is a set of node ids. The set $P(N_i)$ may change over time.

At a given point in time, a particular cache entry is said to be either *live*, *poisoned*, or *dead*. *Live* cache entries are ones that contain the node ids of good nodes that have joined the system and are available to respond to queries with legitimate results. Cache entries containing malicious node ids are not considered live ids, but *poisoned*. *Dead* cache entries contain the node ids of good nodes that have temporarily or permanently left the system. The sum of the number of live, poisoned, and dead cache entries equals the total number of pong cache entries in the system.

A node N_i may decide to *ping* another node N_j to determine if it is live. A ping is simply a "no-operation" query to which a node is expected to respond if it is available. Since message passing incurs network and computational overhead in real systems, a ping may contain a request to process a query instead of just being a "no-op." When node N_j receives the ping, it may respond to N_i with a *pong* that tells N_i that N_j is "live."

In our model, a node N_i may choose to ping any node $N_j \in P(N_i)$ in its pong cache. Node N_i chooses which N_j according to a *ping probe policy*. If node N_j temporarily or permanently dies, N_i may not receive a response to its ping after some timeout period, and N_i may decide to remove N_j from its cache. Node N_i could also decide to keep the cache entry around in the hope that N_j may become available again at some time later.

As part of the pong that N_i receives from N_j, N_i receives a set of node ids $S = \{N_{k1}, N_{k2}, ...N_{k|S|}\}$ where $N_{ki} \in N, 1 \leq i \leq |S|$. The nodes in set S are chosen by N_j from its pong cache ($S \subseteq P(N_j)$) using a *pong choice policy*.

Upon receiving S, N_i may replace any subset of $P(N_i)$ with any subset of S to create $P'(N_i)$, node N_i's updated pong cache. Let X be any subset of $P(N_i)$, and Y be any subset of S. Then, $P'(N_i) \leftarrow (P(N_i) - X) \cup Y$. For example, since the empty set is a subset of $P(N_i)$, and Y could be exactly equal to S, N_i could simply add all of the entries received from N_j to its pong cache. Of course, pong caches in real systems typically have a fixed-size, and require other choices for X and Y. We will refer to the exact method that node N_i uses to determine the sets X and Y as the *cache replacement policy*.

When a node N_k is newly born, its pong cache, $P(N_k)$, needs to be initialized. We say that N_k's pong cache needs to be "seeded" when the node is born, and we explore various *seeding policies* in Section 4.1.

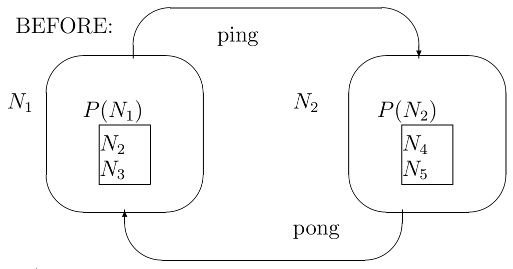
It is important not only for new nodes to have their caches seeded with existing node ids when they are born, but it is equally as important for existing nodes to find out about new nodes that have joined the network. As we will see in Section 5, after some amount of time a majority of the entries in a node's pong cache can become invalid due to the deaths of the corresponding nodes. To eliminate this problem, we propose adding an *introduction protocol (IP)* to GUESS in which a new node N_k may want to make itself available to existing nodes by having its id added to their pong caches. In Section 4.2, we describe an IP that can be used to accomplish this.

Figure 1 shows an example of how resource discovery takes place in a GUESS network. The large, rounded squares represent nodes N_1 and N_2, and the box within each node represents that node's pong cache. The top half of the figure shows the state of the nodes before any interaction between them takes place. Initially, node N_1 has the node ids of nodes N_2 and N_3 in its pong cache. The figure shows what happens when node N_1 pings N_2. In particular, N_1 chooses N_2 from its pong cache, and sends N_2 a ping message. N_1 could have alternatively chosen to ping N_3 since N_3 is also in its pong cache. In response to the ping, N_2 chooses a set, say $S = \{N_4\}$, from its pong cache. N_2 could have also potentially chosen any of the other three possible subsets of $P(N_2)$ as its response to the ping. In the example in Figure 1, N_1 choses to update its pong cache by replacing the N_3 entry ($X = \{N_3\}$) with N_4 ($Y = \{N_4\}$). The lower half of the figure shows the resulting state: Node N_1 computes $P'(N_1) = (P(N_1) - X) \cup Y = \{N_2, N_4\}$ as its updated pong cache.

We may view the model above in one of two (equivalent) ways. In one view, such a network has no overlay topology, and queries are sent to nodes whose ids are chosen from pong caches in an ad-hoc fashion. Another way of thinking about GUESS is that it does have an overlay topology which evolves dynamically. Specifically, we can think of $P(N_i)$ as the neighbors of N_i. In this view, the set of neighbors that a node has is larger and more dynamic than in traditional Gnutella, with the exception that we do not flood, and queries are only processed by direct neighbors.

3. THREAT MODEL

While there might exist various prevention and detection mechanisms in a GUESS system (as mentioned in Section 1), some number of malicious nodes may still be able to join. For instance, we might require that nodes obtain certified node ids as proposed in reference [3]. However, Sybil attacks might still be possible. A Sybil attack [9] is one in which a malicious adversary generates or obtains many node ids (identities). A ma-



1) N_1 sends ping message to N_2
2) N_2 returns pong message $S = \{N_4\}$
3) N_1 chooses $X = \{N_3\}$ and $Y = \{N_4\}$
4) N_1 computes $P'(N_1) = (P(N_1) - X) \cup Y = \{N_2, N_4\}$

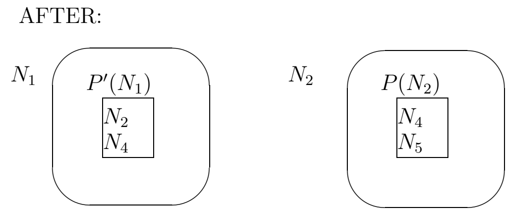


Figure 1: An example of a ping and pong between two good nodes in a GUESS network.

licious adversary could do so by paying for node ids, or compromising the security of existing hosts that have valid node ids. In our threat model, there is some non-negligible cost (i.e., time and/or money) for obtaining a node id, and malicious nodes can acquire some limited number of node ids.

While there are many possible attacks that can be conducted by malicious nodes that have taken control of node ids, we focus on one key class of attacks in this paper. In particular, malicious nodes may collude to conduct "pong-cache poisoning" attacks in which they attempt to propagate malicious node ids in as many pong caches as possible. We believe that the poisoning of pong caches will be the first step in many forms of distributed attacks in a GUESS system.

In a pong-cache poisoning attack, malicious nodes respond to pings from good nodes with node ids of other malicious nodes. Good nodes may then insert the ids of malicious nodes into their cache. There are (at least) three specific reasons why the insertion of malicious node ids into the caches of good nodes is bad:

1. *Denial-of-Service (DoS).* A good node may query a malicious node, and may not receive a response.

2. *Inauthentic Results.* A good node may query a malicious node, and may receive a response that contains incorrect answers to the query.

3. *Propagated Cache Poisoning.* A good node may respond to pings from other good nodes with a malicious node id, thereby making other good nodes susceptible to the two problems above.

Once good nodes rely on the services of malicious nodes, the malicious nodes will have many options available to them as to what type of damage they would like to incur. For instance, the malicious nodes could conduct a distributed denial-of-service attack in which they all decide to "die" at the same time, thereby creating a situation in which good nodes have a large number of "dead" cache entries and are unable to service queries from their clients due to the resulting network fragmentation and/or partition. Alternatively, the malicious nodes could continue to offer service to the good nodes, but all respond with inauthentic copies of documents. We do not consider these post-poisoning attacks here. We only cite them to motivate why containment of poisoning is so critical and why techniques that specifically address containment should be deployed as an important line of defense in a GUESS system.

All malicious nodes in the system have the ability to collude as a single large conspiracy. Doing so gives them a maximal advantage in poisoning pong-caches with as many distinct malicious node ids as possible. Each malicious node is aware of the node ids of all other malicious nodes in the system. When a malicious node receives a ping, it responds with a pong that contains malicious node ids chosen from the entire set of all possible malicious node ids.

Our first goal is to maximize the number of live node ids in pong caches in the steady-state. We say that a GUESS system has achieved "p percent liveness" when over p percent of the node ids in all of the pong caches of good nodes in the system are not dead.

Our second goal is to mitigate pong-cache poisoning, for the reasons described above. We have two subgoals: 1) limit (upper bound) the number of cache entries containing malicious node ids in the steady-state, and 2) reduce the rate at which poisoning occurs.

Note that we are interested in the steady-state behavior of the protocol because if poisoning cannot be controlled in the steady-state in our idealized model of the protocol, there may be little hope of controlling poisoning in an oscillating, real-world system that implements the protocol. However, if we can find policies that work well in the steady-state in our idealized model, such policies may be good candidates for experimentation with in real-world systems.

4. PROTOCOL POLICIES

In this section, we describe in more detail the various protocol policies required in GUESS. Table 1 summarizes the policies and the options that we explore in this paper.

4.1 Seeding Policy (SP)

When a new node wants to join the network (e.g., when it is born), it must initialize, or *seed* its pong cache.

We study three seeding policies (SPs), out of many possible such seeding policies, in this paper:

- *Random-Friend (RF).* A new node's cache is seeded with a copy of some random node's cache. It is possible that a node chosen randomly could be malicious, and could seed a good node's cache with all malicious entries. The probability that a malicious node is chosen as a random friend increases

Table 1: GUESS Protocol Policies

Policy	*Options*
Seeding Policy (SP)	Random-Friend (RF), Popular-Node (PN), Trusted Directory (TD)
Introduction Protocol (IP)	Enabled or Disabled
Cache Replacement Policy (CRP)	Random, Most-Recent-Used (MRU), Least-Recent-Used (LRU)
ID Smearing Algorithm (IDSA)	Enabled or Disabled

as the ratio of the number of malicious nodes to good nodes increases.

- *Popular-Node (PN).* A new node's cache is seeded, without loss of generality, with a copy of N_1's cache. Node N_1 is the "popular" node. We assume that N_1 is not a malicious node, else newly born nodes would have all of their entries poisoned immediately. However, some of N_1's entries may become poisoned as it issues pings and processes pongs itself.

- *Trusted-Directory (TD).* A new node's cache is seeded with node ids that are guaranteed to be non-dead (but could be malicious). A "trusted-directory" node is responsible for maintaining a list of node ids that are guaranteed to be live. Similar to a PN, we assume that a trusted directory itself is not malicious. While a TD SP may not be practical in real systems, we include it as a basis for comparison.

In a real GUESS system that uses popular nodes or trusted directory nodes for seeding, there exist many seeding nodes in the system, such that seeding nodes are not single points of failure. However, to study the basic trends that result from using either the PN or TD SP, we use a single seeding node in our evaluations in Section 5. In addition, since other nodes rely on seeding nodes to initialize their caches, it is especially critical for seeding nodes to use containment techniques to minimize the number of poisoned entries in their caches.

4.2 Introduction Protocol (IP)

Nodes that are newly born do not have their node ids appear in the pong caches of any of the existing nodes. While newly born nodes can query existing nodes because they are seeded appropriately upon birth, we also need a mechanism by which existing nodes can be given the opportunity to query newly born nodes. In this section, we propose a mechanism by which GUESS may "introduce" the node ids of newly born nodes into the pong caches of existing nodes. An introduction protocol (IP) is not part of the original GUESS protocol, but we show that an IP is essential to achieve a steady-state in which most cache entries are live.

The GUESS specification envisions deployment in a hybrid fashion together with the traditional Gnutella flooding protocol. In such a scenario, existing nodes can resort to broadcasting ping messages to allow them to discover newly born nodes. However, doing so requires that the traditional protocol (and all of the performance and security problems associated with it) be carried forward as the network evolves to incorporate GUESS. Hence, we feel it is important to study how to use an IP such that node discovery can take place without any reliance on the traditional protocol.

A natural opportunity at which to introduce a newly born node to an existing node is when a newly born node N_i pings an existing node N_j. After responding to the ping, the node N_j enters N_i into its cache. N_j uses its cache replacement policy (see Section 4.4) to decide which existing entry in its cache to replace.

If an IP is not used, malicious nodes have no incentive to issue pings since it will not help them propagate their node ids into the caches of good nodes. If an IP is used, however, then malicious nodes *do* have an incentive to issue pings, as it can help them propagate their node ids into the caches of pinged nodes.

4.3 Ping Probe and Pong Choice Policies

A node N_i must decide which node N_j to ping. The algorithm that a node uses to decide which node N_j to ping is its *ping probe policy (PPP).* If a node N_i pings N_j and does not receive a response within a preset timeout period, it "marks" N_j as dead. All nodes that receive pings respond with a pong containing up to $|S|$ node ids that are not marked dead. The policy that nodes use to determine exactly which $|S|$ node ids to respond with is called the *pong choice policy (PCP).*

There are many possible options that nodes might have for ping probe and pong choice policies. We conducted some preliminary simulations that experimented with various PPP and PCP options such as most-results-first, most-recently-pinged, and random. We found that the settings for these policies did not have as significant an effect on liveness or poisoning as did the other types of policies that we describe, and we only consider using random PPP and PCPs in this paper. Note that, however, different choices for PPPs and PCPs may have an affect on search performance, and various options for PPPs and PCPs are studied in reference [1].

4.4 Cache Replacement Policy (CRP)

Once N_i receives up to $|S|$ node ids from N_j, it chooses to replace $r \leq |S|$ entries in $P(N_i)$ to form $P'(N_i)$. In other words, N_i chooses $Y \subseteq S$ and $|X| = r$. Node N_i first replaces dead cache entries with those from the set Y. N_i then chooses the contents of the set X using a CRP. We consider three CRPs in this paper:

- *Random.* N_i randomly chooses up to $|S|$ entries from its pong cache $P(N_i)$. Node N_i excludes any entries that are marked dead from its choice.

- *Most Recently Used (MRU).* N_i maintains a *last-time-pinged* timestamp associated with each cache entry, and returns the $|S|$ non-dead entries in $P(N_i)$ that have the *highest* last-time-pinged timestamps.

The last-time-pinged timestamp is set to the current time when a cache entry is chosen to be pinged.

- *Least Recently Used (LRU).* N_i maintains a *last-time-pinged* timestamp associated with each cache entry, and returns the $|S|$ non-dead entries in $P(N_i)$ that have the *lowest* last-time-pinged timestamps. The timestamp is updated similarly.

4.5 ID Smearing Algorithm (IDSA)

In addition to the simple CRPs just described, we develop a more sophisticated cache management policy to help deal with malicious nodes. In particular, if node N_i receives a node id N_k as a response in pongs from many nodes, then it may be the case that either 1) N_k is a malicious node and is working to have its id propagated to as many good node caches as possible as a pre-cursor to an attack, or 2) N_k is a good node that is likely to become overloaded with too many queries because other nodes happen to be including N_k's id in pongs. In either case, it is in the best interest of the good nodes to balance out the number of times that any particular node id appears in the pong caches of other good nodes. That is, good nodes want to replace entries in their pong cache such that for any of the n node ids in the system, each of the ids appears in the pong caches of good nodes with a frequency proportional to $1/n$.

To help accomplish this, we use an "ID smearing" algorithm (IDSA). The algorithm evenly "smears" all of the available node ids across all of the pong caches by doing the following locally at each node. When N_i receives N_k in a pong message from N_j, it checks whether or not N_k already appears in its cache, $P(N_i)$. If $N_k \in P(N_i)$, then there may be too many copies of N_k in the network, and we set $P'(N_i) = P(N_i) - \{N_k\}$. For example, if $P(N_1) = \{N_2, N_3\}$, and N_1 receives N_3 in a pong message from N_2, N_1 updates its pong cache such that $P'(N_1) = P(N_1) - \{N_3\} = \{N_2\}$.

In Section 5, we conduct evaluations that show the IDSA can mitigate pong-cache poisoning attacks. However, malicious nodes can attempt to manipulate good nodes that use the IDSA. For instance, a malicious node M may happen to know that a particular good node id G_2 is in another good node G_1's pong-cache. If M receives a ping from G_1, it can respond with a pong containing G_2, causing G_1 to remove a good node id from its cache, and have one less live id. If an IP is used, M can then instruct another malicious node to introduce itself to G_1 and take the place of G_2's node id in the cache. However, this attack requires M to keep track of which ids it has received in pongs from which nodes and/or communicate this information to other malicious nodes. M may also need to retain this information for quite some time until some malicious node is pinged by the appropriate good node. The IDSA nevertheless does "raise the bar" of effort required on behalf of malicious nodes to poison caches.

4.6 Dynamic Network Partitioning (DNP)

In this subsection, we propose a *dynamic network partitioning (DNP)* technique whose goal is to complement the IDSA in keeping the number of malicious nodes that good nodes interact with low.

The key idea that we employ in DNP is that nodes do not necessarily need to search the entire network at any particular instant. If we can limit the subset of nodes that are searched at any given time, then we can also limit the number of malicious nodes that can impact the search. As search proceeds, the partition of the network that is searched (the "active" partition) can change.

At any given time, a node divides the node id address space into a number of partitions and selects one of those partitions to be its active partition. A node only accepts node ids into its pong cache from the active partition.

The method that we use to partition the network works as follows. We assume that ids are j-length bit strings. Each partition is of size 2^p, where a node has the freedom to choose p such that $0 \leq p \leq j$. There are 2^{j-p} partitions. (If $p = j$ then DNP is not used as the size of the partition is the size of the entire network.)

Each node chooses a random key, κ, and an active partition, ϕ, $(0 < \phi < 2^{j-p})$. A node only accepts an id a into its pong cache if it falls into partition ϕ. The nodes that make up partition ϕ are those nodes that have an id such that the value of the $j - p$ least significant bits of $h(a||\kappa) = \phi$, where h is a hash function and the symbol $||$ is concatenation.

Our DNP scheme allows good nodes to dynamically change how they partition the network much more easily than the amount of effort required by malicious nodes to acquire node ids in different partitions. Acquiring node ids involves some reasonable (but not necessarily prohibitive) cost for malicious nodes. For instance, in order to receive pings and send pongs, a malicious node needs to be able to receive traffic at an id that it either owns or has acquired by compromising an Internet host. Malicious nodes need to expend some effort to acquire ids, and, once acquired, they will not be able to easily spoof a different set of ids immediately.

The parameter κ allows each good node to partition the network in a way that is dynamic and unpredictable. By using DNP, each good node can use a different key κ such that malicious parties are not able to predict which partitions they need to acquire ids in to poison the caches of particular good nodes, and good nodes can periodically change the key that they use to thwart any incoming attack mounted by malicious nodes that have already acquired a particular set of ids.

Note that we do not assume that malicious nodes are clustered in any way. Malicious nodes could be randomly distributed across the node id space, in which case only a fraction of malicious node ids can appear in a good node's cache at a particular time. Alternatively, if malicious nodes are clustered, then they will only be able to affect the part of the search through a particular partition.

4.7 Malicious Node Detection

The policies described thus far attempt to use various cache management strategies to mitigate poisoning. In addition to studying how cache management can help us achieve our goals, we also study malicious node detectors (MNDs). Our goal in this section is to understand how well our cache management mechanisms perform compared to MNDs, and to understand under what conditions we will be required to use MNDs in addition to cache management techniques.

A MND takes a node id as input, and attempts to determine if the corresponding node is malicious. A MND tries to make this determination based on past experience interacting with the node, possibly taking into account the quality and utility of search results received from the node, or using such information from other more "trusted" nodes. Various algorithms based on reputation systems and other mechanisms have been proposed (i.e., [11]), and can serve as MNDs. We do not advocate the use of any particular proposal of a MND, but we are simply interested in how MNDs may help deal with some types of attacks. For instance, MNDs may help us deal with attacks in which malicious nodes send good nodes inauthentic results. (On the other hand, MNDs would not help us in a denial-of-service attack in which malicious nodes poisoned caches until some critical fraction of all good node caches contained malicious ids, and then all the malicious nodes simultaneously disappeared from the network. MNDs would not help in such a denial-of-service attack because the behavior of malicious nodes may be indistinguishable from that of good nodes until the point at which they all disappear, in which case it would be too late for a MND to detect anything.)

We model the accuracy of a MND using a single parameter, p, the probability that the MND will tell us that a node is malicious given a malicious node id. We assume that if a MND is used, it is used as follows: when a node N_i receives a pong from N_j, it submits all of the ids received to the MND before considering them for insertion into its pong cache. In addition, if N_i uses an IP and receives a query from N_k, then N_k's node id is submitted to the MND before its introduction into N_i's pong cache is considered.

The quantity $1 - p$ is the false negative rate of the MND. Practical implementations of MNDs may also have a false positive rate, a probability with which a good node id is deemed malicious. In future work, the effect of MNDs with various false positive rates can be studied, but due to space limitations, we only consider MNDs with false negative rates in this paper.

5. RESULTS

In this section, we present the results of various evaluations that we conducted using a discrete-event simulation of the model described above to address the goals presented in Section 3. While we present the highlights of our findings here, the interested reader is referred to our extended paper [6] for more complete details.

To facilitate simulation, we extended the sets we described in Section 2 to be functions of time. The set of nodes N, is a function of time, $N(t)$, in our simulation. The pong caches $P(N_i), \forall i$ are also functions of time, $P(N_i, t), \forall i$, *etc.* Time advances in discrete-intervals $t = 0, 1, 2, etc.$ Our discrete-event model can approximate the continuous behavior of a real system as the physical time between intervals decreases.

Table 2: Baseline Simulation Parameters

Parameter	*Value*
Number of Nodes (at any one time) (n)	100
Pong Cache Size (c)	5
Births / Deaths Per Round (b)	1
Number of Node Ids in Pongs (r)	1
Number of Experiments	100
Ping Probe Policy (PPP)	Random
Pong Choice Policy (PCP)	Random
Malicious Ping Rate (MPR)	1

We say that all of the events that occur in one time step occur in a "round." If node N_i pings N_j, receives N_k in a pong, and updates its pong cache at time t to additionally contain node N_k, then we say that $P(N_i, t+1) = P(N_i, t) \cup \{N_k\}$. The ping, pong, and cache updates for all of the nodes in the system at a given time t happen sequentially within the same "round."

5.1 Simulation Setup

Our goal in this work is to build a fundamental understanding of the issues and trade-offs involved in using the various policies we outlined in Section 4 to mitigate pong-cache poisoning. *Our evaluations are not designed to predict the performance of an actual system, but to gain an understanding of the trends and trade-offs involved in using the different policies.* In the evaluations described below, we simulated a GUESS network using the baseline parameters in Table 2.

We estimate that nodes in a GUESS network will conservatively be able to store approximately five percent of the live node ids in the network in their pong caches. If a single pong cache entry requires 10-bytes of storage (4 bytes for an IP address, and 6 bytes of additional storage for timestamps and other data), then storing pong cache entries for five percent of the nodes in a 2,000,000 node GUESS system only requires 1 megabyte of main memory. Such a memory requirement is even becoming acceptable for traditionally constrained wireless devices such as cell phones and PDAs. Larger pong caches will result in higher numbers of live ids and a smaller ratio of poisoned ids to live ids than those that we present in the simulations in this section. Additional state may also be used to store performance-related data about some of the node ids in the pong cache, such as the number of query hits received from a given node, but such additional state is only required for a relatively small percentage of the ids in the pong cache.

In our simulations, there are $n = 100$ nodes in the network at any instant, and they have pong caches that can store $c = 5$ node ids. We confirmed that the same trends that we see in our small 100-node simulations can be seen in larger GUESS networks (see our extended paper [6]), and we clearly expect real GUESS networks to have much larger numbers of nodes.

The settings for all of the policies used in each simulation that we present in this paper are shown in Table 3. Each column of Table 3 corresponds to one of our simulations, and we refer to them by the column names in the results below.

Table 3: GUESS Simulation Parameters (Var=Variable, PN=Popular Node, D=Disabled, E=Enabled)

Parameter/Simulation	A1	A2	B	C1	C2	D1	D2
SP	Var	Var	PN	PN	PN	PN	PN
IP	D	E	E	E	E	E	E
CRP	Random	Random	Var	Random	Random	Random	Random
IDSA	D	D	D	Var	E	D	E
Num Rounds	500	500	100000	2000	2000	5000	5000
Mal Nodes (m)	0	0	5	Var	5	Var	Var
MPR	N/A	N/A	1	1	Var	1	1

We assume that each of the good nodes issues a single ping in every round. While this need not be the case in a real network at all times, it allows us to model the effect of pong-cache poisoning when the network is at its "busiest." Also, when an IP is used, we assume malicious nodes issue pings. Malicious nodes may issue more ping messages than good nodes do. The number of pings that each malicious node issues in one round is the *malicious ping rate (MPR)*. While the baseline MPR that we use is 1, we study the impact of malicious nodes that issue pings more frequently in Section 5.5.

For accounting purposes, we assume that malicious nodes have pong caches that are of the same size as good nodes, but that their entries are always poisoned. Furthermore, all good nodes use the same SP, IP, PPP, PCP, CRP, and IDSA settings.

To produce the various graphs we present here, the corresponding simulations were run 100 times, and the results were averaged. The y-axis on the graphs typically measures the total number of live or poisoned pong cache entries in the system.

After describing some simplifying assumptions that we made in our simulations, our first order of business is to analyze which protocol options achieve our first goal of maximizing the number of live node ids in pong caches, without the presence of malicious nodes. We then evaluate which policies minimize susceptibility to attacks. Our simplifying assumptions are as follows:

5.1.1 Deaths

We assume that one good node dies at random, and one new good node is born in place of it each round. That is, $\delta = |\Delta(t)| = |B(t)| = 1$, $G(t+1) = (G(t) - \Delta(t)) \cup B(t)$, and $N(t+1) = G(t+1) \cup M(t+1), \forall t$. Furthermore, we assume that nodes that are born are newly born, and have never previously participated in the system. Nodes that die are not re-born later. In other words, $B(t) \cap (G(0) \cup G(1) \cup ... \cup G(t)) = \emptyset, \forall t$. Therefore, we assume that good nodes are born and die at a constant and equal rate. While this may not be true for a system in which the number of nodes is growing or shrinking, we roughly expect this to be true when the system achieves steady-state. As we recall, our goal is not to model the performance of an actual system, but to understand what steady-state trends occur with respect to cache poisoning when using the policies from Section 4.

5.1.2 Malicious Nodes

While good nodes may be susceptible to death, we assume that the set of malicious nodes is constant: $M(0) = M(1) = ... = M(t), \forall t$. The set of live good nodes $G(t)$, on the other hand, changes over time, but its cardinality is constant, $|G(0)| = |G(1)| = ... = |G(t)|$. While we do not vary the number of malicious nodes in the system in the midst of a simulation, we do study how different numbers of malicious nodes impact our ability to mitigate attacks. In evaluations in which we studied malicious nodes, we measured how the number of poisoned cache entries increases as many nodes become malicious.

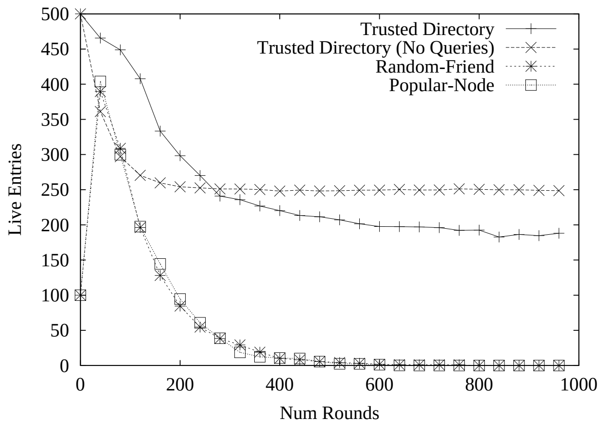


Figure 2: A1: Live Entries vs. Number of Rounds for Various Seeding Policies (No Introductions)

5.2 Seeding and Introductions

R1. A SP must be used in tandem with the IP to achieve liveness. When SPs are used in combination with the IP, a high percentage liveness can be achieved (e.g., a liveness of 95 percent was achieved using our baseline simulation parameters).

Figure 2 shows the results of simulation A1 in which we measured the number of live entries for the three SPs described in Section 4, with no malicious nodes in the system. Simulation A1 was run with no IP, a random CRP, and the IDSA disabled.

At time $t = 0$, seeding from the trusted directory results in 500 live node ids since newly born nodes are given $c = 5$ guaranteed-to-be-live entries from the directory. However, when the RF or PN policies are used, we initialize all of the pong caches at $t = 0$ to contain just one live node id, and the system starts out with a total of 100 live ids.

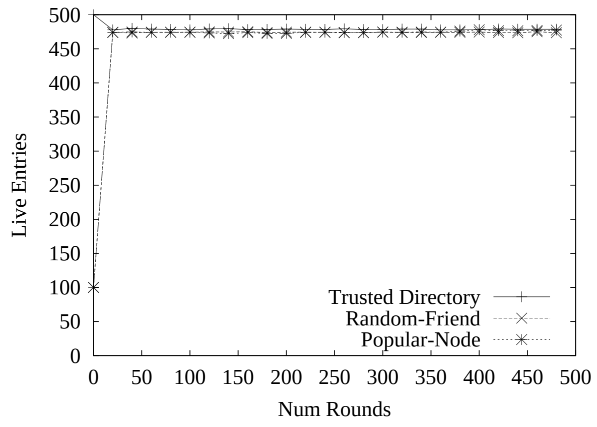


Figure 3: A2: Live Entries vs. Number of Rounds for Various Seeding Policies (With Introductions)

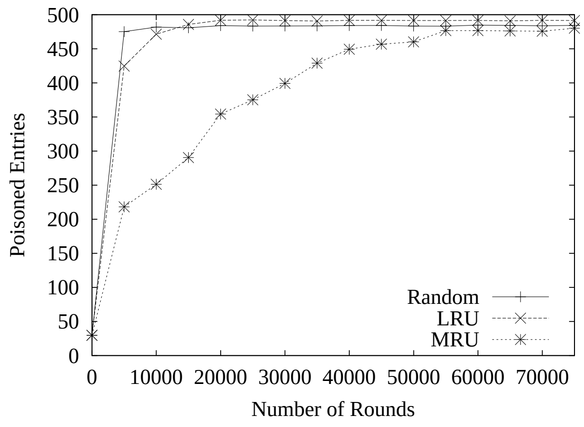


Figure 4: B: Poisoned Ids vs. Number of Rounds for Various CRPs

As time advances, the number of live node ids in the TD case drops because some nodes die, and the ids pointing to them become invalid. In the case of the RF and PN policies, the number of live ids in the system starts increasing as nodes exchange pings and pongs. Unfortunately, after some number of rounds, the RF and PN policies suffer from the same problem. In both policies, when new nodes are born, they seed their caches by copying the caches of some other node. Over time, as nodes die, all of the node ids contained in the pong caches of the initial set of nodes in the system die. Since newly born nodes are simply copying the cache entries that originated from the initial set of the nodes in the system, all of these caches entries are bound to die as well. In other words, in the current scenario, newly born nodes only learn about nodes that existed before them, and never learn about nodes that are born after they are.

Nodes must use an IP if the "current generation" of nodes is to learn about "future generations" of nodes. Figure 3 shows the outcome of simulation A2 in which we employed an IP. The figure shows that any of the SPs we considered can be used to achieve over 475 live node ids out of a possible 500 in steady-state. In other words, we achieved a liveness of 95 percent when SPs were used in combination with an IP.

We also studied the different rates at which poisoning occurs with the RF and PN SPs, and we found that poisoning occurs more slowly with PN than with RF SP. Poisoning occurs slower with PN SP because the popular node must first be poisoned before newly born nodes can be poisoned. We refer the interested reader to our extended report [6] for more details. In the remainder of our evaluations in this paper, we use PN SP by default.

5.3 Cache Replacement Policy (CRP)

R2. While all the CRPs that we study result in inevitable cache poisoning, MRU CRP slows the rate of poisoning the most.

Figure 4 is the result of simulation B in which we held all simulation parameters constant and varied the CRP. The simulation measures the increase in the number of poisoned node ids over an extended number of rounds in a system with $m = 5$ malicious nodes for various CRPs. The figure shows that almost all of the cache entries become poisoned after about 60,000 rounds. The poisioning of almost all of the entries is "inevitable." In our extended paper [6], we present an analysis explaining why this is the case for a random CRP and a TD SP. The result of the analysis is that if a GUESS system is initialized with a non-zero number of malicious nodes and pong-caches are populated with a uniform random distribution of node ids, then, during each ping-pong exchange that a good node conducts, the probability that the number of malicious entries in its cache will increase is always larger than the probability that the number of malicious entries in its cache will either stay constant or decrease. The rate of poisoning and the inevitability of it may vary based on which CRP and SP is used.

From our results in Figure 4, we can see that using non-random CRPs can *slow* the rate of poisoning. When LRU CRP is used, cache entries are discarded (replaced) in the order in which they are pinged. When a particular node is pinged, its id is "pinned" in the cache, and becomes a candidate for being pinged again for up to $c - 1$ rounds. Poisoning occurs slightly slower with LRU CRP than random CRP because a live, good node that is pinged cannot be "randomly" replaced. A malicious node id cannot take the place of a pinged, good node id for $c - 1$ rounds. The good node may even be pinged again, and so long as it is live, can continue to exist in the cache and prevent a malicious node id from taking its place.

MRU CRP slows poisoning more effectively than LRU or random CRP. When a malicious node is pinged under a MRU CRP, its id is replaced by the malicious id from the pong. The number of malicious node ids does not increase. Furthermore, when the next introduction is received, the malicious node id is replaced (since it is the most recently used). If the node introducing itself is not malicious, then the number of poisoned ids decreases. Otherwise, the number of poisoned ids in the cache stays the same. The only case in which the number of malicious nodes in a good node's cache increases is when the good node pings another good node that died since it was either inserted in the cache or pinged last, and it immediately receives an introduction from a malicious node. In this case, the malicious node id takes the place of the good node that is marked dead. Hence, while inevitable cache poisoning does occur with MRU CRP, it occurs slower than with LRU or random CRP.

The "catch" with MRU CRP is that it leads to bad search performance [1] since many cache entries are not updated and die before their next use. Furthermore, live node ids that are introduced into the pong cache via an IP are subsequently replaced with ids that are more likely to be stale from pong messages. To deal with this problem, we suggest dividing the pong cache into two caches– a small pong cache and a large introduction cache. Node ids received in pongs are placed in the pong cache, and the pong cache is managed using a MRU CRP to slow poisoning. Node ids received from introductions are placed in the introduction cache. Node ids in the introduction cache are highly likely to be live since these nodes recently issued queries, and the introduction cache is managed using a FIFO (first-in-first-out) CRP. If many of the entries in the pong cache are dead, ids from the introduction cache can be used to issue queries and provide reasonable search performance. [1]

In the next section, we show that by taking advantage of an IDSA, we can not only slow the rate of poisoning, but we can limit it in the steady-state as well.

5.4 ID Smearing Algorithm (IDSA) and Dynamic Network Partitioning (DNP)

R3. The use of the IDSA limits the number of poisoned entries in the steady-state, and the IDSA mitigates poisoning as the number of malicious nodes increases up to the cache size. DNP can be used together with IDSA to mitigate poisoning as the number of malicious nodes increases beyond the cache size.

Figure 5 shows the results of simulation C1, and demonstrates that an IDSA limits the number of poisoned cache entries *in the steady state* when $c = m = 5$. While just over 450 cache entries are poisoned when the IDSA is not used, approximately 107 cache entries are poisoned when the IDSA is used. The number of poisoned entries is reduced by a factor of four when the IDSA is used and $c = m$. The IDSA is successful in mitigating poisoning because the use of the IDSA causes malicious nodes to "limit their own success" in poisoning caches. If a good node N_g has two poisoned cache entries, N_{m1} and N_{m2}, then when one of the node ids in these cache entries is pinged, say N_{m1}, there is

[1]This dual-cache scheme is similar to that in reference [1]. The pong cache is equivalent to the link cache, and the introduction cache is analagous to the query cache, except the source of the ids are introductions and not pongs.

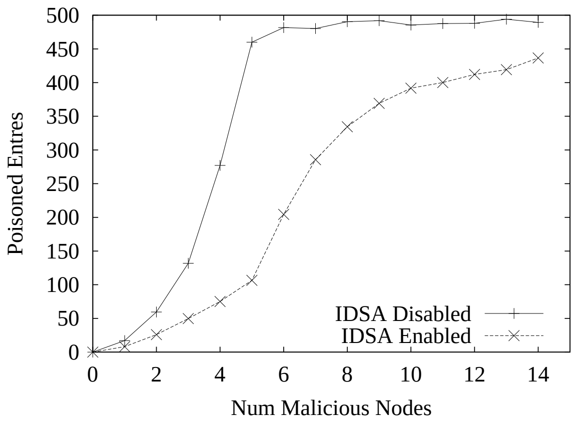


Figure 5: C1: Poisoned Ids vs. Num Malicious Nodes with IDSA

a non-negligible probability that N_{m1} will return N_{m2}'s id, causing N_g to remove N_{m2} from its cache as required by the IDSA. The more malicious entries that a good node's cache contains, the higher the probability that, when pinged, one of these malicious nodes will return an id of one of the other malicious nodes in the cache, and end up causing the good node to remove a malicious id from its cache. A similar effect that occurs (but is less likely) takes place when N_g's cache contains N_m, and receives an introduction (a ping) from N_m itself. Use of the IDSA results in removing N_m from the good node's cache. For these reasons, the success of malicious nodes' poisoning efforts is self-regulated when the IDSA is used.

We assume that a malicious node returns the id of another malicious node at random. Malicious nodes could try to work harder at poisoning by keeping track of exactly which good nodes have received which malicious node ids, such that malicious nodes can collude to ensure that the same malicious node id is never provided to the same good node twice. This could be done in an attempt to thwart the good node's use of the IDSA. However, good nodes could decide to drop ids out of their caches randomly from time to time to easily thwart this attack, at the expense of application performance. Malicious parties would then need to obtain more malicious node ids to poison the good nodes because at any time they would be unsure as to whether or not sending an already used malicious node id would end up causing that malicious node id to be dropped out of the good node's cache or re-inserted.

We can see from Figure 5 that as the number of malicious nodes increases beyond $c = 5$, the IDSA becomes less and less effective. Since the IDSA only eliminates node ids from a cache when it receives an id that is a duplicate of one of its entries, the more unique ids the malicious nodes have, the fewer malicious entries the IDSA is able to eliminate.

IDSA is able to significantly mitigate poisoning so long as the number of malicious nodes does not exceed the cache size ($c = 5$, in Figure 5). For instance, if only 5 percent of the nodes in the system are malicious, approximately 10 percent of a good node's cache is poisoned. However, if the number of malicious nodes is twice the cache size and 10 percent of the nodes are malicious, over 75 percent of a good node's cache is poisoned. To deal with a number of malicious nodes that exceeds the good node's cache size, DNP (as described in Section 4.6) can be employed to keep the number of poisoned entries low. If we expect that the number of malicious nodes in the network is greater than the cache size, then we can partition such that the expected number of malicious nodes per partition is no greater than the cache size. For instance, if we expect that 10 percent of the nodes in the network are malicious, then 2 partitions can be used to keep no more than 10 percent of the good node's caches from being poisoned.

In general, with high probability, it is the case that if m random malicious node ids are in the network, then on average, at most $\frac{m}{2^{j-p}}$ will appear in a node's pong cache. If DNP is not used, then as long as $c > m$, IDSAs significantly mitigate poisoning, whereas when DNP is used, then as long as $c > \frac{m}{2^{j-p}}$, IDSAs are able to significantly mitigate poisoning. The use of DNP allows clients to be able to tolerate more malicious nodes in the network. The downside of using DNP, however, is that it makes it harder for good nodes to systematically search the entire network, due to the fact that partitions created by different values for κ may create overlapping partitions.

5.5 Malicious Ping Rate (MPR)

R4. The use of the IDSA limits the number of poisoned entries in the steady-state, even when malicious nodes issue pings more frequently than do good nodes.

In this subsection, we vary the malicious ping rate (MPR), and measure the effect on the number of poisoned entries. When an IDSA is not used, the rate at which poisoning occurs increases proportionally when malicious nodes increase MPR. That is, malicious nodes will be able to poison a given number of entries in the system in a fewer number of rounds if they increase their MPR.

However, when the IDSA is used, we find that even if malicious nodes increase MPR, the number of cache entries that can be poisoned *in the steady state* does not increase. Figure 6 shows how the number of poisoned entries varies over time when the IDSA is used for different MPRs. Each of the curves in the figure corresponds to different MPRs. At $t = 0$, the number of poisoned entries is over 25 for each of the curves because there are $m = 5$ malicious nodes in the system with a cache size of $c = 5$ each, and a few good nodes have some malicious node ids randomly distributed in their caches.

In the first 50 rounds, we can see from the figure that the higher the MPR, the more cache entries are poisoned. For instance, if the MPR is one, approximately 125 entries are poisoned when $t = 50$, whereas if the MPR is 20, just under 300 entries are poisoned. When malicious nodes issue pings more frequently, they are able to have more of their ids inserted into the caches of good nodes for the first time due to the use of the IP. However, continued pinging by malicious nodes results in having their ids eliminated out of good node caches due to the use of the IDSA. As the system progresses towards steady-state, the more frequently that malicious nodes issue pings, the fewer steady-state poisoned entries they are able to maintain. For instance, with a MPR of one, malicious nodes are able to maintain approximately 100 poisoned entries at $t = 2000$. If malicious nodes issue pings 20 times as often, they are able to maintain only 65 poisoned entries at $t = 2000$. Hence, the use of the IDSA limits the number of poisoned entries in the steady-state, even when malicious nodes issue pings more frequently than do good nodes.

There may be some advantages for malicious nodes to issue pings more frequently than good nodes. Specifically, the higher the MPR, the higher is the peak number of poisoned entries. If MPR is 20, for example, then the peak number of poisoned entries is 290, whereas if MPR is one, then the peak number of poisoned entries is about 125. Malicious nodes could use high MPRs to quickly poison a significant number of cache entries, and execute their attack at the point that the maximum number of entries has been poisoned. For instance, malicious nodes using an MPR of 20 could all leave the system when the peak number of cache entries are poisoned to execute a denial-of-service attack that leaves the network fragmented. Of course, malicious nodes will have to determine when the peak number of cache entires is poisoned.

From Figure 6, we can also see that the higher the MPR, the "sharper" is the peak. At a MPR of 20, for instance, over 250 cache entries are in the poisoned state for only 100 rounds. By contrast, at a MPR of 5, just over 200 cache entries are in the poisoned state for approximately 500 rounds.

What we can learn from the sharpness of the peaks is that if malicious nodes want to quickly poison as many cache entries as possible, and then execute, say, a denial-of-service attack in which they all disappear, the effectiveness of the attack will be dependent upon how well and how quickly malicious nodes can sample the number of poisoned entries in the system. If malicious nodes issue pings at a high rate, they will have a small window of time during which it will be optimal for them to execute their attack.

On the other hand, if malicious nodes are interested in conducting an attack over an extended period of time, the use of the IDSA prevents them from having too many poisoned entries in the steady-state regardless of how fast or slow they issue pings. For instance, if the malicious nodes would like to provide inauthentic results over an extended time period, the IDSA will limit the number of cache entries that the malicious nodes can "own" and use in mounting their attack.

5.6 MND Results

The IDSA limits poisoning by attempting to distribute node ids equally across all of the pong caches. While we are able to eliminate a significant amount of poisoning with the IDSA, if we would like to further eliminate poisoning, we may need to employ a MND. Result R5 tells us how well a MND used on its own (without the IDSA) mitigates poisoning, and we com-

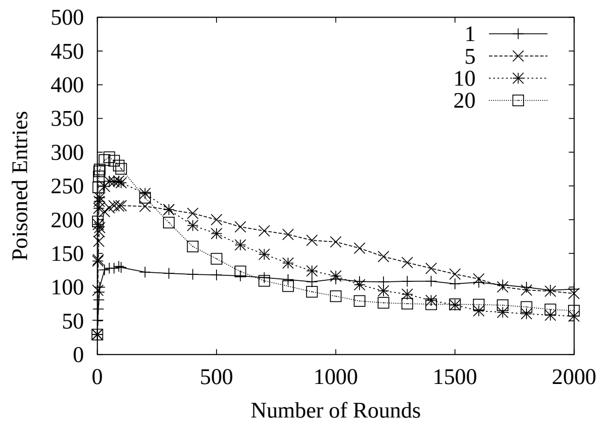


Figure 6: C2: Poisoned Ids vs. Time for Various MPRs with IDSA

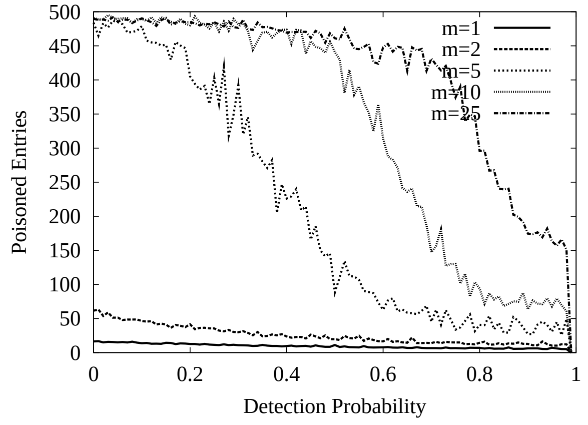


Figure 7: D1: Malicious Node Detection with IDSA Disabled

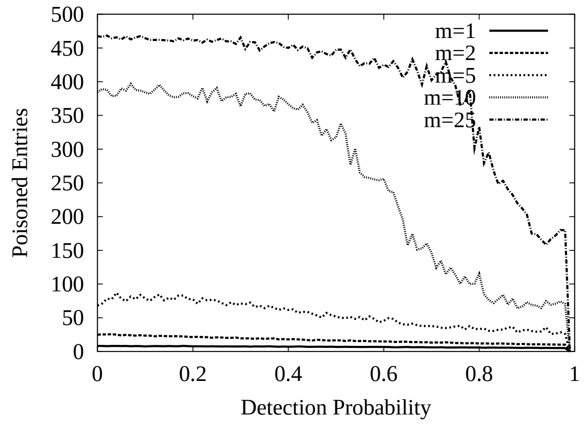


Figure 8: D2: Malicious Node Detection with IDSA Enabled

pare the performance of a MND to the performance of the IDSA. Result R6 considers using a MND and the IDSA together.

R5. A MND needs to be at least 50 percent accurate to achieve equal or better results than the IDSA when $c = m$.

Figure 7 shows how the number of poisoned entries in the steady state decreases as the accuracy of a MND increases. If the number of malicious nodes in the system is equal to the cache size ($m = 5$), we can see that a MND with an accuracy of 50 percent results in about 100 poisoned entries in the steady state. Figure 5 discussed in Section 5.4 shows that enabling the IDSA (without using a MND) also results in about 100 poisoned entries (when $c = m = 5$).

R6. A MND may be required in systems in which the number of malicious nodes is greater than the pong cache size.

Figure 8 shows the level of steady state poisoning that can be achieved when both the IDSA and a MND are employed. If the number of expected malicious nodes in the system is on the order of the cache size ($m = 5$), we can see that employing a MND results in a marginal benefit even if the detector is over 90 (but not 100) percent accurate. On the other hand, as the number of malicious nodes increases significantly beyond the cache size, detectors of increasing accuracies can significantly reduce poisoning.

DNP can also be used to mitigate poisoning as the number of malicious nodes increases beyond pong cache size, but DNP does make it harder for good nodes to systematically search the entire network due to periodic re-keying. On the other hand, MNDs are likely to be more complicated to implement than DNP.

6. RELATED WORK

Much research on security in P2P systems has worked to mitigate attacks against four distinct, but related, system properties: availability, authenticity, access control, and anonymity. As explained in Section 3, our work here focuses on addressing attacks against the first two of these system properties, availability and authenticity, in a GUESS P2P network.

While GUESS is just beginning to be studied by the academic research community [19, 1], some research does exist on security in traditional Gnutella systems. In particular, references [7] and [5] study availability and authenticity issues in traditional Gnutella. Work has also been done that studies how to use a P2P network to prevent DoS attacks on the Internet [12]. Papers such as [4] and [16] study how to use P2P networks to provide anonymity to users. Current P2P networks are plagued by digital rights management and access control issues, and reference [2] outlines some of the problems in this area.

While the papers that we have mentioned so far focus mainly on unstructured P2P networks, references [17] and [3] outline how security issues in DHTs might begin to be addressed. For example, reference [3] imposes additional structure in DHT routing tables to reduce the fraction of entries that can be malicious, just as our DNP scheme imposes additional constraints on the pong-cache to reduce poisoning.

We refer the reader to reference [14] for more complete coverage of related work.

7. CONCLUSION

In this paper, we defined a model of a GUESS network. We outlined the key decisions that nodes need to make to discover new nodes and manage their pong caches to mitigate pong-cache poisoning. We ran simulations based on our model, and evaluated different options for the key decisions. Based on the results of some of our simulations, we found that:

- An IDSA can be used to limit poisoning in the steady-state. An IDSA is most effective when the number of malicious nodes is less than or approximately equal to pong-cache size.
- A DNP scheme can be used to reduce the number of malicious node ids that can poison a pong cache at any one time. A DNP can be used together with an IDSA to tolerate a number of malicious nodes that exceeds pong-cache size.
- If the number of malicious nodes is greater than pong cache size, a MND can also be used together with the IDSA to limit poisoning in the steady-state.
- Introductions are essential to achieving liveness. We propose adding introductions as a basic mechanism in GUESS.
- We suggest using MRU CRP to slow down the rate of poisoning. However, we observe that MRU CRP does not limit poisoning in the steady-state.

Finally, while we studied the GUESS protocol in particular, other P2P protocols such as FastTrack and eDonkey use protocols similar to GUESS for resource discovery, and we expect that our results are applicable to such networks as well.

8. ACKNOWLEDGEMENTS

Thanks to Qi Sun and Prasanna Gannesan for insightful discussions on GUESS behavior. Thanks also to the anonymous referees for helping us identify additional strengths of our work, and for pointing us in directions that helped us address limitations of our work.

9. REFERENCES

[1] H. Garcia-Molina B. Yang, P. Vinograd. Evaluating guess and non-forwarding peer-to-peer search. In *24th International Conference on Distributed Computing Systems (ICDCS 2004), Tokyo, Japan*, 2004.

[2] P. Biddle, P. England, M. Peinado, and B. Willman. The darknet and the future of content distribution. Digital Rights Management Workshop 2002, http://crypto.stanford.edu/ DRM2002/ darknet5.doc.

[3] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Security for peer-to-peer routing overlays. In *Fifth Symposium on Operating Systems Design and Implementation (OSDI '02) (Boston, Massachusetts)*, 2002.

[4] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, 2000.

[5] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati, and F. Violante. A reputation-based approach for choosing reliable resources in peer-to-peer networks. In *Proc. of the 9th ACM Conference on Computer and Communications Security*, Washington, DC, USA, November 2002.

[6] N. Daswani and H. Garcia-Molina. Pong-cache poisoning in guess (extended version). http://dbpubs.stanford.edu/pub/2003-51.

[7] N. Daswani and H. Garcia-Molina. Query-flood dos attacks in gnutella networks. In *Proc. of the 9th ACM Conference on Computer and Communications Security*, Washington, DC, November 2002.

[8] S. Daswani and A. Fisk. Guess protocol specification. http://groups.yahoo.com/ group/ the_gdf/ files/ Proposals/ GUESS/ guess_01.txt.

[9] J. Douceur. The sybil attack. IPTPS, 2002.

[10] Gnutella development forum (gdf). http://groups.yahoo.com/ group/ the_gdf/.

[11] S. Kamvar, M. Schlosser, and H. Garcia-Molina. Eigenrep: Reputation management in p2p systems. In *Proceedings of the 12th International WorldWide Web Conference*, 2003.

[12] A. Keromytis, V. Misra, and D. Rubenstein. Sos: Secure overlay services. In *Proceedings of ACM SIGCOMM'02*, Pittsburgh, PA, August 2002.

[13] C. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th International Conference on Supercomputing*, pages 84–95. ACM Press, 2002.

[14] B. Yang. N. Daswani, H. Garcia-Molina. Open problems in data-sharing peer-to-peer systems. In *International Conference on Database Theory. Siena, Italy.*, 2003.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, San Diego, CA, August 2001.

[16] M. Reiter and A. Rubin. Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[17] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *IPTPS*, Cambridge, MA, USA, March 2002.

[18] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, pages 149–160, San Diego, CA, August 2001.

[19] D. Tsoumakos and N. Roussopoulos. Adaptive probabilistic search (aps) for peer-to-peer networks. http://citeseer.nj.nec.com/568292.html.