Verme: Worm Containment in Overlay Networks

Filipe Freitas[1], Edgar Marques[1], Rodrigo Rodrigues[2], Carlos Ribeiro[1], Paulo Ferreira[1], and Luís Rodrigues[1]

[1]*INESC-ID / Technical University of Lisbon*
[2]*Max Planck Institute for Software Systems (MPI-SWS)*

Abstract

Topological worms, such as those that propagate by following links in an overlay network, have the potential to spread faster than traditional random scanning worms because they have knowledge of a subset of the overlay nodes, and choose these nodes to propagate themselves; and also because they can avoid traditional detection mechanisms. Furthermore, this worm propagation strategy is likely to become prevalent as the deployment of networks with a sparse address space, such as IPv6, makes the traditional random scanning strategy futile.

We present a novel approach for containing topological worms based on the fact that some overlay nodes may not have common vulnerabilities, due to their platform diversity. By properly reorganizing the overlay graph, this can lead to the containment of topological worms in small islands of nodes with common vulnerabilities that only have knowledge of themselves or nodes running on distinct platforms. We also present the design of Verme, a peer-to-peer overlay based on Chord that follows this approach, and VerDi, a DHT layer built on top of the Verme routing overlay.

Simulations show that Verme and VerDi have a low overhead when compared to Chord's corresponding layers, and that our new overlay design helps containing, or at least slowing down the propagation of topological worms.

1. Introduction

In recent years, we have witnessed the outbreak of several Internet worms that have not only caused inconvenience to many users, but also a large societal impact. Many of these are instances of "random scanning worms". This means that once the worm has infected a host, it propagates itself by probing random IP addresses for new nodes to infect. In some cases this choice of IP addresses is biased by some heuristic that increases the chances of finding an IP address that is in use, like choosing other IP addresses in the same network as the infected host [21].

An alternative strategy would be for the worm to discover and try to infect a set of nodes whose IP addresses are obtained either before the deployment or dynamically as the worm is propagated. An instance of the latter case are *topological worms*. These worms choose their next target to infect by following links in a certain graph, which can, for instance, be formed by overlay links in an application-level network (e.g., a multicast overlay).

We argue that topological worms are likely to become more and more prevalent. With the deployment of IPv6, and given its sparse address space, it would be futile for a worm to propagate through blind IP scanning, as many authors pointed out [28]. Thus worm writers will have to devise more clever strategies to choose target nodes, and following a network topology is a natural candidate.

Topological worms are also likely to propagate faster than traditional worms, given that they do not have to probe for random IP addresses, and therefore are more likely to escape prevention mechanisms that are based on "immunizing" nodes against a bad input by disseminating an alert message [7], since the worm might reach other nodes faster than the alert.

An important instance of a topological worm that we address in this paper is a peer-to-peer (p2p) worm [27]. These are topological worms that spread through a p2p overlay. Given the size of p2p systems, they have the potential to target a large node population. Furthermore, p2p worms can avoid traditional detection mechanisms, which are based on anomalous IP traffic patterns [16, 20] (as they do not generate many failed connections and can disguise as normal p2p traffic), and avoid being detected by current honeypots [16] (surveillance machines for early warning and detection that listen in unused IP addresses).

Even though we have not seen specific instances of fast-spreading topological worms, there is some indication that this is a pending problem. For instance, there have been re-

ports of vulnerabilities in p2p client applications like eDonkey, KaZaA and BitComet that would allow for the execution of arbitrary code on the client [1, 2, 4]. Also, there have been some instances of viruses that use file sharing overlays to assist in their propagation by making themselves available for download [3].

In previous work, researchers have pointed out the existence of this problem [21, 27], and even quantified how much faster p2p worms can propagate using simulations [5] and analysis [25]. In this paper we take the next step of proposing that overlays should be modified to incorporate defenses that contain or at least slow down the propagation of topological worms.

We begin by presenting a series of general principles that should guide the design of overlays to achieve the goal of containing topological worms. Then we apply these principles to the design of a new p2p overlay called Verme. Verme is an extension to Chord [23], designed to contain p2p worms in small "islands" of nodes that may have common vulnerabilities (e.g., because all nodes in that island have the same platform). We designed Verme such that nodes inside each island do not have knowledge of other nodes with common vulnerabilities. As a consequence, the worm can be contained within the island. Furthermore, Verme is designed to maintain the good properties of Chord, namely its good lookup performance and low overhead. We also built a distributed hash table (DHT) layer called VerDi on top of the Verme routing overlay.

In the design of our systems a series of interesting problems have arisen, like how to address impersonation attacks (where an attacker could join the overlay with identities of the wrong platform type, and use them to obtain addresses of nodes it should not have access to). In this paper we also discuss possible ways to address such problems.

Performance simulations show that both Verme and its DHT layer (VerDi) do not introduce a significant overhead when compared to Chord, and the corresponding DHT (DHash). Our simulations also show that Verme can be effective in delaying the propagation speed when compared to a p2p worm that spreads through a conventional overlay. While not claiming to have found a panacea, our new overlay design contributes to containing, or at least slowing down the propagation of topological worms, and raising the difficulty level for writing them.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 presents an overview of the problem being addressed and the general solution. Section 4 presents our new p2p routing overlay, Verme, and Section 5 presents the DHT built on top of it, VerDi. Section 6 discusses some open issues in our design. Section 7 presents our experimental evaluation. We conclude in Section 8.

2. Related work

The containment of p2p worms is a recent research area. One of the first papers to point out the existence of this problem was the work of Zhou et al. [27]. In this workshop paper, the authors motivate the problem, and propose as their main research direction populating p2p overlays with guardian nodes. These are special nodes that are running worm-detection software (which was later proposed in a separate paper [7]) that tracks how information from untrusted sources propagates itself in memory. These have to be special purpose nodes, since this detection considerably slows down the execution. This differs from our vision of a true p2p system where all nodes have common responsibilities, and where the overlay graph is modified to contain the propagation of the worms. In this paper, Zhou et al. also mention how the existence of immune nodes could slow down the propagation, but do not propose any reorganization of the overlay to achieve contention.

Previously, a number of other papers had identified the existence of a critical number of infected hosts before which a random scan worm spreads slowly [18], and showed the effect that a hit list (collection of vulnerable hosts gathered previously) may have in minimizing the time to achieve that critical point [21]. P2p systems offer not only a very accurate hit list collection field, but also an almost optimal infection strategy, because by following the overlay structure the number of infection collisions (i.e. two infected hosts wasting time trying infect the same host) is kept to a minimum.

Yu et al. [25] propose a model for p2p worms, and analyze the propagation of these worms depending on the attack model (e.g., whether the worm uses the overlay topology or not), and on the structure of the overlay. They point out that these worms propagate much faster than traditional scanning worms, and that unstructured overlays can also lead to faster propagation. They do not propose, but mention as future work, the design of defense systems.

Ramachandran and Sikdar [19] have proposed an analytical model for the dissemination of worms in p2p overlays. They conclude that an accurate model needs to take into account user characteristics and communication patterns.

Chen and Gray [5] have also studied the propagation of worms in p2p overlays using simulations, but, unlike the previous two papers, they have considered a dynamic peer population instead of a static overlay graph. They also propose a detection mechanism based on the observation that worms distort node popularity, reflected in changes in connection rates.

We contrast with the previous papers in that they focus on a better understanding of the problem using models and simulations, whereas our proposal focuses on the defenses required to contain p2p worms.

Phoenix [15] is a replication protocol that places data

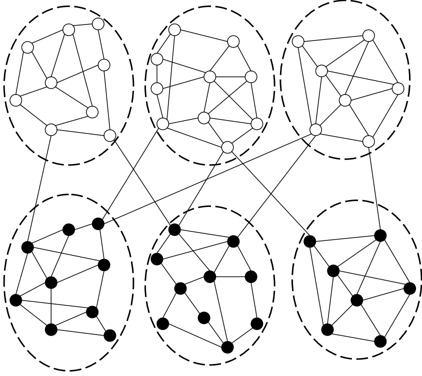

Figure 1. Generic structure of an overlay graph designed for worm containment

on a set of servers running on different platforms to reduce the probability of correlated failures and improve efficiency. This is the same guiding principle that our proposal takes advantage of, but it is used in our case for solving a different problem, which is the containment of topological worms.

Our own workshop paper [12] presented an early version of the design presented in this paper but had not shown any implementation results yet.

3. Overview

In this section we formulate our problem statement and give an overview of our general solution.

Our goal is to redesign overlay networks in such a way that they contain or slow down the propagation of worms that use the overlay topology to choose the next target node to infect.

The simplest possibility for designing such a worm would be to exploit a vulnerability in the overlay application, and use the routing state maintained by the application to choose the next target to infect. However, the overlay application does not have to contain a vulnerability to write such a worm. A worm that is not related to the overlay can use knowledge from this application (e.g., by inspecting open TCP connections) to choose where to propagate.

Our proposal takes advantage of the fact that overlays contain many different types of nodes, running on different platforms, or using different versions of overlay client software. This diversity can be used to contain the propagation of p2p worms, since the vulnerabilities in one particular implementation or platform may not affect the entire population. For instance, the SQLSlammer worm only affects Windows machines running SQLServer 2000 applications. In case the vulnerability is found in the overlay applications it is also not likely to be present in different implementations (e.g., different BitTorrent clients).

In our presentation, instead of referring to a vague definition of platform (which may include different concepts like hardware, OS, or even services and applications that are running), we will introduce the notion of *type*, where we define that two nodes are of the same type if and only if they may have common vulnerabilities.

Given the above observations, we propose that the structure of the overlay should be modified such that the overlay graph forms small "islands" of the same type. The nodes in each island may be adjacent to other nodes from the same island, or to nodes from islands of distinct types, but may not be adjacent to nodes from other islands of the same type.

Figure 1 gives an example of a system with two types of nodes. The overlay graph forms small islands of nodes of the same type (enclosed within the dashed circles). The nodes within an island may have edges among themselves (i.e., they may be present in each other's routing tables) which may lead to the propagation of a topological worm within an island. Nodes may also have edges to nodes that belong to distinct islands of other types, but never to nodes of distinct islands of the same type. Therefore a topological worm will be confined to an island, assuming it is only trying to follow overlay links.

Modifying the overlay graph is not enough to succeed in preventing the propagation of topological worms. For instance, a worm could use overlay maintenance messages or perform lookups to discover the network addresses of nodes of the same type from distinct islands.

4. Verme

In this section we present the design of Verme, an extension of Chord [23] that follows the design principles presented above.

4.1. Assumptions

In this presentation we rely on some assumptions that we will revisit in subsequent sections to discuss how reasonable they are or how they can be enforced.

First, we assume that each node is assigned a certificate that binds its node identifier to the public key that speaks for its principal, and the platform type.

To simplify our presentation, we will assume that nodes may be of two distinct types without common vulnerabilities (generalizing our design to support more than two types of nodes is discussed elsewhere [11]).

4.2. Chord overview

Chord [23] is a peer-to-peer routing overlay that provides a scalable lookup primitive that allows applications to find data stored in a peer-to-peer system.

In Chord nodes have identifiers that are 160-bit integers assigned in such a way that they are uniformly distributed (e.g., as the output of a SHA-1 function applied to the network address and port number of the node).

Figure 2. Identifier structure in Verme

Chord designates the node whose identifier immediately follows a key (called the successor node) as responsible for that key. Lookups map a 160-bit key (the identifier of the data item) to the network address of the successor of that key. In some cases where the layer above Chord requires several replicas for each item stored in the system, lookups will return a list of n successors of the key instead of only the immediate successor.

Each Chord node maintains a small amount of routing state (small enough to keep its maintenance overhead low). This consists of a list of successors (i.e., the identifiers and IP addresses of the nodes that follow it in the ring), the node's predecessor, and a finger table, consisting of the IP addresses and identifiers of nodes that follow it at power-of-two distances in the identifier space. Chord's maintenance protocols work hard to keep the pointer to the next successor up-to-date, by running a stabilization routine that determines if there were any membership changes in the vicinity of the node. On the other hand, the remaining routing state (like finger table entries) can be refreshed in the background by looking up the appropriate id infrequently.

Lookup requests travel through a sequence of nodes (either recursively orn iteratively), where each node in this sequence forwards the request to (or answers the client with) the node from its finger table with highest id still smaller than the desired key. The lookup will conclude when the successor of the id is reached, which happens with high probability after $O(log\ N)$ routing hops.

In Section 5 we will give examples of other layers that use this routing overlay, in particular a DHT.

4.3. Id assignment

The id assignment scheme used by Chord does not obey the principles mentioned in Section 3, since the list of successors of any given node will typically contain nodes of both types. Therefore we modify the way ids are assigned such that the ring is divided into sections, where each section only contains the ids of nodes of a particular type. Furthermore, neighboring sections must always belong to different types. This will cause nodes of the same type from the same section to have knowledge about themselves (through their successor lists) but no knowledge of nodes of the same type in other sections (provided that the number of nodes in each section is large enough that successor lists never span more than two sections).

Verme's id assignment achieves this by dividing the node id in three parts, as depicted in Figure 2. The lower bits are assigned randomly, and the number of bits employed here specifies the length of the section. By adjusting this number properly we can ensure that, with high probability, successor lists do not cross more than one section. The middle bits are fixed according to the node type. With our simplifying assumption of having only two types in the system we could use a single bit. The higher bits are also assigned randomly, and their concatenation with the node type specifies the section number that the node is in.

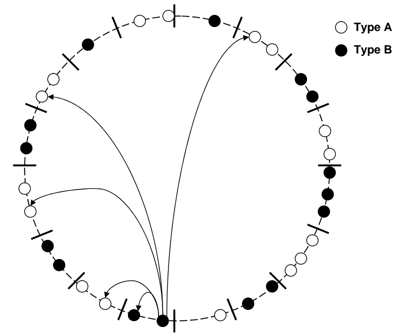

Figure 3. Finger and successor pointers

4.4. Successors and fingers

Each node maintains pointers to a list of successors that are used and maintained just like in Chord. However, finger table entries must be modified to point to a node that is not of the same type as the node itself, to respect the design principles presented in Section 3.

Thus we need to change the way that fingers are defined. Instead of a finger table entry pointing to the node that follows it at power-of-two distances in the identifier space, the finger entry will now correspond to the first successor of the ids at the same distance plus the length of a section (except for nearby nodes that are either in same section where the node is in, or in the subsequent section of nodes of the opposite type where adding a section length is not required), so that the node that follows it belongs to the opposite type. Figure 3 shows a Verme ring with the successors and fingers of a node.

There is a corner case that need to be considered, which happens when the id falls between the last node in the section and the end of that section (in which case the finger table entry would be the first node in the next section of the same type). In this case we chose to assign the responsibility to the predecessor of that id, instead of the successor. The problem with this design choice is that the last node in each section will have a higher load than in Chord, which is compensated by a lower load at the first node of each section. However, fixing this slight load imbalance would require adding a substantial amount of complexity to the design of the system (e.g., to skip the neighboring section), which justifies our choice.

4.5. Lookups

Lookup is the central operation provided by the routing overlay. In Chord (as in most peer-to-peer overlays) any node in the system can issue a *lookup(id)* operation. As mentioned, this returns the address of a node (or set of nodes) that are responsible for the data with that id (in Chord these are the successors of the id).

This is used not only by applications, but also in the overlay maintenance protocols: finger table entries are refreshed periodically by performing a lookup to the appropriate point in the id space; and joins of nodes incoming to the overlay are also initiated by performing a lookup to the id of the incoming node, who then contacts its new successor to update its routing information. We will begin by discussing how lookups are modified for overlay maintenance operations, and we discuss how upper layers can use lookups in Section 5.

The current lookup abstraction allows an infected node to crawl the overlay, by making lookups with different ids, to obtain addresses to attack. We address this issue by changing several aspects of the lookup operation. First, the lookup message must carry the certificate of the node that is performing the lookup. This will allow the predecessor of the id to verify the legitimacy of the initiator in looking up this id. When lookups are being used for joining the overlay or calculating finger table entries, this is straightforward: the node must verify if it is the successor or a correct finger of the id in the certificate.

The second aspect we need to address in lookups is that they cannot be iterative, since many nodes in a lookup path have the same type as the node performing the lookup. Therefore we change the lookup to be recursive (i.e., each node contacts the next node in the lookup path, and the reply travels back through the reverse lookup path). The reply must be encrypted with the public key of the initiator (present in the certificate sent with the lookup) to keep the IP address in the reply from being disclosed to the nodes in the lookup path.

We also rejected the solution of transitive lookups (i.e., the forward path is identical to a recursive lookup, but the replier contacts the initiator directly) because in this case the lookup request must contain the IP address of the initiator node, to allow the replier to contact him. This would open an avenue for an infected node to collect a large number of IP addresses of any type, simply by inspecting the IP addresses that are sent through it.

5. Upper layers: VerDi DHT

The layers above the lookup substrate also need to be modified to preserve the design principles and properties subjacent to our scheme, like not propagating network information about peers.

In this section we will focus on a particular layer that uses the lookup infrastructure: a DHT that supports get and put operations, similar to the DHash layer built on top of Chord lookups [8, 9]. The design of the VerDi DHT is based on the original design of DHash, and it uses Verme as the routing overlay. We believe that a DHT is representative of how other layers, or even applications must be adapted.

5.1. DHash overview

DHash exports a simple interface to applications with two operations:

```
key = put(value)
value = get(key)
```

where the key is computed to be the SHA-1 hash of the value.

In this system data is replicated in the set of n successors of the identifier of the data item. As an optimization, a more recent paper has proposed the use of erasure coded fragments instead of full replicas of the data [9] but we will not consider that optimization in this paper.

Get and put operations are preceded by a lookup that returns the successor list of the key's predecessor. Then one or more of these nodes are contacted directly to store or retrieve the data. Another optimization for the get operation that we did not implement was for the lookup to stop short of the key's predecessor provided that enough successors of the id were returned to reconstruct the original data.

Before returning the output of a get operation to the client, DHash verifies that the hash of the value returned by the replica matches the id being looked up (in other words, the data is self-verifying).

5.2. Replication in VerDi

If we maintained the design of DHash, and only replaced Chord with Verme in the routing layer, we would still have a risk of worm propagation because the replicas of the data may be from the same type of the node making the request. Therefore an infected node could issue a series of get or put requests to harvest IP addresses of any type it wishes to infect.

The first step to address this problem is to change the way that replicas are assigned in VerDi. Instead of replicating in the n successors of the identifier of the data item, we make $n/2$ replicas at that point in the id space, and another $n/2$ in the same position of the subsequent section of the ring (of the opposite type). This replication strategy is depicted in Figure 4.

Again, we addressed the corner case of a data item with an id that falls between the id of the $\frac{n}{2}$th last node in a section and the end of the section by replicating toward the predecessors instead of successors, which causes some load imbalance but avoids a complex design. This implied, though, a small change to Verme's maintenance protocols.

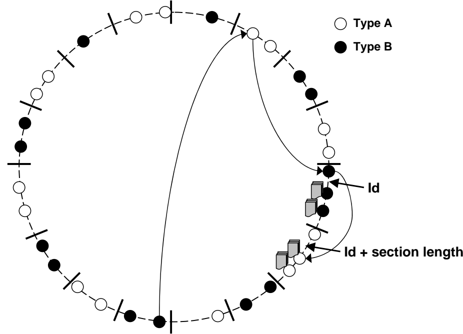

Figure 4. Replication of data items in a DHT

Instead of only trying to keep the successor list up-to-date, Verme also uses the same strategy to maintain a list of predecessors.

Replicating in nodes of both types also has the advantage of increasing reliability, since a worm outbreak that affects nodes from one type will not be able to wipe out all copies of a given object. We intend to further explore the increased availability and reliability guarantees of our system.

5.3. Integrating routing and fetching

We also need to understand how get and put operations are performed in VerDi, in particular how to integrate routing and fetching. We have devised three alternative methods for the implementation of these operations, that represent three distinct points in a tradeoff between performance and security (in terms of the ability to contain p2p worms).

5.3.1. Alternative 1: Fast-VerDi

The most straightforward way to implement operations in VerDi is to allow the lookup primitive exported to the upper layers to return the addresses of the replicas that are of the opposite type of the node that issued the request. For that purpose, the lookup operation adds the section length to the id being looked up if necessary (to avoid returning addresses of the same type as the initiator). The replier then verifies if the initiator is of the opposite type before responding (by checking that the id in the certificate sent along with the message belongs to the opposite type) dropping the message otherwise, and encrypts the reply using the public key in the certificate. The reply travels back through the reverse lookup path.

Once the initiator node receives this reply and decrypts it, the get operation is concluded by fetching the data directly from any of the replicas.

In the case of put operations, after the lookup reply is received, the client sends the data to the responsible node (of the opposite type of the client). After receiving the data, the responsible node has to send the data to the set of replicas of the opposite type, to make the data available to all clients in the overlay (independently of their type). For that, it performs a lookup for the id of the data (or that id plus the section length) to find a responsible node of the opposite type to whom a copy of the data is sent.

The main limitation of Fast-VerDi is that it is vulnerable to an impersonation attack. In this attack, a single node would obtain an identity of the opposite type of the one it wishes to attack, and issue a series of get or put operations to random ids to harvest IP addresses and then infect these nodes. In the next two sections we discuss alternative designs that alleviate this problem.

5.3.2. Alternative 2: Secure-VerDi

In this design we address this problem by combining the lookup operation with the operations get or put. Thus, the operation is piggybacked in the lookup message, which moves recursively until it reaches the predecessor of the identifier associated with the data. In the case of a get operation, one of the replicas is chosen to retrieve the data and the reply travels back through the reverse lookup path. Note that in this case it is not necessary to fetch the reply from a particular type of node, and so data does not need to be replicated in two sections.

In Secure-VerDi, the impersonation attack mentioned in the previous section is highly limited: Each node only contacts its overlay neighbors (finger and successor entries) and therefore a single impersonating node would at most be able to infect the sections corresponding to these nodes (which are $O(\log n)$, hence a relatively small fraction of nodes for a large overlay).

The price to pay for this additional level of security is the latency and bandwidth usage for get and put operations, which imply a costly data transfer for every hop in the lookup path.

5.3.3. Alternative 3: Compromise-VerDi

We also propose a compromise solution that represents an intermediate point in the compromise between performance and security. The idea is to have a single level of indirection when uploading or downloading the data object. Thus the data will be relayed by one of the finger table entries of the initiator of the request, preventing a compromised node from harvesting IP addresses by performing several DHT operations.

In more detail, a get operation works as follows. The initiator of the operation begins by signing a statement vouching for the fact that it wants to perform the operation. Then it sends the request, along with this statement and its certificate to its appropriate finger table entry (who will act as a relay). This node will then append its own node certificate to the request, and will make a get operation like in Fast-Verdi. When the relay receives the data, it forwards it to the initiator.

This way, in case of an impersonation attack, an infected

node pretending to be of the same type that it wishes to attack will not be able to proactively harvest IP addresses by issuing operations, unless it is colluding with one of its finger table entries, which is difficult to do because of the size of the overlay and because of the fact that node ids are random and we envision that they will be difficult to obtain (so nodes cannot easily try out different ids until it works). Again, even though such collusion between a node and its finger would be possible given enough effort to produce two node ids that are correspond to overlay neighbors, the goal of significantly raising the bar for building such a p2p worm has been achieved.

Note that despite this effort to prevent impersonating nodes from harvesting addresses by proactively issuing DHT operations, an impersonating node can still passively listen to requests coming from nodes of whom they are finger table entries, and use this to record IP addresses of nodes of the same type. They are, however, limited to the rate at which their neighbors issue these requests.

6. Discussion

In this section we question the validity of some of the assumptions made previously, and discuss other issues.

6.1. Sybil attacks

In our design we assumed that each node in the system obtained a single certificate containing a correct indication of the type of node.

In the previous section we already discussed how to address an impersonation attack, where an infected node joins the system with a single identity of a type which is different from the vulnerable type.

However, we still need to limit the number of certificates that can be issued to a single entity, since an attacker that populates the system with nodes of arbitrary types under his control (called a Sybil attack [10]) can still harvest a large number of addresses.

Issuing such certificates and limiting Sybil attacks are issues that have been solved with some degree of success in deployed systems like Credence [24] (by asking joining nodes to download a large file or solve cryptographic puzzles), and therefore we intend to use the same strategy. Also, in some cases where the client hardware allows it, we can use remote attestation to verify the identity and platform where the client is running.

6.2. Generalizing to other overlays

Even though DHTs are gaining in popularity, many popular p2p applications are based on unstructured overlays. The design principles stated in Section 3 can also be applied to modify the design of unstructured p2p overlays.

For instance, consider the original (unstructured) design of BitTorrent [6] where a centralized tracker assigns neighbors for nodes to exchange content with.

In this case, and assuming the tracker is not vulnerable to worm infection (e.g., it will not run any services, run behind a firewall, etc.), then it will be able to assign neighbors in a way that forms an overlay graph with the generic structure of Figure 1, therefore achieving the same goal.

Recently, file sharing systems like BitTorrent and eMule have incorporated structured DHTs in their design [22]. For this part of the file sharing system, we could replace their DHTs with VerDi and obtain the benefits of our scheme.

7. Evaluation

In this section we present an experimental evaluation of Verme, and compare it to the Chord overlay in which our design was based.

We implemented Verme by modifying the implementation of Chord for p2psim [13], a discrete event simulator written in C++.

The three variants of the VerDi DHT were implemented based on an incomplete implementation of DHash that was included in p2psim. The incomplete implementation included the get operation and data stabilization routines to maintain certain replication levels for the data. We added the put operation and extended DHash to create the three variants of VerDi.

7.1. Verme overhead

Our first set of experiments evaluates the performance overhead introduced by our new design features. We begin by discussing the overhead of Verme when compared to Chord, and subsequently compare VerDi with DHash.

7.1.1. Simulation setup

We used a simulated network of 1740 nodes, with a matrix derived from measuring the inter-node latencies of DNS servers using the King method [14]. The average round trip time (RTT) was 198 ms. This matrix was obtained from the p2psim web site [13].

In both Verme and Chord overlays, each node has 10 successors, each one runs its stabilization function every 30 seconds and its finger stabilization every 60 seconds. Lookups are issued with random keys by each node at intervals exponentially distributed with a mean of 30 seconds (the values were chosen based on experiments from [17] and [23]).

The Verme overlay was configured with 128 sections, which gives an average of 13 nodes per section and each node has 10 predecessors. The mean node lifetime took the following values: 15 minutes, 30 minutes, 1 hour, 4 hours and 8 hours.

We used the same proportion of nodes of each type. Due to space constraints, we omit the results with an uneven distribution of node types. These results show that such deployments cause a slight load imbalance, which would only become relevant for systems with a very high load.

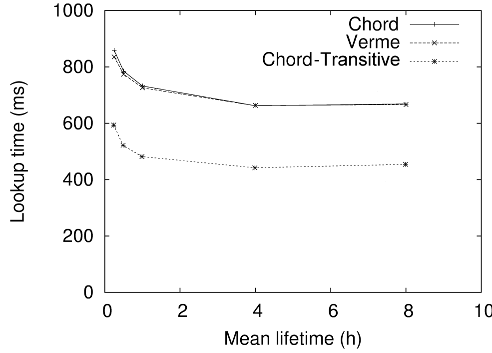

Figure 5. Comparison of lookup latencies

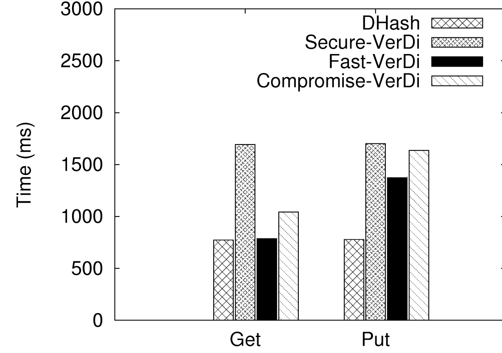

Figure 6. Latencies for get and put

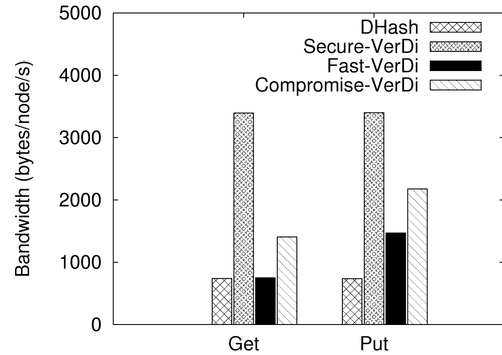

Figure 7. Bandwidth for get and put

For each experiment, the simulation ran for 12 hours and we computed average values. Each simulation was repeated 8 times and we report on the average.

7.1.2. Evaluation results

Figure 5 shows the comparison between the lookup latency for Chord and Verme. Verme lookups have to be recursive, whereas Chord lookups may be transitive, where a recursive route is taken by the lookup request, but the reply is given directly to the initiator. We compared the latency of Verme with transitive and recursive Chord. In the x-axis we varied the mean node lifetime, to determine if our results were affected by node dynamics. The additional overhead introduced by Verme is noticeable when compared to transitive Chord, where lookup latencies are 35% lower than in Verme. Note that the impact of this overhead is minimized when we take into account the fact that an operation in a DHT will include both the lookup and the time to download or upload the data. When compared to recursive Chord the latency of Verme is similar, thus the changes in the finger assignment and in the lookup strategy did not introduce a significant overhead. The node dynamics did not affect the comparison, since all implementations were equally affected by the need to route around failures (every time a node tried to contact a node that had failed it chose another neighbor).

In additional experiments, reported in a separate document [11], we also show that both lookup failure rates and the bandwidth used for overlay maintenance and lookups does not differ significantly between Chord and Verme.

7.2. VerDi overhead

Now we evaluate the overhead of VerDi with respect to Chord's DHT layer, DHash. In this set of experiments, the King data set revealed itself less useful, since we were simulating data operations, and it did not have reference values for bandwidth between nodes, so we slightly modified the setup to use the GT-ITM model [26].

Figure 6 shows the latency for DHT get and put operations in the different systems. The results confirm that there is a tradeoff between performance and security. If we analyze the latency of get operations, we can see that Fast-VerDi has the lowest latency, which is very close to the latency of DHash, since they work very similarly by having the initiator perform a recursive lookup followed by a direct download from the responsible node. Secure-VerDi has the highest latency due to the fact that there is a costly data transfer between every pair of nodes along the reverse lookup path. Compromise-VerDi performs in between the other two designs, since it only has one level of indirection when downloading the data – up to 31% slower than DHash.

In terms of the latency of put operations, Secure-VerDi has the highest latency because the data is sent through the forward lookup path. Fast-VerDi and Compromise-VerDi show a larger difference when compared to DHash due to the fact that the reply is only sent to the client after the responsible node that receives the data makes a copy to the other responsible node of the opposite type (to ensure that the data is available to nodes of any type).

We also analyzed the bandwidth used for DHT get and put operations. The results are shown in Figure 7.

The results for bandwidth usage show that DHash and Fast-VerDi use more or less the same bandwidth for get op-

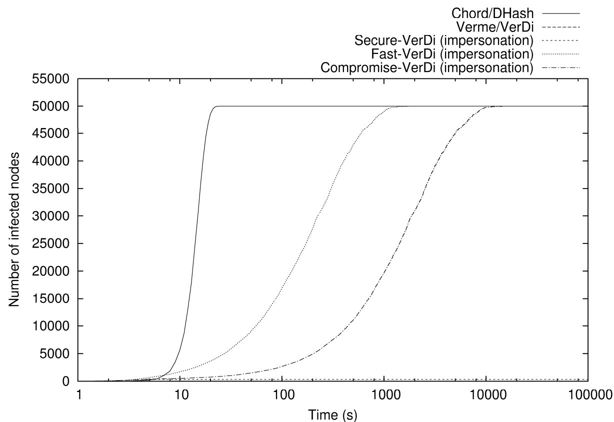

Figure 8. Simulated propagation speeds

erations, since the bulk of the bandwidth is spent on the data transfer from the responsible node to the client. Get operations in Secure-VerDi are costly because the data is piggybacked on the reply that travels through every pair of nodes in the lookup path, and Compromise-VerDi only has one level of indirection, which approximately doubles the bandwidth consumption when compared to the more effective strategies. When we analyze the bandwidth used by put operations we can see that the results are similar to the results for get operations, because the sequence of data transfers are also similar (albeit in the reverse direction). The differences for Fast-VerDi and Compromise-VerDi are due to the fact that before returning to the client, an extra copy is done to the other responsible node (of the opposite type). Note that the results shown in Figure 7 do not include the bandwidth used for the creation of additional replicas (other than the copies held by one or two responsible nodes, depending on the implementation) that takes place in background.

The conclusion of this comparison between the different DHT layers is that there is a marked tradeoff between the security offered by our DHT against more sophisticated worms, and the overhead introduced by the DHT, especially in terms of bandwidth usage. Latencies, however, are more affected in the case of put operations (which are probably less frequent in most scenarios) or in the case of Secure-VerDi where all operations work in a completely recursive manner.

7.3. Worm propagation speed

The experiments so far have focused on the overhead introduced by the system when compared to the original version of Chord. Now we shift our focus to benefits from using Verme in terms of slowing down the propagation of worms. To analyze these benefits, it was necessary to develop a model for worm propagation under different conditions. The adopted model was based on an existing model to simulate the propagation of worms that used parameters derived from real worms [21].

In particular, we considered a model where nodes can be in one of four states: *not infected, scanning, infecting, inactive*. A machine that is in the *scanning* state starts scanning nodes at some rate (according to some strategy to obtain addresses of peers that depends on the type of worm, as we discuss next). If it finds a vulnerable node, it switches to the *infecting* state. After some period of time, if the target node is in the *not infected* state, the infection is complete, the target node goes to the *inactive* state, and the original node returns to *scanning*. After some more time the worm is activated in the new node, and it also starts *scanning*.

We used the following parameters (based on the aforementioned model [21]): a scanning rate of 100 scans/machine/second; the time to infect a machine was 100 ms; and the time between the node infection and worm activation was 1 second.

We considered a $100,000$-node static overlay where 50% of the machines were vulnerable to the worm being simulated. The Verme overlay was configured with 4096 sections which gives an average of 24 nodes per section.

Our simulation compares the propagation speeds for different strategies: a p2p worm that propagates exclusively in a Chord overlay, a p2p worm that propagates exclusively in Verme, and a p2p worm that propagates with the help of an impersonating node (i.e., a node that joins the overlay with a type that is opposite from the one it wishes to attack) that issues a series of DHT operations to harvest IP addresses. In the latter case we considered the three variants of VerDi. For the case of Fast-VerDi, the impersonating node was issuing lookups at a rate of 10 lookups per second, and in the case of Compromise-VerDi (where the impersonating node does not gain from issuing lookups, but has to wait for other nodes to issue so it can act as a relay) every overlay node would issue 1 lookup per second. Each strategy was simulated 10 times and we report on the average.

Figure 8 shows the number of infected machines as a function of the time since the start of the infection. Note that we used a logarithmic scale in the x-axis so there are substantial differences between the different curves. These results show that an overlay like Chord can be the ideal substrate to achieve a very fast propagation, taking only 32 seconds to infect the entire system. On the other end, if Verme is used without any impersonation attack then the worm would be confined to the nodes in a single section. In the case of Secure-VerDi with an infected machine impersonating the appropriate type, the infection is limited to a logarithmic number of sections, which also represents a very small fraction of the system (352 nodes). These two curves are almost imperceptible because they are so close to the x-axis. Fast-VerDi and Compromise-VerDi succeed in slowing down worm propagation, even in the presence of

an impersonation attack, with approximately 160 and 1600 seconds to infect half of the vulnerable population of the overlay.

8. Conclusion

This paper presented a novel overlay called Verme, and a new DHT built on top of it called VerDi, which were designed to contain, or at least slow down the propagation of p2p worms.

We implemented Verme and VerDi using p2psim. Our simulations show that the overhead of using our overlay and DHT is reasonable when compared to Chord and DHash, and that using Verme can contain, or at least slow down the propagation of p2p worms.

Our new overlay design is a first step towards the containment of worms that propagate using the aid of an overlay network, and is at least raising the difficulty level for writing them.

9. Acknowledgments

We would like to thank the anonymous reviewers for valuable feedback.

References

[1] http://www.securityfocus.com/bid/4951.

[2] http://www.securityfocus.com/bid/6747.

[3] http://www.cert.org/incident_notes/IN-2004-01.html.

[4] Bitcomet torrent file handling remote buffer overflow vulnerability. http://www.securityfocus.com/bid/16311.

[5] G. Chen and R. S. Gray. Simulating non-scanning worms on peer-to-peer networks. In *Proceedings of 1th International Conference on Scalable Information Systems (INFOSCALE 2006)*, May 2006.

[6] B. Cohen. Incentives build robustness in bittorrent. In *Proceedings of the First Workshop on the Economics of Peer-to-Peer Systems*, 2003.

[7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, Oct. 2005.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, Oct. 2001.

[9] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a dht for low latency and high throughput. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[10] J. Douceur. The sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Mar. 2002.

[11] F. Freitas. Verme: Worm containment in overlay networks. Master Dissertation in Computer Engineering, Technical University of Lisbon, 2008.

[12] F. Freitas, R. Rodrigues, C. Ribeiro, P. Ferreira, and L. Rodrigues. Verme: Worm containment in peer-to-peer overlays. In *6th International Workshop on Peer-to-Peer Systems (IPTPS'07)*, 2007.

[13] T. Gil, F. Kaashoek, J. Li, R. Morris, and J. Stribling. p2psim: a simulator for peer-to-peer (p2p) protocols. http://pdos.csail.mit.edu/p2psim/.

[14] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: estimating latency between arbitrary internet end hosts. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, 2002.

[15] F. Junqueira, R. Bhagwan, A. Hevia, K. Marzullo, and G. M. Voelker. Surviving internet catastrophes. In *Proceedings of USENIX Annual Tech. Conference*, 2005.

[16] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proceedings of HotNets 2003*, Nov. 2003.

[17] J. Li, J. Stribling, R. Morris, M. F. Kaashoek, and T. M. Gil. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *Proceedings of the INFOCOM'05*, 2005.

[18] D. Moore, C. Shannon, and J. Brown. Code-red: a case study on the spread and victims of an internet worm. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, 2002.

[19] K. Ramachandran and B. Sikdar. Modeling malware propagation in gnutella type peer-to-peer networks. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Apr. 2006.

[20] S. Singh, G. V. C. Estan, , and S. Savage. Automated worm fingerprinting. In *Proceedings of 6th Symposium on operating design and implementation 2004 (OSDI 2004)*, Dec. 2004.

[21] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the internet in your spare time. In *Proceedings of USENIX Security Symposium 2002*, Aug. 2002.

[22] M. Steiner, E. W. Biersack, and T. Ennajjary. Actively monitoring peers in kad. In *6th International Workshop on Peer-to-Peer Systems (IPTPS'07)*.

[23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, Aug. 2001.

[24] K. Walsh and E. G. Sirer. Experience with an object reputation system for peer-to-peer filesharing. In *3rd Symposium on Networked Systems Design and Implementation (NSDI 06)*, May 2006.

[25] W. Yu, C. Boyer, S. Chellappan, and D. Xuan. Peer-to-peer system-based active worm attacks: Modeling and analysis. In *IEEE International Conference on Communications (ICC)*, May 2005.

[26] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOM*, pages 594–602, 1996.

[27] L. Zhou, L. Zhang, F. McSherry, N. Immorlica, M. Costa, and S. Chien. A first look at peer-to-peer worms: Threats and defenses. In *Proceedings of the 4th International Workshop on Peer-To-Peer Systems (IPTPS'05)*, Feb. 2005.

[28] C. C. Zou, W. Gong, D. Towsley, and L. Gao. The monitoring and early detection of internet worms. *IEEE/ACM Trans. Netw.*, 13(5):961–974, 2005.