Computer Science Department
University of Crete

Network Address Space Randomization: A Proactive Defense Against Hitlist Worms

Master's Thesis

Spiros Antonatos

October 2005
Heraklion, Greece

Abstract

Worms are self-replicating malicious programs that represent a major security threat for the Internet, as they can infect and damage a large number of vulnerable hosts at timescales where human responses are unlikely to be effective. Sophisticated worms that use precomputed hitlists of vulnerable targets are especially hard to contain, since they are harder to detect, and spread at rates where even automated defenses may not be able to react in a timely fashion.

This thesis examines a new proactive defense mechanism called Network Address Space Randomization (NASR) whose objective is to harden networks specifically against hitlist worms. The idea behind NASR is that hitlist information could be rendered stale if nodes are forced to frequently change their IP addresses. NASR limits or slows down hitlist worms and forces them to exhibit features that make them easier to contain at the perimeter. We explore the design space for NASR and present a prototype implementation as well as preliminary experiments examining the effectiveness and limitations of the approach.

Supervisor: Evangelos P. Markatos

Τυχαιοποίηση του Χώρου Δικτυακών Διευθύνσεων: Ένα Μέσο Προληπτικής Άμυνας Ενάντια σε Hitlist Worms

Αντωνάτος Σπύρος

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Περίληψη

Τα worms είναι κακόβουλα προγράμματα τα οποία έχουν την ιδιότητα να αυτο-πολλαπλασιάζονται με αποτέλεσμα να αποτελούν μιας πρώτης τάξης απειλή για το διαδίκτυο, αφού μπορούν να καταστρέψουν ένα μεγάλο αριθμό μηχανημάτων σε τόσο λίγο χρόνο όπου η ανθρώπινη παρέμβαση καθίσταται αναποτελεσματική. Πιο εξελιγμένα worms, τα οποία χρησιμοποιούν προϋπολογισμένες λίστες ευάλωτων στόχων (hitlists), είναι ιδιαίτερα δύσκολο να περιοριστούν για δυο λόγους. Πρώτον, είναι πιο δύσκολο να εντοπιστούν και επιπλέον εξαπλώνονται με τέτοιο ρυθμό ώστε ακόμα και αυτόματα αμυντικά συστήματα να μην μπορούν να αντιδράσουν έγκαιρα.

Σκοπός της εργασίας αυτής είναι η μελέτη του Network Address Space Randomization (NASR), ενός προληπτικού αμυντικού μηχανισμού που έχει ως σκοπό να δυσκολέψει το έργο των hitlist worms. Η ιδέα πίσω από τον μηχανισμό αυτό είναι ότι η πληροφορία που περιέχεται στις προϋπολογισμένες λίστες είναι δυνατόν να γίνει αναχρονιστική αν οι κόμβοι αλλάζουν συχνά τις IP διευθύνσεις τους. Το NASR περιορίζει ή καθυστερεί τα hitlist worms αναγκά-

ζοντας τα να εμφανίσουν χαρακτηριστικά που επιτρέπουν τον πιο εύκολο εντοπισμό τους στην περίμετρο του δικτύου. Παρουσιάζουμε τον σχεδιασμό του NASR, μια πρότυπη υλοποίηση καθώς επίσης και αποτελέσμα που εξετάζουν την αποδοτικότητα και τους περιορισμούς του μηχανισμού μας.

Επόπτης Μεταπτυχιακής Εργασίας: Ευάγγελος Π. Μαρκάτος

Acknowledgments

I would like to deeply thank my supervisor, Evangelos P. Markatos, for his great support and feedback. His simple way of thinking and broad perspective will be a guide to my academic route. I would also like to express my deep gratitude to Kostas Anagnostakis, a valuable partner whose contribution was a key for writing this thesis. It is truly an unprecedented experience to work with these people.

I would also like to deeply thank my friends and colleagues Michalis Polychronakis, Dimitris Koukis and Manos Moschous as well as all the members of the Distributed Computing Systems lab, especially Dimitris Antoniadis and Elias Athanasopoulos. I am also very grateful to the ISPs that provided me with useful data for this work.

Στη μητέρα μου

An early report on this work appeared in the proceedings of the 3rd ACM Workshop on Rapid Malcode (WORM'05), in conjunction with the 12th ACM Conference on Computer and Communications Security (CCS), November 2005[13].

Contents

1 Introduction **1**

1.1 Thesis organization 4

2 Background **5**

2.1 Worms 5

2.2 Hitlists 7

2.3 Worm defenses 9

3 Network Address Space Randomization **11**

3.1 Abstract model of NASR 11

3.2 Practical constraints 12

3.2.1 Scope 13

3.2.2 Static addressing 13

3.2.3 DNS updates 14

3.2.4 Tolerance to address changes 14

3.3 Implementation 15

4 Measurements **19**

4.1 Hitlist generation strategies 20

4.1.1 Random scanning 21
4.1.2 Passive P2P snooping 23
4.1.3 Search-engine harvesting 24
4.2 Subnet address space utilization 25

5 Impact of NASR on worm infection **29**
5.1 Impact of NASR 32
5.2 Partial deployment scenario 34
5.3 Interaction with scan-blocking 35

6 The cost of NASR **37**
6.1 Address change frequency vs. application failures 39

7 Transparent NASR **43**

8 Discussion **51**

9 Related Work **55**

10 Summary **59**

List of Figures

1.1 Propagation speed of different types of worm attacks 4

4.1 Decay of addresses harvested using random scanning 21

4.2 Decay of addresses harvested by monitoring peer-to-peer traffic routed through a node. 22

4.3 Decay of addresses harvested by querying a popular web search engine. 23

4.4 Number of distinct addresses harvested by monitoring Gnutella traffic as a function of time and number of monitoring nodes. . 25

4.5 Subnet address space utilization 26

5.1 Worm spread time(time to 90% infection) vs. time between host address changes for different hitlist generation rates . . . 30

5.2 Effect of NASR on hitlist decay 30

5.3 Effect of NASR vs. subnet usage density 31

5.4 Effect of NASR for different vulnerable host populations . . . 31

5.5 Effect of network address space randomization on worm spread time when partially deployed 34

5.6 Maximum fraction of infected hosts vs. time between host address changes for different hitlist rates assuming scan-blocking mechanisms 36

6.1 Distribution of host uptimes in 5 different networks 38
6.2 Percentage of aborted connections as a function of the hard change limit 38
6.3 Percentage of aborted connections as a function of the soft change limit 39

7.1 An example of NASR using the randomization box 45
7.2 A more advanced example of NASR using the randomization box. Host has two public IP addresses, one (139.91.70.50) devoted for the SSH session to calliope and the other (139.91.70.60) for new connections 46
7.3 The percentage of extra IP space needed 47
7.4 The percentage of extra IP space needed relative to the load of subnets 48

1

Introduction

Worms are widely regarded to be a major security threat facing the Internet today. Incidents such as Code Red[2, 37] and Slammer[36] have clearly demonstrated that worms can infect tens of thousands of hosts in less than half an hour, a timescale where human intervention is unlikely to be feasible, causing financial damage up to 38.5 billion dollars [24].

More recent research studies have estimated that worms can infect one million hosts in less than two seconds [49, 50, 54]. Unlike most of the currently known worms that spread by targeting random hosts, these extremely fast worms rely on predetermined lists of vulnerable targets, called *hitlists*, in

order to spread efficiently.

The threat of worms and the speed at which they can spread have motivated research in automated worm defense mechanisms. For instance, several recent studies have focused on detecting scanning worms [57, 27, 56, 40, 48, 55]. These techniques detect scanning activity and either block or throttle further connection attempts. These techniques are unlikely to be effective against hitlist worms, given that hitlist worms do not exhibit the failed-connection feature that scan detection techniques are looking for. To improve the effectiveness of worm detection, several distributed early-warning systems have been proposed [60, 38, 61, 12]. The goal of these systems is to aggregate and analyze information on scanning or other indications of worm activity from different sites. The accuracy of these systems is improved as they have a more "global" picture of suspicious activity. However, these systems are usually slower than local detectors, as they require data collection and correlation among different sites. Thus, both reactive mechanisms and cooperative detection techniques are unlikely to be able to react to an extremely fast hitlist worm in a timely fashion.

Observing this *gap* in the worm defense space, we consider the question of whether it is possible to develop defenses *specifically* against hitlist worms. We start by looking at likely strategies for building hitlists and examine how effective these strategies can be. We observe that hitlists tend to *decay* naturally for various reasons, as hosts disconnect and applications are abnormally terminated. A rapidly decaying hitlist is likely to decrease the spread rate of a worm. It may also increase the number of unsuccessful connections it initiates and may thus increase the exposure of the worm to scan-detection methods.

Starting with this observation, we ask whether it is possible to *inten-*

tionally accelerate hitlist decay, and propose a specific technique for this purpose called *network address space randomization* (NASR). This technique is primarily inspired by similar efforts for security at the host-level [58, 19, 59, 18, 42, 31, 17]. It is also similar in principle to the "IP hopping" mechanism in the APOD architecture[15], BBN's DYNAT[33] and Sandia's DYNAT[35] systems, all three designed to confuse targeted attacks by dynamically changing network addresses. In this thesis, we examine the same basic idea in the context of defending against hitlist worms. In its simplest form, NASR can be implemented by adapting dynamic network address allocation services such as DHCP[23][1] to *force* more frequent address changes. This simple approach may be able to protect enabled networks against hitlist worms, and, if deployed at a large enough scale, may be able to significantly hamper their spread.

We must emphasize that, like most (if not all) other worm containment proposals, NASR is only a partial solution to the worm containment problem. Where applicable, our approach succeeds in limiting the extent or slowing down the rate of a worm infection. However, the mechanism is specific to IP-hitlist worms, and may be less effective against DNS hitlists (we discuss such issues in Section 8). Furthermore, it cannot always completely squash hitlist-based worm epidemics and it cannot be used universally. Nevertheless, being able to slow down the fastest known propagation mechanism is likely to be valuable, as it may allow more time for other reactive defenses to kick in. Furthermore, note that our analysis does not invalidate the worst-case estimates provided in previous work, nor is our goal to play down the threat

[1] Another known address allocation service is `bootp`[22], but it allocates addresses semi-permanently, without any mechanism for renewing the allocation and is thus not usable for our purposes.

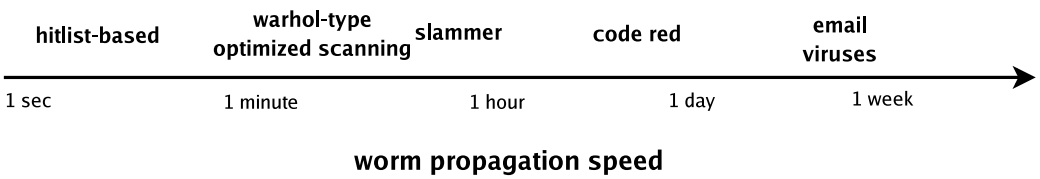

FIGURE 1.1: Propagation speed of different types of worm attacks

posed by such worms. The purpose of this thesis is to help examine whether NASR is worth considering as part of a broader worm defense portfolio.

In the rest of this thesis, we present NASR in more detail and examine issues of applicability, effectiveness and implementation cost.

1.1 Thesis organization

The rest of this thesis is organized as follows. In Chapter 2 we provide a background on worms and current defenses.In Chapter 3 we explore in more detail the idea of network address space randomization, and outline a randomized DHCP server implementation. In Chapter 4 we analyze various hitlist generation strategies and present measurements exploring the properties of a small subset of the IP address space. In Chapter 5 we present a simulation study analyzing the effectiveness of network address space randomization in terms of how much it would slow down a hitlist worm and how it would expose such a worm to scan detection. In Chapter 6 we study the damage caused by NASR on active connections and in Chapter 7 we present a mechanism to overcome this damage. We summarize our results and conclude in Chapter 8.

2

Background

For the purpose of placing our work in context, we first give a brief overview of what is known about worms, with emphasis on hitlist worms. We also present a summary of proposals for defending against worms and how they relate to hitlist worms which are the focus of this thesis.

2.1 Worms

Computer viruses have been studied extensively over the last couple of decades. Cohen was the first to define and describe computer viruses in their present form. In [21], he gave a theoretical basis for the spread of computer viruses.

The strong analogy between biological and computer viruses led Kephart *et al.* [32] to investigate the propagation of computer viruses based on epidemiological models. They extend the standard epidemiological model by placing it on a directed graph and use a combination of analysis and simulation to study its behavior. They conclude that if the rate at which defense mechanisms detect and remove viruses is sufficiently high relative to the rate at which viruses spread, it is possible to prevent widespread virus propagation.

The Code Red worm [2] was analyzed extensively in [62]. The authors conclude that even though epidemic models can be used to study the behavior of Internet worms, they are not accurate enough because they cannot capture some specific properties of the environment, in which worms operate: the effect of human countermeasures against worm spreading (*i.e.,* patching, filtering, disconnecting, *etc.*) and the slowing down of the worm infection rate due to the impact of worm on Internet traffic and infrastructure. They derive a new general Internet worm model called *two-factor worm* model, which they then validate in simulations that match the observed Code Red data available to them. Their analysis seems also to be independently supported by the data on Code Red propagation in [37].

A similar analysis on the SQL "Slammer" (or Sapphire) worm [5] can be found in [6]. Sapphire, the fastest worm to day, was able to infect more than 70,000 victim computers in less than 15 minutes.

The Blaster/Welchia epidemic is an interesting example of a "vigilante" worm (Welchia) causing more trouble than the original outbreak (Blaster). A "vigilante" worm attempts to clean-up another worm by using the same vulnerability. However, the very notion of "vigilante" worms is rendered useless if worms immediately disable the vulnerability after compromising a machine.

The Witty worm [44] is of interest for several reasons. First, it was the first widely propagated Internet worm to carry a destructive payload. Second, Witty was started in an organized manner with an order of magnitude more ground-zero hosts than any previous worm and also began to spread as early as only one day after the vulnerability was publicized, which is an indication that the worm authors had already prepared all the worm infrastructure, including the ground-zero hosts and the replication mechanisms, and were only waiting for an exploit to become available in order to launch the worm.

All these worms use (random) scanning to determine their victims, by using a random number generator to select addresses from the entire IP address space. Although some worms chose their next target uniformly among all the available IP addresses, other worms seemed to prefer local addresses over distant ones, so as to spread the worm to as many local computers as possible. Once inside an organization, these worms make sure that they will infect several of its computers before trying to infect any outside hosts.

2.2 Hitlists

Instead of attempting to infect random targets, a worm could first determine a large vulnerable population before it starts spreading. The worm creator can assemble a list of potentially vulnerable machines prior to releasing the worm, for example, through a slow port scan. The list of known vulnerable hosts is called a hitlist. Using hitlists, worms do not need to waste time scanning for potential targets during the time of the attack and will not generate as many unsuccessful connections as when scanning randomly. This allows them to spread much faster, and it also makes them less visible to scan-based worm detection tools. A hitlist can be either a collection of IP addresses, a set of DNS names or a set of Distributed Hash Table identities

(for infecting DHT systems irrelevantly of the network infrastructure).

Hitlist worms have not been observed in the wild, perhaps because the co-evolution of worms and defenses has not reached that stage yet: they are not currently *necessary* for a successful worm epidemic, since neither scan-blocking nor distributed detection systems are widely deployed yet. However, hitlists are certainly feasible today and worm creators are very likely to use them in the future.

Hitlist worms have attracted some attention lately because they are easy to model off-line [50, 49]. In this context, several hitlist construction methods have been outlined: random scanning, DNS searches, web crawling, public surveys and indexes, as well as monitoring of control messages in peer-to-peer networks.

Random scanning can be used to compile a list of IP addresses that respond to active probing. Since the addresses will not be (ab)used immediately, the worm author can use so-called stealth, low rate, scanning techniques to make the scan pass unnoticed. On the other hand, if the duration of the low-rate scanning phase is very long, some IP addresses will eventually expire.

Hitlists of Web servers can be assembled by sending queries to search engines and by harvesting Web server names off the replies. Similar single-word queries can also be sent to DNS servers in order to validate web server names and find their IP addresses. Interestingly enough, these types of scans can be used to easily create large lists of web servers and are very likely to go unnoticed.

However, any form of active scanning, probing, or searching, has the potential risk of being detected. This gives special appeal to passive techniques, such as those based on peer-to-peer networks. Such networks typically ad-

vertise many of their nodes and this information can be collected by just observing the traffic that is routed through a peer. The creation of the hitlist does not require any active operation from the peer-to-peer node and therefore cannot raise suspicion easily.

2.3 Worm defenses

We discuss some recent proposals for defending against worms and whether they could be effective against hitlist worms.

Approaches such as the one by Wu *et al.* [57] attempt to detect worms by monitoring unsolicited probes to unassigned IP addresses ("dark space") or inactive ports. Worms can be detected by observing statistical properties of scan traffic, such as the number of source/destination addresses and the volume of the captured traffic. By measuring the increase on the number of source addresses seen in a unit of time, it is possible to infer the existence of a new worm when as little as 4% of the vulnerable machines have been infected.

An approach for isolating infected nodes inside an enterprise network is discussed in [48, 27]. The authors show that as little as 4 probes may be sufficient in detecting a new port-scanning worm. Weaver *et al.* [55] describe a practical approximation algorithm for quickly detecting scanning activity that can be efficiently implemented in hardware. Schechter *et al.* [40] use a combination of reverse sequential hypothesis testing and credit-based connection throttling to quickly detect and quarantine local infected hosts. These systems are effective only against scanning worms (not topological, or "hit-list" worms), and rely on the assumption that most scans will result in non-connections.

Several cooperative, distributed defense systems have been proposed.

DOMINO is an overlay system for cooperative intrusion detection [60]. The system is organized in two layers, with a small core of trusted nodes and a larger collection of nodes connected to the core. The experimental analysis demonstrates that a coordinated approach has the potential of providing early warning for large-scale attacks while reducing potential false alarms. Zou *et al.* [61] describes an architecture and models for an early warning system, where the participating nodes/routers propagate alarm reports towards a centralized site for analysis. The question of how to respond to alerts is not addressed, and, similar to DOMINO, the use of a centralized collection and analysis facility is weak against worms attacking the early warning infrastructure. Fully distributed defense mechanisms, such as [38, 12] may be more robust against infrastructure attacks, yet all distributed defense mechanisms that we are aware of are likely to be too slow for the estimated timescales of hitlist worms.

3

Network Address Space Randomization

The goal of network address space randomization (NASR) is to force hosts to change their IP addresses frequently enough so that the information gathered in hitlists is rendered stale by the time the worm is unleashed.

3.1 Abstract model of NASR

To illustrate the basic idea more formally, consider an abstract system model, with an address space $R = \{1, 2, ..., n\}$, a set of hosts $H = \{h_1, ..., h_m\}$ where

$m < n$, and a function A that maps all hosts h_k to addresses $A(h_k) = r \in R$. Assume that at time t_a, the attacker can (instantly) generate a hitlist $X \subset R$ containing the addresses of hosts that are live and vulnerable at that time. If the attack is started at time t_x and all hosts in X are still live and vulnerable and have the same address as at time t_a, then the worm can very quickly infect $|X|$ hosts.

In a system implementing NASR, consider that at time t_b, where $t_a < t_b < t_x$, all hosts are assigned a new address from R. Thus, at the time of the attack t_x the probability that a hitlist entry x_k still corresponds to a live host is $p = m/n$ and thus the attacker will be able to infect $(m/n)|X|$ hosts. Besides reducing the number of successfully infected nodes in the hitlist, the attack will also result in a fraction $1-m/n$ of all attempts failing (which may be detectable using existing techniques). In this simple model, the density m/n of the address space seems to be a crucial factor in determining the effectiveness of NASR. So far we have assumed a homogeneous set of nodes, all with the same vulnerability and probability of getting infected. If only a subset of the host population is vulnerable to a certain type of attack, then the effectiveness of NASR in reducing the fraction of infected hitlist nodes and the number of failed attempts is proportionally higher, according to our simulation results.

3.2 Practical constraints

The model we presented illustrates the basic intuition of how NASR can affect a hitlist worm. Mapping the idea to the reality of existing networks requires us to look into several practical issues.

3.2.1 Scope

Random assignment of an address from a global IP address space pool is not practical for several reasons: (i) it would explode the size of routing tables, the number of routing updates and the frequency of recomputing routes, (ii) it would result in tremendous administrative overhead for reconfiguring mechanisms that make address-based decisions, such as those based on access lists and (iii) it would require global coordination for being implemented. The difficulty of implementing NASR decreases as we restrict its scope to more local regions. Each AS could implement AS- or prefix-level NASR, but this would still create administrative difficulties with interior routing and access lists. It seems that a reasonable strategy would be to provide NASR at the subnet-level, although this does not completely remove the problems outlined above. For instance, access lists would need to be reconfigured to operate on DNS names and DNS would need to be dynamically updated when hosts change addresses. It is also obvious that it is pointless to implement NASR behind NATs, as the internal addresses have no global significance. It is sufficient to change the address of the NAT endpoint (e.g., DSL/home router) to protect the internal hosts.

3.2.2 Static addressing

Some nodes cannot change addresses and those that can may not be able to do so as frequently as we would want. The reason for this is that addresses have first-class transport- and application-level semantics. For instance, DNS server addresses are usually hardcoded in system configurations. Even for DHCP-configured hosts, changing the address of a DNS server would require synchronizing the lease durations so that the DNS server can change its address at exactly the same time when *all* hosts refresh their DHCP leases.

While technically feasible, this seems too complex to implement and such complexity should be avoided. Similar constraints hold for routers.

3.2.3 DNS updates

For services referenced through the DNS name, such as email, FTP and Web servers, implementing NASR requires the DNS name to accurately reflect the current IP address of the host. This means that the DNS time-to-live timers need to be set low enough so that remote clients and name servers do not cache stale data when an address is changed. The NASR mechanism also needs to interact with the DNS server to keep the address records up to date. It is reasonable to ask whether this could increase the load on the DNS system, given that lower TTLs will negatively affect DNS caching performance. Fortunately, a recent study of DNS performance suggests that reducing the TTLs of address records to values as low as a few hundred seconds does not significantly affect DNS cache hit rates [28].

3.2.4 Tolerance to address changes

Generally, all active TCP connections on a host that changes its address would be killed, unless connection migration techniques such as [26, 47, 16] are used. Such techniques are not widely deployed yet and it is unrealistic to expect that they will be deployed in time to be usable for the purposes of NASR. Many applications are not designed to tolerate connection failures. For instance, NFS clients often hang when the server is lost, and do not transparently re-resolve the NFS server address from DNS before reconnecting.

Fortunately, many applications are designed to deal with occasional connectivity loss by automatically reconnecting and recovering from failure, and

more recent research prototypes even explicitly deal with such failures[29]. For such applications, we can assume that infrequent address changes can be tolerated. Examples of these applications are many P2P clients, like Kazaa and DirectConnect as well as Windows/SAMBA sharing (when names are used), messengers, FTP clients, chat tools, etc. However, tolerance does not always come for free: frequent address changes may result in churn in DHT-based applications and would generally have the side-effect of increasing stale state in other distributed applications, including P2P indexing and Gnutella-like host caches.

There exist ways to make systems more robust to address changes. Rocks [16] is one solution providing reliable sockets for protecting applications sensitive to IP address changes. However, it must be present at both ends of the connection, so it is not practical for connections with external third parties. In a LAN environment, a solution using an "address randomization box" may be applicable in some cases, with the client host not being oblivious to address changes and the randomization box making sure that address changes do not affect applications. However, we must admit that the overall approach seems to require an infrastructure overhaul that we would prefer to avoid. We will discuss this approach in Chapter 7.

Another option, which appears more attractive, is to make the NASR mechanism aware of the active connections on each host, so that address changes can be timed to coincide with the host being inactive. We will discuss one possible approach to address this problem in the next section.

3.3 Implementation

The practical constraints presented in the previous sections suggest that NASR should be implemented very carefully. A plausible scenario would

involve NASR at the subnet level and particularly for client hosts in DHCP-managed address pools. How such concessions affect NASR, as well as the rate at which address changes should be made for NASR to be effective will be explored in more detail in Chapters 5 and 8.

A basic form of NASR can be implemented by configuring the DHCP server to expire DHCP leases at intervals suitable for effective randomization. The DHCP server would normally allow a host to renew the lease if the host issues a request before the lease expires. Thus, forcing addresses changes even when a host requests to renew the lease before it expires requires some minor modifications to the DHCP server. Fortunately, it does not require any modifications to the protocol or the client. We have implemented an advanced NASR-enabled DHCP server, called Wuke-DHCP, based on the ISC open-source DHCP implementation[25]. To minimize the "collateral damage" caused by address changes we introduce two modules in our DHCP implementation: an *activity monitoring* module, and a *service fingerprinting* module.

The activity monitoring module keeps track of open connections for each host with the goal of avoiding address changes for hosts whose services could be disrupted. In our prototype, we only consider long-lived TCP connections (that could be, for example, FTP downloads). More complicated policies can be implemented but are outside the scope of our proof-of-concept implementation. Wuke-DHCP communicates with a flow monitor that records all active sessions of all hosts in the subnet. The flow monitor responds with the number of active connections that are sensitive to address changes. The communication between the DHCP server and the flow monitor is minimal (the question is 4 bytes as well as the answer) and does not impose additional overhead to the server. The flow monitor must be placed at the edge of the

subnet served by the DHCP server and is a lightweight process that processes packet headers of the underlying traffic.

Service fingerprinting examines traffic on the network and attempts to identify what services are running on each host. The purpose of service fingerprinting is two-fold. First, we want to supplement activity monitoring with some context to make address change decisions by indicating whether a connection failure is tolerable by the end-system. Second, we want to avoid assigning an address to a host that has significant overlap in services (and potential vulnerabilities) with hosts that recently used the same address. For instance, randomization between hosts with different operating systems, e.g., between a Windows and a Linux platform appears as a reasonable strategy. Our implementation of service fingerprinting is rudimentary: we only use port number information obtained through passive monitoring to identify OS and application characteristics. For instance, a TCP connection to port 80 suggests that the host is running a Web server, and port 445 is an indication that a host might be a Windows platform. In an operational setting, more elaborate techniques would be necessary, such as the passive techniques described in [30, 41] and active probing techniques implemented as part of open-source tools[11, 9, 8, 7].

In our implementation, we use three timers on the DHCP server for controlling host addresses. The *refresh* timer determines the duration of the lease communicated to the client. The client is forced to query the server when the timer expires. The server may or may not decide to renew the lease using the same address. The *soft-change* timer is used internally by the server to specify the interval between address changes, assuming that the flow monitor does not report any activity for the host. A third, *hard-change* timer is used to specify the maximum time that a host is allowed to keep the

same address. If this timer expires, the host is forced to change address, as the DHCP server does not renew the lease, despite the damage that may be caused. We explore the configuration of these timers in Section 6.

4

Measurements

To explore the design space of network address space randomization we first need to consider some basic hitlist characteristics, such as the speed at which a hitlist can be constructed, the rate at which addresses already change (without any form of randomization), and how address space is allocated and utilized. We perform measurements on the Internet to obtain a more clear picture of these characteristics.

4.1 Hitlist generation strategies

There are two key issues that need to be examined to determine how hitlist generation strategies relate to the effectiveness of NASR. First, we need to have a rough estimate of the speed at which an attacker can generate a hitlist. Second, we need to determine whether these strategies produce reasonably accurate hitlists, given that hitlists may decay naturally.

Unfortunately, we cannot accurately measure hitlist generation speeds. The speed that can be achieved will depend heavily on the defense mechanisms deployed, for which we do not have any robust operational data, as well as the generation strategies used, which we could not exhaustively analyze to produce a safe estimate.

We must note that although it seems reasonable to assume that IP-level stealth scans can take days or weeks to do properly, a skilled attacker may be able to use a botnet to speed up data collection. Systems such as DShield[1] and DOMINO[60] should be able to detect this activity, but the exact thresholds under which the attacker would have to operate to evade detection are unclear at this point.

We must also note that application-level probing appears as a bigger threat, as some distributed applications provide protocol functionality for crawling that can be exploited by an attacker to rapidly build hitlists. For example, by crawling through selected Gnutella superpeers, we were able to collect 520,000 unique IPs within 5 minutes. Normal crawling through regular peers was significantly slower, as we will discuss briefly in Section 4.1.2. Of course, additional probing would be needed to determine client software and version information, assuming that the worm can only infect specific software versions.

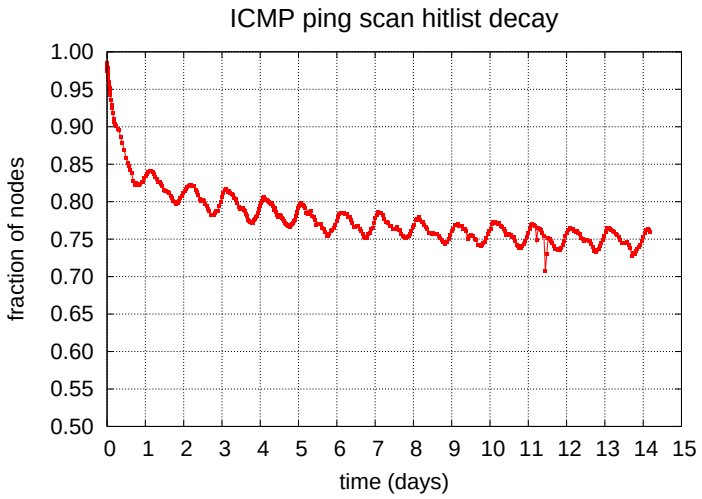

FIGURE 4.1: Decay of addresses harvested using random scanning

Given the complexity and intricacies of this question, we defer the answer to future work. For the purposes of this paper, it seems reasonable to expect that if such discovery functionality is determined to be dangerous, it may be disabled or at least carefully monitored. Recent experience with the *Santy* worm[10], that used *Google* to search for victims, seems to support this assumption, as *Google* quickly responded by blocking requests originating from the worm.

Next, we briefly present three different hitlist generation strategies and focus on their effectiveness in terms of natural decay rates.

4.1.1 Random scanning

We determine the effectiveness of random scanning for building hitlists. We first generate a list of all /16 prefixes that have a valid entry with the `whois` service, in order to increase scan success rates and avoid reserved address space. We then probe random targets within those prefixes using ICMP ECHO messages. Using this approach, we generated a hitlist of 20,000 ad-

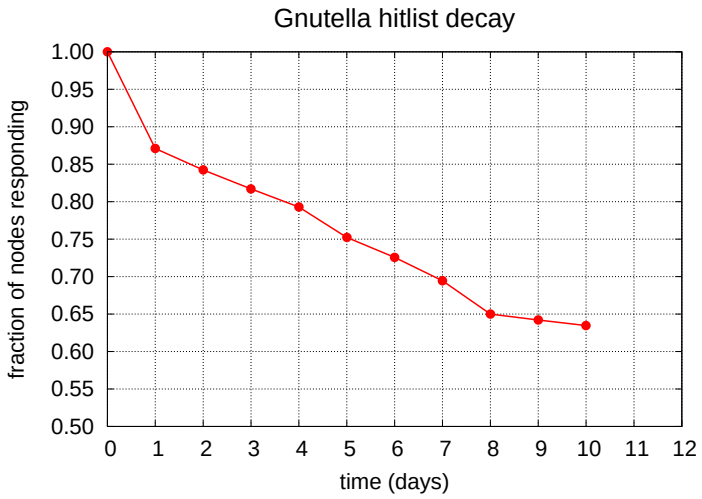

FIGURE 4.2: Decay of addresses harvested by monitoring peer-to-peer traffic routed through a node.

dresses. Given this hitlist, we probe each target in the hitlist once every hour for a period of two weeks. Every probe consists of four ICMP ECHO messages spaced out over the one-hour run in order to reduce the probability of accidentally declaring an entry stale (e.g., because of short-term congestion or connectivity problems). Note that these measurements do not give us exactly the probability of the worm successfully infecting the target host, but only a rough estimate. Although we were tempted to perform more insightful reconnaissance probes on the nodes in the hitlist, this would result in a much higher cost in terms of traffic and a high risk of causing (false) alarms at the target networks. More accurate results could be obtained using full port scans, application-level fingerprinting and more frequent probes needed for `ipid`-based detection of host changes[20, 34].

The results of the ICMP ECHO experiment are shown in Figure 4.1. We observe that the hitlist decays rapidly during the first day and continues to decay, albeit very slowly, over the rest of the two-week run. The number of

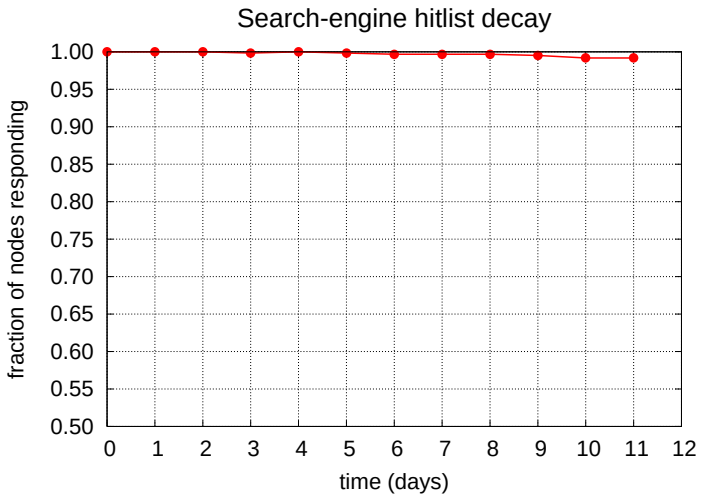

FIGURE 4.3: Decay of addresses harvested by querying a popular web search engine.

reachable nodes tends to vary during the time of day, apparently peaking on business hours in the US with minor peaks that may coincide with working hours elsewhere in the world. Overall, the decay of the hitlist slows down over time, reaching an almost stable level of 75% of hitlist nodes reachable.

4.1.2 Passive P2P snooping

In the Gnutella P2P network, node addresses are carried in QueryHit and Pong messages. By snooping on these messages, a Gnutella client can harvest thousands of addresses without performing any atypical operations. In our experiments, a 24-hour period sufficed for gathering 200K unique IP addresses, as shown in Figure 4.4. Intensive searches and the use of other, more popular P2P networks will probably result in a higher yield.

Most P2P nodes are short-lived, and therefore addresses harvested through P2P networks become unavailable very quickly. Figure 4.2 shows the decay of the hitlist as a function of elapsed time. Note that in this experiment we

only check whether the nodes respond to ICMP ECHO probes, not whether the Gnutella client is still up and running. Thus, it is possible that the IP address is not used by the same host recorded in the hitlist. This may or may not be important for the attacker, depending on how much the attack depends on software versions and whether version information has been used in constructing the hitlist.

4.1.3 Search-engine harvesting

Querying a popular search engine for *the* or similar keywords returns hundreds of millions of results. Retrieving a thousand results gave 612 unique alive hosts and 30 dead hosts. The retrieval time is equivalent to a typical request to a search engine, in our case nearly one second. Most search engines restrict the number of results that can be retrieved, but the attacker can use multiple keywords either randomly generated or taken from a dictionary.

The hosts that immediately appear as dead are a result of the frequency of the indexing by the search engine. It plays a role in the speed of harvesting the addresses and must be considered for the decay if the addresses are not checked.

Figure 4.3 shows the decay of the hitlist created using the search engine results. We observe that, compared to the other address sources, the search engine results are very stable. This was expected, since web servers have to be online and use stable addresses. It does not mean, however, that addresses retrieved through search engines are better suited for attackers. Depending on the vulnerability at hand, unprotected, client PCs, such as those returned by crawling peer-to-peer networks may be preferred.

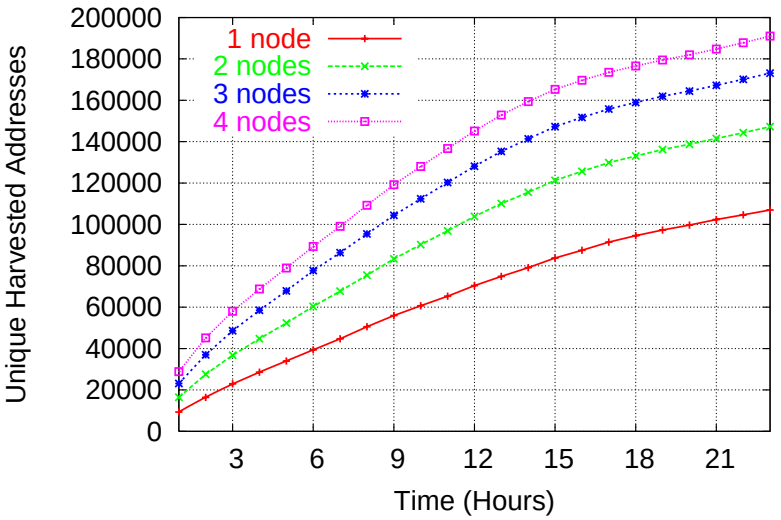

FIGURE 4.4: Number of distinct addresses harvested by monitoring Gnutella traffic as a function of time and number of monitoring nodes.

4.2 Subnet address space utilization

The feasibility and effectiveness of NASR depend on the fraction of unused addresses in NASR-enabled subnets. Performing randomization on a sparse subnet will result in more connection failures for the hitlist worm compared to a dense subnet. Such failures could expose the worm as they could be picked up by scan-detection mechanisms. In a dense subnet with homogeneous systems (e.g., running the same services) the worm is more likely to succeed in infecting a host, even if the original host recorded in the hitlist has actually changed its address. Finally, in the extreme (and probably rare) case of a subnet that is always fully utilized, there will never be a free address slot to facilitate address changes.

We attempt to get an estimate of typical subnet utilization levels. Because

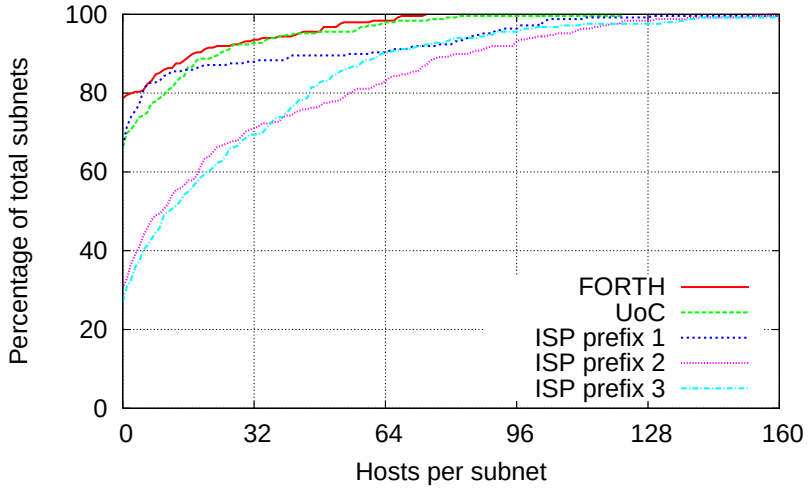

FIGURE 4.5: Subnet address space utilization

of the high scanning activity, we cannot perform this experiment globally without tripping a large number of alerts. We therefore opted for scanning five /16 prefixes that belong to FORTH, the University of Crete and a large ISP, after first obtaining permission by the administrators of the networks. We performed hourly scans on all prefixes using `ICMP ECHO` messages over a period of one month.

A summary of the results is shown in Figure 4.5. For simplicity, we assume that all prefixes are subnetted in /24's. We see that many subnets were completely dark with no hosts at all (not even a router). Nearly 30% of the subnets in two ISP prefixes were totally empty, while for the FORTH and UoC the percentage reaches 70%. This means that swapping subnets would likely be an effective NASR policy, but unfortunately it is not practical, as discussed in Section 3.2.1. We also see that 95% of these subnets have less than 50% utilization and the number of maximum live hosts observed does

not exceed 100. If subnet utilization at the global level is similar to what we see in our limited experiment, then NASR at the level of /24 subnets is likely to be quite effective, as there is sufficient room to move hosts around, reducing the effectiveness of the worm and causing it to make failed connections.

5

Impact of NASR on worm infection

It is infeasible to run experiments on the scale of the global Internet. To evaluate the effectiveness of our design, we simulated a small-scale (compared to the Internet) network of 1,000,000 hosts, each of which could be a potential target of worms.

Because of the variety of operating systems used and services provided, we assume that a fraction of hosts v is vulnerable to the worm. For simplicity, we ignore the details of the network topology, including the effect of end-to-end delays and traffic generated by the worm outbreak. We simply consider a flat topology of routers, each serving a subnet of end-hosts.

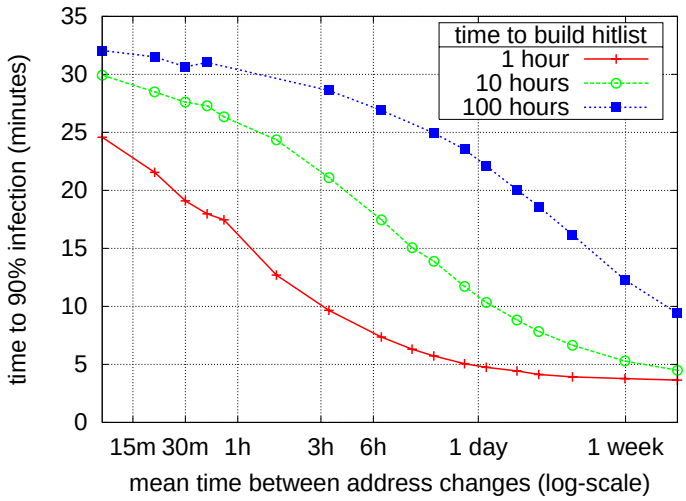

FIGURE 5.1: Worm spread time(time to 90% infection) vs. time between host address changes for different hitlist generation rates

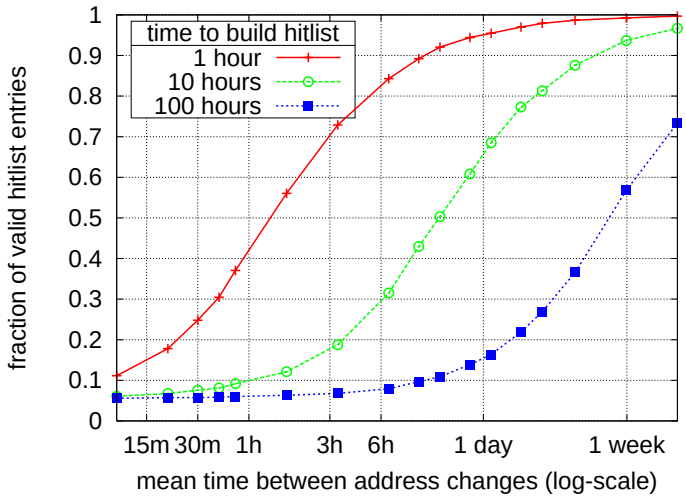

FIGURE 5.2: Effect of NASR on hitlist decay

A fraction of addresses is allocated in each subnet, which affects the probability of successful scan attempts within the subnet. This probability is an important parameter in the case where a host in the hitlist has changed its address, because it determines if *another* live host would be available

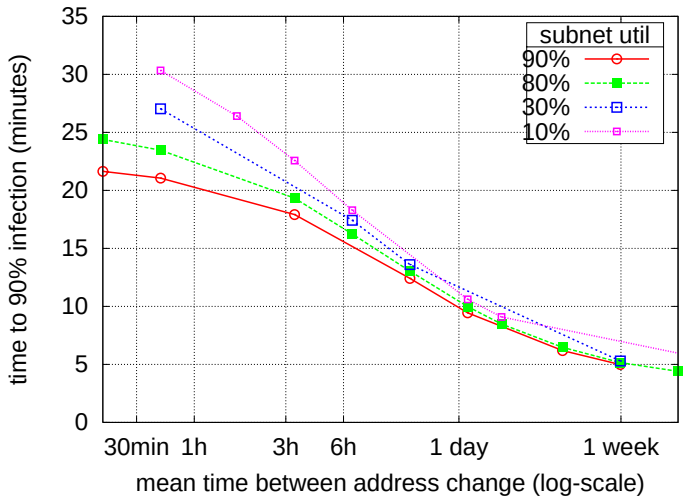

FIGURE 5.3: Effect of NASR vs. subnet usage density

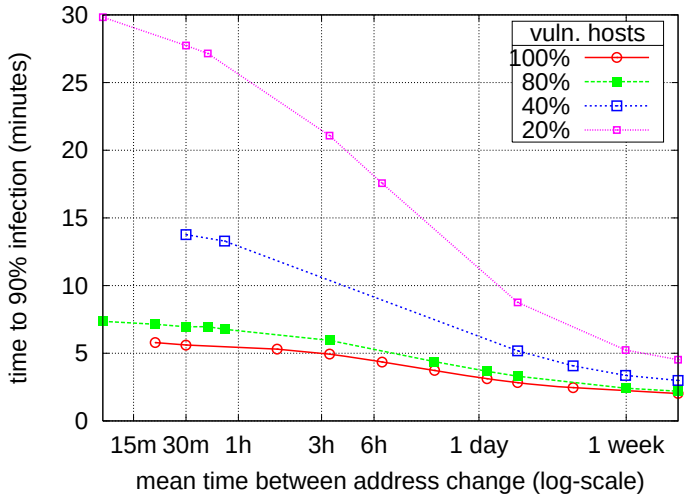

FIGURE 5.4: Effect of NASR for different vulnerable host populations

at the same address. A separate parameter is used for random scanning, reflecting the fraction of the overall address space that is completely unused.

The hitlist is generated at variable times (from 1 hour up to 4 days), and we assume that the worm starts spreading immediately after finishing with generating the hitlist. Because the early hitlist entries are more likely to have

become stale between their discovery and the start of the attack, the worm starts attacking the freshest addresses in the hitlist first. For simplicity, we ignore the details of how the hitlist is distributed and encoded in the payload of the worm: we assume that every worm instance can obtain the next available entry at zero cost. After finishing with the hitlist, we assume that the worm may continue trying to infect hosts using random scanning.

5.1 Impact of NASR

In the first experiment, we simulate worm outbreaks with different parameters, and measure the worm spread time, expressed in terms of the time required for the worm to infect 90% of the vulnerable hosts. We compare the impact of network address space randomization, varying how fast the hitlist is generated and how fast the host addresses are changed. The fraction of vulnerable hosts is 20%, the internal scan success probability is 0.3 (based on the subnet utilization measurements of Section 4.2) and the random scanning success probability is 0.05 (based on the measurements presented in Section 4.1.1).

The results are shown in Figure 5.1. We observe that NASR achieves the goal of slowing down the worm outbreak, in terms of the time to reach 90% infection, from 5 minutes when no NASR is used to between 24 and 32 minutes when hosts change their addresses very frequently. As expected, defending against hitlists that are generated very fast requires more frequent address changes. It appears that the mean time between address changes needs to be 3-5 times less than the time needed to generate the hitlist for the approach to reach around 80% of its maximum effectiveness, while more frequent address changes give diminishing returns. Considering the observations of Section 4.1, it appears that daily address changes could significantly

slow down a worm whose hitlist is generated by passive snooping on a P2P network.

Note that when using NASR, the hitlist worm is not completely reduced to a random-scanning worm: knowledge of subnets that have even one host available already gives the worm some advantage over a purely random-scanning worm. In this particular experiment, it would take roughly 30 minutes for the hitlist worm to infect the whole network (under NASR), and 2 hours for a purely scanning worm. This is the result of performing subnet-level instead of global-level NASR; global-level NASR would indeed reduce the hitlist worm to random-scanning. We must also note that although the spread times reported depend on scanning frequency, the relative improvement when using NASR appears to be constant.

The above experiment assumed that the hitlist worm will fall back to random scanning after exhausting the hitlist. For a pure hitlist worm, the fraction of nodes that are successfully infected is equal to the fraction of valid hitlist entries. The fraction of valid hitlist entries for different address change and hitlist generation times is shown in Figure 5.2. Again we observe that NASR is quite effective, even for short hitlist generation times.

We also simulated NASR with different fractions of vulnerable hosts, and average subnet utilization. The impact of NASR is greater in terms of slowing down the infection for smaller vulnerable populations. This is expected, as in such cases the failure rate for stale entries is higher compared to a network where every available host is vulnerable. The results for the impact of NASR as a function of subnet utilization are similar: higher subnet utilization results in a higher success rate when hitting stale entries. However, NASR remains effective even for 90% subnet utilization.

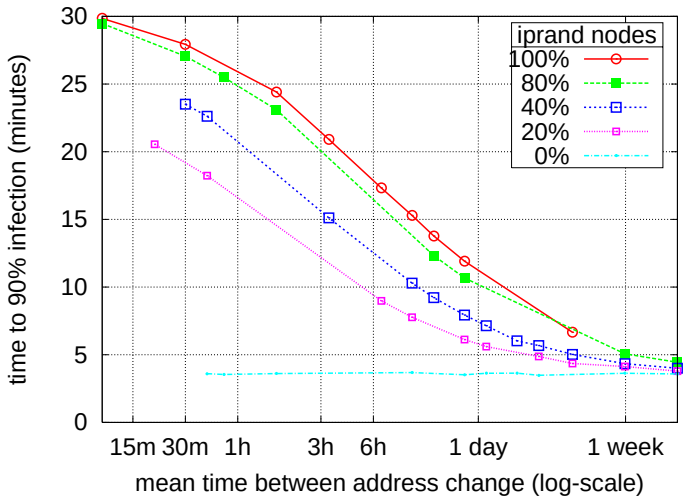

FIGURE 5.5: Effect of network address space randomization on worm spread time when partially deployed

5.2 Partial deployment scenario

We have so far assumed that NASR is deployed globally throughout the network. In reality, it is more likely that only a fraction of subnets will employ the mechanism, such as dynamic address pools. As we are not aware of any studies estimating the fraction of DHCP pools in the Internet, we measure the effectiveness of NASR for different values for the fraction of NASR-enabled subnets. The results are shown in Figure 5.5. We observe that NASR continues to be effective in slowing down the worm, even when deployed in 20% or 40% of the network. The worm still infects the non-NASR subnets quite rapidly, with a slowdown in the order of 50% caused by the worm failing to infect NASR subnets. In other words, NASR has a milder but still notable impact on non-NASR hosts. However, the worm will have to resort to random scanning after exhausting the hitlist, and it will take significantly more time to infect NASR compared to non-NASR

subnets. This observation suggests that there is a clear incentive for network administrators to deploy NASR, as it may provide them the critical amount of time needed to react to a worm outbreak.

5.3 Interaction with scan-blocking

Hitlist worms are generally immune to scan-blocking mechanisms such as [55]. Even for the natural decay rates measured in Section 4.1, such worms would still be *under* the detection threshold most of the time. Randomization, however, will cause many infection attempts to fail, as hosts change addresses and their previous addresses are either unused or used by a different host that may or may not run the same service, and thus may or may not be vulnerable. To determine the interaction between NASR and scan-blocking mechanism we simulate worm outbreaks in a network where both NASR and scan-blocking are deployed. As scan-blocking *contains* the outbreak, in this experiment we measure the maximum fraction of hosts that are infected in the presence of NASR together with scan-blocking. The results are shown in Figure 5.6. We observe that if NASR is performed according to the rule-of-thumb observation made previously (e.g., with address changes at a rate that is 3-5x faster than hitlist generation), the infection can be contained to under 15% of the vulnerable population.

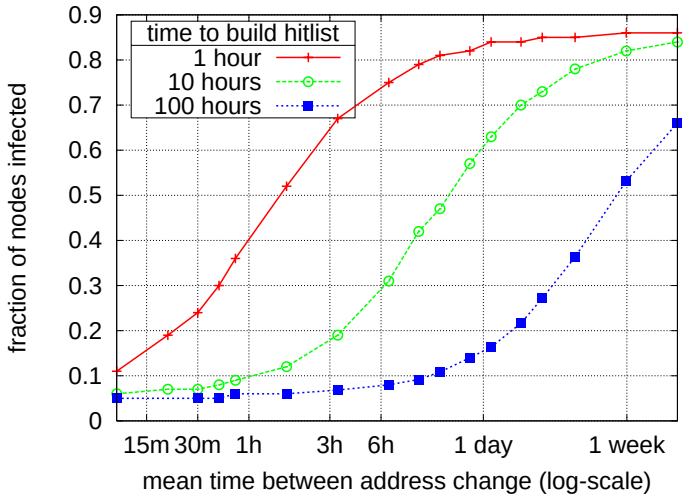

FIGURE 5.6: Maximum fraction of infected hosts vs. time between host address changes for different hitlist rates assuming scan-blocking mechanisms

6

The cost of NASR

We attempt to estimate the "collateral damage" caused by NASR. The damage depends on how frequently the address changes occur, whether hosts have active connections that are terminated and whether the applications can recover from the transient connectivity problems caused by an address change.

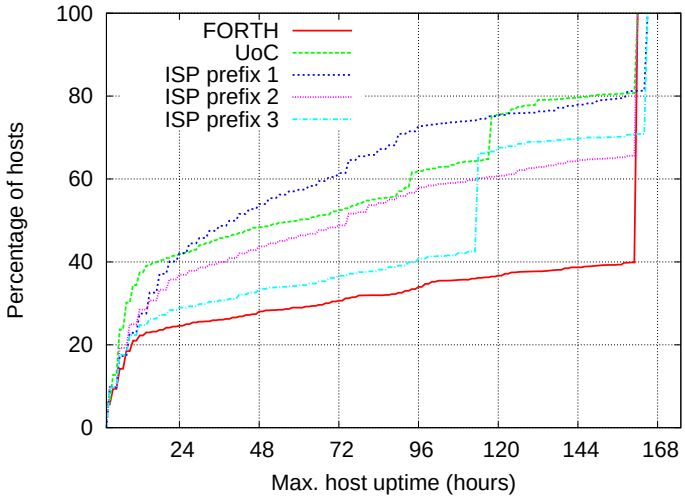

FIGURE 6.1: Distribution of host uptimes in 5 different networks

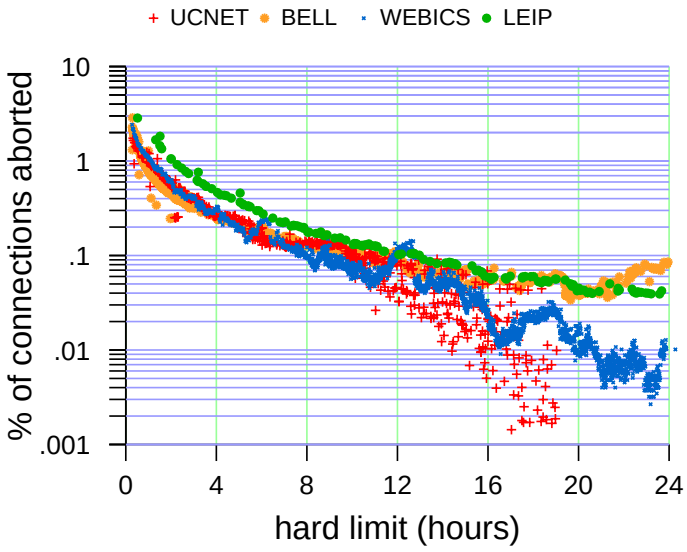

FIGURE 6.2: Percentage of aborted connections as a function of the hard change limit

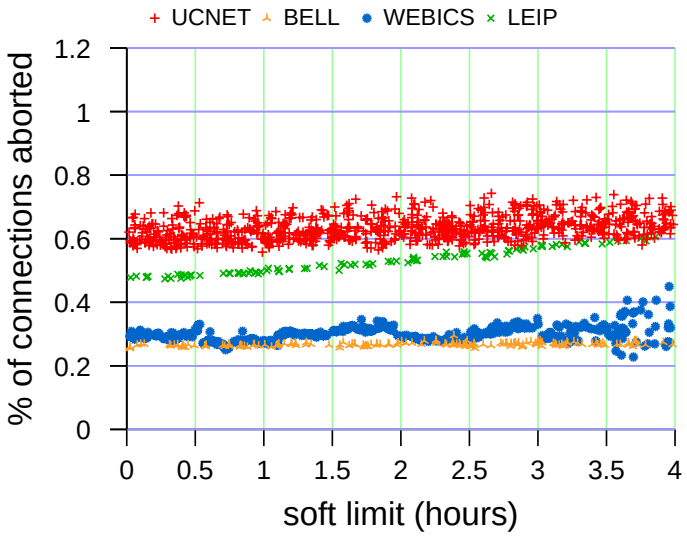

FIGURE 6.3: Percentage of aborted connections as a function of the soft change limit

6.1 Address change frequency vs. application failures

We first consider a scenario where host addresses are only changed when a node is rebooted. In this case, we know that the failure rate is zero, and try to determine what address change frequency this would permit. We measured the maximum uptime for hosts on the three networks presented previously.

The measured distribution is shown in Figure 6.1. The liveness of the hosts was monitored for a full week by sending `ping` messages every hour. Almost 60% of the hosts inside FORTH were always up, which seems reasonable for an environment consisting mostly of workstations. In more dynamic environments, like the ISP and the University of Crete networks only 20-30% of the hosts were continuously up and running, while nearly 40% of the hosts

had a maximum uptime of 10 hours. These results lead to two observations. First, although it may be possible to perform NASR once every 1-4 days for hosts only when they reboot, thus not causing any disruption, a significant fraction of hosts has a longer uptime. Considering that we may want to change addresses more aggressively, this trivial form of randomization is unlikely to be sufficient. Second, although such dynamic environments perform some form of natural randomization on their address space, mostly due to DHCP, most of the DHCP servers are configured to maintain leases for machines connecting to the network. The usual scenario is that a DHCP server is giving the same IP to a specific host (by caching its Ethernet address). Typically, a lease expires in 15 days, so hosts that do not refresh the lease before it expires (e.g., because they are not connected) would obtain a new address. Although we do not have measurements on how often this happens, it appears that this minor, slow form of randomization is unlikely to be effective by itself.

Given the above, we try to estimate the aborted connections caused by more aggressive randomization, by simulating NASR with different parameters on four different traces: a one-week contiguous IP header trace collected at Bell Labs research[3], a 5-day trace from the University of Leipzig[4], a 1-day trace from the University of Crete, and a 20-day trace from a link serving a single Web server at FORTH-ICS. For the first experiment, we use a refresh timer of 1 minute (the DHCP renewal period), a soft-change timer of 2 hours (interval on which we check if we can perform randomization) and vary the hard-change timer (interval on which randomization is enforced). The results are shown in Figure 6.2. As expected, there is a clear downward trend as the timer increases, consistent among different traces. An observation that initially surprised us was that the means of our samples did

not converge towards a smooth, monotonically decreasing function, despite hundreds of simulations for each value of the hard-timer and the initial "last-lease" times for each host randomized. The samples we obtained indicated a behavior that was almost deterministic. Indeed, a closer look revealed that the address change process for the same value of the hard-change timer is synchronized for each host across different simulations. The first synchronization point is the first successful soft-change event, which depends only on the timings of the flows in the trace and the soft-change timer, which both remain constant across different experiments. Thus, we consider this to be an artifact of our experiment.

We also examine how the failure rate is affected when we keep the hard-change timer constant, at 4 hours and vary the soft-change timer. The results are shown in Figure 6.3. We see very little change as we vary the soft-change timer. There is a small improvement as soft-change decreases, because we can hit a small number of additional connection-free hosts.

A closer examination of the raw data reveals that more than 90% of the failures come from a few highly active hosts. These hosts almost always have some active connections which will always be aborted, regardless of how much we relax the timer. Thus, it might make sense for the DHCP server to also make exceptions and not strictly enforce the hard-change limit for such hosts that are highly active, assuming they represent only a small fraction of hosts on the network. We also note that our analysis overestimates the failure rates because we do not filter out those applications that are resilient to aborted connections.

Overall, we observe that the failure rates are reasonable when compared to typical connection failure measurements on network links[14] and typical false positive rates of attack detection heuristics [51, 53, 46, 39]. However,

as we need to perform randomization in small timescales, where the failure rates wave between 3 and 5%, failure rates may not be acceptable. We can avoid network failures by using *transparent NASR*, an approach which needs more deployment resources than the standard NASR implementation. We describe the *transparent NASR* in the following section.

7

Transparent NASR

As shown in the previous Section, the damage caused by NASR in terms of aborted connections may not be acceptable in some cases. Terminating, for example, a large web transfer or an SSH session would be both irritating and frustrating. Additionally, it would potentionally increase network traffic as users or applications may repeat the aborted transfer or try to reconnect. To address these issues, we suggest `transparent NASR`, an external mechanism for deploying NASR avoiding connection failures.

The idea behind the mechanism is the existence of an "address randomization box" inside the LAN environment. This box performs the random-

ization on behalf of the end hosts, without the need of any modifications to the DHCP behavior, as suggested in Section 3.3. "Randomization box" controls all traffic passing by the subnet(s) it is responsible for, analogous to the firewall concept. The address used for communication between the host and the box remains the same. We should note that there is no need for private addresses, as end hosts can obtain any address from the organization they belong. The public address of the end host – that is the IP that outside world sees – changes from time to time according to soft and hard timers, similar to the procedure described in Section 3.3. Old connections continue to operate over the old address, the one that host had before the change, until they are terminated. The "randomization box" is responsible for two things. First, to prevent new connections on the old addresses reaching the host. Second, to perform address translation to the packets based on which connection they belong, similar to the NAT case. Until all old connections are terminated, a host would require multiple addresses to be allocated. The upper limit is when all addresses of the subnet are allocated, a case in which we revert to killing applications.

An example of how the "randomization box" works is illustrated in Figure 7.1. The box is responsible for address randomization on the 139.91.70.0/24 subnet, that is it can pick up addresses only from this subnet. Initially the host has the IP address 139.91.70.40 and "randomization box" sets that the public IP address of this host is 139.91.70.50. The host starts a new SSH connection to calliope.ics.forth.gr and sends packets with its own IP address (139.91.70.40). The box translates the source IP address based on the public one, setting it to 139.91.70.50. Simultaneously, the box keeps state that the connection from port 2000 to calliope.ics.forth.gr on port 22 belongs to the host with behind-the-box IP address 139.91.70.40 and public IP address

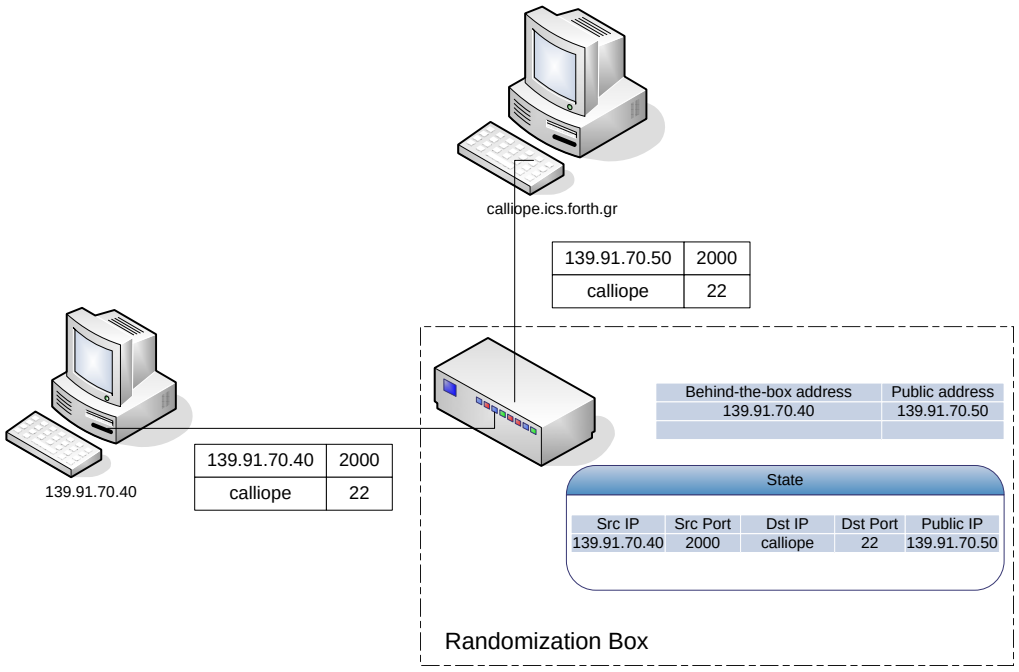

FIGURE 7.1: An example of NASR using the randomization box

139.91.70.50. Thus, on the calliope.ics.forth.gr side we see packets coming from 139.91.70.50. When calliope responds back to 139.91.70.50, box has to perform address translation. Consulting his state, it sees that this connection was initiated by host 139.91.70.40 so it rewrites the destination IP address. We should note here that "randomization box" is different from a NAT box instrumented to perform randomization. NAT box handles a single public IP address and shares it to multiple hosts (one-to-many) while our approach handles multiple IP addresses and assigns them to multiple hosts (many-to-many).

After an interval `t`, the time for public address change has arrived. The box sets the public IP address of the host to 139.91.70.60. Any connections

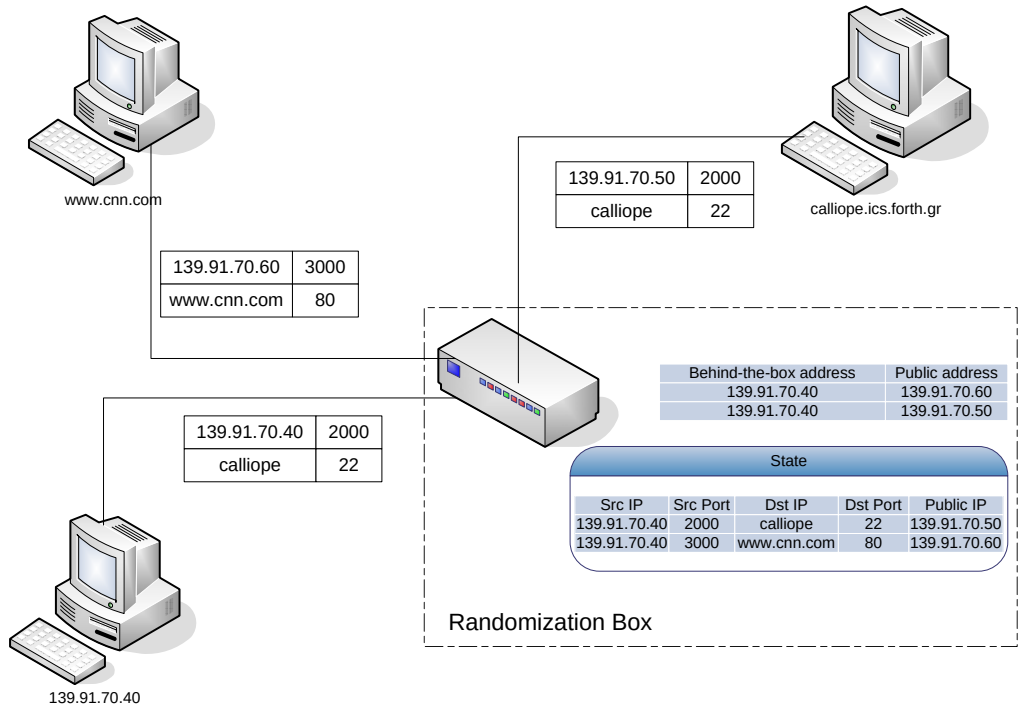

FIGURE 7.2: A more advanced example of NASR using the randomization box. Host has two public IP addresses, one (139.91.70.50) devoted for the SSH session to calliope and the other (139.91.70.60) for new connections

initiated by external hosts can reach the host through this new public IP address. As it can be seen in Figure 7.2 the new connection to www.cnn.com website has the new public IP as source. Note that in the behind-the-box and public address mapping table host now has two entries (the top is chosen for new connections). The only connection permitted to communicate with the host at 139.91.70.50 address is the SSH connection from calliope. For each incoming packet, the box checks its state to find an entry. If no entry is found, then packet is not forwarded to the internal hosts, else the "src IP"

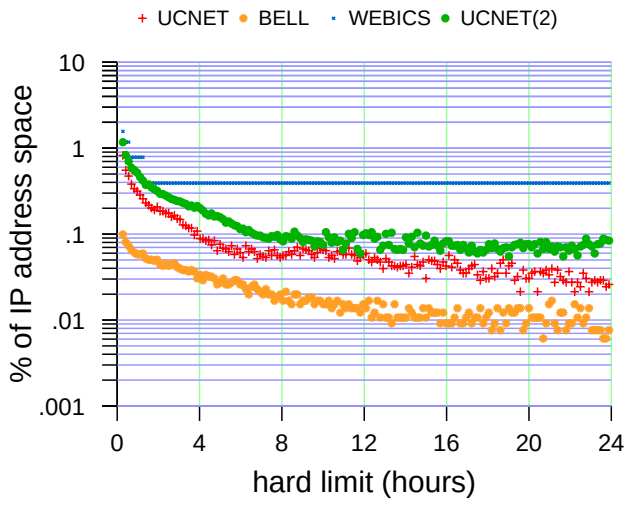

FIGURE 7.3: The percentage of extra IP space needed

field of the state is used to forward the packet. As long as the SSH connection lasts, the 139.91.70.50 IP will be bound to the particular host and cannot be assigned to any other internal host. When SSH session finishes, the address will be released. For stateless transport protocols, like UDP or ICMP, only the latest mapping between public and behind-the-box IP address is used.

The drawback of the "randomization box" is the extra address space required for keeping alive old connections. An excessive requirement of address space would empty the address pool, making the box abort connections. We tried to quantify the amount of extra space needed by simulating the "randomization box" on top of four traffic traces. The first two traces, `UCNET` and `UCNET(2)`, come from a local ISP and include traffic from 760 and 1675 hosts respectively. All hosts of this trace belong to a /16 subnet. The second trace, `BELL`, is a one-week contiguous IP header trace collected at Bell Labs research with 395 hosts located in a /16 subnet. Finally, the `WEBICS` trace

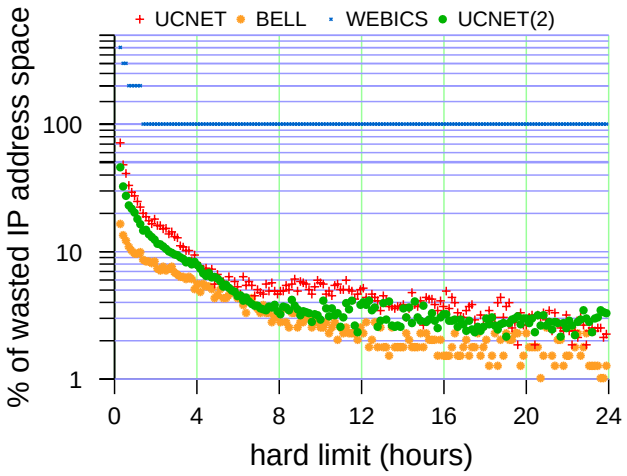

FIGURE 7.4: The percentage of extra IP space needed relative to the load of subnets

is a 20-day trace from a link serving a single Web server at FORTH-ICS. In this trace, we have only one host and we assume it is the only host in a /24 subnet. In our simulation, the soft timer had a constant value of 90 seconds, while the hard timer varied from 15 minutes to 24 hours. The results of the simulation are presented in Figure 7.3. In almost all cases, we need 1% more address space in order to keep alive the old connections. Combining this result with our observation from Section 4.2 that 95% of the subnets are less than half-loaded, we can safely assume that this 1% of extra space is not an obstacle in the operation of the "randomization box". However, the little extra address space needed derives from the fact that subnets are lightly loaded. For example, the 760 hosts of the `UCNET` trace correspond to the 1.15% of the /16 address space. In Figure 7.4, the relative results of the previous simulation are shown. On average, 10% more address space for hard timer over one hour is needed, which seems a reasonable overhead. In

the case of the `WEBICS` trace the percentage is 100% but this is expected as we have only one host.

We must admit that although transparent NASR overcomes the problem of aborted connections, it poses deployment overhead. Despite the fact that its configuration is simple, the setup of an extra network device always causes administrative overhead. A possible scenario would be to incorporate transparent NASR inside routers or switches.

8

Discussion

The experiments presented in Sections 5 suggest that network address space randomization is likely to be useful. However, these results should only be treated as preliminary, as there are several issues that need to be examined more closely before reaching any definite conclusions.

First, the interaction between NASR and other defense mechanisms needs to be studied in more depth. Our simulation results show that NASR enables scan-blocking mechanisms to contain the worm to under 15% infection. However, scan-blocking is not entirely foolproof, at least in its current form. For example, a list of *known repliers* can be used to defeat the failed-connection

test used by these mechanisms, by padding infection attempts with successful probes to the known repliers. Whether it is possible to design better mechanisms for detecting and containing scanning worms is thus still an open question. Therefore, we should also consider other possibilities, including reactive defenses and distributed detection mechanisms. As NASR is likely to at least slow down worms, it *may* provide the critical amount of time needed for distributed detectors such as DOMINO[60] to kick in, and for reactive approaches to deploy patches[45] or short-term filters[52]. Determining whether this is indeed a possibility requires further experimentation and analysis.

Second, we have so far focused entirely on IP-level address randomization, as IP hitlist worms seem to have the most efficient propagation properties. On the one hand, we have only considered IPv4 as deployed today. In an IPv6 Internet, the address space is so much bigger that randomization could be even more effective. On the other hand, we need to also consider worms that use higher-level addressing schemes, such as DNS or DHT identifiers. DNS hitlist worms will defeat NASR, assuming that hosts also update their DNS records. This would be true for Web servers, but when the DNS name is only a descriptor (such as a string containing the IP address), which is typical for DHCP and broadband address pools, a DNS-based hitlist worm would not be successful. DNS hitlist worms would also suffer the additional lookup latency, a slightly larger payload and the added risk of being detected. While we are not aware of any such detection mechanism in place today, it could be deployed, for example, on DNS servers.

With some simplifications, we can see how this is true for the case of DNS. IP-level NASR would be rendered useless if a DNS name hitlist is used, for example, for attacking Web servers, for which the DNS name will have to be updated under IP randomization so that *www.site.com* always points to the

correct IP. We measured the fully qualified domain name (FQDN) for several entries from search engine results. The average length was 16 bytes. Servers that hold web content tend to have shorter, more memorable names, so we expect that this is a conservative estimate. We measured a 46% compression ratio for these strings, and therefore on average each entry will take up 7.5 bytes in the hitlist. IP addresses take up 4 bytes, so storing DNS names causes almost a doubling of the hitlist size. The DNS lookups required for resolving the names also introduce latency. Resolving the names used in the previous paragraph results in an average latency of 1 second. It is possible to pipeline these requests, but massive DNS lookups may raise suspicion. While no such detection mechanism is in place now, it could be deployed, for example, on DNS servers.

An estimate of the time needed to resolve a hitlist of 1 million DNS names is described as follows. We assume that a compressed list of 1M DNS names needs 9MB of storage (average length of a DNS name and compression ratio are described above) and that attacker has X zombies at his disposal. Each zombie will receive 9/X MB of the compressed list and will try to translate 2M/X DNS names. If zombies are behind common DSL lines, with download rate of 512 KBps and upload rate of 128 KBps, then each zombie needs (9/X)/0.06 to receive the list, where 0.06 is measured rate of download rate of a 512 KBps DSL line in Mbytes/sec. According to [43] each DNS name takes 186 msec to be translated. Assuming that each zombie performs translation in a pipelined way with 5 stages, the total translation time is (2M/5/X)*0.186 secs. After translation is finished, the zombie must send back to the attacker 2M/X IP addresses, that is 8/X Mbytes. Measured upload rate of a 512 KBps line was found to be 0.01 Mbytes/sec, thus the transfer needs (8/X)/0.01 secs. The whole process needs (9/X)/0.06 + (400000/X)*0.186 + (8/X)/0.01

seconds. For X=1, this time is 75350 seconds, for X=100 decreases linearly to 753 seconds, while to get a translation time under 10 secs more than ten thousand zombies are needed. The maximum pipelining that can be achieved is upload_rate/53, where 53 is a typical size of a DNS request in bytes. With maximum pipelining (188 for 512 KBps DSL line), the time needed for X=1 decreases from 75350 to 2928 seconds, but generating this amount of DNS requests is extremely suspicious. For typical T1 lines, the time for one zombie is around 74435 (assuming pipeline of depth 5), showing that the dominant cost is the DNS requests. For T1 line with maximum pipelining (18867) and one zombie the time is decreased from 74435 to 55,38 seconds. Thus, while the worm is spread through the entire Internet, in practice, every infection has to be processed by the DNS system, involving orders of magnitude less hosts.

Third, we have not considered how worm creators would react to the widespread deployment of NASR. One option would be for the attacker to perform a second round of (stealthy) probing, and retain only entries that seem to be stable over time. If NASR is partially deployed, then the worm could infect the non-NASR part of the Internet, without being throttled by stale entries or generating too many failed connections. Interestingly, in this scenario all networks that employ NASR will be worm-free, unless the worm switches to random scanning after finishing with the hitlist. Even if this happens, NASR-enabled networks will still get infected much later than the nodes in the hitlist.

9

Related Work

Our work on network address space randomization was inspired by similar techniques for randomization performed at the OS level [58, 19, 59, 18, 42, 31, 17]. The general principle in randomization schemes is that attacks can be disrupted by reducing the knowledge that the attacker has about the system. For instance, instruction set randomization[31] changes the instruction set opcodes used on each host, so that an attacker cannot inject compiled code using the standard instruction set opcodes. Similarly, address obfuscation[18] changes the locations of functions in a host's address space so that buffer-overflow exploits cannot predict the addresses of the functions they would

like to utilize for hijacking control of the system. Our work at the network level is similar, as it reduces the ability of the attacker to build accurate hitlists of vulnerable hosts.

The use of IP address changes as a mechanism to defend against attacks was proposed independently in [15], [33] and [35]. Although these mechanisms are similar to ours, there are several important differences in the threat model as well as the way they are implemented. The main difference is that they focus on targeted attacks, performing address changes to confuse attackers during reconnaissance and planning. Neither project discusses or analyzes the use of such a mechanism for defending against worm attacks.

More specifically, the BBN DYNAT system[33] was developed as part of the DARPA Information Assurance Program exploring the area of dynamic network defense, with the hypothesis that dynamic network reconfiguration would inhibit an adversary's ability to gather intelligence, and thus degrade the ability to successfully launch an attack. BBN's DYNAT operates by obfuscating host identity information in TCP/IP headers when packets enter public parts of the network. The obfuscation algorithm depends on a pre-esta-blished keying parameter that varies with time. The evaluation shows that the adversary was a) severely time limited by the dynamic nature of the network, and b) forced into more vulnerable and detectable behavior. We raise the same arguments for defending typical LANs against hitlist worm attacks, the main difference being that in our case the clients are loosely coupled to the servers and therefore pre-established keying parameters were undesirable. In particular, the BBN approach requires a "shim" module to be installed on the client to coordinate address changes with the (modified) server, while in our approach we consider a DHCP-based implementation that is easier to deploy as it does not require any changes to the

client. However, client-side modifications make it easier for DYNAT to manage address changes without affecting applications, unlike the DHCP-based approach that requires additional care to minimize application disruption. The reason behind this difference in the two designs is that DYNAT assumes an adversary that can passively listen to client-server communication. In contrast, our work focuses on attackers performing scans and other active harvesting activities to build a worm hitlist.

The APOD (Applications That Participate in Their Own Defense) project [15] set out to develop technologies that increase an application's resilience against attacks. One of the mechanisms they describe, called Port and Address Hopping, is relevant to our work as it is designed to evade attacks against a service by constantly changing its IP address and TCP port using random numbers. The intention is both to hide the service's real identity and confuse the attacker during reconnaissance. Packets intercepted by attackers will reveal random addresses and ports, which are valid only for a small period of time, e.g., 1 minute. For an attack to be successful, the attacker must discover the current addresses and ports and execute the attack all within one refresh cycle. A stated additional benefit is the increased likelihood of an attacker being detected. This mechanism too relies on synchronization of random number generators and time synchronization between the two components. Port hopping, as opposed to address hopping, was not an option in our design due to the loose coupling between clients and servers. APOD also provides hopping functionality on protocol layers above TCP, such as distributed CORBA calls, which requires additional modification of TCP/IP data in the IIOP protocol. This feature would be a reasonable addition to our proposal.

Sandia's Dynamic Network Address Translation for network protection is

a similar proposal [35]. The authors discuss several types of dynamic address translation and point out that the use of this approach is dependent on many different factors which can influence overall effectiveness. With this in mind, they provide a detailed decision tree which allows the designer to determine which type of address translation is suitable for a particular environment.

10

Summary

We have explored the design and effectiveness of *network address space randomization* (NASR), a technique that hardens IP networks against hitlist worms. The idea behind NASR is to force network nodes to frequently change their network address in order to increase the staleness of information contained in hitlists. The approach is appealing in several ways. First, it is effective in limiting the infection for pure hitlist worms, or slowing down the infection for hybrid hitlist-scanning worms. Second, it forces both types of worms to exhibit scan-like behavior that exposes them to scan detection mechanisms, where available. Third, it is relatively easy to implement and

deploy. Unlike network-level detection mechanisms, NASR does not add any additional packet-level processing on network elements. Unlike host-based detection or other proactive mechanisms, it does not require any changes to the end-points.

We have discussed various constraints that limit the applicability of the proposed approach, such as the administrative overhead for managing address changes, services that require static addresses (such as routers and DNS servers) and applications that cannot tolerate address changes or suffer performance-wise when addresses change frequently. Our experiments indicate that the connection failure rates due to NASR are comparable to typical connection failure rates on modern networks and typical false positive rates of attack detection heuristics.

Our analysis suggests that network segments that *already* perform dynamic address allocation, such as DHCP pools for broadband connections, laptop subnets, wireless networks, etc., are the most suitable environment for deploying NASR without significantly impairing functionality or increasing administrative overhead. Assuming that broadband users are less likely to be vigilant and keep their systems secure, NASR appears promising. However, given that most worms so far have targeted servers, and until better defenses are available, we believe that the administrative overhead for implementing NASR is offset by the benefits of NASR, that effectively allow administrators to "opt-out" from hitlists.

Bibliography

[1] DShield: Distributed Intrusion Detection System. http://www.dshield.org.

[2] CERT Advisory CA-2001-19: 'Code Red' Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. http://www.cert.org/advisories/CA-2001-19.html, July 2001.

[3] NLANR-PMA Traffic Archive: Bell Labs-I trace. http://pma.nlanr.net/Traces/Traces/long/bell/1, 2002.

[4] NLANR-PMA Traffic Archive: Leipzig-I trace. http://pma.nlanr.net/Traces/Traces/long/leip/1, 2002.

[5] Cert Advisory CA-2003-04: MS-SQL Server Worm. http://www.cert.org/advisories/CA-2003-04.html, Jan. 2003.

[6] The Spread of the Sapphire/Slammer Worm. http://www.silicondefense.com/research/worms/slammer.php, Feb. 2003.

[7] DISCO: The Passive IP Discovery Tool. http://www.altmode.com/disco/, 2004.

[8] Fingerprinting: The complete documentation. http://www.l0t3k.org/security/docs/fingerprinting/, 2004.

[9] Fingerprinting: The complete toolsbox. http://www.l0t3k.org/security/tools/fingerprinting/, 2004.

[10] Net Worm Uses Google to Spread. http://it.slashdot.org/it/04/12/21/2135235.shtml, Dec. 2004.

[11] THC-Amap. http://thc.org/releases.php, 2004.

[12] K. G. Anagnostakis, M. B. Greenwald, S. Ioannidis, A. D. Keromytis, and D. Li. A Cooperative Immunization System for an Untrusting Internet. In *Proceedings of the 11th IEEE International Conference on Networking (ICON)*, pages 403–408, Sept./Oct. 2003.

[13] S. Antonatos, P. Akritidis, E. P. Markatos, and K. G. Anagnostakis. Defending against hitlist worms using network address space randomization. In *Proceedings of the 3rd ACM Workshop on Rapid Malcode*, Nov. 2005.

[14] M. Arlitt and C. Williamson. An Analysis of TCP Reset Behaviour on the Internet. *ACM SIGCOMM Computer Communication Review*, 35(1):37–44, 2005.

[15] M. Atighetchi, P. Pal, F. Webber, R. Schantz, and C. Jones. Adaptive use of network-centric mechanisms in cyber-defense. In *Proceedings of the 6th IEEE International Symposium on Object-oriented Real-time Distributed Computing*, May 2003.

[16] R. A. Baratto, S. Potter, G. Su, and J. Nieh. Mobidesk: mobile virtual desktop computing. In *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking (MOBICOM)*, pages 1–15. ACM Press, 2004.

[17] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection at-

tacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, Oct. 2003.

[18] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *In Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Aug. 2003.

[19] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, 1994.

[20] W. Chen, Y. Huang, B. F. Ribeiro, K. Suh, H. Zhang, E. de Souza e Silva, J. Kurose, and D. Towsley. Exploiting the IPID field to infer network path and end-system characteristics. In *Proceedings of the 6th Passive and Active Measurement Workshop (PAM 2005)*, Mar. 2005.

[21] F. Cohen. Computer Viruses: Theory and Practice. *Computers & Security*, 6:22–35, Feb. 1987.

[22] B. Croft and J. Gilmore. Bootstrap Protocol (BOOTP). RFC 951, http://www.rfc-editor.org/, Sept. 1985.

[23] R. Droms. Dynamic Host Configuration Protocol. RFC 2131, http://www.rfc-editor.org/, Mar. 1997.

[24] C. Fosnock. Computer worms: Past, present and future, 2005. `http://www.infosecwriters.com/text_resources/pdf/Computer_Worms_Past_Present_and_Future.pdf`.

[25] Internet Systems Consortium Inc. Dynamic host configuration protocol (DHCP) reference implementation. http://www.isc.org/sw/dhcp/.

[26] J. Ioannidis and G. Q. Maguire Jr. The design and implementation of a mobile internetworking architecture. In *USENIX Winter*, pages 489–502, 1993.

[27] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[28] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. In *Proceedings of the 1st ACM SIGCOMM Internet Measurement Workshop (IMW)*, Nov. 2001.

[29] M. Kaminsky, E. Peterson, D. B. Giffin, K. Fu, D. MaziG•res, and M. F. Kaashoek. REX: Secure, extensible remote execution. In *In Proceedings of the 2004 USENIX Technical Conference*, pages 199–212, June-July 2004.

[30] T. Karagiannis, A. Broido, M. Faloutsos, and K. claffy. Transport layer identification of P2P traffic. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 121–134, New York, NY, USA, 2004. ACM Press.

[31] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization . In *Proceedings of the ACM Computer and Communications Security Conference (CCS)*, pages 272–280, Oct. 2003.

[32] J. O. Kephart. A Biologically Inspired Immune System for Computers. In *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 130–139. MIT Press, 1994.

[33] D. Kewley, J. Lowry, R. Fink, and M. Dean. Dynamic approaches to thwart adversary intelligence gathering. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX)*, 2001.

[34] T. Kohno, A. Broido, and kc Claffy. Remote physical device fingerprinting. In *IEEE Symposium on Security and Privacy*, May 2005.

[35] J. Michalski, C. Price, E. Stanton, E. L. Chua, K. Seah, W. Y. Heng, and T. C. Pheng. Final Report for the Network Security Mechanisms Utilizing Network Address Translation LDRD Project. Technical Report SAND2002-3613, Sandia National Laboratories, November 2002.

[36] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security & Privacy*, pages 33–39, July/Aug. 2003.

[37] D. Moore, C. Shannon, and J. Brown. Code-Red: a case study on the spread and victims of an Internet worm. In *Proceedings of the 2nd Internet Measurement Workshop (IMW)*, pages 273–284, Nov. 2002.

[38] D. Nojiri, J. Rowe, and K. Levitt. Cooperative response strategies for large scale attack mitigation. In *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition (DISCEX)*, Apr. 2003.

[39] A. Pasupulati, J. Coit, K. Levitt, S. F. Wu, S. H. Li, J. C. Kuo, and K. P. Fan. Buttercup: On Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities. In *Proceedings of the Network Operations and Management Symposium (NOMS)*, pages 235–248, vol. 1, Apr. 2004.

[40] S. E. Schechter, J. Jung, and A. W. Berger. Fast Detection of Scanning Worm Infections. In *Proceedings of the 7^{th} International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 59–81, Oct. 2004.

[41] S. Sen, O. Spatscheck, and D. Wang. Accurate, scalable in-network identification of P2P traffic using application signatures. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 512–521, New York, NY, USA, 2004. ACM Press.

[42] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of*

the 11th ACM Conference on Computer and Communications Security, pages 298–307, New York, NY, USA, 2004. ACM Press.

[43] A. Shaikh, R. Tewari, and M. Agrawal. On the effectiveness of DNS-based server selection. In *Proceedings of the IEEE Infocom Conference*, Apr. 2001.

[44] C. Shannon and D. Moore. The spread of the witty worm, 2004. http://www.caida.org/analysis/security/witty/.

[45] S. Sidiroglou and A. D. Keromytis. A network worm vaccine architecture. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security*, June 2003.

[46] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the* 6^{th} *Symposium on Operating Systems Design & Implementation (OSDI)*, Dec. 2004.

[47] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 155–166, New York, NY, USA, 2000. ACM Press.

[48] S. Staniford. Containment of Scanning Worms in Enterprise Networks. *Journal of Computer Security*, 2004.

[49] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The top speed of flash worms. In *Proc. ACM CCS WORM*, Oct. 2004.

[50] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, Aug. 2002.

[51] T. Toth and C. Krügel. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Oct. 2002.

[52] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM'04*, pages 193–204, 2004.

[53] K. Wang and S. J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the* 7^{th} *International Symposium on Recent Advanced in Intrusion Detection (RAID)*, pages 201–222, Sept. 2004.

[54] N. Weaver and V. Paxson. A worst-case worm. In *Proc. Third Annual Workshop on Economics and Information Security (WEIS'04)*, May 2004.

[55] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 29–44, Aug. 2004.

[56] M. Williamson. Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code. Technical Report HPL-2002-172, HP Laboratories Bristol, 2002.

[57] J. Wu, S. Vangala, L. Gao, and K. Kwiat. An Effective Architecture and Algorithm for Detecting Worms with Various Scan Techniques. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 143–156, Feb. 2004.

[58] J. Xu, Z. Kalbarczyk, and R. Iyer. Transparent runtime randomization for security. In *A. Fantechi, editor, Proc. 22nd Symp. on Reliable Distributed Systems –SRDS 2003*, pages 260–269, Oct. 2003.

[59] C. Yarvin, R. Bukowski, and T. Anderson. Anonymous RPC: Low-latency protection in a 64-bit address space. In *In Proc. USENIX Summer 1993 Technical Conference*, pages 175–186, June 1993.

[60] V. Yegneswaran, P. Barford, and S. Jha. Global Intrusion Detection in the DOMINO Overlay System. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2004.

[61] C. C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and Early Warning for Internet Worms. In *Proceedings of the* 10^{th} *ACM International Conference on Computer and Communications Security (CCS)*, pages 190–199, Oct. 2003.

[62] C. C. Zou, W. Gong, and D. Towsley. Code Red Worm Propagation Modeling and Analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 138–147, Nov. 2002.