Surviving Internet Catastrophes

Flavio Junqueira, Ranjita Bhagwan, Alejandro Hevia, Keith Marzullo and Geoffrey M. Voelker
Department of Computer Science and Engineering
University of California, San Diego
{flavio,rbhagwan,ahevia,marzullo,voelker}@cs.ucsd.edu

Abstract

In this paper, we propose a new approach for designing distributed systems to survive Internet catastrophes called informed replication, and demonstrate this approach with the design and evaluation of a cooperative backup system called the Phoenix Recovery Service. Informed replication uses a model of correlated failures to exploit software diversity. The key observation that makes our approach both feasible and practical is that Internet catastrophes result from shared vulnerabilities. By replicating a system service on hosts that do not have the same vulnerabilities, an Internet pathogen that exploits a vulnerability is unlikely to cause all replicas to fail. To characterize software diversity in an Internet setting, we measure the software diversity of host operating systems and network services in a large organization. We then use insights from our measurement study to develop and evaluate heuristics for computing replica sets that have a number of attractive features. Our heuristics provide excellent reliability guarantees, result in low degree of replication, limit the storage burden on each host in the system, and lend themselves to a fully distributed implementation. We then present the design and prototype implementation of Phoenix, and evaluate it on the PlanetLab testbed.

1 Introduction

The Internet today is highly vulnerable to Internet epidemics: events in which a particularly virulent Internet pathogen, such as a worm or email virus, compromises a large number of hosts. Starting with the Code Red worm in 2001, which infected over 360,000 hosts in 14 hours [27], such pathogens have become increasingly virulent in terms of speed, extent, and sophistication. Sapphire scanned most IP addresses in less than 10 minutes [25], Nimda reportedly infected millions of hosts, and Witty exploited vulnerabilities in firewall software explicitly designed to defend hosts from such pathogens [26]. We call such epidemics *Internet catastrophes* because they result in extensive wide-spread damage costing billions of dollars [27]. Such damage ranges from overwhelming networks with epidemic traffic [25, 27], to providing zombies for spam relays [30] and denial of service attacks [35], to deleting disk blocks [26]. Given the current ease with which such pathogens can be created and launched, further Internet catastrophes are inevitable in the near future.

Defending hosts and the systems that run on them is therefore a critical problem, and one that has received considerable attention recently. Approaches to defend against Internet pathogens generally fall into three categories. Prevention reduces the size of the vulnerable host population [38, 41, 42]. Treatment reduces the rate of infection [9, 33]. Finally, containment techniques block infectious communication and reduce the contact rate of a spreading pathogen [28, 44, 45].

Such approaches can mitigate the impact of an Internet catastrophe, reducing the number of vulnerable and compromised hosts. However, they are unlikely to protect all vulnerable hosts or entirely prevent future epidemics and risk of catastrophes. For example, fast-scanning worms like Sapphire can quickly probe most hosts on the Internet, making it challenging for worm defenses to detect and react to them at Internet scale [28]. The recent Witty worm embodies a so-called *zero-day worm*, exploiting a vulnerability soon after patches were announced. Such pathogens make it increasingly difficult for organizations to patch vulnerabilities before a catastrophe occurs. As a result, we argue that defenses are necessary, but not sufficient, for fully protecting distributed systems and data on Internet hosts from catastrophes.

In this paper, we propose a new approach for designing distributed systems to survive Internet catastrophes called informed replication. The key observation that makes informed replication both feasible and practical is that Internet epidemics exploit shared vulnerabilities. By replicating a system service on hosts that do not have the same vulnerabilities, a pathogen that exploits one or more vulnerabilities cannot cause all replicas to fail. For example, to prevent a distributed system from failing due to a pathogen that exploits vulnerabilities in Web servers, the system can place replicas on hosts running different Web server software.

The software of every system inherently is a shared vulnerability that represents a risk to using the system, and systems designed to use informed replication are no different. Substantial effort has gone into making systems themselves more secure, and our design approach can certainly benefit from this effort. However, with the dramatic rise of worm epidemics, such systems are now increasingly at risk to large-scale failures due to vulnerabilities in *unrelated* software running on the host. Informed replication reduces this new source of risk.

This paper makes four contributions. First, we develop a system model using the *core* abstraction [15] to represent failure correlation in distributed systems. A core is a reliable minimal subset of components such that the probability of having all hosts in a core failing is negligible. To reason about the correlation of failures among hosts, we associate *attributes* with hosts. Attributes represent characteristics of the host that can make it prone to failure, such as its operating system and network services. Since hosts often have many characteristics that make it vulnerable to failure, we group host attributes together into *configurations* to represent the set of vulnerabilities for a host. A system can use the configurations of all hosts in the system to determine how many replicas are needed, and on which hosts those replicas should be placed, to survive a worm epidemic.

Second, the efficiency of informed replication fundamentally depends upon the degree of software diversity among the hosts in the system, as more homogeneous host populations result in a larger storage burden for particular hosts. To evaluate the degree of software heterogeneity found in an Internet setting, we measure and characterize the diversity of the operating systems and network services of hosts in the UCSD network. The operating system is important because it is the primary attribute differentiating hosts, and network services represent the targets for exploit by worms. The results of this study indicate that such networks have sufficient diversity to make informed replication feasible.

Third, we develop heuristics for computing cores that have a number of attractive features. They provide excellent reliability guarantees, ensuring that user data survives attacks of single- and double-exploit pathogens with probability greater than 0.99. They have low overhead, requiring fewer than 3 copies to cope with single-exploit pathogens, and fewer than 5 copies to cope with double-exploit pathogens. They bound the number of replica copies stored by any host, limiting the storage burden on any single host. Finally, the heuristics lend themselves to a fully distributed implementation for scalability. Any host can determine its replica set (its core) by contacting a constant number of other hosts in the system, independent of system size.

Finally, to demonstrate the feasibility and utility of our approach, we apply informed replication to the design and implementation of Phoenix. Phoenix is a cooperative, distributed remote backup system that protects stored data against Internet catastrophes that cause data loss [26]. The usage model of Phoenix is straightforward: users specify an amount F of bytes of their disk space for management by the system, and the system protects a proportional amount F/k of their data using storage provided by other hosts, for some value of k. We implement Phoenix as a service layered on the Pastry DHT [32] in the Macedon framework [31], and evaluate its ability to survive emulated catastrophes on the PlanetLab testbed.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes our system model for representing correlated failures. Section 4 describes our measurement study of the software diversity of hosts in a large network, and Section 5 describes and evaluates heuristics for computing cores. Section 6 describes the design and implementation of Phoenix, and Section 7 describes the evaluation of Phoenix. Finally, Section 8 concludes.

2 Related work

Most distributed systems are not designed such that failures are independent, and there has been recent interest in protocols for systems where failures are correlated. Quorum-based protocols, which implement replicated update by reading and writing overlapping subsets of replicas, are easily adapted to correlated failures. A model of dependent failures was introduced for Byzantine-tolerant quorum systems [23]. This model, called a *fail-prone system*, is a dual representation of the model (*cores*) that we use here. Our model was developed as part of a study of lower bounds and optimal protocols for Consensus in environments where failures can be correlated [15].

The ability of Internet pathogens to spread through a vulnerable host population on the network fundamentally depends on three properties of the network: the number of susceptible hosts that could be infected, the number of infected hosts actively spreading the pathogen, and the contact rate at which the pathogen spreads. Various approaches have been developed for defending against such epidemics that address each of these properties.

Prevention techniques, such as patching [24, 38, 42] and overflow guarding [7, 41], prevent pathogens from exploiting vulnerabilities, thereby reducing the size of the vulnerable host population and limiting the extent of a worm outbreak. However, these approaches have the traditional limitations of ensuring soundness and completeness, or leave windows of vulnerability due to the time required to develop, test, and deploy.

Treatment techniques, such as disinfection [6, 9] and vaccination [33], remove software vulnerabilities after they have been exploited and reduce the rate of infection as hosts are treated. However, such techniques are reactive in nature and hosts still become infected.

Containment techniques, such as throttling [21, 44] and filtering [28, 39], block infectious communication between infected and uninfected hosts, thereby reducing or potentially halting the contact rate of a spreading pathogen. The efficacy of reactive containment fundamentally depends upon the ability to quickly detect a new pathogen [19, 29, 37, 46], characterize it to create filters specific to infectious traffic [10, 16, 17, 34], and deploy such filters in the network [22, 40]. Unfortunately, containment at Internet scales is challenging, requiring short reaction times and extensive deployment [28, 45]. Again, since containment is inherently reactive, some hosts always become infected.

Various approaches take advantage of software heterogeneity to make systems fault-tolerant. N-version programming uses different implementations of the same service to prevent correlated failures across implementations. Castro's Byzantine fault tolerant NFS service (BFS) is one such example [4] and provides excellent fault-tolerant guarantees, but requires multiple implementations of every service. Scrambling the layout and execution of code can introduce heterogeneity into deployed software [1]. However, such approaches can make debugging, troubleshooting, and maintaining software considerably more challenging. In contrast, our approach takes advantage of existing software diversity.

Lastly, Phoenix is just one of many proposed cooperative systems for providing archival and backup services. For example, Intermemory [5] and Oceanstore [18] enable stored data to persist indefinitely on servers distributed across the Internet. As with Phoenix, Oceanstore proposes mechanisms to cope with correlated failures [43]. The approach, however, is reactive and does not enable recovery after Internet catastrophes. With Pastiche [8], pStore [2], and CIBS [20], users relinquish a fraction of their computing resources to collectively create a backup service. However, these systems target localized failures simply by storing replicas offsite. Such systems provide similar functionality as Phoenix, but are not designed to survive wide-spread correlated failures of Internet catastrophes. Finally, Glacier is a system specifically designed to survive highly correlated failures like Internet catastrophes [11]. In contrast to Phoenix, Glacier assumes a very weak failure model and instead copes with catastrophic failures via massive replication. Phoenix relies upon a stronger failure model, but replication in Phoenix is modest in comparison.

3 System model

As a first step toward the development of a technique to cope with Internet catastrophes, in this section we describe our system model for representing and reasoning about correlated failures, and discuss the granularity at which we represent software diversity.

3.1 Representing correlated failures

Consider a system composed of a set $\mathcal{H}$ of hosts each of which is capable of holding certain objects. These hosts can fail (for example, by crashing) and, to keep these objects available, they need to be replicated. A simple replication strategy is to determine the maximum number t of hosts that can fail at any time, and then maintain more than t replicas of each object.

However, using more than t replicas may lead to excessive replication when host failures are correlated. As a simple example, consider three hosts $\{h_1, h_2, h_3\}$ where the failures of h_1 and h_2 are correlated while h_3 fails independent of the other hosts. If h_1 fails, then the probability of h_2 failing is high. As a result, one might set $t = 2$ and thereby require $t + 1 = 3$ replicas. However, if we place replicas on h_1 and h_3, the object's availability may be acceptably high with just two replicas.

To better address issues of optimal replication in the face of correlated failures, we have defined an abstraction that we call a *core* [15]. A core is a minimal set of hosts such that, in any execution, at least one host in the core does not fail. In the above example, both $\{h_1, h_3\}$ and $\{h_2, h_3\}$ are cores. $\{h_1, h_2\}$ would not be a core since the probability of both failing is too high and $\{h_1, h_2, h_3\}$ would not be a core since it is not minimal. Using this terminology, a central problem of informed replication is the identification of cores based on the correlation of failures.

An Internet catastrophe causes hosts to fail in a correlated manner because all hosts running the targeted software are vulnerable. Operating systems and Web servers are examples of software commonly exploited by Internet pathogens [27, 36]. Hence we characterize a host's vulnerabilities by the software they run. We associate with each host a set of *attributes*, where each attribute is a canonical name of a software package or system that the host runs; in Section 3.2 below, we discuss the tradeoffs of representing software packages at different granularities. We call the combined representation of all attributes of a host the *configuration* of the host. An example of a configuration is $\{$*Windows, IIS, IE*$\}$, where *Windows* is a canonical name for an operating system, *IIS* for a Web server package, and *IE* for a Web browser. Agreeing on canonical names for attribute values is essential to ensure that dependencies of host failures are appropriately captured.

An Internet pathogen can be characterized by the set of attributes A that it targets. Any host that has none of the attributes in A is not susceptible to the pathogen. A core is a minimal set C of hosts such that, for each pathogen, there is a host h in C that is not susceptible to the pathogen. Internet pathogens often target a single (possibly cross-platform) vulnerability, and the ones that target multiple vulnerabilities target the same operating system. Assuming that any attribute is susceptible to attack, we can re-define a core using attributes: a core is a minimal set C of processes such that no attribute is common to all hosts in C. In Section 5.4, we relax this assumption and show how to extend our results to tolerate pathogens that can exploit multiple vulnerabilities.

To illustrate these concepts, consider the system described in Example 3.1. In this system, hosts are characterized by six attributes which we classify for clarity into operating system, Web server, and Web browser.

H_1 and H_2 comprise what we call an *orthogonal core*, which is a core composed of hosts that have disjoint configurations. Given our assumption that Internet pathogens target only one vulnerability or multiple vulnerabilities on one platform, an orthogonal core will contain two hosts. $\{H_1, H_3, H_4\}$ is also a core because there is no attribute present in all hosts, and it is minimal.

Example 3.1

Attributes: *Operating System* = {Unix, Windows}*;*
Web Server = {Apache, IIS}*;*
Web Browser = {IE, Netscape}.
Hosts: $H_1 = \{$Unix, Apache, Netscape$\}$*;*
$H_2 = \{$Windows, IIS, IE$\}$*;*
$H_3 = \{$Windows, IIS, Netscape$\}$*;*
$H_4 = \{$Windows, Apache, IE$\}$.
Cores $= \{\{H_1, H_2\}, \{H_1, H_3, H_4\}\}$.

The smaller core $\{H_1, H_2\}$ might appear to be the better choice since it requires less replication. Choosing the smallest core, however, can have an adverse effect on individual hosts if many hosts use this core for placing replicas. To represent this effect, we define *load* to be the amount of storage a host provides to other hosts. In environments where some configurations are rare, hosts with the rare configurations may occur in a large percentage of the smallest cores. Thus, hosts with rare configurations may have a significantly higher load than the other hosts. Indeed, having a rare configuration can increase a host's load even if the smallest core is not selected. For example, in Example 3.1, H_1 is the only host that has a flavor of Unix as its operating system. Consequently, H_1 is present in both cores.

To make our argument more concrete, consider the worms in Table 1, which are well-known worms unleashed in the past few years. For each worm, given two hosts with one not running Windows or not running a specific server such as a Web server or a database, at least one survives the attack. With even a very modest amount of heterogeneity, our method of constructing cores includes such pairs of hosts.

3.2 Attribute granularity

Attributes can represent software diversity at many different granularities. The choice of attribute granularity balances resilience to pathogens, flexibility for placing replicas, and degree of replication. An example of the coarsest representation is for a host to have a configuration comprising a single attribute for the generic class of operating system, e.g., "Windows", "Unix", etc. This single attribute represents the potential vulnerabilities of all versions of software running on all versions of the same class of operating system. As a result, replicas would always be placed on hosts with different operating systems. A less coarse representation is to have attributes for the operating system as well as all network services running on the host. This representation yields more freedom for placing replicas. For example, we can place replicas on hosts with the same class of operating system if they run different services. The core $\{H_1, H_3, H_4\}$ in Example 3.1 is an example of this situation since H_3 and H_4 both run Windows. More fine-grained representations can have attributes for different versions of operating systems and applications. For example, we can represent the various releases of Windows, such as "Windows 2000" and "Windows XP", or even versions such as "NT 4.0sp4" as attributes. Such fine-grained attributes provide considerable flexibility in placing replicas. For example, we can place a replica on an NT host and an XP host to protect against worms such as Code Red that exploit an NT service but not an XP service. But doing so greatly increases the cost and complexity of collecting and representing host attributes, as well as computing cores to determine replica sets.

Worm	Form of infection (Service)	Platform
Code Red	port 80/http (MS IIS)	Windows
Nimda	multiple: email; Trojan horse versions using open network shares (SMB: ports 137–139 and 445); port 80/HTTP (MS IIS); Code Red backdoors	Windows
Sapphire	port 1434/udp (MS SQL, MSDE)	Windows
Sasser	port 445/tcp (LSASS)	Windows
Witty	port 4000/udp (BlackICE)	Windows

Table 1: Recent well-known pathogens.

Our initial work [14] suggested that informed replication can be effective with relatively coarse-grained attributes for representing software diversity. As a result, we use attributes that represent just the class of operating system and network services on hosts in the system, and not their specific versions. In subsequent sections, we show that, when representing diversity at this granularity, hosts in an enterprise-scale network have substantial and sufficient software diversity for efficiently supporting informed replication. Our experience suggests that, although we can represent software diversity at finer attribute granularities such as specific software versions, there is not a compelling need to do so.

4 Host diversity

With informed replication, the difficulty of identifying cores and the resulting storage load depend on the actual distribution of attributes among a set of hosts. To better understand these two issues, we measured the software diversity of a large set of hosts at UCSD. In this section, we first describe the methodology we used, and discuss the biases and limitations our methodology imposes. We then characterize the operating system and network service attributes found on the hosts, as well as the host configurations formed by those attributes.

4.1 Methodology

On our behalf, UCSD Network Operations used the *Nmap* tool [12] to scan IP address blocks owned by UCSD to determine the host type, operating system, and network services running on the host. Nmap uses various scanning techniques to classify devices connected to the network. To determine operating systems, Nmap interacts with the TCP/IP stack on the host using various packet sequences or packet contents that produce known behaviors associated with specific operating system TCP/IP implementations. To determine the network services running on hosts, Nmap scans the host port space to identify all open TCP and UDP ports on the host. We anonymized host IP addresses prior to processing.

Due to administrative constraints collecting data, we obtained the operating system and port data at different times. We had a port trace collected between December 19–22, 2003, and an operating system trace collected between December 29, 2003 and January 7, 2004. The port trace contained 11,963 devices and the operating system trace contained 6,395 devices.

Because we are interested in host data, we first discarded entries for specialized devices such as printers, routers, and switches. We then merged these traces to produce a combined trace of hosts that contained both operating system data and open port data for the same set of hosts. When fingerprinting operating systems, Nmap determines both a class (e.g., Windows) as well as a version (e.g., Windows XP). For added consistency, we discarded host information for those entries that did not have consistent OS class and version info. The result was a data set with operating system and port data for 2,963 general-purpose hosts.

Our data set was constructed using assumptions that introduced biases. First, worms exploit vulnerabilities that are present in network services. We make the assumption that two hosts that have the same open port are running the same network service and thus have the same vulnerability. In fact, two hosts may use a given port to run different services, or even different versions (with different vulnerabilities) of the same service. Second, ignoring hosts that Nmap could not consistently fingerprint could bias the host traces that were used. Third, DHCP-assigned host addresses are reused. Given the time elapsed between the time operating system information was collected and port information was collected, an address in the operating system trace may refer to a different host in the port trace. Further, a host may appear multiple times with different addresses either in the port trace or in the operating system trace. Consequently, we may have combined information from different hosts to represent one host or counted the same host multiple times.

The first assumption can make two hosts appear to share vulnerabilities when in fact they do not, and the second assumption can consistently discard configurations that otherwise contribute to a less skewed distribution of configurations. The third assumption may make the distribution of configurations seem less skewed, but operating system and port counts either remain the same (if hosts do not appear multiple times in the traces) or increase due to repeated configurations.

OS	
Name	**Count (%)**
Windows	1604 (54.1)
Solaris	301 (10.1)
Mac OS X	296 (10.0)
Linux	296 (10.0)
Mac OS	204 (6.9)
FreeBSD	66 (2.2)
IRIX	60 (2.0)
HP-UX	32 (1.1)
BSD/OS	28 (0.9)
Tru64 Unix	22 (0.7)

(a)

Port	
Number	**Count (%)**
139 (netbios-ssn)	1640 (55.3)
135 (epmap)	1496 (50.4)
445 (microsoft-ds)	1157 (39.0)
22 (sshd)	910 (30.7)
111 (sunrpc)	750 (25.3)
1025 (various)	735 (24.8)
25 (smtp)	575 (19.4)
80 (httpd)	534 (18.0)
21 (ftpd)	528 (17.8)
515 (printer)	462 (15.6)

(b)

Table 2: Top 10 operating systems (a) and ports (b) among the 2,963 general-purpose hosts.

The net effect of our assumptions is to make operating system and port distributions appear to be less diverse than it really is, although it may have the opposite effect on the distribution of configurations.

Another bias arises from the environment we surveyed. A university environment is not necessarily representative of the Internet, or specific subsets of it. We suspect that such an environment is more diverse in terms of software use than other environments, such as the hosts in a corporate environment or in a governmental agency. On the other hand, there are perhaps thousands of universities with a large setting connected to the Internet around the globe, and so the conclusions we draw from our data are undoubtedly not singular.

4.2 Attributes

Together, the hosts in our study have 2,569 attributes representing operating systems and open ports. Table 2 shows the ten most prevalent operating systems and open ports identified on the general purpose hosts. Table 2.a shows the number and percentage of hosts running the named operating systems. As expected, Windows is the most prevalent OS (54% of general purpose hosts). Individually, Unix variants vary in prevalence (0.03–10%), but collectively they comprise a substantial fraction of the hosts (38%).

Table 2.b shows the most prevalent open ports on the hosts and the network services typically associated with those port numbers. These ports correspond to services running on hosts, and represent the points of vulnerability for hosts. On average, each host had seven ports open. However, the number of ports per host varied considerably, with 170 hosts only having one port open while one host (running a firewall software) had 180 ports open. Windows services dominate the network services running on hosts, with netbios-ssn (55%), epmap (50%), and domain services (39%) topping the list. The most prevalent services typically associated with Unix are sshd (31%) and sunrpc (25%). Web servers on port 80 are roughly as prevalent as ftp (18%).

These results show that the software diversity is signifi-

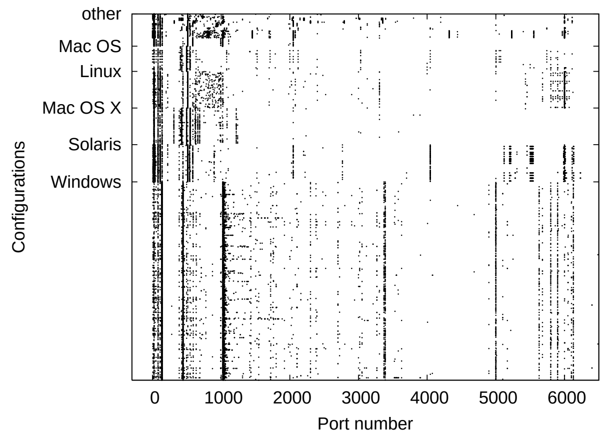

Figure 1: Visualization of UCSD configurations.

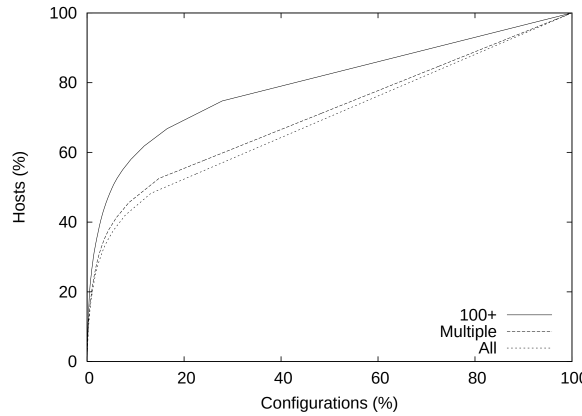

Figure 2: Distribution of configurations.

cantly skewed. Most hosts have open ports that are shared by many other hosts (Table 2.b lists specific examples). However, most attributes are found on few hosts, i.e., most open ports are open on only a few hosts. From our traces, we observe that the first 20 most prevalent attributes are found on 10% or more of hosts, but the remaining attributes are found on fewer hosts.

These results are encouraging for the process of finding cores. Having many attributes that are not widely shared makes it easier to find replicas that cover each other's attributes, preventing a correlated failure from affecting all replicas. We examine this issue next.

4.3 Configurations

Each host has multiple attributes comprised of its operating system and network services, and together these attributes determine its configuration. The distribution of configurations among the hosts in the system determines the difficulty of finding core replica sets. The more configurations shared by hosts, the more challenging it is to find small cores.

Figure 1 is a qualitative visualization of the space of host configurations. It shows a scatter plot of the host configurations among the UCSD hosts in our study. The x-axis is the port number space from 0–6500, and the y-axis covers the entire set of 2,963 host configurations grouped by operating system family. A dot corresponds to an open port on a host, and each horizontal slice of the scatter plot corresponds to the configuration of open ports for a given host. We sort groups in decreasing size according to the operating systems listed in Table 2: Windows hosts start at the bottom, then Solaris, Mac OS X, etc. Note that we have truncated the port space in the graph; hosts had open ports above 6500, but showing these ports did not add any additional insight and obscured patterns at lower, more prevalent port numbers.

Figure 1 shows a number of interesting features of the configuration space. The marked vertical bands within each group indicate, as one would expect, strong correlations of network services among hosts running the same general operating system. For example, most Windows hosts run the epmap (port 135) and netbios (port 139) services, and many Unix hosts run sshd (port 22) and X11 (port 6000). Also, in general, non-Windows hosts tend to have more open ports (8.3 on average) than Windows hosts (6.0 on average). However, the groups of hosts running the same operating system still have substantial diversity within the group. Although each group has strong bands, they also have a scattering of open ports between the bands contributing to diversity within the group. Lastly, there is substantial diversity among the groups. Windows hosts have different sets of open ports than hosts running variants of Unix, and these sets even differ among Unix variants. We take advantage of these characteristics to develop heuristics for determining cores in Section 5.

Figure 2 provides a quantitative evaluation of the diversity of host configurations. It shows the cumulative distribution of configurations across hosts for different classes of port attributes, with configurations on the x-axis sorted by decreasing order of prevalence. A distribution in which all configurations are equally prevalent would be a straight diagonal line. Instead, the results show that the distribution of configurations is skewed, with a majority of hosts accounting for only a small percentage of all configurations. For example, when considering all attributes, 50% of hosts comprise just 20% of configurations. In addition, reducing the number of port attributes considered further skews the distribution. For example, when only considering ports that appear on more than one host, shown by the "Multiple" line, 15% of the configurations represent over 50% of the hosts. And when considering only the port attributes that appear on at least 100 hosts, only 8% of the configurations represent over 50% of the hosts. Skew in the configuration distribution makes it more difficult to find cores for those hosts that share more prevalent configurations with other hosts. In the next section, however, we show that host populations with diversity similar to UCSD are sufficient for efficiently constructing cores that result in a low storage load.

5 Surviving catastrophes

With informed replication, each host h constructs a core *Core*(h) based on its configuration and the configuration of other hosts.[1] Unfortunately, computing a core of optimal size is NP-hard, as we have shown with a reduction from SET-COVER [13]. Hence, we use heuristics to compute *Core*(h). In this section, we first discuss a structure for representing advertised configurations that is amenable to heuristics for computing cores. We then describe four heuristics and evaluate via simulation the properties of the cores that they construct. As a basis for our simulations, we use the set of hosts $\mathcal{H}$ obtained from the traces discussed in Section 4.

5.1 Advertised configurations

Our heuristics are different versions of greedy algorithms: a host h repeatedly selects other hosts to include in *Core*(h) until some condition is met. Hence we chose a representation that makes it easier for a greedy algorithm to find good candidates to include in *Core*(h). This representation is a three-level hierarchy.

The top level of the hierarchy is the operating system that a host runs, the second level includes the applications that run on that operating system, and the third level are hosts. Each host runs one operating system, and so each host is subordinate to its operating system in the hierarchy (we can represent hosts running multiple virtual machines as multiple virtual hosts in a straightforward manner). Since most applications run predominately on one platform, hosts that run a different operating system than h are likely good candidates for including in *Core*(h). We call the first level the *containers* and the second level the *sub-containers*. Each sub-container contains a set of hosts. Figure 3 illustrates these abstractions using the configurations of Example 3.1.

More formally, let $\mathcal{O}$ be the set of canonical operating system names and $\mathcal{C}$ be the set of containers. Each host h has an attribute $h.os$ that is the canonical name of the operating system on h. The function $m_c : \mathcal{O} \rightarrow \mathcal{C}$ maps operating system name to container; thus, $m_c(h.os)$ is the container that contains h.

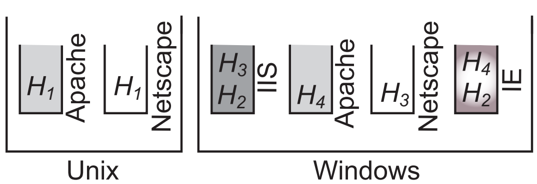

Figure 3: Illustration of containers and sub-containers.

Let $h.apps$ denote the set of canonical names of the applications that are running on h, and let $\mathcal{A}$ be the canonical names of all of the applications. We denote with $\mathcal{S}$ the set of sub-containers and with $m_s : \mathcal{C} \rightarrow 2^{\mathcal{S}}$ the function that maps a container to its sub-containers. The function $m_h : \mathcal{C} \times \mathcal{A} \rightarrow \mathcal{S}$ maps a container and application to a sub-container; thus, for each $a \in h.apps$, host h is in each sub-container $m_h(m_c(h.os), a)$.

At this high level of abstraction, advertising a configuration is straightforward. Initially $\mathcal{C}$ is empty. To advertise its configuration, a host h first ensures that there is a container $c \in \mathcal{C}$ such that $m_c(h.os) = c$. Then, for each attribute $a \in h.apps$, h ensures that there is a sub-container $m_h(c, a)$ containing h.

5.2 Computing cores

The heuristics we describe in this section compute *Core*(h) in time linear with the number of attributes in $h.apps$. These heuristics reference the set $\mathcal{C}$ of containers and the three functions m_c, m_s and m_h, but they do not reference the full set $\mathcal{A}$ of attributes. In addition, these heuristics do not enumerate $\mathcal{H}$, but they do reference the configuration of hosts (to reference the configuration of a host h', they reference $h'.os$ and $h'.apps$). Thus, the container/sub-container hierarchy is the only data structure that the heuristics use to compute cores.

5.2.1 Metrics

We evaluate our heuristics using three metrics:

- **Average core size**: $|$*Core*$(h)|$ averaged over all $h \in \mathcal{H}$. This metric is important because it determines how much capacity is available in the system. As the average core size increases, the total capacity of the system decreases.

- **Maximum load**: The load of a host h' is the number of cores *Core*(h) of which h' is a member. The maximum load is the largest load of any host $h' \in \mathcal{H}$.

- **Average coverage**: We say that an attribute a of a host h is *covered* in *Core*(h) if there is at least one other host h' in *Core*(h) that does not have a. Thus, an exploit of attribute a can affect h, but not h', and so not all hosts in *Core*(h) are affected. The *coverage* of *Core*(h) is the fraction of attributes of h that are covered. The *average coverage* is the average of the coverages of *Core*(h) over all hosts $h \in \mathcal{H}$. A high average coverage indicates a higher resilience to Internet catastrophes: many hosts have most or all of their attributes covered. We return to this discussion of what coverage means in practice in Section 5.3, after we present most of our simulation results for context.

For brevity, we use the terms core size, load, and coverage to indicate average core size, maximum load, and average coverage, respectively. Where we do refer to these terms in the context of a particular host, we say so explicitly.

[1] More precisely, *Core*(h) is a core constrained to contain h. That is, *Core*$(h) \setminus \{h\}$ may itself be minimal, but we require $h \in$ *Core*(h).

	Core size	Coverage	Load
Random	5	0.977	12
Uniform	2.56	0.9997	284
Weighted	2.64	0.9995	84
DWeighted	2.58	0.9997	91

Table 3: A typical run of the heuristics.

A good heuristic will determine cores with small size, low load, and high coverage. Coverage is the most critical metric because it determines how well it does in guaranteeing service in the event of a catastrophe. Coverage may not equal 1 either because there was no host h' that was available to cover an attribute a of h, or because the heuristic failed to identify such a host h'. As shown in the following sections, the second case rarely happens with our heuristics.

Note that, as a single number, the coverage of a given *Core*(h) does not fully capture its resilience. For example, consider host h_1 with two attributes and host h_2 with 10 attributes. If *Core*(h_1) covers only one attribute, then *Core*(h_1) has a coverage of 0.5. If *Core*(h_2) has the same coverage, then it covers only 5 of the 10 attributes. There are more ways to fail all of the hosts in *Core*(h_2) than those in *Core*(h_1). Thus, we also use the number of cores that do not have a coverage of 1.0 as an extension of the coverage metric.

5.2.2 Heuristics

We begin by using simulation to evaluate a naive heuristic called **Random** that we use as a basis for comparison. It is not a greedy heuristic and does not reference the advertised configurations. Instead, h simply chooses at random a subset of $\mathcal{H}$ of a given size containing h.

The first row of Table 3 shows the results of **Random** using one run of our simulator. We set the size of the cores to 5, i.e., **Random** chose 5 random hosts to form a core. The coverage of 0.977 may seem high, but there are still many cores that have uncovered attributes and choosing a core size smaller than five results in even lower coverage. The load is 12, which is significantly higher than the lower bound of 5.[2]

Our first greedy heuristic **Uniform** ("uniform" selection among operating systems) operates as follows. First, it chooses a host with a different operating system than $h.os$ to cover this attribute. Then, for each attribute $a \in h.apps$, it chooses both a container $c \in \mathcal{C} \setminus \{m_c(h.os)\}$ and a sub-container $sc \in m_s(c) \setminus \{m_h(c,a)\}$ at random. Finally, it chooses a host h' at random from sc. If $a \notin h'.apps$ then it includes h' in *Core*(h). Otherwise, it tries again by choosing a new container c, sub-container sc, and host h' at random. **Uniform** repeats this procedure `diff_OS` times in an attempt to cover a with *Core*(h). If it fails to cover a, then the heuristic tries up to `same_OS` times to cover a by choosing a sub-container $sc \in m_c(h.os)$ at random and a host h' at random from sc.

The goal for having two steps, one with `diff_OS` and another with `same_OS`, is to first exploit diversity across operating systems, and then to exploit diversity among hosts within the same operating system group. Referring back to Figure 1, the set of prevalent services among hosts running the same operating system varies across the different operating systems. In the case the attribute cannot be covered with hosts running other operating systems, the diversity within an operating system group may be sufficient to find a host h' without attribute a.

In all of our simulations, we set `diff_OS` to 7 and `same_OS` to 4. After experimentation, these values have provided a good trade-off between number of useless tries and obtaining good coverage. However, we have yet to study how to in general choose good values of `diff_OS` and `same_OS`.

Pseudo-code for **Uniform** is as follows.

```
Algorithm Uniform on input h:
integer i;
core ← {h};
C' ← C \ {m_c(h.os)}
for each attribute a ∈ h.apps
  i ← 0
  while (a is not covered) ∧
        (i ≤ diff_OS + same_OS)
   if (i ≤ diff_OS) choose randomly c ∈ C'
      else c ← m_c(h.os)
   choose randomly sc ∈ m_s(c) \ {m_h(c,a)}
   choose a host h' ∈ sc : h' ≠ h
   if (h' covers a) add h' to core
   i ← i + 1
return core
```

The second row of Table 3 shows the performance of **Uniform** for a representative run of our simulator. The core size is close to the minimum size of two, and the coverage is very close to the ideal value of one. This means that using **Uniform** results in significantly better capacity and improved resilience than **Random**. On the other hand, the load is very high: there is at least one host that participates in 284 cores. The load is so high because h chooses containers and sub-containers uniformly. When constructing the cores for hosts of a given operating system, the other containers are referenced roughly the same number of times. Thus, **Uniform** considers hosts running less prevalent operating systems for inclusion in cores a disproportionately large number of times. A similar argument holds for hosts running less popular applications.

This behavior suggests refining the heuristic to choose containers and applications weighted on the popularity of their operating systems and applications. Given a container c, let $N_c(c)$ be the number of distinct hosts in the sub-containers of c, and given a set of containers C, let $N_c(C)$ be the sum of $N_c(c)$ for all $c \in C$. The heuristic **Weighted** ("weighted" OS selection) is the same as **Uniform** except that for the

[2]To meet this bound, number the hosts in $\mathcal{H}$ from 0 to $|\mathcal{H}| - 1$. Let *Core*(h) be the hosts $\{h + i \pmod{|\mathcal{H}|} : i \in \{0,1,2,3,4\}\}$.

first *diff_OS* attempts, h chooses a container c with probability $N_c(c)/N_c(\mathcal{C} \setminus \{m_c(h.os)\})$. Heuristic **DWeighted** ("doubly-weighted" selection) takes this a step further. Let $N_s(c,a)$ be $|m_h(c,a)|$ and $N_s(c,A)$ be the size of the union of $m_h(c,a)$ for all $a \in A$. Heuristic **DWeighted** is the same as **Weighted** except that, when considering attribute $a \in h.apps$, h chooses a host from sub-container $m_h(c,a')$ with probability $N_s(c,a')/N_s(c,\mathcal{A} \setminus \{a\})$.

In the third and fourth rows of Table 3, we show a representative run of our simulator for both of these variations. The two variations result in comparable core sizes and coverage as **Uniform**, but significantly reduce the load. The load is still very high, though: at least one host ends up being assigned to over 80 cores.

Another approach to avoid a high load is to simply disallow it at the risk of decreasing the coverage. That is, for some value of L, once a host h' is included in L cores, h' is removed from the structure of advertised configurations. Thus, the load of any host is constrained to be no larger than L.

What is an effective value of L that reduces load while still providing good coverage? We answer this question by first establishing a lower bound on the value of L. Suppose that a is the most prevalent attribute (either service or operating system) among all attributes, and it is present in a fraction x of the host population. As a simple application of the pigeonhole principle, some host must be in at least l cores, where l is defined as:

$$l = \left\lceil \frac{|\mathcal{H}| \cdot x}{|\mathcal{H}| \cdot (1-x)} \right\rceil = \left\lceil \frac{x}{(1-x)} \right\rceil \tag{1}$$

Thus, the value of L cannot be smaller than l. Using Table 2, we have that the most prevalent attribute (port 139) is present in 55.3% of the hosts. In this case, $l = 2$.

Using simulation, we now evaluate our heuristics in terms of core size, coverage, and load as a function of the load limit L. Figures 4–7 present the results of our simulations. In these figures, we vary L from the minimum 2 through a high load of 10. All the points shown in these graphs are the averages of eight simulated runs with error bars (although they are too narrow to be seen in some cases). For Figures 4–6, we use the standard error to determine the limits of the error bars, whereas for Figure 7 we use the maximum and minimum observed among our samples. When using load limit as a threshold, the order in which hosts request cores from $\mathcal{H}$ will produce different results. In our experiments, we randomly choose eight different orders of enumerating $\mathcal{H}$ for constructing cores. For each heuristic, each run of the simulator uses a different order. Finally, we vary the core size of **Random** using the load limit L to illustrate its effectiveness across a range of core sizes.

Figure 4 shows the average core size for the four algorithms for different values of L. According to this graph, **Uniform**, **Weighted**, and **DWeighted** do not differ much in terms of

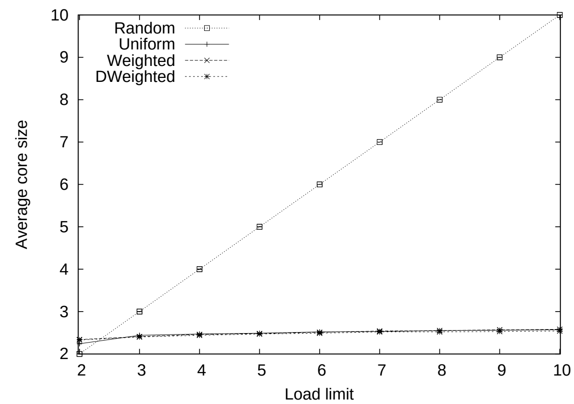

Figure 4: Average core size.

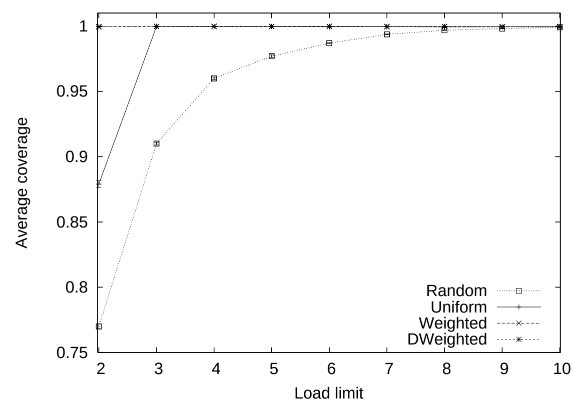

Figure 5: Average coverage.

core size. The average core size of **Random** increases linearly with L by design.

In Figure 5, we show results for coverage. Coverage is slightly smaller than 1.0 for **Uniform**, **Weighted**, and **DWeighted** when L is greater or equal to three. For $L = 2$, **Weighted** and **DWeighted** still have coverage slightly smaller than 1.0, but **Uniform** does significantly worse. Using weighted selection is useful when L is small. **Random** improves coverage with increasing L because the size of the cores increases. Note that, to reach the same value of coverage obtained by the other heuristics, **Random** requires a large core size of 9.

There are two other important observations to make about this graph. First, coverage is roughly the same for **Uniform**, **Weighted**, and **DWeighted** when $L > 2$. Second, as L continues to increase, there is a small decrease in coverage. This is due to the nature of our traces and to the random choices made by our algorithms. Ports such as 111 (portmapper, rpcbind) and 22 (sshd) are open on several of the hosts with operating systems different than Windows. For small values of L, these hosts rapidly reach their threshold. Consequently, when hosts that do have these services as attributes request a core, there are fewer hosts available with these same at-

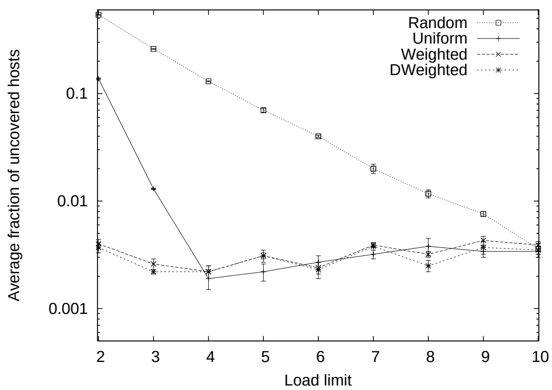

Figure 6: Average fraction of uncovered hosts.

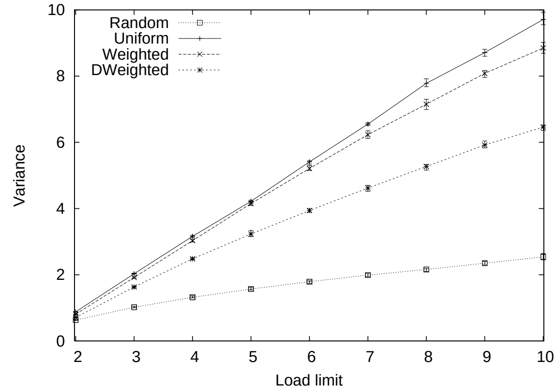

Figure 7: Average load variance.

tributes. On the other hand, for larger values of L, these hosts are more available, thus slightly increasing the probability that not all the attributes are covered for hosts executing an operating system different than Windows. We observed this phenomenon exactly with ports 22 and 111 in our traces.

This same phenomenon can be observed in Figure 6. In this figure, we plot the average fraction of hosts that are not fully covered, which is an alternative way of visualizing coverage. We observe that there is a share of the population of hosts that are not fully covered, but this share is very small for **Uniform** and its variations. Such a set is likely to exist due to the non-deterministic choices we make in our heuristics when forming cores. These uncovered hosts, however, are not fully unprotected. From our simulation traces, we note the average number of uncovered attributes is very small for **Uniform** and its variations. In all runs, we have just a few hosts that do not have all their attributes covered, and in the majority of the instances there is just a single uncovered attribute.

Finally, we show the resulting variance in load. Since the heuristics limit each host to be in no more than L cores, the maximum load equals L. The variance indicates how fairly the load is spread among the hosts. As expected, **Random** does well, having the lowest variance among all the algorithms and for all values of L. Ordering the greedy heuristics by their variance in load, we have **Uniform** $\succ$ **Weighted** $\succ$ **DWeighted**. This is not surprising since we introduced the weighted selection exactly to better balance the load. It is interesting to observe that for every value of L, the load variance obtained for **Uniform** is close to L. This means that there were several hosts not participating in any core and several other hosts participating in L cores.

A larger variance in load may not be objectionable in practice as long as a maximum load is enforced. Given the extra work of maintaining the functions N_s and N_c, the heuristic **Uniform** with small L ($L > 2$) is the best choice for our application. However, should load variance be an issue, we can use one of the other heuristics.

5.3 Translating to real pathogens

In this section, we discuss why we have chosen to tolerate exploits of vulnerabilities on a single attribute at a time. We do so based on information about past worms to support our choices and assumptions.

Worms such as the ones in Table 1 used services that have vulnerabilities as vectors for propagation. Code Red, for example, used a vulnerability in the IIS Web server to infect hosts. In this example, a vulnerability on a single attribute (Web server listening on port 80) was exploited. In other instances, such as with the Nimda worm, more than one vulnerability was exploited during propagation, such as via e-mail messages and Web browsing. Although these cases could be modeled as exploits to vulnerabilities on multiple attributes, we observe that previous worms did not propagate across operating system platforms: in fact, the worms targeted services on various versions of Windows.

By covering classes of operating systems in our cores, we guarantee that pathogens that exploit vulnerabilities on a single platform are not able to compromise all the members of a core C of a particular host h, assuming that C covers all attributes of h. Even if *Core*(h) leaves some attributes uncovered, h is still protected against attacks targeting covered attributes. Referring back to Figure 6, the majority of the cores have maximum coverage. We also observed in the previous section that, for cores that do not have maximum coverage, usually it is only a single uncovered attribute.

Under our assumptions, informed replication mitigates the effects of a worm that exploits vulnerabilities on a service that exists across multiple operating systems, and of a worm that exploits vulnerabilities on services in a single operating system. Figure 6 presents a conservative estimate on the percentage of the population that is unprotected in the case of an outbreak of such a pathogen. Assuming conservatively that every host that is not fully covered has the same uncovered attribute, the numbers in the graph give the fraction of the population that can be affected in the case of an outbreak. As can be seen, this fraction is very small.

With our current use of attributes to represent software heterogeneity, a worm can be effective only if it can exploit vulnerabilities in services that run across operating systems, or if it exploits vulnerabilities in multiple operating systems. To the best of our knowledge, there has been no large-scale outbreak of such a worm. Of course, such a worm could be written. In the next section, we discuss how to modify our heuristics to cope with exploits of vulnerabilities on multiple attributes.

5.4 Exploits of multiple attributes

To tolerate exploits on multiple attributes, we need to construct cores such that, for subsets of attributes possessed by members of a core, there must be a core member that does not have these attributes. We call a *k-resilient core C* a group of hosts in $\mathcal{H}$ such that, for every k attributes of members of C, there is at least one host in C that does not contain any of these attributes. In this terminology, the cores we have been considering up to this point have been 1-resilient cores.

To illustrate this idea, consider the following example. Hosts run *Windows*, *Linux*, and *Solaris* as operating systems, and *IIS*, *Apache*, and *Zeus* as Web servers. An example of a 2-resilient core is a subset composed of hosts h_1, h_2, h_3 with configurations: $h_1 = \{$Linux, Apache$\}$; $h_2 = \{$Windows, IIS$\}$; $h_3 = \{$Solaris, Zeus$\}$. In this core, for every pair of attributes, there is at least one host that contains none of them.

As before, every host h builds a k-resilient core *Core*(h). To build *Core*(h), host h uses the following heuristic:

Step 1 Select randomly $k-1$ hosts, h_1 through h_{k-1}, such that $h_i.os \neq h.os$, for every $i \in \{1, \ldots, k-1\}$;

Step 2 Use **Uniform** to search for a 1-resilient core C for h;

Step 3 For each $i \in \{1, \ldots, k-1\}$, use **Uniform** to search for a 1-resilient core C_i for h_i;

Step 4 *Core*$(h) \leftarrow C \cup C_1 \cup \ldots \cup C_{k-1}$.

Intuitively, to form a k-resilient core we need to gather enough hosts such that we can split these hosts into k subsets, where at least one subset is a 1-resilient core. Moreover, if there are two of these subsets where, for each subset, all of the members of that subset share some attribute, then the shared attribute of one set must be different from the shared attribute of the other set. Our heuristic is conservative in searching independently for 1-resilient cores because the problem does not require all such sets to be 1-resilient cores. In doing so, we protect clients and at the same time avoid the complexity of optimally determining such sets. The sets output by the heuristic, however, may not be minimal, and therefore they are approximations of theoretical cores. We discuss this heuristic further in [13].

In Table 4, we show simulation results for this heuristic for $k = 2$. The first column shows the values of load limit (L) used by the **Uniform** heuristic to compute cores. We chose

L	Avg. 2–coverage	Avg. 1–coverage	Avg. Core size
5	0.829 (0.002)	0.855 (0.002)	4.19 (0.004)
6	0.902 (0.002)	0.917 (0.002)	4.59 (0.005)
7	0.981 (0.001)	0.987 (0.001)	5.00 (0.005)
8	0.995 (0.0)	1.0 (0.0)	5.11 (0.005)
9	0.996 (0.0)	1.0 (0.0)	5.14 (0.005)
10	0.997 (0.0)	1.0 (0.0)	5.17 (0.003)

Table 4: Summary of simulation results for $k = 2$ for 8 different runs.

values of $L \geq 5$ based on an argument generalized from the one given in Section 5.2 giving the lower bound of L [13]. In the second and third columns, we present our measurements for coverage with standard error in parentheses. For each computed core *Core*(h), we calculate the fraction of pairs of attributes such that at least one host $h' \in$ *Core*(h) contains none of attributes of the pair. We name this metric **2-coverage**, and in the table we present the average across all hosts and across all eight runs of the simulator. **1-coverage** is the same as the average coverage metric defined in Section 5.2. Finally, the last column shows average core size.

According to the coverage results, the heuristic does well in finding cores that protect hosts against potential pathogens that exploit vulnerabilities in at most two attributes. A beneficial side-effect of protecting against exploits on two attributes is that the amount of diversity in a 2-resilient core permits better protection to its client against pathogens that exploit vulnerabilities on single attributes. For values of L greater than seven, all clients have all their attributes covered (the average 1-coverage metric is one and the standard error is zero).

Having a system that more broadly protects its hosts requires more resources: core sizes are larger to obtain sufficiently high degrees of coverage. Compared to the results in Section 5.2, we observe that we need to double the load limit to obtain similar values for coverage. This is not surprising. In our heuristic, for each host, we search for two 1-resilient cores. We therefore need to roughly double the amount of resources used.

Of course, there is a limit to what can be done with informed replication. As k increases, the demand on resources continues to grow, and a point will be reached in which there is not enough diversity to withstand an attack that targets $k+1$ attributes. Using our diversity study results in Table 2, if a worm were able to simultaneously infect machines that run one of the first four operating systems in this table, the worm could potentially infect 84% of the population. The release of such a worm would most likely cause the Internet to collapse. An approach beyond informed replication would be needed to combat an act of cyberterrorism of this magnitude.

6 The Phoenix Recovery Service

A cooperative recovery service is an attractive architecture for tolerating Internet catastrophes. It is attractive for both individual Internet users, like home broadband users, who do not

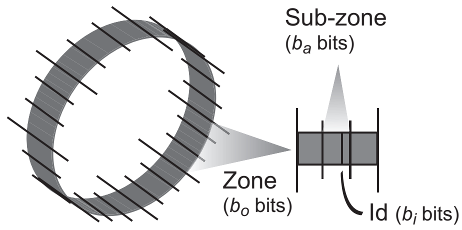

Figure 8: Phoenix ring.

wish to pay for commercial backup service or deal with the inconvenience of making manual backups, as well as corporate environments, which often have a significant amount of unused disk space per machine. If Phoenix were deployed, users would not need to exert significant effort to backup their data, and they would not require local backup systems. Phoenix makes specifying what data to protect as straightforward as specifying what data to share on file-sharing peer-to-peer systems. Further, a cooperative architecture has little cost in terms of time and money; instead, users relinquish a small fraction of their disk, CPU, and network resources to gain access to a highly resilient backup service.

As with Pastiche [8], we envision using Phoenix as a cooperative recovery service for user data. However, rather than exploiting redundant data on similar hosts to reduce backup costs for operating system and application software, we envision Phoenix users only backing up user-generated data and relying upon installation media to recover the operating system and application software. With this usage model, broadband users of Phoenix can recover 10 GB of user-generated data in a day. Given the relatively low capacity utilization of disks in desktop machines [3], 10 GB should be sufficient for a wide range of users. Further, users can choose to be more selective in the data backed up to reduce their recovery time. We return to the issue of bandwidth consumption and recovery time in Section 7.3.

6.1 System overview

A Phoenix host selects a subset of hosts to store backup data, expecting that at least one host in the subset survives an Internet catastrophe. This subset is a core, chosen using the **Uniform** heuristic described above.

Choosing cores requires knowledge of host software configurations. As described in Section 5, we use the container mechanism for advertising configurations. In our prototype, we implement containers using the Pastry [32] distributed hash table (DHT). Pastry is an overlay of nodes that have identifiers arranged in a ring. This overlay provides a scalable mechanism for routing requests to appropriate nodes.

Phoenix structures the DHT identifier space hierarchically. It splits the identifier space into *zones*, mapping containers to zones. It further splits zones into *sub-zones*, mapping sub-containers to equally-sized sub-zones. Figure 8 illustrates this hierarchy. Corresponding to the hierarchy, Phoenix creates host identifiers out of three parts. To generate its identifier, a host concatenates the hash representing its operating system $h.os$, the hash representing an attribute $a \in h.apps$, and the hash representing its IP address. As Figure 8 illustrates, each part has b_o, b_a, and b_i bits, respectively. To advertise its configuration, a host creates a hash for each one of its attributes. It therefore generates as many identifiers as the number of attributes in $h.apps$. It then joins the DHT at multiple points, each point being characterized by one of these identifiers. Since the hash of the operating system is the initial, "most significant" part of all the host's identifiers, all identifiers of a host lie within the same zone.

To build *Core*(h) using **Uniform**, host h selects hosts at random. When trying to cover an attribute a, h first selects a container at random, which corresponds to choosing a number c randomly from $[0, 2^{b_o} - 1]$. The next step is to select a sub-container and a host within this sub-container both at random. This corresponds to choosing a random number sc within $[0, 2^{b_a} - 1]$ and another random number id within $[0, 2^{b_i} - 1]$, respectively. Host h creates a Phoenix identifier by concatenating these various components as $(c \circ sc \circ id)$. It then performs a lookup on the Pastry DHT for this identifier. The host h' that satisfies this lookup informs h of its own configuration. If this configuration covers attribute a, h adds h' to its core. If not, h repeats this process.

The hosts in h's core maintain backups of its data. These hosts periodically send announcements to h. In the event of a catastrophe, if h loses its data, it waits for one of these announcements from a host in its core, say h'. After receiving such a message, h requests its data from h'. Since recovery is not time-critical, the period between consecutive announcements that a host sends can be large, from hours to a day.

A host may permanently leave the system after having backed up its files. In this situation, other hosts need not hold any backups for this host and can use garbage collection to retrieve storage used for the departed host's files. Thus, Phoenix hosts assume that if they do not receive an acknowledgment for any announcement sent for a large period of time (e.g., a week), then this host has left the system and its files can be discarded.

Since many hosts share the same operating systems, Phoenix identifiers are not mapped in a completely random fashion into the DHT identifier space. This could lead to some hosts receiving a disproportionate number of requests. For example, consider a host h that is either the first of a populated zone that follows an empty zone or is the last host of a populated zone that precedes an empty zone. Host h receives requests sent to the empty zone because, by the construction of the ring, its address space includes addresses of the empty zone. In our design, however, once a host reaches its load limit, it can simply discard new requests by the Phoenix protocol.

Experimenting with the Phoenix prototype, we found that constructing cores performed well even with an unbalanced ID space. But a simple optimization can improve core construction further. The system can maintain an OS hint list that contains canonical names of operating systems represented in the system. When constructing a core, a host then uses hashes of these names instead of generating a random number. Such a list could be maintained externally or generated by sampling. We present results for both approaches in Section 7.

We implemented Phoenix using the Macedon [31] framework for implementing overlay systems. The Phoenix client on a host takes a tar file of data to be backed up as input together with a host configuration. In the current implementation, users manually specify the host configuration. We are investigating techniques for automating the configuration determination, but we expect that, from a practical point of view, a user will want to have some say in which attributes are important.

6.2 Attacks on Phoenix

Phoenix uses informed replication to survive wide-spread failures due to exploits of vulnerabilities in unrelated software on hosts. However, Phoenix itself can also be the target of attacks mounted against the system, as well as attacks from within by misbehaving peers.

The most effective way to attack the Phoenix system as a whole is to unleash a pathogen that exploits a vulnerability in the Phoenix software. In other words, Phoenix itself represents a shared vulnerability for all hosts running the service. This shared vulnerability is not a covered attribute, hence an attack that exploits a vulnerability in the Phoenix software would make it possible for data to be lost as a pathogen spreads unchecked through the Phoenix system. To the extent possible, Phoenix relies on good programming practices and techniques to prevent common attacks such as buffer overflows. However, this kind of attack is not unique to Phoenix or the use of informed replication. Such an attack is a general problem for any distributed system designed to protect data, even those that use approaches other than informed replication [11]. A single system fundamentally represents a shared vulnerability; if an attacker can exploit a vulnerability in system software and compromise the system, the system cannot easily protect itself.

Alternatively, hosts participating in Phoenix can attack the system by trying to access private data, tamper with data, or mount denial-of-service attacks. To prevent malicious servers from accessing data without authorization or from tampering with data, we can use standard cryptographic techniques [13]. In particular, we can guarantee the following: (1) the privacy and integrity of any data saved by any host is preserved, and (2) if a client host contacts an honest server host for a backup operation, then the client is able to recover its data after a catastrophe. From a security perspective, the most relevant part of the system is the interaction process between a host client and a host server which has agreed to participate in the host's core.

Malicious servers can mount a denial-of-service attack against a client by agreeing to hold a replica copy of the client's data, and subsequently dropping the data or refusing recovery requests. One technique to identify such misbehavers is to issue *signed receipts* [13]. Clients can use such receipts to claim that servers are misbehaving. As we mentioned before, servers cannot corrupt data assuming robustness of the security primitives.

Hosts could also advertise false configurations in an attempt to free-ride in the system. By advertising attributes that make a host appear more unreliable, the system will consider the host for fewer cores than otherwise. As a result, a host may be able to have its data backed up without having to back up its share of data.

To provide a disincentive against free-riders, members of a core can maintain the configuration of hosts they serve, and serve a particular client only if their own configuration covers at least one client attribute. By sampling servers randomly, it is possible to reconstruct cores and eventually find misbehaving clients.

An important feature of our heuristic that constrains the impact of malicious hosts on the system is the load limit: if only a small percentage of hosts is malicious at any given time, then only a small fraction of hosts are impacted by the maliciousness. Hosts not respecting the limit can also be detected by random sampling.

7 Phoenix evaluation

In this Section, we evaluate our Phoenix prototype on the PlanetLab testbed using the metrics discussed in Section 5. We also simulate a catastrophic event — the simultaneous failure of all Windows hosts — to experiment with Phoenix's ability to recover from large failures. Finally, we discuss the time and bandwidth required to recover from catastrophes.

7.1 Prototype evaluation

We tested our prototype on 63 hosts across the Internet: 62 PlanetLab hosts and one UCSD host. To simulate the diversity we obtained in the study presented in Section 4, we selected 63 configurations at random from our set of 2,963 configurations of general-purpose hosts, and made each of these configurations an input to the Phoenix service on a host. In the population we have chosen randomly, out of the 63 configurations 38 have Windows as their operating system. Thus, in our setting roughly 60% of the hosts represent Windows hosts. From Section 5.2, the load limit has to be at least three.

For the results we present in this section, we use an OS hint list while searching for cores. Varying L, we obtained the values in Table 5 for coverage, core size, and load variance for a representative run of our prototype. For comparison, we

Load limit (L)	Core size		Coverage		Load var.	
	Imp.	Sim.	Imp.	Sim.	Imp.	Sim.
3	2.12	2.23	1.0	1.0	1.65	1.88
5	2.10	2.25	1.0	1.0	2.88	3.31
7	2.10	2.12	1.0	1.0	4.44	3.56

Table 5: Implementation results on PlanetLab ("Imp") with simulation results for comparison ("Sim").

also present results from our simulations with the same set of configurations used for the PlanetLab experiment. From the results in the table, coverage is perfect in all cases, and the average core size is less than 3 (less than 2 replica copies).

The major difference in increasing the value of L is the respective increase in load variance. As L increases, load balance worsens. We also counted the number of requests issued by each host in its search for a core. Different from our simulations, we set a large upper bound on the number of request messages ($\texttt{diff\_OS + same\_OS} = 100$) to verify the average number of requests necessary to build a core, and we had hosts searching for other hosts only outside their own zones ($\texttt{same\_OS}$ = 0). The averages for number of requests are 14.6, 5.2, and 4.1 for values of L of 3, 5, and 7, respectively. Hence, we can tradeoff load balance and message complexity.

We also ran experiments without using an OS hint list. The results are very good, although worse than the implementation that uses hint lists. We observed two main consequences in not using a hint list. First, the average number of requests is considerably higher (over 2x). Second, for small values of L ($L = 3, 5$), some hosts did not obtain perfect coverage.

7.2 Simulating catastrophes

Next we examine how the Phoenix prototype behaves in a severe catastrophe: the exploitation and failure of all Windows hosts in the system. This scenario corresponds to a situation in which a worm exploits a vulnerability present in *all* versions of Windows, and corrupts the data on the compromised hosts. Note that this scenario is far more catastrophic than what we have experienced with worms to date. The worms listed in Table 1, for example, exploit only particular services on Windows.

The simulation proceeded as follows. Using the same experimental setting as above, hosts backed up their data under a load limit constraint of $L = 3$. We then triggered a failure in all Windows hosts, causing the loss of data stored on them. Next we restarted the Phoenix service on the hosts, causing them to wait for announcements from other hosts in their cores (Section 6.1). We then observed which Windows hosts received announcements and successfully recovered their data.

All 38 hosts recovered their data in a reasonable amount of time. For 35 of these hosts, it took on average 100 seconds to recover their data. For the other three machines, it took several minutes due to intermittent network connectivity (these machines were in fact at the same site). Two important parameters that determine the time for a host to recover are the frequency of announcements and the backup file size (transfer time). We used an interval between two consecutive announcements to the same client of 120 seconds, and a total data size of 5 MB per host. The announcement frequency depends on the user expectation on recovery speed. In our case, we wanted to finish each experiment in a reasonable amount of time. Yet, we did not want to have hosts sending a large number of announcement messages unnecessarily. For the backup file size, we chose an arbitrary value since we are not concerned about transfer time in this experiment. On the other hand, this size was large enough to hinder recovery when connectivity between client and server was intermittent.

It is important to observe that we stressed our prototype by causing the failure of these hosts almost simultaneously. Although the number of nodes we used is small compared to the potential number of nodes that Phoenix can have as participants, we did not observe any obvious scalability problems. On the contrary, the use of a load limit helped in constraining the amount of work a host does for the system, independent of system size.

7.3 Recovering from a catastrophe

We now examine the bandwidth requirements for recovering from an Internet catastrophe. In a catastrophe, many hosts will lose their data. When the failed hosts come online again, they will want to recover their data from the remaining hosts that survived the catastrophe. With a large fraction of the hosts recovering simultaneously, a key question is what bandwidth demands the recovering hosts will place on the system.

The aggregate bandwidth required to recover from a catastrophe is a function of the amount of data stored by the failed hosts, the time window for recovery, and the fraction of hosts that fail. Consider a system of 10,000 hosts that have software configurations analogous to those presented in Section 4, where 54.1% of the hosts run Windows and the remaining run some other operating system. Next consider a catastrophe similar to the one above in which all Windows hosts, independent of version, lose the data they store. Table 6 shows the bandwidth required to recover the Windows hosts for various storage capacities and recovery periods. The first column shows the average amount of data a host stores in the system. The remaining columns show the bandwidth required to recover that data for different periods.

The first four rows show the aggregate system bandwidth required to recover the failed hosts: the total amount of data to recover divided by the recovery time. This bandwidth reflects the load on the network during recovery. Assuming a deployment over the Internet, even for relatively large backup sizes and short recovery periods, this load is small. Note that these results are for a system with 10,000 hosts and that, for an equivalent catastrophe, the aggregate bandwidth require-

Size (GB)	1 hour	1 day	1 week
Aggregate bandwidth			
0.1	1.2 Gb/s	50 Mb/s	7.1 Mb/s
1	12 Gb/s	0.50 Gb/s	71 Mb/s
10	120 Gb/s	5.0 Gb/s	710 Mb/s
100	1.2 Tb/s	50 Gb/s	7.1 Gb/s
Per-host bandwidth ($L = 3$)			
0.1	0.7 Mb/s	28 Kb/s	4.0 Kb/s
1	6.7 Mb/s	280 Kb/s	40 Kb/s
10	66.7 Mb/s	2.8 Mb/s	400 Kb/s
100	667 Mb/s	28 Mb/s	4.0 Mb/s

Table 6: Bandwidth consumption after a catastrophe.

ments will scale linearly with the number of hosts in the system and the amount of data backed up.

The second four rows show the average per-host bandwidth required by the hosts in the system responding to recovery requests. Recall that the system imposes a load limit L that caps the number of replicas any host will store. As a result, a host recovers at most L other hosts. Note that, because of the load limit, per-host bandwidth requirements for hosts involved in recovery are independent of both the number of hosts in the system and the number of hosts that fail.

The results in the table show the per-host bandwidth requirements with a load limit $L = 3$, where each host responds to at most three recovery requests. The results indicate that Phoenix can recover from a severe catastrophe in reasonable time periods for useful backup sizes. As with other cooperative backup systems like Pastiche [8], per-host recovery time will depend significantly on the connectivity of hosts in the system. For example, hosts connected by modems can serve as recovery hosts for a modest amount of backed up data (28 Kb/s for 100 MB of data recovered in a day). Such backup amounts would only be useful for recovering particularly critical data, or recovering frequent incremental backups stored in Phoenix relative to infrequent full backups using other methods (e.g., for users who take monthly full backups on media but use Phoenix for storing and recovering daily incrementals). Broadband hosts can recover failed hosts storing orders of magnitude more data (1–10 GB) in a day, and high-bandwidth hosts can recover either an order magnitude more quickly (hours) or even an order of magnitude more data (100 GB). Further, Phoenix could potentially exploit the parallelism of recovering from all surviving hosts in a core to further reduce recovery time.

Although there is no design constraint on the amount of data hosts back up in Phoenix, for current disk usage patterns, disk capacities, and host bandwidth connectivity, we envision users typically storing 1–10 GB in Phoenix and waiting a day to recover their data. According to a recent study, desktops with substantial disks (> 40 GB) use less than 10% of their local disk capacity, and operating system and temporary user files consume up to 4 GB [3]. Recovery times on the order of a day are also practical. For example, previous worm catastrophes took longer than a day for organizations to recover, and recovery using organization backup services can take a day for an administrator to respond to a request.

8 Conclusions

In this paper, we proposed a new approach called informed replication for designing distributed systems to survive Internet epidemics that cause catastrophic damage. Informed replication uses a model of correlated failures to exploit software diversity, providing high reliability with low replication overhead. Using host diversity characteristics derived from a measurement study of hosts on the UCSD campus, we developed and evaluated heuristics for determining the number and placement of replicas that have a number of attractive features. Our heuristics provide excellent reliability guarantees (over 0.99 probability that user data survives attacks of single- and double-exploit pathogens), result in low degree of replication (less than 3 copies for single-exploit pathogens; less than 5 copies for double-exploit pathogens), limit the storage burden on each host in the system, and lend themselves to a fully distributed implementation. We then used this approach in the design and implementation of a cooperative backup system called the Phoenix Recovery Service. Based upon our evaluation results, we conclude that our approach is a viable and attractive method for surviving Internet catastrophes.

Acknowledgements

We would like to express our gratitude to Pat Wilson and Joe Pomianek for providing us with the UCSD host traces. We would also like to thank Chip Killian for his valuable assistance with Macedon, Marvin McNett for system support for performing our experiments, Stefan Savage for helpful discussions, and our shepherd Steve Gribble, whose comments significantly improved this paper. Support for this work was provided in part by AFOSR MURI Contract F49620-02-1-0233 and DARPA FTN Contract N66001-01-1-8933.

References

[1] E. G. Barrantes et al. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM CCS*, pages 281–289, Washington D.C., USA, Oct. 2003.

[2] C. Batten, K. Barr, A. Saraf, and S. Treptin. pStore: A secure peer-to-peer backup system. Technical Report MIT-LCS-TM-632, MIT, Dec. 2001.

[3] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Hu. Kosha: A peer-to-peer enhancement for the network file system. In *Proc. of ACM/IEEE Supercomputing*, Pittsburgh, PA, Nov 2004.

[4] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of the 3rd ACM/USENIX OSDI*, pages 173–186, New Orleans, LA, Feb. 1999.

[5] Y. Chen. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, pages 28–37, Berkeley, CA, Aug. 1999.

[6] Codegreen. `http://www.winnetmag.com/Article/ArticleID/22381/22381.html`.

[7] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan. 1998.

[8] L. P. Cox and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. of the 5th ACM/USENIX OSDI*, pages 285–298, Boston, MA, Dec. 2002.

[9] CRclean. `http://www.winnetmag.com/Article/ArticleID/22381/22381.html`.

[10] H. A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proc. of the 13th USENIX Security Symposium*, pages 271–286, Aug. 2004.

[11] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of NSDI*, May 2005.

[12] Insecure.org. The nmap tool. `http://www.insecure.org/nmap`.

[13] F. Junqueira, R. Bhagwan, A. Hevia, K. Marzullo, and G. M. Voelker. Coping with internet catastrophes. Technical Report CS2005–0816, UCSD, Feb 2005.

[14] F. Junqueira, R. Bhagwan, K. Marzullo, S. Savage, and G. M. Voelker. The Phoenix Recovery System: Rebuilding from the ashes of an Internet catastrophe. In *Proc. of HotOS-IX*, pages 73–78, Lihue, HI, May 2003.

[15] F. Junqueira and K. Marzullo. Synchronous Consensus for dependent process failures. In *Proc. of ICDCS*, pages 274–283, Providence, RI, May 2003.

[16] J. O. Kephart and W. C. Arnold. Automatic extraction of computer virus signatures. In *Proceedings of the 4th Virus Bulletin International Conference*, pages 178–184, Abingdon, England, 1994.

[17] C. Kreibich and J. Crowcroft. Honeycomb – Creating intrusion detection signatures using honeypots. In *Proc. of HotNets-II*, pages 51–56, Cambridge, MA, Nov. 2003.

[18] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ACM ASPLOS*, pages 190–201, Cambridge, MA, 2000.

[19] J. Levin, R. LaBella, H. Owen, D. Contis, and B. Culver. The use of honeynets to detect exploited systems across large enterprise networks. In *Proc. of the IEEE Information Assurance Workshop*, pages 92–99, Atlanta, GA, June 2003.

[20] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative Internet backup scheme. In *Proc. of USENIX Annual Technical Conference*, pages 29–42, San Antonio, TX, 2003.

[21] T. Liston. Welcome To My Tarpit: The Tactical and Strategic Use of LaBrea. Technical report, 2001. `http://www.threenorth.com/LaBrea/LaBrea.txt`.

[22] J. W. Lockwood, J. Moscola, M. Kulig, D. Reddick, and T. Brooks. Internet worm and virus protection in dynamically reconfigurable hardware. In *Proc. of MAPLD*, Sept. 2003.

[23] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. of ACM STOC*, pages 569–578, El Paso, TX, 1997.

[24] Microsoft Corporation. Microsoft windows update. `http://windowsupdate.microsoft.com`.

[25] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer worm. *IEEE Privacy & Security*, 1(4):33–39, Jul 2003.

[26] D. Moore and C. Shannon. The spread of the Witty worm. http://www.caida.org/analysis/security/sitty/.

[27] D. Moore, C. Shannon, and J. Brown. Code Red: A case study on the spread and victims of an Internet worm. In *Proc. of ACM IMW*, pages 273–284, Marseille, France, Nov. 2002.

[28] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Internet quarantine: Requirements for containing self-propagating code. In *Proc. of IEEE Infocom*, pages 1901–1910, San Francisco, CA, Apr. 2003.

[29] D. Moore, G. M. Voelker, and S. Savage. Inferring Internet denial of service activity. In *Proc. of the USENIX Security Symposium*, Washington, D.C., Aug. 2001.

[30] Lurhq. mydoom word advisory. `http://www.lurhq.com/mydoomadvisory.html`.

[31] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proc. of NSDI*, pages 267–280, San Francisco, CA, Mar 2004.

[32] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. of ACM Middleware*, pages 329–350, Heidelberg, Germany, Nov. 2001.

[33] S. Sidiroglou and A. D. Keromytis. A network worm vaccine architecture. In *Proc. of IEEE Workshop on Enterprise Security*, pages 220–225, Linz, Austria, June 2003.

[34] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *Proc. of the 6th ACM/USENIX OSDI*, pages 45–60, San Francisco, CA, Dec. 2004.

[35] Lurhq. sobig.a and the spam you received today. `http://www.lurhq.com/sobig.html`.

[36] Sophos anti-virus. W32/sasser-a worm analysis. `http://www.sophos.com/virusinfo/analyses/w32sassera.html`, May 2004.

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

[38] Symantec. Symantec Security Response. `http://securityresponse.symantec.com/`.

[39] T. Toth and C. Kruegel. Connection-history Based Anomaly Detection. Technical Report TUV-1841-2002-34, Technical University of Vienna, June 2002.

[40] J. Twycross and M. M. Williamson. Implementing and testing a virus throttle. In *Proc. of the 12th USENIX Security Symposium*, pages 285–294, Washington, D.C., Aug. 2003.

[41] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First step towards automated detection of buffer overrun vulnerabilities. In *Proc. of NDSS*, pages 3–17, San Diego, CA, Feb. 2000.

[42] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proc. of ACM SIGCOMM*, pages 193–204, Portland, Oregon, Aug. 2004.

[43] H. Weatherspoon, T. Moscovitz, and J. Kubiatowicz. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proc. of International Workshop on Reliable Peer-to-Peer Distributed Systems*, Oct. 2002.

[44] M. Williamson. Throttling Viruses: Restricting propagation to defeat malicious mobile code. Technical Report HPL-2002-172, HP Laboratories Bristol, June 2002.

[45] C. Wong et al. Dynamic quarantine of Internet worms. In *Proc. of DSN*, pages 73–82, Florence, Italy, June 2004.

[46] C. C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and early warning for Internet worms. In *Proceedings of the 10th ACM CCS*, pages 190–199, Washington D.C., USA, Oct. 2003.