Coping with 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

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 replication factors, 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.**

I. 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 [49], such pathogens have become increasingly virulent in terms of speed, extent, and sophistication. Sapphire scanned most IP addresses in less than 10 minutes [47], Nimda reportedly infected millions of hosts, and Witty exploited vulnerabilities in firewall software explicitly designed to defend hosts from such pathogens [48]. We call such epidemics *Internet catastrophes* because they result in extensive wide-spread damage costing billions of dollars [49]. Such damage ranges from overwhelming networks with epidemic traffic [47], [49], to providing zombies for spam relays [52] and denial of service attacks [64], to deleting disk blocks [48]. 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 techniques, such as patching and overflow guarding, preclude pathogens from exploiting vulnerabilities and thereby reduce the size of the vulnerable host population and limit the extent of an outbreak [68], [72], [73]. Treatment techniques, such as disinfection and vaccination, remove software vulnerabilities after they have been exploited and thereby reduce the rate of infection as hosts are treated [18], [62]. Containment techniques, such as throttling and filtering, block infectious communication and reduce the contact rate of a spreading pathogen [50], [76], [77].

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 [50]. The recent Witty worm embodies a so-called *zero-day worm*, exploiting a vulnerability very 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 entirely 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 a vulnerability 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.

A service implemented as a distributed system is at risk to vulnerabilities in its own software as well as the other software on its hosts. The software of every system inherently is a shared vulnerability that represents a risk to using the system, and substantial effort has gone into making systems themselves more secure. 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 is an approach that reduces this new source of increased risk.

This paper makes four contributions. First, we develop a system model using the *core* abstraction [35] 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. More homogeneous host populations require more replicas. 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: 1) They provide excellent reliability guarantees; 2) They have low overhead; 3) They bound the number of replicas stored by any host, limiting the storage burden on any single host. Additionally, 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 [48]. 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 [60] in the Macedon framework [59]. We evaluate the performance of Phoenix running on the PlanetLab testbed, and validate its ability to survive emulated catastrophes.

The rest of this paper is organized as follows. Section II discusses related work. Section III describes our system model for representing correlated failures. Section IV describes our measurement study of the software diversity of hosts in a large network, and Section V describes and evaluates heuristics for computing cores. Section VI describes the design and implementation of the Phoenix Recovery Service, and Section VII describes the evaluation of Phoenix on PlanetLab. Finally, Section VIII concludes.

II. Related work

Informed replication bridges fault-tolerant systems with the prevention of network epidemics. The study of fault-tolerant systems, distributed or not, is a major area of research. Several sophisticated tools exist for evaluating the reliability and availability of a system given failure probabilities and covariances (for example, see [61]). The SIFT project [75] was the first to separate such analysis from system design in the context of distributed systems. It assumed that no more than a certain number t of components could be simultaneously faulty. A value for t can then be computed off-line using the tools mentioned above. Considerable work has been done in the analysis of various problems in distributed computing as a function of the failure model, the environmental assumptions, and the upper bound t.

The SIFT project was the formal design and verification of a fly-by-wire system. This was an embedded system that was designed so that failures were independent. Most distributed systems, though, 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 [43]. 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 [35].

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 principally address each of these properties.

Prevention techniques prevent pathogens from exploiting vulnerabilities, thereby reducing the size of the vulnerable host population and limiting the extent of a worm outbreak. Such techniques include static and dynamic testing to eliminate vulnerabilities [16], [72], patching vulnerabilities before they are exploited using virus detection software [68] and software update mechanisms [46], and *shields* that filter traffic exploiting known vulnerabilities in applications [73]. 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 remove software vulnerabilities after they have been exploited, thereby reducing the rate of infection as hosts are treated. Recent proposals include anti-worms that disinfect hosts using the same propagation methods as the original worm [15], [18], and vaccinating hosts by automatically detecting infection in software and generating and applying patches online [62]. Such techniques are reactive in nature, and some hosts still become infected. Further, counter-worms have questionable legality, and automatic vaccination has limiting constraints on deployment (e.g., requiring source to patch).

Containment techniques block infectious communication between infected and uninfected hosts, thereby reducing or potentially halting the contact rate of a spreading pathogen.

Such techniques include reducing connection rates [41], [76] and network filtering [50], [70]. The efficacy of reactive containment fundamentally depends upon the ability to quickly detect the onset of a new pathogen [39], [51], [56], [66], [78], characterize the pathogen to create filters specific to infectious traffic [29], [36], [37], [63], and deploy such filters in the network [42], [71]. Unfortunately, containment at Internet scales is challenging, requiring short reaction times and extensive deployment [50], [77]. Again, since containment is inherently reactive, some hosts will 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 [13] 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 among hosts.

Lastly, Phoenix is just one of many proposed cooperative systems for providing archival and backup services. For example, Intermemory [14] and Oceanstore [38] enable stored data to persistent indefinitely on servers distributed across the Internet. As with Phoenix, Oceanstore proposes mechanisms to cope with correlated failures [74]. The approach, however, is reactive and does not enable recovery after Internet catastrophes. With Pastiche [17], pStore [2], and CIBS [40], 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 [30]. In contrast to Phoenix, Glacier copes with catastrophic failure via massive replication.

III. System Model

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.

A. Representing correlated failures

Consider a system made up of a set $\mathcal{H}$ of hosts each of which is capable of holding certain files. These hosts can fail (for example, by crashing), and to keep these files 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 file.

However, using this value of t 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. This implies that one might wish to set $t = 2$. However, if we place replicas on h_1 and h_3, the file's availability may be acceptably high with just two replicas, and we need not make $t + 1$ or 3 replicas as this strategy suggests.

To better address issues of optimal replication in the face of correlated failures, we defined an abstraction that we call a *core* [35]. 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 (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 [49], [65]. 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 III-B below, we discuss the granularity at which we can represent different software packages. 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 in C that is not susceptible to the pathogen. Most Internet pathogens target a single (possibly cross-platform) vulnerability, and the ones that target multiple vulnerabilities target the same operating system. Hence, for now, we assume that any attribute is susceptible to attack and that a core is a minimal set C of processes such that no attribute is common to all hosts in C. In Section V-D, we relax this assumption and we show how to extend our results to tolerate pathogens that can attack 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 into operating system, Web server, and Web browser.

Example 3.1:

Attributes: Operating System = {*Unix, Windows*};
Web Server = {*Apache, IIS*};
Web Browser = {*IE, Netscape*}.

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 I

RECENT WELL-KNOWN PATHOGENS AND THEIR FORMS OF INFECTION.

Hosts: $H_1 = \{$*Unix, Apache, Netscape*$\}$;
$H_2 = \{$*Windows, IIS, IE*$\}$;
$H_3 = \{$*Windows, IIS, Netscape*$\}$;
$H_4 = \{$*Windows, Apache, IE*$\}$.
Cores $= \{\{H_1, H_2\}, \{H_1, H_3, H_4\}\}$.

H_1 and H_2 comprise what we call an *orthogonal core*, which is a core composed of hosts that have different values for every attribute. 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.

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 configurations 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 version of Unix as the operating system. Consequently, H_1 is present in both cores.

To make our argument more concrete, consider the worms summarized in Table I, which are well-known worms unleashed in the past 3 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 would have survived the attack. Given even a very modest amount of heterogeneity, our method of constructing cores would include such pairs of hosts.

B. 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 of a single attribute for the generic class of operating system, e.g., "*Windows*", "*Unix*", "*MacOS*", 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 since H_3 and H_4 both run Windows. More fine-grained representations would 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 in the attributes. Such fine-grained attributes provide considerable flexibility in placing replicas – e.g., we could place a replica on an NT host and an XP host to protect against worms such as CodeRed 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.

Our initial work [34] 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 no compelling need to do so.

IV. HOST DIVERSITY

Two issues with informed replication are (1) the difficulty of identifying cores and (2) the resulting storage load. Both of these issues are influenced by 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 what 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.

A. Methodology

On our behalf, UCSD Network Operations used the *nmap* tool [32] 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.

- 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.
- Ignoring hosts that nmap could not consistently fingerprint could bias the host traces that were used.
- DHCP-assigned host addresses are reused. This implies the following: 1) 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; 2) A host may appear multiple times with different addresses either in the port trace or in the operating system trace. As a consequence, 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. 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 such university with large settings connected to the Internet around the globe, and so the results we draw from our data is undoubtedly not singular.

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 II

TOP 10 OPERATING SYSTEMS (A) AND PORTS (B) AMONG THE 2,963 GENERAL-PURPOSE HOSTS.

B. Attributes

Table II shows the ten most prevalent operating systems and open ports identified on the general purpose hosts (ignoring all other device types), from a total of 2,569 attributes representing operating systems and open ports. Columns one and two show the number and percentage of hosts running the named operating systems. As expected, Windows is the most prevalent OS (54% of general purpose hosts). Separately Unix variants vary in prevalence (0.03–10%), but collectively they comprise a substantial fraction of the hosts (38%).

Columns three and four show 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 (a firewall) 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 ssh (31%) and sunrpc (25%). Web servers on port 80 are roughly as prevalent as ftp (18%).

These results show that the software diversity is significantly skewed. Most hosts have open ports that are shared by many other hosts (Table II 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.

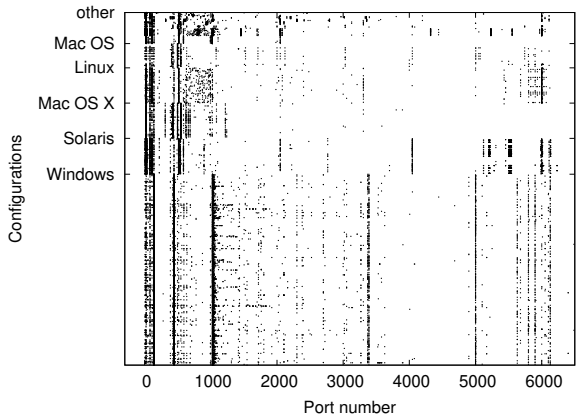

Fig. 1. Visualization of configurations from UCSD traces.

C. 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 II: Windows hosts start at the bottom, then Solaris, MacOS X, Linux, 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 insights and made it more difficult to see 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 with 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 V.

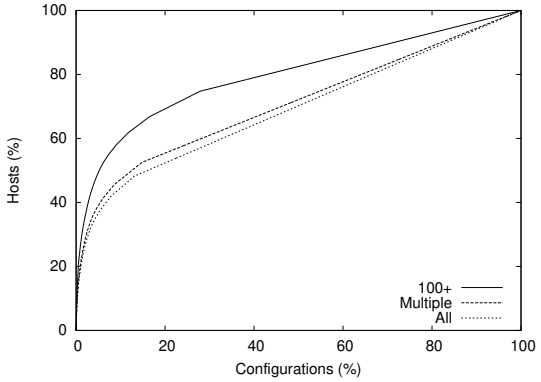

Fig. 2. Configuration distribution.

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. If every host had a unique configuration, then the curves would be straight diagonal lines. 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, 50% of hosts comprise 15% of the configurations. And considering only the top 100 port attributes, 50% of hosts comprise 8% of the configurations. 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 construct cores that result in a low storage load.

V. Surviving catastrophes

With informed replication, each host h constructs a core $\mathit{Core}(h)$ 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 [33]. Hence, we use heuristics to compute $\mathit{Core}(h)$. 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 IV.

A. Representing advertised configurations

Our heuristics are different versions of greedy algorithms: a host h repeatedly selects other hosts to include in $\mathit{Core}(h)$ 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 $\mathit{Core}(h)$. This representation is a three-level hierarchy.

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

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.

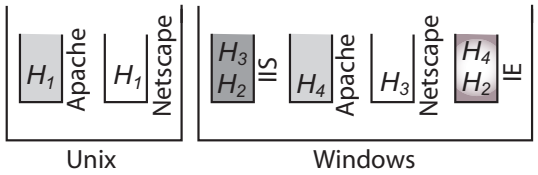

Fig. 3. Illustration of containers and sub-containers.

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 $\mathcal{S}(c)$ the sub-containers associated with a container c. The function $m_s : \mathcal{C} \times \mathcal{A} \rightarrow \mathcal{S}$ maps a container and application to a sub-container; thus, for each $a \in h.app$, host h is in each sub-container $m_s(m_c(h.os), a)$.

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 $m_s(c, a)$ containing h.

B. 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_s, m_c and $S(c)$, but they do not reference the full set $\mathcal{A}$ of attributes. In addition, a host h does not need to enumerate $\mathcal{H}$, but it does need to be able to reference the configuration of any host h' (which it will find by enumerating sub-containers). Thus, the container/sub-container hierarchy is the only data structure that the heuristics use to compute cores.

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 $h' \in \mathcal{H}$. If one heuristic results in a higher maximum load than a second heuristic, then the first heuristic is less fair in its distribution of work than the second one.
- **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 the attributes of h . 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 V-C, 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.

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 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 five of the ten attributes. There are more ways to fail all of the hosts in *Core*(h_2) than those in *Core*(h_1). Consequently, we also use the number of cores that do not have a coverage of 1 as an extension of the coverage metric.

2) Heuristics: We begin by evaluating 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 III shows the performance of **Random** using a single run of our simulator. We set the size of the cores to five, i.e., **Random** chose five random hosts to form a core. The coverage of 0.977 may seem high, but there are still many cores that have attributes not covered. The load is 12, which is significantly higher than the lower bound of 5.[2] And choosing a core size smaller than five results in even lower coverage.

Our first greedy heuristic **Uniform** ("uniform" selection

[2]To meet this bound, number the hosts in $\mathcal{H}$ from 0 to $|\mathcal{H}|-1$. Let *Core*(h) be the hosts $\{h, h \oplus 1, h \oplus 2, h \oplus 3, h \oplus 4\}$ where $\oplus$ is addition modulo $|\mathcal{H}|$.

	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 III

A TYPICAL RUN OF THE HEURISTICS.

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 a both a container $c \in \mathcal{C} \setminus \{m_c(h.os)\}$ and a sub-container $sc \in \mathcal{S}(c) \setminus \{m_s(c,a)\}$ 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 within hosts within the same operating system class. Referring back to Figure 1, the set of prevalent services among hosts running the same operating system varies across the different operating systems. If, for some reason, 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, since these values seem to provide 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 III 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 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 $|m_s(c,a)|$ and $N_s(c,A)$ be the size of the union of $m_s(c,a)$ 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 $m_s(c,a')$ with probability $N_s(c,a')/N_s(c,\mathcal{A} \setminus \{a\})$.

In the third and fourth rows of Table III, 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:

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

Thus, the value of L cannot be smaller than l. Using Table II, 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, 5, 6, and 7 present the results our of 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. (We also plot 95% confidence intervals for each point, although they are too narrow to be seen). 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

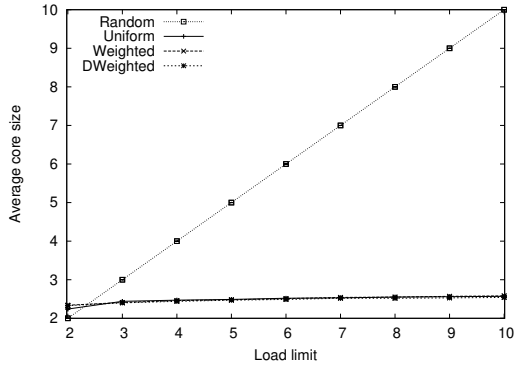

Fig. 4. Average core size.

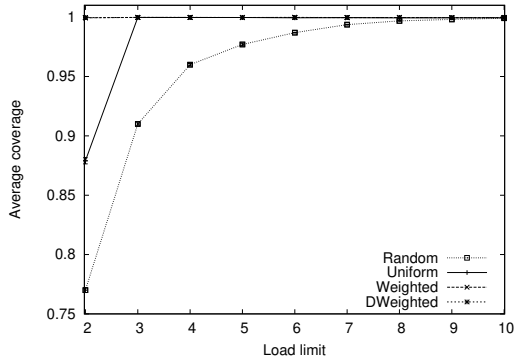

Fig. 5. Average coverage.

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, there is not much difference in terms of core size among **Uniform**, **Weighted**, and **DWeighted**. 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** has better coverage with increasing L because the size of the cores increases. Note that, to reach the same value of coverage obtained by **Uniform**, **Weighted**, and **DWeighted**, **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 $L > 3$. 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

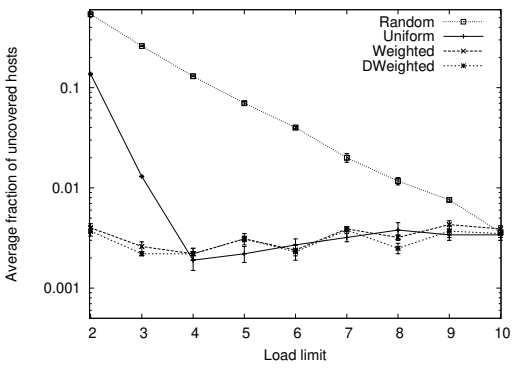

Fig. 6. Average fraction of uncovered hosts.

are fewer hosts available with these same attributes. 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 attribute not covered.

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 in actual indicates how fair 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 large 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 a small load limit greater than three is the best choice for our application. However, should load variance be an issue, one can use one of the other heuristics.

C. 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

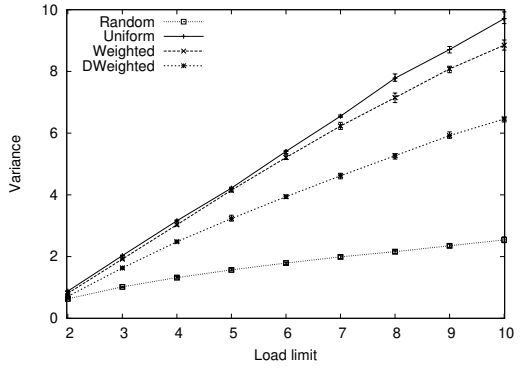

Fig. 7. Average load variance.

do so based on information about past worms to support our choices and assumptions.

Worms such as the ones in Table I used services that have vulnerabilities as vector 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. And, from Figure 6, the majority of the cores have a coverage of 1. We also observed in the previous section that, for cores that do not have a coverage of 1, usually it is only a single attribute that is not covered.

Under the assumptions we have made in this paper, 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 attribute not covered, 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 against informed replication 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.

D. Tolerating exploits of vulnerabilities on multiple attributes

To tolerate exploits on multiple attributes, we need to build cores such that, for subsets of attributes possessed by members of a core, there must be a number of core members that do 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 in such a system is a subset composed of hosts h_1, h_2, h_3 as follows:

- h_1 = {*Linux*, *Apache*};
- h_2 = {*Windows*, *IIS*};
- h_3 = {*Solaris*, *Zeus*};

In this core, every pair of attributes is such that at least one host contains none of them.

To build a k–resilient core for a host h, we use the following heuristic:

1) Search for $k-1$ hosts, h_1 through h_k, that have an operating system different than h;
2) Use **Uniform** to search for a 1–resilient core C for h
3) For each $i \in [1 \ldots k]$, use **Uniform** to search for a 1–resilient core C_i for h_i;
4) Merge all C_i into a single set C;
5) Output $C \cup C_i \cup \ldots \cup C_k$ as a k–resilient core for h.

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.

From Appendix A, we have that searching for a k–resilient core is at least as hard as searching for a 1–resilient core and we also show that searching for a 1–resilient core is an NP-complete problem. Thus, there is no polynomial-time optimal solution for such a problem unless P = NP. Our greedy heuristic, however, searches for cores in polynomial time. More specifically, it runs in $O(k \cdot \Lambda)$, where Λ is the size of the largest configuration. This upper bound on time complexity assumes constant-time access to containers. If access to containers is not done in constant time, and instead it takes at most $c \cdot f(\mathcal{C})$, where $c > 0$ is a constant and $f(\cdot, \cdot)$ is a polynomial time function of a set of containers $\mathcal{C}$, then the heuristic runs in $O(k \cdot f(\mathcal{C}) \cdot \Lambda)$.

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 IV

SUMMARY OF SIMULATION RESULTS FOR $k = 2$ FOR 8 DIFFERENT RUNS.

In Table IV, 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 values of $L \geq 5$ based on the argument presented in Appendix C. In the second and third columns, we present our measurements for coverage with 95% confidence limits 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 V-B. Finally, the last column shows average core size.

The coverage results show that 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, the average 1–coverage metric is one with a null 95% confidence interval, indicating that all clients have all their attributes covered.

Having a system that more broadly protects its hosts requires more resources: core sizes are much larger to obtain sufficiently high degrees of coverage. Compared to the results in Section V-B, 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 available for use.

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 II, 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.

VI. THE PHOENIX RECOVERY SERVICE

A cooperative recovery service is an attractive architecture for tolerating Internet catastrophes. It is an attractive system for individual Internet users, like home broadband users, who do not wish to pay for commercial backup service or deal with the inconvenience of making manual backups. 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.

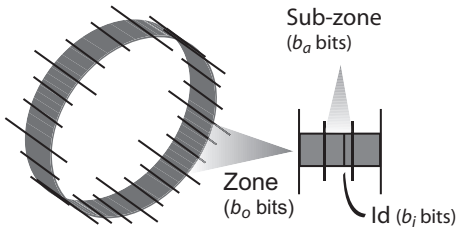

Fig. 8. Phoenix ring.

A. 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 V, we use the container mechanism for advertising configurations. In our prototype, we implement containers using the Pastry [60] 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 an identifier for itself, 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. Figure 8 illustrates the three parts of a host identifier. 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. Therefore, it 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, or the "most significant" part of all the host's identifiers, all identifiers of a host lie within the same zone.

To build a core for itself according to heuristic **Uniform**, a host h selects hosts at random from sub-containers, also selected at random. Selecting a container corresponds to choosing a number c randomly from $[0, 2^{b_o} - 1]$. Similarly, to select a sub-container and a host within this sub-container, we choose a random number sc within $[0, 2^{b_a} - 1]$ and another random number id within $[0, 2^{b_i} - 1]$, respectively. A host

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 lookup with another randomly chosen sub-container.

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 periodic 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. Consequently, we assume that hosts send such announcements once a day, although the parameter is configurable and can be changed according to system demands.

A host may permanently leave the Phoenix 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. Therefore, 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. Although the load limit the heuristic imposes guarantees that the amount of storage required of h is constrained, such hosts may still receive a disproportionate number of requests.

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 VII.

B. Service design

The Phoenix software a host runs is composed of two mechanisms, an *agent* and a *server*. The Phoenix agent is responsible for interacting with a user application on top of it and with a Pastry agent underneath. Figure 9 is a state machine description of the behavior of the Phoenix agent. The agent begins in the `Init` state, and changes to `Joining` when the user application requests it to join the Phoenix ring. In the `Joining` state, it creates a session for each Phoenix address of the host. Each of these sessions has its own routing table and leaf set, and thus participates in the DHT as an independent Pastry agent. After joining all the sessions, a Phoenix agent change its state to `Uncovered`. At this point, the agent requires input from the user application. If the application specifies that the host needs a core to backup data, the agent undergoes transition "3", changing to state to `Covering`. If, on the other hand, the application specifies that the host has lost data, it generates a request that causes the agent to use transition "7", changing its state to `Waiting`.

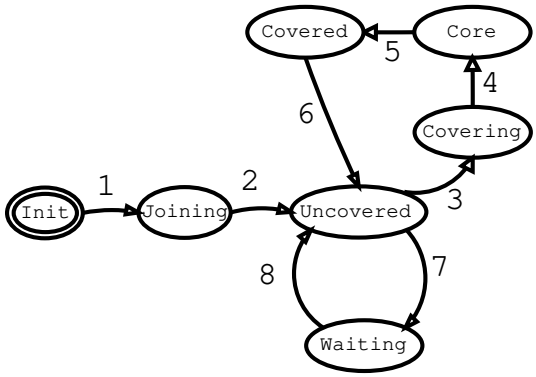

Fig. 9. State machine for Phoenix agent.

In state `Covering`, the agent uses heuristic **Uniform** to select a core. Note that containers and sub-containers in the original specification of **Uniform** map to zones and sub-zones, respectively. After selecting a core, the agent notifies the application and changes its state to `Core`. The application then has to decide if the core satisfies its expectations, or if the agent should try again. If it decides to accept the core, then the agent sends `Data` messages containing the data to be backed up to the Phoenix servers of the core components. When the data backup completes on all all the hosts in the core, the host transitions to state `Covered`.

If, while in state `Uncovered`, the Phoenix agent undergoes a transition to `Waiting`, then it waits until it receives an announcement from a host in its core. Hosts holding data on behalf of other hosts send these messages periodically so that hosts learn of the members of their core in the event of a catastrophe. Upon reception of an announcement, a host in state `Waiting` sends a request to the core member that made the announcement to restore its data. The Phoenix server of the core member receives the request and replies with the content requested.

The two main responsibilities of the Phoenix server of a host are managing storage, and sending announcements and responding to requests to restore data. When a Phoenix server of a host h participates in a core, it commits to store the data of the requester. If it receives data from the requester, then h stores this data, and starts sending announcements to this host. The requester, however, may decide not to include h in its core. In this case, the requester may ignore the reply or send a release message. If the requester sends a release message, then h removes the data it it already have it. Otherwise, the

acceptance eventually times out, and h again rejects data from the requester. Release messages also serve the purpose of releasing data stored on core members. This happens in the case that a user decides to select another, core or if a core partially fails.

Since recovery is not time-critical, the period between consecutive announcements sent to the same host can be relatively large, from hours to a day. For this reason, we assume that hosts send such announcements once a day, although the parameter is configurable and can be changed according to the demands on the system. These messages are acknowledged by the receiver. Note that not getting a reply to an announcement does not necessarily mean that the host left the system ungracefully, since the particular host that did not reply might have failed. At the same time, it is necessary to garbage collect backups of hosts that are not part of the system anymore. For this reason, we assume that if a host h does not receive a reply to announcement messages it sent to h' within a large period of time, say a week, then it garbage collects the data it holds on behalf of h'. A week is sufficient time for users to notice that they lost their data and request a restore.

We now turn our attention to the protocol used by Phoenix to communicate among servers. Phoenix implements this protocol using the following message types:

- `request`: requests participation in a core;
- `reply`: in response to a request to participate in a core, a host h replies indicating whether it agrees to participate or not. This decision depends on the number of other hosts already being serviced by this host. If h decides to accept, then it sends its own configuration along with the reply message;
- `release:` if a host h' decides not to use h as a core member for its data, it sends this message to h so that h is notified that it is not a core member for h'.
- `announce`: a host h periodically sends this message to host h' if h is in h's core and stores a copy of h's data;
- `data`: a host h sends this message containing its data to be backed up to a host h' if h' has agreed to participate in the core constructed by h;
- `request_restore`: after a catastrophe, a host sends this message as soon as it discovers a member of its core storing its data, *i.e.*, as soon as it receives an `announce` message;
- `restore`: once a host receives a `restore_request`, it replies with the data it stored on behalf of the requesting host.

We implemented Phoenix using the Macedon [59] 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.

C. 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 [30]. 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 [33]. In what follows, we detail the security measures that the client and server must carry out to achieve the above goals. Here, we only provide a high level description of the design; the full description and analysis appear in Appendix D.

The system uses symmetric encryption and signatures (cf. [45]) to guarantee the privacy and integrity of the stored data. In the basic design, a client uses a secret key to encrypt and sign the data, which is then sent to the server. In addition, requests from client hosts are signed to prevent impersonation by third parties. In particular, the client uses the user's passphrase to derive a pair of public and signing keys (pk_c, sk_c) and a symmetric encryption key k. The public key pk_c is sent to each server in the client's core. To prevent attacks that overwrite data or send unauthorized release messages, the client deletes all key material after data storage operations. Any subsequent operation that requires the client's secret key must request the passphrase from the user and re-generate the keys. After a catastrophe, rogue third parties may return old copies of the data. In order to preclude this attack, if multiple announcement messages are received by the client, the client must reply to each and request its data. The client decrypts the data returned and keeps the latest copy of the data received.

The security of the design assumes some authenticated mechanism is in place in the underlying network or, alternatively, that the network is such that "man in the middle"

attacks are infeasible to mount.[3] Privacy and integrity of the data is guaranteed using encryption and signatures as long as no secret key is compromised. Since all secret material is erased after each sensitive operation, the key is safe as long as the user does not supply the passphrase to the application during the time the host is infected. Regarding availability, if the server host contacted by the client host is honest, the server eventually sends an announcement message to the client host and the client recovers an authentic copy of the stored data.

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. Although the basic design (Appendix D.2) does not protect against dishonest servers that purportedly delete data, the system does guarantee that no other disruptive behavior may happen. Dishonest servers, for example, cannot tamper nor obtain information from other host's saved data, nor can dishonest clients replace or modify some other client's data. Nonetheless, intentional erasure of data may still be a concern in some environments. One technique to identify such misbehavior is to issue *signed receipts*. A signed receipt is a message signed by the server that binds a specific backup operation with the identities of both the client and server involved (see Appendix D.3 for details). Clients can use such receipts to claim that servers are misbehaving. Thus, servers that purposely delete other host's data can be eventually identified and removed from the system.

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.[4]

Similarly, host may attempt to overload the servers in the system. 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.

We can further enhance the security of phoenix when extra resources are available (see Appendix D.3). For instance, if users have smartcards or other protected devices with (possibly limited) computing power, it is possible to "split" the secret keys between the smartcard and the client host using *key insulation* techniques [22]. This method guarantees the availability of some recent copy of the host's data even in the presence of pathogens that quietly corrupt the client host and attempt to erase the host's data by performing store and delete operations without the user's consent. We stress that this last technique is orthogonal to the security enhancements mentioned before (eg. that of signed receipts), and therefore either one can be implemented independently if needed.

[3]Entity authentication can be strictly enforced if some distributed authority is implemented.

[4]Indeed, detection can be guaranteed if the configurations are digitally signed.

VII. Phoenix Evaluation

In this Section, we evaluate our Phoenix prototype on the PlanetLab testbed using the metrics discussed in Section V. We also simulate a catastrophic event – the simultaneous failure of all Windows hosts – to experiment with Phoenix's ability to recover from large failures.

A. 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 IV, we selected 63 configurations at random from our set of 2963 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 its operating system. Thus, in our setting roughly 60% of the hosts represent Windows hosts. This configuration implies that the load limit L must be at least three, otherwise some hosts would not be fully covered even assuming an optimal distribution.

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 V for coverage, core size, and load variance for one run of our prototype. For comparison, we also present results from our simulations with the same set of configurations used for the PlanetLab experiment for coverage, core size, and load variance.

Coverage is perfect in all cases, demonstrating that we were able to protect all the hosts using these values of L. And fully covering hosts did not require extensive replication. From the average core size, the majority of cores had two replicas. Note that core sizes account for the clients as well. Thus, if a core has size two, it contains the client and some other host. In addition, the average core size showed no significant variation as we increased the value of L.

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 simulations, we set a large upper bound on the number of request messages ($\texttt{diff\_OS} + \texttt{same\_OS} = 100$) 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

Load limit (L)	Core size		Coverage		Load var.	
	Imp.	Sim.	Imp.	Sim.	Imp.	Sim.
3	2.12	2.22	1	1	1.65	1.94
5	2.10	2.23	1	1	2.88	2.72
7	2.10	2.22	1	1	4.44	3.33

TABLE V

RESULTS FROM PLANETLAB EXPERIMENT. SIMULATION RESULTS ARE ALSO PRESENTED FOR COMPARISON. "IMP." STANDS FOR IMPLEMENTATION, AND "SIM." STANDS FOR SIMULATION.

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.

B. 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. For the worms listed in Table I, for example, there were Windows hosts not vulnerable at the time these worms were unleashed because they were patched or not running a particular service required by the worm to infect.

The catastrophe 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 VI-A). 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 unecessarily. For the backup file size, we chose an arbitrary value since we are not concerned about tranfer 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.

Size (GB)	1 hour	1 day	1 week
Aggregate bandwidth			
0.1	1.20 Gb/s	50.1 Mb/s	7.16 Mb/s
1	12 Gb/s	0.5 Gb/s	71.6 Mb/s
10	120 Gb/s	5 Gb/s	716 Mb/s
100	1.2 Tb/s	50 Gb/s	7.2 Gb/s
Per-host bandwidth ($L = 3$)			
0.1	0.6 Mb/s	27.8 Kb/s	4.0 Kb/s
1	6.7 Mb/s	278 Kb/s	39.7 Kb/s
10	66.7 Mb/s	2.8 Mb/s	397 Kb/s
100	667 Mb/s	28 Mb/s	3.97 Mb/s

TABLE VI

BANDWIDTH CONSUMPTION AFTER A CATASTROPHE.

C. Recovering from a catastrophe

Finally, we 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 IV, 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 VI 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 recovery 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 Internet during recovery. 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 requirements 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 will only have to recover 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 during a catastrophe.

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 [17], 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 DVD 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).

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 1–10GB 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 [11]. 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 through organization backup services can often take a day for an administrator to respond to a backup request.

VIII. Conclusions

In this paper, we proposed a new approach for designing distributed systems to survive Internet epidemics that cause catastrophic damage. In contrast to previous approaches for defending against Internet epidemics, our approach uses informed replication and a model of correlated failures to survive them. 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 objects survive attacks of single- and double-exploit pathogen), result in low replication factors (less than 3 replicas for single-exploit pathogens; less than 5 replicas for double-exploit pathogens), limit the storage burden on each host in the system, and lend themselves to a fully distributed implementation. We then demonstrated the use of this approach in the design and evaluation of a cooperative backup system called the Phoenix Recovery Service. Based upon our results, we conclude that our approach is a viable and attractive method for surviving Internet catastrophes.

Acknowledgements

We would like to thank 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.

References

[1] E. G. Barrantes et al. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. of CCS*, 2003.

[2] C. Batten, K. Barr, A. Saraf, and S. Treptin. pStore: A secure peer-to-peer backup system. Unpublished report, Dec. 2001.

[3] M. Bellare, A. Boldyreva, and S. Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 259–274. Springer-Verlag, 2000.

[4] M. Bellare, R. Canetti, and H. Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *37th Annual Symposium on Foundations of Computer Science (FOCS '96)*, pages 514–523. IEEE, Oct. 1996.

[5] M. Bellare, R. Canetti, and H. Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing, STOC'98*, pages 419–428. ACM Press, 1998.

[6] M. Bellare, A. Desai, E. Jokipii, and R. Rogaway. A concrete security treatment of symmetric encryption. In *38th Annual Symposium on Foundations of Computer Science*, pages 394–403. IEEE, 1997.

[7] M. Bellare and S. Goldwasser. The complexity of decision versus search. *SIAM Journal on Computing*, 23(1), Feb 1994.

[8] M. Bellare and P. Rogaway. The exact security of digital signatures—how to sign with RSA and Rabin. In *Advances in Cryptology—EUROCRYPT 96*, volume 1070 of *LNCS*, pages 399–416. Springer-Verlag, 1996.

[9] M. Bellare and A. Sahai. Non-malleable encryption: Equivalence between two notions, and an indistinguishability-based characterization. In *Advances in cryptology — CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 519–536. Springer-Verlag, 1999.

[10] M. Bellare, H. Shi, and C. Zhang. Foundations of group signatures: The case of dynamic groups. In *CT-RSA'05, The Cryptographers' Track at RSA Conference*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

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

[12] R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *Advances in Cryptology – EUROCRYPT ' 01*, volume 2045 of *Lecture Notes in Computer Science*, pages 453–472. Springer-Verlag, 2001.

[13] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[14] Y. Chen. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, 1999.

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

[16] 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.

[17] L. P. Cox and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of Fifth USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[18] Crclean. http://www.winnetmag.com/Article/ArticleID/22381/22381.html.

[19] W. Dai. Crypto++ library. http://www.eskimo.com/∼weidai/cryptlib.html.

[20] A. Desai, A. Hevia, and Y. L. Yin. A practice-oriented treatment of pseudorandom number generators. In *Advances in Cryptology—EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 368–383. Springer-Verlag, 2002.

[21] H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A strengthened version of RIPEMD. In *Fast Software Encryption FSE'96*, volume 1039 of *LNCS*, pages 71–82. Springer-Verlag, 1996.

[22] Y. Dodis, J. Katz, S. Xu, and M. Yung. Key-insulated public key cryptosystems. In *Advances in Cryptology – EUROCRYPT' 2002*, volume 2332 of *LNCS*, pages 65–82. Springer-Verlag, 2002.

[23] A. Fujioka, T. Okamoto, and S. Miyaguchi. ESIGN: An efficient digital signature implementation for smart cards. In *Advances in Cryptology (EUROCRYPT '91)*, volume 547 of *LNCS*, pages 446–457. Springer, Apr. 1991.

[24] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[25] O. Goldreich. *Foundations of Cryptography*, volume Basic Tools. Cambridge University Press, 2001.

[26] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[27] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[28] P. Gutmann. Cryptlib security toolkit. http://www.cs.auckland.ac.nz/∼pgut001/cryptlib/index.html.

[29] H. A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proc. of the 13th Usenix Security Symposium*, 2004.

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

[31] IEEE. P1363: Standard specifications for public key cryptography, 2004.

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

[33] F. Junqueira, R. Bhagwan, A. Hevia, K. Marzullo, and G. M. Voelker. Coping with internet catastrophes. Technical Report CS2005–815, UCSD, Feb 2005.

[34] 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*, May 2003.

[35] F. Junqueira and K. Marzullo. Synchronous Consensus for dependent process failures. In *Proceedings of the ICDCS 2003*, pages 274–283, May 2003.

[36] 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.

[37] C. Kreibich and J. Crowcroft. Honeycomb – Creating Intrusion Detection Signatures Using Honeypots. In *Proceedings of the USENIX/ACM Workshop on Hot Topics in Networking*, Cambridge, MA, Nov. 2003.

[38] 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 *Proceedings of ACM ASPLOS*, 2000.

[39] 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 WIA*, June 2003.

[40] 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.

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

[42] 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.

[43] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 569–578. ACM Press, 1997.

[44] NTT Multimedia Communications Laboratory. ESIGN-EMSA5 source code, 2001. http://www.nttmcl.com/sec/Esign/esign.html.

[45] A. J. Menezes, P. C. van Oorschot, and S. A. Vanston, editors. *Handbook of Applied Cryptography*. CRC Press, 1996.

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

[47] 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.

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

[49] 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 ACM/USENIX Internet Measurement Workshop (IMW)*, Marseille, France, Nov. 2002.

[50] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Internet quarantine: Requirements for containing self-propagating code. In *Proceedings of the IEEE Infocom Conference*, San Francisco, California, Apr. 2003.

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

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

[53] National Institute of Standards and T. Technology. DES model of operation. FIPS PUB 81, 1980.

[54] National Institute of Standards and T. Technology. Advanced Encryption Standard (AES). FIPS PUB 197, 2001.

[55] National Institute of Standards and Technology. Secure hash standard (sha1). FIPS PUB 180-1, 1995. Supersedes FIPS PUB 180 1993 May 11.

[56] D. Plonka. FlowScan - Network Traffic Flow Visualization and Reporting Tool.

[57] C. Rackoff and D. R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Advances in Cryptology – CRYPTO '91*, volume 576 of *LNCS*, pages 433–444. Springer-Verlag, 1991.

[58] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. of the ACM*, 21(2):120, Feb. 1978.

[59] 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*, Mar 2004.

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

[61] R. A. Sahner, K. S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*. Kluwer Academic, 1996.

[62] S. Sidiroglou and A. D. Keromytis. A network worm vaccine architecture. In *Proc. of IEEE Workshop on Enterprise Security*, 2003.

[63] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. Poster at SOSP 2003.

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

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

[66] S. Staniford. Containment of Scanning Worms in Enterprise Networks. *to appear in the Journal of Computer Security*, 2004.

[67] J. Stern, D. Pointcheval, M.-L. Malone-Lee, and N. P. Smart. Flaws in applying proof methodologies to signature schemes. *Lecture Notes in Computer Science*, 2442:93–110, 2002.

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

[69] O. Team. OpenSSL project. http://www.openssl.org/.

[70] T. Toth and C. Kruegel. Connection-history Based Anomaly Detection. In *Proceedings of the IEEE Workshop on Information Assurance and Security*, West Point, NY, June 2002.

[71] J. Twycross and M. M. Williamson. Implementing and Testing a Virus Throttle. In *Proceedings of the 12th USENIX Security Symposium*, Washington, D.C., Aug. 2003.

[72] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, Feb. 2000.

[73] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the ACM SIGCOMM Conference*, Portland, Oregon, Aug. 2004.

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

[75] J. Wensley et al. Sift design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, Oct. 1978.
[76] M. Williamson. Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code. Technical Report HPL-2002-172, HP Laboratories Bristol, June 2002.
[77] C. Wong et al. Dynamic quarantine of internet worms. In *Proceedings of DSN*, 2004.
[78] C. C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and early warning for internet worms. In *Proceedings of the CCS*, Oct. 2003.

APPENDIX

A. The complexity of finding cores

In this section, we discuss the complexity of searching for cores in a set of hosts. Informally, a core is a subset of hosts that is diverse enough for a given task. What "enough" means depends on the application. In the case of the Phoenix recovery system, the members of a core must have sufficiently different software configurations so that not all members share a non-empty set of exploitable vulnerabilities.

We show that finding such cores is an intractable problem. First, we provide a few definitions to formalize our ideas. We define a system as follows:

Definition 1.1: A system is a triple $\langle \mathcal{H}, \mathcal{A}, \alpha \rangle$, where $\mathcal{H}$ is a finite set of hosts, $\mathcal{A}$ is a finite set of attributes, and a mapping $\alpha : h \in \mathcal{H} \rightarrow A \subseteq \mathcal{A}$.

We define a k–resilient core as follows:

Definition 1.2: Given a system $\langle \mathcal{H}, \mathcal{A}, \alpha \rangle$, a set of hosts $C \subseteq \mathcal{H}$ is a k–resilient core, $k > 0 \wedge k \in \mathbb{N}$, if and only if there is no $A \subseteq \mathcal{A}$, $|A| \leq k$, such that for every $h \in C$, $\alpha(h) \cap A \neq \emptyset$, where $\alpha : h \in \mathcal{H} \rightarrow A \subseteq \mathcal{A}$. Such a subset must be also minimal: $\forall h_c \in C, \exists a \in \alpha(h) : \forall h'_c \in C \setminus \{h_c\}, a \notin \alpha(h'_c)$.

Although not necessary, we define for convenience what it means when we say that a subset is a k–resilient core for some host h in particular.

Definition 1.3: A k–resilient core C is a k–resilient core for $h \in \mathcal{H}$ if and only if $h \in C$.

This definition is convenient because hosts use cores to accomplish tasks. Thus, it is useful to refer to the requester of the service.

We now move on to define two important problems for the purposes of this paper. The Set-Cover problem is a well-studied NP-Complete problem [24]. We repeat its definition here for the sake of clarity:

Problem: SC decision problem
Instance: Collection C of subsets of a finite set S, positive integer $k \leq |C|$;
Question: Does C contain a cover for S of size at most k?

Now the problem we are interested in:

Problem: k-Core decision problem
Instance: A system $\langle \mathcal{H}, \mathcal{A}, \alpha \rangle$, a host $h \in \mathcal{H}$, a positive integer $s > 0$;
Question: Is there a k–resilient core $C \subseteq \mathcal{H}$ for h of size at most s?

We show with the following two claims that the 1-Core decision problem is NP-Complete. By doing so, we later argue in this section that there cannot be a polynomial-time algorithm that outputs a minimal 1–resilient core, unless P= NP . In other words, the correspondent search problem cannot be easier to solve than the decision problem. In the next section, we discuss in more details the problem of searching for a minimal k–resilient core for a value of k greater than one. Intuitively, such a problem is at least as hard as searching for a minimal 1–resilient core. As such, if there is no polynomial-time algorithm that outputs a minimal 1–resilient core given a system $\langle \mathcal{H}, \mathcal{A}, \alpha \rangle$, there cannot be a polynomial-time algorithm that outputs a minimal k–resilient core given a system $\langle \mathcal{H}, \mathcal{A}, \alpha \rangle$, for values of k greater than one.

Claim 1.4: SC $\leq_m$ 1-Core

Proof: We need to provide a polynomial-time algorithm that, given an instance $\langle S, C, k \rangle$ of the SC problem, returns an instance $\langle \langle \mathcal{H}, \mathcal{A}, \alpha \rangle, h, s \rangle$ of the Core Problem, such that the following holds:

i) If $\langle S, C, k \rangle \in$ SC, then $\langle \langle \mathcal{H}, \mathcal{A}, \alpha \rangle, h, s \rangle \in$ 1-Core;
ii) If $\langle \langle \mathcal{H}, \mathcal{A}, \alpha \rangle, h, s \rangle \in$ 1-Core, then $\langle S, C, k \rangle \in$ SC.

We describe such an algorithm as follows:

Algorithm **SCtoC**: $\langle S, C, k \rangle$

1) $\mathcal{A} \leftarrow \emptyset$; $\mathcal{H} \leftarrow \{h\}$;
2) $A \leftarrow \emptyset$;
3) For every element x of S,
   $\mathcal{A} \leftarrow \mathcal{A} \cup \{a_x, \widehat{a_x}\}$;
   $A \leftarrow A \cup \{a_x\}$;
4) $\alpha \leftarrow [\alpha : h \leftarrow A]$;
5) For every element c of C,
   $\mathcal{H} \leftarrow \mathcal{H} \cup \{h_c\}$;
   $A \leftarrow \emptyset$;
   $\forall a_x \in A$, if $(x \in c)$ then $A \leftarrow A \cup \{\widehat{a_x}\}$;
   else $A \leftarrow A \cup \{a_x\}$;
   $\alpha \leftarrow [\alpha : h_c \leftarrow A]$;
6) $s \leftarrow k$;

Every step of the algorithm runs in polynomial time. Consequently, time complexity is given by the sum of the complexities of the individual steps. This is clearly polynomial.

It remains to show that Properties i) and ii) hold for **SCtoC**. First, we show i. For an instance $\langle S, C, k \rangle$ of the SC problem, suppose there is a subset C' of C such that $|C'| \leq k$ and C' is a cover for S. We construct a 1–resilient core Ω for the instance of the Core Problem returned by our algorithm as follows:

1) $\forall c \in C' : \Omega \leftarrow \Omega \cup \{h_c\}$;
2) $\Omega \leftarrow \Omega \cup \{h\}$.

By construction, for every attribute $a \in \alpha(h)$, there is in Ω at least one host h_c such that $a \in \alpha(h_c)$. According to the description of **SCtoC**, a host h_c only covers an attribute a_x of h if $x \in c$. Because C' is a cover for S, Ω must be a 1–resilient core for h. Moreover, Ω must have size at most $s = k$.

Now we show ii). Given an instance $\langle \langle \mathcal{H}, \mathcal{A}, \alpha \rangle, h, s \rangle$ of the 1-Core problem, suppose there is a 1–resilient core Ω for h of

size at most s. From the definition of a core, we have that a host h_c is in Ω only if it covers at least one attribute of h. By the construction of **SCtoC**, if a host h_c covers an attribute a_x of h, then $x \in c$. Thus, we can construct a cover C' for S by including in C' all the sets $c \in C$ such that $h_c \in \Omega$. Again by construction, C' must cover S, and $|C'| \leq k$. This completes our proof.

■

Now we show that 1-Core is in NP .

Claim 1.5: 1-*Core* $\in$ *NP* .

Proof: We need to provide a polynomial-time verifier for 1-Core. The verifier takes as input an instance $\langle\langle \mathcal{H}, \mathcal{A}, \alpha\rangle, h, s\rangle$ of the 1-Core problem and a certificate C. This certificate consists of a subset of $\mathcal{H}$. Thus, the verifier has to check whether the subset provided as a certificate is an 1–resilient core of size at most s for the instance provided. We now describe such a verifier as follows:

Verifier **V**: $\langle\langle \mathcal{H}, \mathcal{A}, \alpha\rangle, h, s\rangle$, C

1) Parse C into a subset Ω of processes;
2) Check if $\Omega \subseteq \mathcal{H}$;
3) Check if $|\Omega| \leq s$;
4) Check if $h \in \Omega$;
5) For every attribute $a \in \mathcal{A}$, check if there are at least one host in Ω that does not contain a;
6) If any of these checks fail, then reject, otherwise accept.

Each step of the verifier executes in polynomial time on the size of the input. Thus, the total execution time has to be polynomial on the size of the input. This concludes the proof of our claim. ■

With these these two claims, we have shown that there is no polynomial-time algorithm for 1-Core if P $\neq$ NP . From [7], we have that search reduces to decision for NP-Complete problems. The search problem for k-Core is as follows:

Problem: k-Core search problem
Instance: A system $\langle \mathcal{H}, \mathcal{A}, \alpha\rangle$ a host $h \in \mathcal{H}$, a positive integer $s > 0$;
Search for k-Core: Find a subset $H \subseteq \mathcal{H}$ such that H is a k–resilient core, $h \in H$, and $|H| \leq s$, or output $\perp$.

Thus, we conclude that there is a polynomial-time algorithm for the 1-Core search problem if and only if there is a polynomial-time algorithm that solves the 1-Core decision problem. Furthermore, the following optimization problem clearly reduces to the k-Core search problem:

Definition 1.6:

Problem*: k-MinCore*
Input *: A system* $\langle \mathcal{H}, \mathcal{A}, \alpha\rangle$ *a host* $h \in \mathcal{H}$*;*
Output *: a subset* $H \subseteq \mathcal{H}$ *such that* H *is a* k*–resilient core and* $h \in H$*;*
Cost function*:* $f(H) = |H|$*;*
Goal *: Minimize.*

Given an oracle $\text{OS}_{k\text{-Core}}$ that solves the k-Core search problem in polynomial time on the size of the input, we can solve the optimization problem by calling the oracle with increasing values of s, until the oracle outputs a k–resilient core. Note that we need to call the oracle at most $|\mathcal{H}|$ times in the worst case. Thus, running such an algorithm still takes polynomial time on the size of the input. Note also that this description is valid for any $k > 0$, and consequently it is valid for the case $k = 1$, which is the one we discussed above.

B. Searching for k–resilient cores

In the previous section, we concentrated on the 1-Core problem, although we stated all the definitions in terms of k. The problem of searching for a 1–resilient core consists more descriptively in determining a subset H of hosts such that the intersection of the set of attributes across all hosts of H is empty. The generalization for $k > 1$ is as follows:

$\exists\ H_1, \ldots, H_k \subseteq \mathcal{H}$:
$\wedge \exists i \in [1 \ldots k] : H_i$ is a 1–resilient core
$\wedge \forall i, j \in [1 \ldots k], i \neq j : (\cap_{h \in (H_i \cup H_j)} h) = \emptyset$

Claim 1.7: k-Core $\leq_m$ $(k+1)$-Core $k \geq 1$

Proof: We have to show that there is a polynomial-time algorithm **KtoK+1** such that given an instance of the K-Core problem, it outputs an instance of the $(k+1)$-Core. Such an instance must be such that:

i) If $\langle\langle \mathcal{H}, \mathcal{A}, \alpha\rangle, h, s\rangle \in k$-Core, then $\langle\langle \mathcal{H}', \mathcal{A}', \alpha'\rangle, h', s'\rangle \in (k+1)$-Core;
ii) If $\langle\langle \mathcal{H}', \mathcal{A}', \alpha'\rangle, h', s'\rangle \in (k+1)$-Core, then $\langle\langle \mathcal{H}, \mathcal{A}, \alpha\rangle, h, s\rangle \in k$-Core.

We now describe **KtoK+1**:

Algorithm **KtoK+1**:$\langle\langle \mathcal{H}, \mathcal{A}, \alpha\rangle, h, s\rangle$

1) $\mathcal{H}' \leftarrow \mathcal{H} \cup \{h^*\}$;
2) $\mathcal{A}' \leftarrow \mathcal{A} \cup \{a^*\}$;
3) $\alpha \leftarrow [\alpha : h^* \rightarrow \{a^*\}]$;
4) $h' \leftarrow h$;
5) $s' \leftarrow s + 1$;
6) output $\langle\langle \mathcal{H}, \mathcal{A}, \alpha\rangle, h', s'\rangle$;

The algorithm clearly runs in polynomial time, since every step is executed in polynomial time on the size of the input. It remains to show i) and ii). First, we show i. Let $C \subseteq \mathcal{H}$ be a k–resilient core of size at most s. We then have that $C' = C \cup \{h^*\}$ is a $(k+1)$–resilient core. By assumption, C is a k–resilient core. Consequently, there is no subset A of k or less attributes such that for all $h \in C$, $\alpha(h) \cap A$ is not empty. The host we add to C to form C' has a single attribute that is not shared by any other host. There are two cases to analyze. First, let A' be a subset of $k+1$ attributes that does not include a^*. Such a subset of attributes cannot intersect every host in C' because it does not intersect at least h^*. Second, let A'' be a subset of $k+1$ attributes such that A'' includes a^*. Such subset cannot intersect the configuration of every host in C' either. Otherwise, there is a subset of k attributes in A'' that intersects the configuration of every host in C, thereby contradicting our assumption that C is a k–resilient core. We thus have that there is no subset of $k+1$ or less attributes such that for all $h \in C$, $\alpha(h) \cap A$ is not empty, and C' has size at most $s' = s + 1$.

We now show ii. Let C' be a $(k+1)$–resilient core of size at most s' for the instance of $(k+1)$-Core output by **KtoK+1**. If C' does not contain h^*, then there is a host h' in C' such that $C' \setminus \{h'\}$ is a k–resilient core. This must be true, otherwise there is a subset of at $k+1$ that intersects the configurations of all the hosts in C'. Now, if C' does contain h^*, then $C' \setminus \{h^*\}$ is a k–resilient core of size at most s. To see why $C' \setminus \{h^*\}$ must be a k–resilient core observe that if it is not, then there exists a set A' of k attributes of $\mathcal{A}$ such that they intersect the configurations of all the hosts in $C' \setminus \{h^*\}$. In this case, C' cannot be a $(k+1)$–resilient core either because $A' \cup \{a^*\}$ intersects all the configurations of C'. This concludes our proof.

■

Using a simple recursive argument, we have that 1-Core reduces to k-Core for any $k > 1$. We therefore have that k-Core cannot be solved in polynomial time, unless 1-Core has a polynomial-time solution.

C. Lower bound on the value of L for 2-resilient cores

In Section V-B, we compute a lower bound for the value of L when constructing cores that are 1-resilient. In this Section, we also compute a lower bound on the value of L, but now we want to construct 2-resilient cores.

Compare to the 1–resilient core case, the value of the lower bound is computed differently because all pairs of attributes in a core must be covered. That is, for every attribute a of the client, its core must contain at least two other hosts with different configurations to be 2-resilient. Thus, a core must have at least three members to be 2-resilient. To compute a lower bound, we assume that every host has a 2–resilient core of minimal size three. Let x be the fraction of hosts that have the most prevalent attribute. Out of the remaining $(1-x)\cdot|\mathcal{H}|$ hosts, suppose a fraction $y \cdot (1-x)$ has the most prevalent attribute for this group of hosts. As a simple application of the pigeonhole principle, some host must be in at least the following number of cores:

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

Using the data from our diversity study, we have the following. The most prevalent attribute is port 139, which is present in 1,640 of the hosts. The hosts that have port 139 as an attribute are mostly hosts running the Windows operating system: out of the 1,640 hosts, 1,491 run Windows. The first most prevalent attribute that is mostly present in hosts not running Windows is port 22. This attribute is present in 910 hosts, out of which 901 run operating systems different than Windows. The difference between the total number of hosts and the number of hosts with port 139 as an attribute is 1,323. Out of these 1,323 hosts, 780 hosts have port 22 as an attribute, but not port 139, which is approximately 59% of the remaining hosts. Plugging these values into the equation for l ($x = 1,640/2,963 = 0.553$ and $y = 780/1,323 = 0.590$.), we have that $l = 4$.

In the experiments of Section V-D, the minimum value of L we considered is five, because the efficiency measured with coverage is poor for lower values. The reason for not meeting the lower bound relies both in the nature of the configurations we used, and the way we compute cores. First, many configurations overlap in their attributes, thus requiring more than four hosts to form a 2–resilient core. To meet the lower bound, most of the cores must have size at most three. Second, the heuristic computes two 1–resilient cores independently and merge into one 2–resilient core. Consequently, the 2–resilient cores we compute have size at least four.

D. Phoenix Security

In this section, we elaborate on the techniques we suggested to secure the data in the Phoenix system in Section VI-C. First, we generalize the problem and define a class of (abstract) protocols, the archival-recovery protocols, which allow hosts to save data on behalf of other hosts – precisely the goal of the Phoenix system. We explain the syntax and meaning of the messages exchanged by the hosts when one of them initiates a backup process. Then, we present two protocols: the basic protocol and the enhanced protocol. Both of them are concrete instantiations of an (abstract) archival-recovery protocol. Our first protocol, the basic protocol, is easily implemented using available open-source software libraries and provides some basic security guarantees. This protocol is explained and analyzed in Section D.2. The enhanced protocol requires more resources (namely, user smartcards and support for public keys), but provides stronger security guarantees. The protocol and its analysis is presented in Section D.3.

1) Functional View of the Protocol: In this section, we revisit the concept of an archival-recovery service. Our goal is to identify the components whose implementation and analysis are critical for security. First, we define explicitly what it means for a protocol to provide an archival-recovery system. Then, we precisely define when an archival-recovery system is secure.

An archival-recovery service is a system in which hosts store (save) data on behalf of other hosts. Moreover, hosts can continuously submit pieces of data, ask for the deletion (release) of such pieces and, when needed, ask for a copy of the stored data. A more concrete description follows. In an archival-recovery system, hosts can act both as clients and as servers. A host acts as client when it generates the data (which needs to be saved by some other host); it acts as a server when it saves data on behalf of some other host. Each host has an identifier id. For simplicity, we assume that each client may need to backup, at most, a single piece of data at a time – if more pieces of data are to be backup, each new piece overwrites the previous piece.[5]

In what follows, we formally describe an abstract archival-recovery protocol in terms of functions both the client and the server must execute to generate and process the exchanged

[5] Generalizations to multiple pieces are possible by requiring the client to issue an explicit `release` message for a given piece before such piece is replaced.

messages. In particular, this protocol depends on four client functions (with prefix $client\_$) and four server functions (with prefix $server\_$). Any protocol that implement an archival-recovery service must implement these functions.

Formally, we say a protocol implements an **Archival-Recovery system** if it follows the sequence of messages and actions detailed below. Every archival-recovery protocol is divided in three main phases: Backup, Release, and Recovery phase.

- **Backup Phase**: This phase is triggered each time a new piece of data D is available in the client. The goal of this phase is to store a copy of D on the server.

  The parties proceed as follows:. first, the client computes a message M as the output of the function $client\_gen\_data(D)$. We call this message M a "`data` message". This message is then sent to the server. The message includes a data identifier did that uniquely identifies D. Upon reception of this message, the server executes function $server\_save\_data(M)$ which saves M locally. Additionally, once the server holds data on behalf of other hosts, it must advertise it. At least once, the server must generate an "`announce` message". This message is used by the server to announce that it currently stores a piece of data with identifier did on behalf of client cid. The actual message is computed as the output of the function $server\_gen\_announce(did, cid)$. The message, which includes a data identifier did and a client identifier cid, is then sent to all clients.
- **Release Phase**: This optional phase is triggered each time a client decides it no longer needs the data associated to identifier did. The goal of this phase is to erase the copy of the data with such identifier stored on the server.

  The parties proceed as follows:. First the client computes a message M as the output of the function $client\_gen\_release(did)$. We call this message M a "`release` message". The message is then sent from the client to the server. The message includes a data identifier did. Upon reception of this message, the server executes function $server\_erase\_data(M)$ which, if applicable, erases the data with identifier did locally.
- **Recovery Phase**: This phase is triggered after the client has suffered a catastrophe. The goal of this phase is to recover a copy of the data stored on the server.

  The parties proceed as follows: whenever client cid receives an `announce` message (containing client identifier cid and data identifier did), the client computes a message M as the output of the function $client\_request\_restore(did)$. We call this message M a "`request_restore` message". The message, which includes data identifier did, is then sent to the sender of the `announce` message. Upon reception of this message, the server executes function $server\_retrieve\_data(M)$ which first extracts did from M and then retrieves the data with identifier did from the data stored locally. The output of function $server\_retrieve\_data(M)$ is another message M'. We call this message M' a "`restore` message". Message M' is then sent from the server to the client. Upon reception of this message, the client executes the function $client\_restore\_data(M)$ which outputs the client's data D or $\perp$ if message M is not valid.

In Section D.2 and Section D.3, we show how to use some cryptographic tools to provide two concrete instantiations for archival-recovery protocols. In next section, we present a definition of security of any archival-recover protocol.

WHAT SECURITY MEANS: THE MODEL. In what follows, we say a host is "honest" if it strictly follows all protocol instructions, including deleting information.

There are three security goals for the system.

1) Data privacy: "Any host other than the client host should not obtain any partial information from the data stored on the server host".
2) Data integrity: "Any tampering of the backup data should be detectable by the client host".
3) Data availability: "If a backup is made by the client host on an honest server host before the catastrophe, the client host eventually is able to recover a copy of the same backup from that server host".[6]

In order to formally prove the above properties, we define a security model using the concept of experiments, adversaries and "oracles" (initiated in Goldwasser and Micali's work [26] but refined extensively later [4], [6], [9], [3], [10]). Defining an experiment or "game" for the system aims to formally capture the type of interactions (actions and messages) an adversary may exploit when attacking the system.[7] The resources available to the adversary are represented by *oracles*, interactive programs that reply to queries from the adversary. Oracles simulate the type of actions (and messages) adversaries can trigger (or obtain) in the system. In particular, in our setting, oracles play the role of honest clients and servers. The adversary can freely interact with the oracles, that is, we make no assumption on how messages are delivered in the system, other than messages are eventually delivered. Thus, messages can be delivered in any order or at any time as long as they eventually arrive.The output of the experiment under a given adversary is a single bit; the bit is set to 1 if and only if any of the security goals of the system are broken. The idea is to prove that no practical adversary can cause the experiment to return 1 with high probability, no matter what the adversary does.

As usual, adversaries in the experiment are modeled by arbitrary polynomial-time computable programs with oracle access. The oracles provided to the adversary are *Client-*$\mathcal{O}_b(\cdot)$ and *Server-*$\mathcal{O}(\cdot)$. The former simulates the execution of honest clients while the latter simulates the execution of honest

[6] Notice that, if the backup is made on a *dishonest* server, no claims are made. In Section D.3, we show how to achieve a stronger property where such dishonest servers are always detected using "signed receipts".

[7] In practice, we consider "worst case" adversaries, Internet pathogens that may attempt to target the logic of the system, that is, possible flaws in the protocol itself.

servers. They capture the adversary's ability to interact with many clients and servers in parallel. Both oracles are stateful and simulate the actions of multiple hosts (clients and servers) following the protocol, except for a few modifications we now explain. First, the oracles allow the adversary to schedule the actions taken by client and servers (by sending queries for specific actions at specific times). This means the adversary can control, for example, when and how many times each client submits data for backup or send out "`release`" messages, when a catastrophe occurs for a given client, or when "`announce`" and "`request_restore`" messages are sent. Although, in practice, the adversary may not have such control, this choice only strengthen our results, since we show that no practical adversary, not even one with that power, can break the security guarantees of the system. Secondly, we allow the adversary to restrict the data that each (simulated) client submits to the server for archival. Specifically, the simulated client must select one of two strings $data_0$ and $data_1$, both chosen by the adversary. The selection only depends on a bit b fixed at the onset of the experiment, the same for all the simulated clients. This aspect of the model follows the standard notion of indistinguishability under *chosen-plaintext attacks* [6], and allows us to define privacy: any adversary guessing bit b can be seen as somehow extracting information from the data. Our design does not tolerate even such (apparently innocuous) action, so we see such adversary as violating the privacy of the system. Finally, the adversary's ability to eavesdrop the interaction between pairs of honest clients and servers is modeled by allowing the adversary to start a "virtual" interaction between *Client-*$\mathcal{O}_b$ and *Server-*$\mathcal{O}$. The interaction is virtual in the sense adversary only gets a copy of the exchanged messages between a given client (simulated by *Client-*$\mathcal{O}_b$) and a given server (simulated by *Server-*$\mathcal{O}$). However, the adversary still can affect the interaction by controlling the schedule of messages and by impersonating third clients and servers.

More concretely, oracle *Client-*$\mathcal{O}_b(\cdot)$ works as follows. All the information for the simulated clients are maintained in a table Q. Concretely, for any client cid, $Q[cid]$ contains all local variables used by client cid. Additionally, table Q is used by the oracle to keep track of some information per simulated client: variable $Q[cid].ActiveBackup[sid, did]$ is set to *Yes* if client cid has submitted data with identifier did for backup to server sid; variable $Q[cid].AllData$ maintains the set of all pieces of data that a client cid has submitted for backup; and variable $Q[cid].failed$ is set to *Yes* if client cid has failed. All queries accepted by the oracle take two parameters: a client identifier cid and a server identifier sid. The adversary's identifier is a fixed but arbitrary string different from any client or server identifier: without lost of generality, we assume that it is the sender identifier specified in the first query made to any oracle. The sid identifier must always be the arbitrary identifier *unless* the adversary has started a virtual interaction for such client; in this case, sid can be set to any of the servers for which the client has a virtual interaction.

The types of queries accepted by oracle *Client-*$\mathcal{O}_b(\cdot)$ are the following:

1) `start`(cid), starts the simulation of client with identifier cid. It sets up any local variables if required by the protocol.
2) `start_data`(cid, sid, D_0, D_1), instructs simulated client cid to initiate a `data` message with D_b as the intended data to backup. The oracle executes function $client\_gen\_data(D_b)$ on behalf of simulated client cid and obtains an output M. This output includes a data identifier did. Also, the oracle adds D_b in set $Q[cid].AllData$, and sets $Q[cid].ActiveBackup[sid, did] \leftarrow Yes$. The query returns as output M.
3) `start_release`(cid, sid, did), instructs simulated client cid to initiate a `release` message with data identifier did. The oracle first sets $Q[cid].ActiveBackup[sid, did] \leftarrow No$. Then, the oracle returns the output of function $client\_gen\_release(did)$ (as it were executed by client cid) as the reply to the query.
4) `fail`(cid), instructs simulated client cid to fail, as if a catastrophe has occurred; it sets $Q[cid].failed \leftarrow Yes$.
5) `announce`(cid, sid, did), instructs simulated client cid to act as if an `announce` message specifying data identifier did was received. The query returns nothing.
6) `start_request_restore`(cid, sid, did), instructs simulated client cid to to initiate a `request_restore` message with data identifier did. The query returns the output of function $client\_request\_restore(did)$ as it were executed by client cid.
7) `start_restore`(cid, sid, M'), instructs simulated client cid to act as if a `restore` message M' was received. In particular, the oracle computes $d \leftarrow client\_restore\_data(M')$. The query returns "Good backup" if $d \neq \perp$ (in which case it sets $Q[cid].ActiveBackup[sid, did] \leftarrow No$) and "Bad backup" otherwise.

Oracle *Server-*$\mathcal{O}(\cdot)$ works as follows. All the information for each simulated server (its local variables and tables) are maintained in a table Q' indexed by server identifier. As with the client oracle *Client-*$\mathcal{O}_b$, all queries accepted by this oracle take two parameters: a client identifier cid and a server identifier sid. The cid identifier must be set to the identity of the adversary (an arbitrary but fixed string) unless the adversary has started a virtual interaction for such server; in this case, cid can be set to any of the clients for which the server has a virtual interaction.

The types of queries accepted by oracle *Server-*$\mathcal{O}(\cdot)$ are:

1) `start`(sid), starts the simulation of server with identifier sid by initializing any local variables for sid.
2) `data`(sid, cid, M), instructs simulated server sid to act as if `data` message M was received. In particular, server sid executes function $server\_save\_data(M)$. This query returns nothing.
3) `release`(sid, cid, M), instructs simulated server

sid to act as if `release` message M was received. In particular, server sid executes function $server\_erase\_data(M)$. This query returns nothing.

4) `start_announce`(sid, cid), instructs simulated server sid to initiate an `announce` message for any data held from client cid. The query returns the output of function $server\_gen\_announce(M)$ as it were executed by server sid.
5) `request_restore`(sid, cid, M), instructs simulated server sid to act as if a `request_restore` message M was received. The query returns the output of function $server\_retrieve\_data(M)$ as it were executed by server sid.

In order to make the model meaningful, we impose some restriction for the queries. A query to oracle *Client-*$\mathcal{O}_b$ (respectively, *Server-*$\mathcal{O}$) is ignored if: (a) the query specifies a server (respectively, client) identifier which is already simulated by the opposite oracle;[8] (b) the query is not `start` and specifies a client (respectively, server) identifier not yet defined; and (c) it is a `start_data`$(cid, \cdot)$ query with a data identifier did smaller than that of any previous `start_data`$(cid, \cdot)$ query, for the same client.[9] Also, the adversary is required to send a query of type `start_announce`$(sid, cid, \cdot)$ at least once after submitting a query `data`$(sid, cid, \cdot)$, for any sid, cid. Otherwise, the adversary is considered invalid. This restriction models that all servers interacting with the adversary are honest and, consequently, they must send announcements if they hold backups.

Additionally, we add a new set of queries to initiate the "virtual" interaction between *Client-*$\mathcal{O}_b$ and *Server-*$\mathcal{O}$ oracles. (Recall that virtual interactions aim to capture the adversary's ability to eavesdrop and affect the interaction between an honest client and an honest server.) We define the special query `start`(cid, sid) as the query that has the same effect than `start`(cid) and `start`(sid) when submitted to both oracles at the same time. Once a query `start`(cid^*, sid^*) is made for some client cid^* and server sid^*, we say that a virtual interaction between virtual client cid and virtual server sid^* has begun. Thereafter, the effect of any of the previously defined queries for oracles *Client-*$\mathcal{O}_b$ and *Server-*$\mathcal{O}$ remains unchanged *except* in the case the same pair cid^*, sid^* ever appears again in a query. In such a case, the answer to this query is not only returned to the adversary, but it is also sent to the *other* oracle, in the form of a message to the virtual client or sender indicated in the query (namely sid^* if the query was originally for oracle *Client-*$\mathcal{O}_b$ or cid^* if it was for *Server-*$\mathcal{O}$). Even though the adversary receives the replies of such queries immediately, the adversary is allowed to schedule the delivery of the virtual replies to the virtual clients or server. Furthermore, the adversary must deliver all such messages before it finishes execution. Intuitively, in a virtual interaction, the two oracles exchange messages almost undisturbedly (except for the fact that the adversary may delay the delivery of some of the messages), only sending copies of such messages to the adversary. We remark that the adversary still has control on which messages are sent (if they are not message replies), the order and delivery times of them, and the choice of the identities of the communicating parties.

Now we describe the experiment. For any adversary A and archival-recovery system Γ we define the *Archival-Recovery* experiment:

$\mathsf{Exp}_{\Gamma}^{\mathtt{Arch-Rec}}(A)$:

1) Select a random bit b.
2) A starts executing. A is allowed to make multiple queries to *Client-*$\mathcal{O}_b(\cdot)$, *Server-*$\mathcal{O}(\cdot)$ until it stops and outputs a bit d.
3) Experiment outputs 1 (that is, adversary "wins") if any of the following holds:
   - (*Data privacy condition*) Adversary guess bit b correctly, that is, $d = b$.
   - (*Data integrity condition*) There exists client cid and data D such that
     a) A made a `start_restore`(cid, sid, M') query such that *Client-*$\mathcal{O}_b$ internally recovered data equal to D and returned "Good Backup", but
     b) $D \notin Q[cid].AllData$, that is, no adversary's query was ever of the form `start_data`$(cid, \_, D_0, D_1)$ where $D_b = D$.
   - (*Data availability condition*) There exists values cid, sid, did, that satisfy the following conditions:
     a) A made a `start`(cid, sid) query,
     b) A made a `start_announce`(cid, sid) query, and
     c) After all `restore`$(cid, sid, \_)$ messages have been delivered, it holds that $Q[cid].failed = Yes$ and $Q[cid].ActiveBackup[sid, did] = Yes$.

In the above experiment, if adversary A wins the experiment because $c = 1$, we say A wins trivially.

The advantage of adversary A, denoted by $\mathsf{Adv}_{\Gamma}^{\mathtt{Arch-Rec}}(A)$, is defined as follows

$$\mathsf{Adv}_{\Gamma}^{\mathtt{Arch-Rec}}(A) \stackrel{\text{def}}{=} 2 \cdot \Pr\left[\mathsf{Exp}_{\Gamma}^{\mathtt{Arch-Rec}}(A) = 1\right] - 1$$

Notice this is a value between 0 and 1.

Let $\ell > 0$ be an integer, the security parameter. We define the *advantage function* for the archival-recovery system Γ, denoted $\mathsf{Adv}_{\Gamma}^{\mathtt{Arch-Rec}}(\ell)$ as the maximum of function $\mathsf{Adv}_{\Gamma}^{\mathtt{Arch-Rec}}(A)$ over all possible adversaries A with time-complexity polynomial in the security parameter ℓ. We say archival-recovery system Γ is *secure* if $\mathsf{Adv}_{\Gamma}^{\mathtt{Arch-Rec}}(k)$ is small for all reasonable values of ℓ.

Next section presents one of the concrete instance of archival-recovery system that we propose for the Phoenix system.

2) Basic Protocol: The protocol uses only software tools. We make the following assumptions about the client host. The user controlling a client host knows a (long enough) passphrase which the user enters to the client host. The setup, backup and recovery operations will require this passphrase.

[8] This captures the fact that the channel is authenticated.

[9] This captures that the identifiers must be strictly increasing values.

The client host also has access to a reliable local clock – not necessarily synchronized with any other clock – which we assume external entities cannot modify. There are no extra assumptions about the host when acting as server. With respect to the network, we only assume message delivery is reliable (all sent messages are eventually received) and the communication between each pair of host is authenticated (each message has an unforgeable sender identifier).

The cryptographic tools used in this solution are standard. We use secure symmetric encryption (cf. [6]), secure digital signature schemes (cf. [27], [58], [8], [23], [44]). pseudo-random bit generators [25], [20], and cryptographic hash functions [55], [21] (see also [45], [31] for background and standards). Open-source implementations of these primitives can be found in [19], [28], [69]. In what follows, $\mathcal{SE} = (\mathcal{E}, \mathcal{D})$ denotes the symmetric encryption scheme used (e.g. AES in counter mode [53], [54]) and $\mathcal{SS} = (\mathcal{K}, Sig, Vf)$ the signature scheme (e.g. RSA [58] or ESIGN [23] with PSS [8]). When describing the protocol we use the following notation. We write $C \leftarrow \mathcal{E}(k, M)$ to denote that C is the encryption of message M under key k, and $M' \leftarrow \mathcal{D}(k, C)$ means M' is the decryption of ciphertext C under key k. For the signature scheme, $(pk, sk) \leftarrow \mathcal{K}(r)$ denotes the process where the key pair pk, sk is generated by the key-generation algorithm $\mathcal{K}$ using random bits r; $\sigma \leftarrow Sig(sk, M)$ means σ is the signature of M under signing key sk and $b \leftarrow Vf(pk, M, \sigma)$ denotes the process of verifying if σ is a valid signature of message M under verification key pk: if $b = 1$ then the signature is valid, $b = 0$ otherwise. We also write $k_1 \circ k_2 \leftarrow G(k)$ to denote that the concatenation of k_1 and k_2 is the result of running the pseudorandom bit generator G on input k. Finally, we write $h \leftarrow H(M)$ when h is the output of the cryptographic hash function on input M.

THE BASIC PROTOCOL: The input for each client is a `pass` – provided interactively by the user – and one or more piece of data D to backup.[10] Each server host maintains a local counter `numClients` with the number of client it is currently serving, and a table T indexed by client identifier that stores data and identifiers for all served clients. Each server stores at most one backup (one copy of data) for each client.

In this implementation, the data identifiers are timestamps generated from the client's local clock. Without lost of generality, we see these values are positive and strictly increasing integers. Additionally, the client employs local keys which must be generated from its passphrase. In order to generate such keys, the client uses function *setup*, which takes a single argument, a passphrase `pass`, and deterministically computes the keys by $(r, k) \leftarrow G(H(\texttt{pass}))$, $(pk, sk) \leftarrow \mathcal{K}(r)$. The output is (pk, sk, k), where pk, sk are the signing key and verification key and k is the encrypting key.[11]

We now explain how to implement the $client\_*(\cdot)$ and $server\_*(\cdot)$ functions for each phase. We call this protocol simply `Phoenix`.

Backup Phase:

- $client\_gen\_data(D)$:
  1) Recompute keys (pk, sk, k) from its passphrase `pass` by running $setup(\texttt{pass})$. Let cid be the identifier of the client running this function.
  2) Compute $C \leftarrow \mathcal{E}(k, D)$ as the encryption of D under key k. Let $R \leftarrow (C \circ \texttt{ts} \circ pk \circ cid)$ where `ts` is the value in the local time (timestamp), and $\circ$ denotes string concatenation.
  3) Compute the signature of R as $s \leftarrow Sig(sk, R)$ and set $M \leftarrow R \circ s$.
  4) Erase keys (pk, sk, k).
  5) Output M.
- $server\_save\_data(M)$:
  1) Parse M as $R \circ s$ and then R as $C \circ \texttt{ts} \circ pk \circ cid$.
  2) If cid is a new client, check that the number `numClients` of currently served clients is less than L. If so, increase the counter `numClients` by one. Otherwise (if $\texttt{numClients} = L$), abort.
  3) If cid is not a new client (that is, there exists an entry $M' = (C' \circ \texttt{ts}' \circ pk' \circ cid \circ s')$ associated to the same cid in T), check if the signing keys match ($pk = pk'$), and that the timestamp `ts` is more recent than $\texttt{ts}'$. If any of the checks fail, abort.
  4) Check that the request was properly signed by the client, namely that $Vf(pk, R, s) = 1$. If so, store $T[cid] \leftarrow M$. If not, abort.
- $server\_gen\_announce(did, cid)$:
  1) If it holds data (with identifier did) from client cid in local table T, compute $M \leftarrow$ "`announce`" $\circ\, cid \circ did$ and output M. Otherwise, abort.

Release Phase:

- $client\_gen\_release(did)$:
  1) Recompute keys (pk, sk, k) from its passphrase `pass` by running $setup(\texttt{pass})$. Let cid be the identifier of the client running this function.
  2) Compute a signed release request by $R \leftarrow$ "`release`" $\circ\, did \circ pk \circ cid$ and $s \leftarrow Sig(sk, R)$.
  3) Set $M \leftarrow R \circ s$.
  4) Erase keys (pk, sk, k).
  5) Output M.
- $server\_erase\_data(M)$:
  1) Parse M as $R \circ s$ and then R as "`release`" $\circ\, \texttt{ts} \circ pk \circ cid$.
  2) Verify that both the signing key pk and timestamp `ts` match the ones stored in local table T under index cid.
  3) Check the request was properly signed, namely that $Vf(pk, R, \sigma) = 1$.
  4) If all the checks pass, erase entry $T[cid]$. Otherwise, if any of the checks fail, abort.

Recovery Phase:

[10] In fact, we assume new pieces of data are generate dynamically and continually by the client.

[11] Formally, our analysis treats H as a random oracle but this assumption can be removed by using more sophisticated randomness extraction techniques.

- After a catastrophe, the client sets its local variable $latest\_ts$ to $(-\infty)$.
- $client\_request\_restore(did)$:
  1) Recompute keys (pk, sk, k) from the passphrase using function $setup$. Let cid be the identifier of the client running this function.
  2) Let $R \leftarrow$ "`request_restore`" $\circ$ did $\circ$ pk $\circ$ cid. Compute the signature of R as $s \leftarrow Sig(sk, R)$.
  3) Set $M \leftarrow R \circ s$.
  4) Erase keys (pk, sk, k).
  5) Output M.
- $server\_retrieve\_data(M)$:
  1) Parse M as $R \circ s$ and then R as "`request_restore`" $\circ$ `ts` $\circ$ pk $\circ$ cid.
  2) If it holds no data for client cid in local table T, abort. Otherwise, verify that both verification key pk and identifier `ts` match the ones stored in table T under index cid.
  3) Checks that the request was properly signed, namely that $Vf(pk, R, \sigma) = 1$. If any of the checks fail, abort.
  4) Retrieve entry $M' \leftarrow T[cid]$. Output M'.
- $client\_restore\_data(M')$:
  1) Recompute keys (pk, sk, k) from the passphrase using function $setup$. Let cid be the identifier of the client running this function.
  2) Parse M as $R' \circ s'$ and then R' as $C' \circ$ `ts`$' \circ pk' \circ cid' \circ s'$.
  3) Check that `ts`$'$ is more recent than $latest\_ts$. If so, set $latest\_ts \leftarrow$ `ts`.
  4) Check the authenticity of the received data R' by verifying that pk equals pk' and that the signature of R' is valid, that is $Vf(pk, R', s') = 1$.
  5) If all checks pass, compute $D \leftarrow \mathcal{D}(k, C')$. Otherwise, compute $D \leftarrow \perp$.
  6) Erase keys (pk, sk, k).
  7) Output D.

Next section analyzes the security of this protocol.

SECURITY ANALYSIS: Next result shows that data privacy and data integrity are preserved. Indeed, privacy is guaranteed by the usage of encryption, and integrity is guaranteed by the use of signatures, as long as the key is not compromised. In particular, malicious hosts cannot tamper with saved (backup) data.

Claim 1.8: Consider the `Phoenix` *archival-recovery protocol. If the symmetric encryption scheme $\mathcal{SE}$ is secure (in the sense of [57]) and the signature scheme scheme $\mathcal{SS}$ is secure (in the sense of [27]) then* data privacy *and* data integrity *hold with overwhelming probability.*

Proof: The proof is by reduction to the security of the encryption scheme and the security of the signature scheme. Given an adversary A that makes experiment $\mathsf{Exp}^{\mathtt{Arch-Rec}}_{\mathtt{Pheonix}}(A)$ output 1 with noticeable probability by violating either the privacy or the integrity condition, we can build either an adversary B_1 that breaks the security of the underlying encryption scheme (in the sense of privacy under chosen-plaintext attacks [6]) or an adversary B_2 that breaks the security of the signature scheme (in the sense of unforgeability under chosen-message attacks [27]). Let $\mathsf{Adv}^{\mathtt{Arch-Rec}}_{\mathtt{Pheonix}}(A)$ denote the advantage function of Phoenix when restricted to the experiment outputting 1 because the data privacy or the data integrity condition.

For the case of adversary B_1, the proof is almost identical to the one by Bellare, Boldyreva and Micali [3, Theorem 4.1]. If fact, their model and ours are essentially the same when considering only the queries that deal with encryption (even though their setting is asymmetric encryption). Following the construction of [3], we obtain an adversary B_1 that satisfies $\mathsf{Adv}^{\mathtt{ind-cpa}}_{\mathcal{SE}}(B_1) \geq 1/(nq_e) \cdot \mathsf{Adv}^{\mathtt{Arch-Rec}}_{\mathtt{Phoenix}}(A)$, where $\mathsf{Adv}^{\mathtt{ind-cpa}}_{\mathcal{SE}}(B_1)$ is the advantage function of the encryption scheme $\mathcal{SE} = (\mathcal{E}, \mathcal{D})$ for privacy under chosen-plaintext attack, n is a bound on the number of clients, and q_e is the number of `start_data` queries.

For the case of adversary B_2, the reduction is straightforward. Adversary B_2 simulate completely the execution environment for adversary A (including oracles), guesses the client identifier under whose key A will produce the forgery and then assign it the challenge verification key (that is, the key given as input to B_2) to that client. The probability we obtain a forgery under the challenge verification key is $1/n$ the probability A breaks the data integrity condition. Then $\mathsf{Adv}^{\mathtt{uf-cpa}}_{\mathcal{SS}}(B_2) \leq 1/n \cdot \mathsf{Adv}^{\mathtt{Arch-Rec}}_{\mathtt{Phoenix}}(A)$, where $\mathsf{Adv}^{\mathtt{uf-cpa}}_{\mathcal{SS}}(B_2)$ is the advantage function of the signature scheme $\mathcal{SS} = (\mathcal{K}, Sig, Vf)$ for unforgeability under chosen-message attach.

As before, n is a bound on the number of client hosts started by A.

Neither B_1 nor B_2 is guaranteed to work with noticeable probability if A does, but both adversaries combined are. Indeed, we obtain

$$\begin{aligned} &\mathsf{Adv}^{\mathtt{Arch-Rec}}_{\mathtt{Pheonix}}(A) \\ &\quad \leq \quad nq_e \cdot \mathsf{Adv}^{\mathtt{ind-cpa}}_{\mathcal{SE}}(B_1) + n \cdot \mathsf{Adv}^{\mathtt{uf-cpa}}_{\mathcal{SS}}(B_2) \end{aligned} \quad (2)$$

This result says that, as long as encryption scheme $\mathcal{SE}$ is secure (that is, its advantage function is small) and signature scheme $\mathcal{SS}$ is secure (that is, its advantage function is small) are small, the advantage function for the archival-recovery scheme `Phoenix` under data privacy and data integrity is small, and in consequence, `Pheonix` is secure. ■

Regarding availability, if the server host contacted by the client host is honest, the server will eventually send an announce message to the client host, and the client will obtain the stored data D. If any other server host does provide the client host with a "valid" backup copy D', then the copy is either equal to D or older than D. This is detected if there is a more recent backup copy. Also, client impersonating during the backup and release phases cannot occur since every server checks for valid client signature in all `data` and `release` messages.

Claim 1.9: If the communication in the system is authenticated then data availability *holds.*

Proof: It follows from the operation of the "virtual interaction" between oracle *Client-*$\mathcal{O}_b$ and *Server-*$\mathcal{O}$. Indeed, assume the experiment returns 1 because of the availability condition. Then, there exists a tuple cid^*,sid^*, and did^*, such that there was a $\texttt{start}(cid^*, sid^*)$ query (that is, a virtual interaction), $Q[cid^*].failed = Yes$ (that is, client cid^* failed) and $Q[cid^*].ActiveBackup[sid, did] = Yes$ (that is, there is an existent backup). The intuition is that, the simulated client cid^* cannot recover its backup data, even though it submitted the data ($Q[cid^*].ActiveBackup[sid, did] = Yes$), never erase it (otherwise $Q[cid^*].ActiveBackup[sid, did]$ would be No), and contacted an honest server ($\texttt{start}(cid^*, sid^*)$ query).

We now show that the probability that any adversary A causes experiment $\mathsf{Exp}^{\texttt{Arch}-\texttt{Rec}}_{\texttt{Phoenix}}(A)$ output 1 (because of the availability condition) is 0. If $Q[cid^*].ActiveBackup[sid, did] = Yes$, since sid^* is honest, it must be that (a) either the `announce`, the `request_restore`, or the `restore` message was tampered with or dropped, or (b) the server received a `release` message from cid, or (c) the client received an invalid backup (`restore` message) from sid. Case (a) contradicts the assumption on authenticated channels or the assumption that messages are eventually delivered, and (b) and (c) again contradict the authenticated channel assumption. Since the simulation is perfect and the assumptions hold with probability 1, we have that the $\mathsf{Adv}^{\texttt{Arch}-\texttt{Rec}}_{\texttt{Phoenix}}(A) = 0$ when restricted to the availability condition. ■

DISCUSSION: There are some threats to the system that are not capture by the above model: malicious servers that purposely delete other host's data, malicious servers that advertise false configurations, and malicious clients that attempt to overload a server. First, we notice that none of these threats can be fully prevented due to the nature of the system. Nonetheless, we claim that the security impact of those is small. Indeed, a malicious server that purposely deletes other host's data, specially if repeated offender, is likely to be capture by external monitoring.[12] Similarly, external monitoring can be used to detect repeated offenders among those that advertise false configurations. For example, contrasting the configuration advertised as a server with the configuration used as a client by the same host can help to identify offenders. Finally, malicious clients that attempt to overload a given server can only do it up to the maximum load limit L. This bound can be set to reasonable levels for most servers.

The security claims of this section (Claim 1.8 and Claim 1.9) rely on two assumptions. First, we assume that the communication is authenticated. This assumption is reasonable in many environments, specially close networks. If it does not hold, the availability property may be compromised but not the privacy or integrity. Indeed, authenticated channels are only required in the Recovery Phase to prevent third hosts from preventing successful recoveries: a third party, impersonating an honest server, may inject fake `restore` messages which may lead a client to think a server's backup is invalid. Therefore, the protocol can still be successfully implemented in practice in environments where message injection or message tampering is detectable. Moreover, in next section, we show how to eliminate the authenticated channels by using a distributed public key infrastructure. The second assumption is in the failure mode; we assume only fail-stop failures. In this model, after a catastrophe, the compromised client only fails by stopping. A way to enforce this property is by requiring the passphrase not be supplied to the host during the time the host is compromised. Although reasonable in some settings, this last condition may be hard to enforce in practice – pathogens may silently infect the host without the user noticing it. Such an attack can be very effective: once this "silent pathogen" gains access the client's secret keys it can issue `release` or `data` requests, thus causing the data to be deleted or overwritten. In the following section, we show how to prevent these attacks, provided that users have access to smartcards.

3) Enhanced Protocol (Sketch): In case extra resources are available, it is possible to design protocols with stronger security guarantees. In particular, we show how to cope (detect) misbehaving servers who purposely drop the backup data, and how to relax the two assumptions described in the previous section (authenticated channels and fail-stop failure model). In this section, we describe an enhanced solution that provides extra security as long as the following assumptions hold:

- The user of a client host has access to a smartcard (or similar device with limited computational power). It is not required that the device be tamper resistant.
- The setup, backup and recovery operations require access to the user smartcard.
- The infection of a client host can be detected within some fixed amount of time, say d minutes.
- There is a public key infrastructure (PKI) infrastructure available. Namely, the public keys can be authenticated, that is, anyone who receives a public key knows the identity of the corresponding owner. The PKI can be open or closed. As standard, we assume each entity first chooses its own keys and then the link between identities and verification keys is made public.

First, using the existent PKI, authenticated channels can easily be implemented on top of unauthenticated ones using standard cryptographic techniques [5], [12].

In order to cope with "silent pathogens", we do the following. The overall strategy is to split time into periods of length $t > d$ minutes. Any server stores the backup data from the two last periods. Only one update operation is allowed within one period of time. If more than one update is done for one period, the update will apply only the data for the current period. The data for the previous period is not modified. After a new update is issued in a new period, the oldest backup data is removed in the server.

We remark that requiring the server store two backups restricts the amount of storage available (indeed, it is a tradeoff:

[12] For settings in which detection of even a single misbehaving server is critical, "signed receipts" must be used. Those are discussed in next section.

if server space cannot be increased, this strategy effectively halves the space available to clients; if available client space cannot be decreased, server space must be doubled).

In this protocol, we use *key-insulated* signature and encryption schemes [22]. A key-Insulated signature scheme works like a signature scheme but "splits" the signing key between a physically protected device (e.g. the smartcard) and an insecure computer (the client host). By splitting the key, these schemes guarantee that, even though the insecure computer may be compromised (attacked) at time t, as long as the device is not compromised, no other time periods $t' \neq t$ will be compromised. Key-Insulated encryption schemes satisfy a similar property. We use this property to prevent a compromised client from erasing the current backup copy as follows. The system enforces the restriction that only one copy of saved data can be erased or overwritten during a single period of time. Therefore, as long as the host compromise is discovered (and the host restored to the honest state) in time less than t (the time of a period) no compromised client host is able to delete both data backups.

DETECTING SERVERS WHO PURPOSELY DELETE DATA: A malicious server that purposely deletes other host's data, if repeated offender, is likely to be capture by external monitoring. Nonetheless, this approach may not work if the malicious server only acts occasionally. To detect these cases, the enhanced protocol uses "signed receipts". A signed receipt is an unforgeable message that the server returns to the client after a successful completion of a backup phase. This message, signed with the publicly-known verification key of the server, must include the identity of the client requesting the backup operation, a timestamp, and a cryptographic digest of the data. Accordingly, the backup operation is not considered successful unless the client obtains a valid receipt. This receipt can be exhibited as a proof of misbehavior if later the server refuses to restore the client's backup. Hosts who are shown to be dishonest servers may then be removed from the system.

Notice that the problem of preserving the receipt is not trivial. Since the client is subject to failures, the receipt can either be saved in append-only storage or shared with a third party (say, the other servers in the client's core). We leave as open the interesting engineering problem of designing a practical system that allow host to save the signed receipts and use them to "prove" any server misbehavior in a simple but efficient way.

THE ENHANCED PROTOCOL: The following protocol illustrates the techniques described in this section.

The user smartcard stores a 2ℓ-bit-long key $K = K_1 \circ K_2$ (where ℓ is a large enough security parameter, say 128), and the client stores one or more pieces of data D that needs to backup.

Setup Phase: Before the client contacts any server.

1) The client smartcard deterministically generates a long-lived (master) secret and public key pair (msk_c, pk_c) from key K_1. Then, the smartcard sends the client the public key pk_c. This public key remains constant throughout the periods.
2) The client certifies key pk_c with the PKI authority.

Backup Phase: (period $i > 0$)

1) The client smartcard computes the secret keys $sk_c[i]$, $k[i]$ for the current period, by using the long-lived master secret key sk_c and K_2. Then $sk_c[i]$,$k[i]$ are sent to the client.
2) The client computes the encryption of $D \circ \texttt{ts}$ under key $k[i]$ and then signs it under key $sk_c[i]$. We denote by M the result.
3) The client sends M to the server host as part of the data message.
4) The server verifies that the request was signed by the client under the public key corresponding to the client's identifier.
5) The server sends an acknowledgment of the operation (a "signed receipt"), $A =$ "backup" $\circ\, cid \circ i \circ H(M)$, and its signature $s = Sig(sk_s, A)$ back to the client.

Recovery Phase: (period $i > 0$)

1) The client smartcard generates the secret keys corresponding to the current time period $sk_c[i]$ and $k[i]$, and sends them the the client.
2) The client waits until an `announce` message is received from a server host. The client sends out a `request_restore` message.
3) The server host send the stored data to the client host.
4) The client host verifies the authenticity of the received data by using key $sk_c[i]$. If it fails, abort the communication with the server host.
5) The client decrypts the backup data using $k[i]$, and compares the decrypted timestamp with the timestamps of any previously received data. If the timestamp is the most recent one received, it keeps the data. Otherwise, the client erases the data.
6) If any other `announce` message is received, repeat the above steps until no other `announce` messages exist.

SECURITY ANALYSIS: (SKETCH) The enhanced protocol achieves the same properties that the basic protocol achieves (the proof is analogous and hence omitted). Availability, however, is strengthen by means of the "signed receipts". Indeed, in the enhanced protocol any client host is able prove the misbehavior of servers who claim that no backup was ever performed by the client when, instead, the backup was done and they purposely deleted the client's data.

Claim 1.10: If the signature scheme $\mathcal{SS}$ is secure (in the sense of [27]) any server who purposely deletes a client's data after a successful backup operation (provided that no release request was made by the client) is detected with overwhelming probability.

Proof: The proof is straightforward. After a successful backup operation, the client obtains a signature from the server (the signed receipt). As long as the client checks the validity of the signature, the server (who may drop the data or refuse to honor recovery requests from the client) cannot later repudiate the signed message. More technically, there are

two cases: one, in which the client generates a signed receipt on its own, and two, where the server generates more than one message for the same signature. The latter case, called a *duplicate signature* [67], is not ruled out by the standard unforgeability notion [27] provided that the malicious signer can generate its own signing and verification keys (see [67] for a discussion). For the first case, assume a message/signature pair (m, s) corresponding to the signed receipt is output by an adversarial client with non-negligible probability, with the condition that the signature s is valid for message m under the verification key vk corresponding to the server. Then, such adversary contradicts the unforgeability property of the signature scheme.

In the second case, there may exist a second pair (m', s) (where the message m' is different from the message m shown as proof of backup by the client), also valid under verification key vk, such that the server may claim that it was message m' (and not m) the actual signed value. We claim that, even in such a case, the server *cannot claim it did not signed* that message (m, s). The reason is that such "duplicate signatures" can only exist in settings where the signer (the server) purposely chooses key pairs that are "weak" [67]. Since, in a public key infrastructure where signers (hosts) choose their own keys (as the one assumed here), no other host may affect such choice, no signer (server) can repudiate a signature of its own. ■

Additionally, the enhanced protocol prevents against pathogens that manage to access the client's secret keys. and issue `data` or `release` requests.

Claim 1.11: If the encryption scheme $\mathcal{SE}$ is a secure key-insulation encryption scheme, and the signature scheme $\mathcal{SS}$ is a secure key-insulation signature schemes [22], then any honest client recovers the most recent or second-most-recent saved data with overwhelming probability.

Proof: The proof is by reduction to the security of the key-insulation encryption and signature schemes. ■