Improving Robustness of DNS to Software Vulnerabilities

Ahmed Khurshid, Firat Kiyak, Matthew Caesar
University of Illinois at Urbana-Champaign
{khurshi1, caesar}@illinois.edu, firatkiyak@gmail.com

ABSTRACT

The ability to forward packets on the Internet is highly intertwined with the availability and robustness of the Domain Name System (DNS) infrastructure. Unfortunately, the DNS suffers from a wide variety of problems arising from implementation errors, including vulnerabilities, bogus queries, and proneness to attack. In this work, we present a preliminary design and early prototype implementation of a system that leverages diversified replication to increase tolerance of DNS to implementation errors. Our design leverages software diversity by running multiple redundant copies of software in parallel, and leverages data diversity by sending redundant requests to multiple servers. Using traces of DNS queries, we demonstrate our design can keep up with the loads of a large university's DNS traffic, while improving resilience of DNS.

1. INTRODUCTION

The Domain Name System (DNS) is a hierarchical system for mapping hostnames (e.g., www.illinois.edu) to IP addresses (e.g., 128.174.4.87). The DNS is a ubiquitous and highly crucial part of the Internet's infrastructure. Availability of the Internet's most popular services, such as the World Wide Web and email rely almost completely on DNS in order to provide their functionality. Unfortunately, the DNS suffers from a wide variety of problems, including performance issues [9, 17], high loads [11, 27], proneness to failure [25], and vulnerabilities [7]. Due to the propensity of applications and services that share fate with DNS, these problems can bring significant harm to the Internet's availability.

Much DNS research focuses on dealing with *fail-stop* errors in DNS. Techniques to more efficiently cache results [17], to cooperatively perform lookups [23, 24], to localize and troubleshoot DNS outages [22], have made great strides towards improving DNS availability. However, as fail-stop errors are reduced by these techniques, Byzantine errors become a larger bottleneck in achieving availability. Unlike fail-stop failures, where a system stops when it encounters an error, Byzantine errors include the more arbitrary class of faults where a system can violate protocol. For example, software errors in DNS implementations lead to bogus queries [27] and vulnerabilities, which can be exploited by attackers to gain access to and control DNS servers. These problems are particularly serious for DNS – while the root of the DNS hierarchy is highly *physically* redundant to avoid hardware failures, it is *not* software redundant, and hence multiple servers can be taken down with the same attack. For example, while there are 13 geographically distributed DNS root clusters, each comprised of hundreds of servers, they only run two distinct DNS software implementations: BIND and NSD (see [8] and references therein). While coordinated attacks to DoS these servers are hard, the fact these servers may share vulnerabilities makes these attacks simpler. Not as much work has been done in dealing with such problems in the context of DNS.

In this paper, we revisit the classic idea of using *diverse replication* to improve system availability. These techniques have been used to build a wide variety of robust software, especially in the context of operating systems and runtime environments [10, 12, 13, 18, 19, 28]. Several recent systems have also been proposed to decrease costs of replication, by skipping redundant computations [30], and by eliminating storage of redundant states [16]. However, to the best of our knowledge, such techniques have not been widely investigated in improving resilience of DNS. Applying these techniques in DNS presents new challenges. For example, the DNS relies on distributed operations and hence some way to coordinate responses across the wide area is required. Moreover, the DNS relies on caching and hence a faulty response may remain resident in the system for long periods of time.

In this paper we present *DR-DNS*, a design and early prototype DNS service that leverages diverse replication to mask Byzantine errors. In particular, we design and implement a *DNS hypervisor*, which allows multiple diverse replicas of DNS software to simultaneously execute, with the idea being that if one replica crashes or generates a faulty output, the other replicas will remain available to drive execution. To reduce the need to implement new code, our prototype leverages the several already-existing diverse open-source DNS implementations. Our hypervisor maintains *isolation* across running instances, so software errors do not affect other instances. It uses a simple voting procedure to select the majority result across instances, and includes a cache to offset the use of redundant queries. Voting is per-

ACSAC '11 Dec. 5-9, 2011, Orlando, Florida USA

formed in the *inbound* direction, to protect end-hosts from errors in local implementations or faulty responses returned by servers higher up in the DNS hierarchy. As our voting mechanism selects the majority result, it is able to protect end-hosts from t faulty replicas if we run $2t+1$ diverse DNS software replicas side-by-side.

Roadmap: To motivate our approach, we start by surveying common problems in DNS, existing work to address them, as well as performing our own characterization study of errors in open-source DNS software (Section 2). We next present a design that leverages diverse replication to mitigate software errors in DNS (Section 3). We then describe our prototype implementation (Section 4), and characterize its performance by replaying DNS query traces (Section 5). We then consider an extension of our design that leverages existing diversity in the current DNS hierarchy to improve resilience, and measure the ability of this approach in the wide-area Internet (Section 6). Next, we consider Content Distribution Networks and their effects on DR-DNS (Section 6.3). We finally conclude with a brief discussion of related work (Section 7) and future research directions (Section 8).

2. MOTIVATION

In this section, we make several observations that motivate our design. First, we survey the literature to enumerate several kinds of Byzantine faults that have been observed in the DNS infrastructure. Next, we study several alternatives towards achieving diversity across replicas. Finally, we study the costs involved in running diverse replicas.

Errors in DNS software: The highly-redundant and overprovisioned nature of the DNS makes it very resilient to physical failures. However, the DNS suffers from a variety of software errors that introduce correctness issues. For example, Wessels et al. [27] found large numbers of *bogus queries* reaching DNS root servers. In addition, some DNS implementation bugs are *vulnerabilities*, which can be exploited by attackers to compromise the DNS server [7] and corrupt DNS operations. While possibly more rare than physical failures, incorrect behavior is potentially much more serious, as faulty responses can be cached for long periods of time, and since a single faulty DNS server may send incorrect results to many clients (e.g., a single DNS root name server services on average 152 million queries per hour, to 382 thousand unique hosts [27]). With increasing deployments of physical redundancy and fast-failover technologies, software errors and vulnerabilities stand to make up an increasingly large source of DNS problems in the future.

Approaches to achieving diversity: Our approach leverages diverse replicas to recover from bugs. There are a wide variety of ways diversity could be achieved, and our architecture is amenable to several alternatives: the execution environment could be made different for each instance (e.g., randomizing layout in memory [10]), the data/inputs to each instance could be manipulated (e.g., by ordering queries differently for each server), and the software itself could be diverse (e.g., running different DNS implementations). For simplicity, in this paper we focus on software diversity. Software diversity has been widely used in other areas of computing, as diverse instances of software typically fail on different inputs [10, 12–14, 18, 28].

To estimate the level of diversity achieved across different DNS implementations, we performed static code analysis of nine popular DNS implementations (listed in the column headings of Figure 1b). First, to evaluate code diversity, we used *MOSS*, a tool used by a number of universities to detect student plagiarism of programming assignments. We used MOSS to gauge the degree to which code is shared across DNS implementations and versions. Second, to evaluate fault diversity, we used *Coverity Prevent*, an analyzer that detects programming errors in source code. We used Coverity to measure how long bugs lasted across different versions of the same software. We did this by manually investigating each bug reported by Coverity Prevent, and checking to see if the bug existed in other versions of the same software. Our results are shown in Figure 1. We found that most DNS implementations are diverse, with no code bases sharing more than one bug, and only one pair of code bases achieving a MOSS score of greater than 2% (Figure 1b). Operators of our system may wish to avoid running instances that achieve a high MOSS score, as bugs/vulnerabilities may overlap more often in implementations that share code. Also, we found that while implementation errors can persist for long periods across different versions of code, code after a major rewrite (e.g., BIND versions 8.4.7 and 9.0.0 in Figure 1a) tended to have different bugs. Hence, operators of our system may wish to run multiple versions of the same software in parallel to recover from bugs, but only versions that differ substantially (e.g., major versions).

3. DESIGN

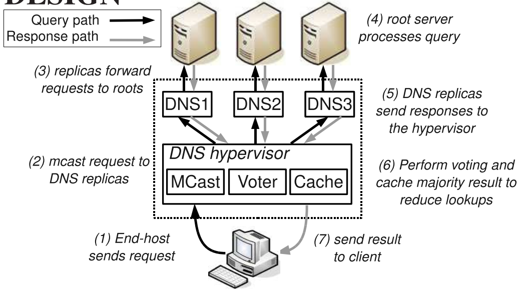

Query path
Response path
(4) root server processes query
(3) replicas forward requests to roots
DNS1
DNS2
DNS3
(5) DNS replicas send responses to the hypervisor
DNS hypervisor
(2) mcast request to DNS replicas
MCast
Voter
Cache
(6) Perform voting and cache majority result to reduce lookups
(1) End-host sends request
(7) send result to client

Figure 2: Design of DNS hypervisor.

In this section we describe the details of the design of our DNS service, which uses diverse replication to improve resilience to Byzantine failures. Our overall architecture is shown in Figure 2. Our design runs multiple replicas of DNS software atop a *DNS hypervisor*. The DNS hypervisor is responsible for mediating inputs and outputs of the DNS replicas, to make them collectively operate like a single DNS server. Our design interacts with other DNS servers using the standard DNS protocol to simplify deployment. The hypervisor is also responsible for masking bugs by using a simple voting procedure: if one replica produces an incorrect result due to a bug, or due to the fact that it is compromised by an attacker, or if it crashes, and if the instances are sufficiently diverse, then it is likely that another replica will remain available to drive execution. There are a few design choices related to DNS replicas that may affect the DR-DNS operations.

1. How many replicas to run (r)? To improve resilience to faults, the hypervisor can spawn additional replicas. Increasing the number of replicas can improve resilience, but

	8.3.0	8.4.0	8.4.7	9.0.0	9.2.0	9.3.0	9.4.0	9.5.0	9.6.0
8.3.0	200 (100%)								
8.4.0	181 (83%)	198 (100%)							
8.4.7	109 (82%)	118 (90%)	123 (100%)						
9.0.0	2 (5%)	2 (5%)	2 (5%)	108 (100%)					
9.2.0	0 (16%)	0 (15%)	0 (15%)	63 (34%)	135 (100%)				
9.3.0	0 (13%)	0 (14%)	0 (14%)	55 (26%)	119 (58%)	143 (100%)			
9.4.0	0 (12%)	0 (12%)	0 (13%)	24 (22%)	65 (50%)	76 (68%)	116 (100%)		
9.5.0	0 (11%)	0 (12%)	0 (12%)	18 (21%)	50 (47%)	59 (63%)	93 (76%)	98 (100%)	
9.6.0	0 (0%)	0 (0%)	0 (0%)	18 (23%)	47 (38%)	55 (53%)	89 (65%)	94 (71%)	110 (100%)

(a)

	BIND	djbdns	MyDNS	NSD	PowerDNS	RBLDNSD	Unbound	Posadis	dnsjava
BIND	110 (100%)								
djbdns	0 (0%)	2 (100%)							
MyDNS	0 (0%)	0 (0%)	81 (100%)						
NSD	0 (0%)	0 (0%)	0 (0%)	10 (100%)					
PowerDNS	0 (0%)	0 (0%)	0 (0%)	1 (2%)	11 (100%)				
RBLDNSD	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	5 (100%)			
Unbound	0 (0%)	0 (0%)	0 (1%)	1 (2%)	1 (2%)	0 (0%)	30 (100%)		
Posadis	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	6 (100%)	
dnsjava	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	6 (100%)

(b)

Figure 1: Number of overlapping bugs across code bases, with MOSS scores given in parenthesis, for (a) different versions of BIND (b) latest versions of different code bases. We find a high correlation between MOSS score and bug overlap.

incurs additional run time overheads (CPU, memory usage). In addition, there may be diminishing returns after a point. For example, we were only able to locate nine diverse copies of DNS software, and hence running more than that number of copies would not attain benefits from increased software diversity (though data diversity techniques may be applied, by manipulating inputs and execution environment of multiple replicas of the same software code base [10, 13]). Similarly, the hypervisor can kill or restart a misbehaving replica. A replica is misbehaving if it regularly produces different output than the majority result or if it crashes. In this case, the hypervisor first restarts the replica and if the problem persists, then the replica is killed and a new replica is spawned. This new replica may have different software or configuration.

2. How to select software that run in replicas? In order to increase the fault tolerance, DR-DNS administrators should choose diverse DNS implementations to run in replicas. For instance, using the same software with minor version changes (e.g., BIND 9.5.0 and BIND 9.6.0) in replicas should be avoided since those two versions will be likely to have common bugs. Instead, different software implementations (e.g., BIND and PowerDNS) or the same software implementation with major version changes (e.g., BIND 8.4.7 and BIND 9.6.0) are more suitable to run in replicas.

3. How to configure the replicas? Each DNS replica is independently responsible for returning a result for the query, though due to implementation and configuration differences, each replica may use a different procedure to achieve the result. For example, some replicas may perform iterative queries, while others perform recursive queries. To determine the result to send to the client, the DNS replicas may either recursively forward the request towards the DNS root, or may respond immediately (if they are authoritative, or have the response for the query cached). Furthermore, different cache sizes can affect the response times of replicas. For instance, a query can be cached in a replica, whereas another replica with a smaller cache may have to do a lookup for the same query.

4. How to select upstream DNS servers for replicas? Upstream DNS servers should be selected such that the possibility of propagating an incorrect result to the client is minimized. For instance, if all replicas use the same upstream DNS server to resolve the queries and if this upstream DNS server produces an incorrect result, then this incorrect result will be propagated to the end-host. However, one can easily configure replicas to select diverse upstream DNS servers that in result protects the end-users from misbehaving upstream DNS servers. External replication and path diversity techniques are further discussed in Section 6.

The hypervisor has a more complex design than replicas and it includes multiple modules: *Multicast, Voter* and *Cache*. Upon receiving an incoming query from the end-host, the hypervisor follows multiple steps. First, the *Multicast* module replicates the incoming query from the end-host and forwards the replicated queries to DNS replicas. Next, the *Voter* module waits for a set of answers received from the DNS replicas and then it generates the best answer depending on the voting scheme. For instance, a simple majority voting scheme selects the most common answer and returns it to the end-host. Finally, the answer is stored in the cache. The *Cache* module is responsible for storing the answers to common queries to reduce the response time. If the cache already has the answer to the incoming query of the end-host, then DR-DNS directly replies the answer without any further processing.

To mediate between the outputs of replicas, we use a simple *voting* scheme, which selects the majority result to send to downstream DNS/end-host clients. We propose a single voting procedure with several tunable parameters:

How long to wait (t, k)? Each replica in the system may take different amounts of time to respond to a request. For example, a replica may require additional processing time: it may be due to a less-efficient implementation, because it does not have the response cached and must perform a remote lookup, or because the replica is frozen/locked-up and not responding. To avoid waiting for an arbitrary amount of time, the voter only waits for a maximum amount of time t before continuing, and is allowed to return the majority early when k replicas return their responses.

Even though DR-DNS uses the simple majority voting scheme as default, a different voting scheme can be selected by the administrator. There are three main voting schemes DR-DNS currently supports: *Simple Majority Voting, Weighted Majority Voting*, and *Rank Preference Majority Voting*. Note that a DNS answer may include multiple ordered IP addresses. The end-host usually tries to communicate with the first IP address in the answer. The second IP address is used only if the first one fails to reply. Similarly the third address is used if the first two fails, and so on.

Simple Majority Voting: In this voting scheme, the ranking of IP addresses in a given DNS answer is ignored. IP addresses seen in majority of the replica answers win regardless of the ordering in replica answers. The final answer, however, orders the majority IP addresses according to their final counts. This voting scheme is a simplified version of the weighted majority voting scheme with all weights being equal to one.

Weighted Majority Voting: This voting scheme is based on the simple majority voting. The main difference of this voting scheme is that replicas have weights affecting the final result proportional to their weights. Replicas with more weights contribute more to the final result. Weights can be determined dynamically, or they can be assigned by the administrator statically in the configuration file. A dynamic weight of a replica is increased if the replica answer and the final answer has at least one common IP address. Otherwise, the replica is likely to have an incorrect result and its weight is decreased. In the static approach, the administrator may prefer to assign static weights to replicas. For instance, one may want to assign a larger weight to the replica using latest version of the same software compared to replicas using older versions. Similarly, an administrator may trust replicas using well-known software such as BIND more than replicas using other DNS software. The dynamic approach can adjust to transient buggy states much better than the static approach, but it includes an additional performance cost. Finally, a hybrid approach is also possible where each replica has two weights: a static and a dynamic weight. As a result, static weight is assigned by the administrator, whereas the dynamic weight is adjusted as DR-DNS processes queries.

Rank Preference Majority Voting: This voting scheme is also based on the simple majority voting. In the simplest rank preference voting, the IP addresses are weighted based on their ordering in the DNS answer. For instance, the first IP address in a replica answer is weighted more than the second IP address in the same answer. The final answer is generated by applying simple majority voting on the cumulative weights of IP addresses.

4. IMPLEMENTATION

To better understand the practical challenges of our design, we built a prototype implementation in Java, which we refer to as "Diverse Replica DNS" (DR-DNS). We had several goals for the prototype. First, we would like to ensure that the multiple diverse replicas are *isolated*, so that incorrect behavior/crashes of one replica do not affect performance of the other replicas. To achieve this, the DNS hypervisor runs each instance within its own process, and uses socket communication to interact with them. Second, we wanted to eliminate the need to modify the code of existing DNS software implementations running within our prototype. To do this, our hypervisor's voter acts like a DNS proxy, by maintaining a separate communication with each running replica and mediating across their outputs. In addition, we wanted our design to be as simple as possible, to avoid introducing potential for additional bugs and vulnerabilities that may lead to compromising the hypervisor. To deal with this, we focused on only implementing a small set of basic functionality in the hypervisor, relying on the replicas to perform DNS-specific logic. Our implementation consisted of 2,391 lines of code, with 1,700 spent on DNS packet processing, 378 lines on hypervisor logic including caching and voting, and the remaining 313 lines on socket communication. (by comparison, BIND has 409,045 lines of code, and the other code bases had 28,977-114,583 lines of code). Finally, our design should avoid introducing excessive additional traffic into the DNS system, and respond quickly to requests. To achieve this, our design incorporates a simple cache, which is checked before sending requests to the replicas. Our cache implementation uses the Least Recently Used (LRU) eviction policy.

On startup, our implementation reads a short configuration file describing the location of DNS software packages on disk, spawns a separate process corresponding to each, and starts up a software instance (replica) within each process. Each of these software packages must be configured to start up and serve requests on a different port[1]. The hypervisor then binds to port 53 and begins listening for incoming DNS queries. Upon receipt of a query, the hypervisor checks to see if the query's result is present in its cache. If present, the hypervisor responds immediately with the result. Otherwise, it forwards a copy of the query to each of the replicas. The hypervisor then waits for the responses, and selects the majority result to send to the client. To avoid waiting arbitrarily long for frozen/deadlocked/slow replicas to respond, the hypervisor waits no longer than a timeout (t) for a response. Note each replica's approach to processing the query may be different as well, increasing potential for diversity. For example, one replica may decide to iteratively process the query, while others may perform recursive lookups. In addition, different implementations may perform different caching strategies or have different cache sizes, and hence one copy may be able to satisfy the request from its cache while another copy may require a remote lookup. Regardless, the responses are processed by the hypervisor's voter to agree on a common answer before returning the result to the client.

[1] As part of future work, we are investigating use of virtual machine technologies to eliminate this requirement.

Our implementation has three main features to achieve high scalability, fast response and correctness. First, DR-DNS is implemented using threads with a thread pool. Upon start up, DR-DNS generates a thread pool including the threads that are ready to handle incoming queries. Whenever a query is received, it is assigned to a worker thread and run in parallel to other queries. The worker is responsible for keeping all the state information about the query including the replica answers. After the answer to the query is replied, the worker thread returns to the pool and waits for a new query. High scalability in our implementation can be reached by increasing the size of the thread pool as the load on the server increases. Second, DR-DNS is implemented in an event-driven architecture. The main advantage of the event-driven architecture is that it provides flexibility to process an event without any delay. In our implementation, almost all events related to replicas are time critical and need to be processed quickly to achieve fast response time. Finally, our hypervisor implementation consistently checks replicas for possible misbehavior. The replica answers are regularly checked against the majority result to notice any misbehavior to achieve high correctness.

5. REPLICATION WITHIN A SINGLE NODE

Setup: To study performance under heavy loads, we replayed traces of DNS requests collected at a large university (the University of Illinois at Urbana-Champaign (UIUC), which has roughly 40,000 students) against our implementation (DR-DNS) running on a single-core 2.5 GHz Pentium 4. The trace contains two days of traffic, corresponding to 1.7 million requests. Since some of the DNS software implementations we use make use of caches, we replay 5 minutes worth of trace before collecting results, as we found this amount of time eliminated any measurable cold start effects. We configure DR-DNS to run four diverse DNS implementations, namely: BIND version 9.5.0, PowerDNS version 3.17, Unbound version 1.02, and djbdns version 1.05. We run each replica with a default cache size of 32MB. Some implementations resolve requests iteratively, while others resolve recursively, and we do not modify this default behavior. Since modeling bug behavior is in itself an extremely hard research topic, for simplicity we consider a simple two-state model where a DNS server can be either in a *faulty* or *non-faulty* state. When faulty, all its responses to requests are incorrect, and the interarrival times between faulty states is sampled from a Poisson distribution with mean rate $\lambda_{nf} = 100000$ milliseconds. The duration of faulty states is also sampled from a Poisson distribution with mean rate $\lambda_f = \mu * \lambda_{nf}$. While for traditional failures μ is on the order of 0.0005 [20], to stress test our system under more frequent bugs (where our system is expected to perform more poorly), we consider of $\mu = 0.01$, $\mu = 0.003$, and $\mu = 0.001$.

Metrics: There are several benefits associated with our approach. For example, running multiple copies can improve resilience to Byzantine faults. To evaluate this, we measure the *fault rate* as the fraction of time when a DNS server is generating an incorrect output. At the same time, there are also several costs. For example, it may slow response time, as we must wait for multiple replicas to finish computing their results. To evaluate this, we measure the *processing delay* of a request through our system. In this section, we quantify the benefits (Section 5.1) and costs (Section 5.2) of our design.

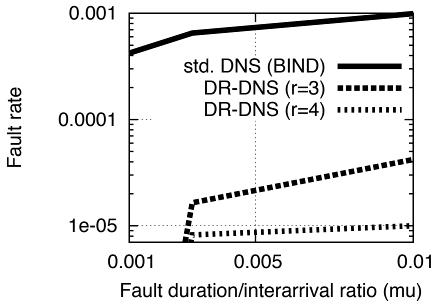


Figure 3: Effect of μ on fault rate, with t fixed at 4000ms.

5.1 Benefits

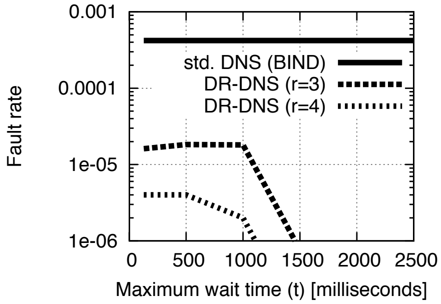


Figure 4: Effect of timeout on fault rate, with μ fixed at 0.001.

The primary benefit of our design is in improving resilience to Byzantine behavior. However, the precise amount of benefit achieved is a function of several factors, including how often Byzantine behavior occurs, how long it tends to last, the level of diversity achieved across replicas, etc. Here, we evaluate the amount of benefit gained from diverse replication under several different workloads.

First, using λ_f and λ_{nf} we measured the fraction of buggy responses returned to clients (i.e., the *fault rate*). In particular, we vary $\mu = \lambda_f / \lambda_{nf}$. For simplicity, since performance of DR-DNS is a function primarily of the ratio of these two values, we can measure performance as a function of this ratio. We found that DR-DNS reduces fault rate by multiple orders of magnitude when run with $\mu = 0.0005$. To evaluate performance under more stressful conditions, we plot in Figure 3 performance for higher ratios. We find that even under these more stressful conditions, DR-DNS reduces fault rate by an order of magnitude. We find a similar result when we vary the timeout value t, as shown in Figure 4.

Our system also can leverage spare computational capacity to improve resilience further. It does this by running additional replicas. We evaluate the effect of the number of replicas on fault rate in Figures 3 and 4. As expected, we find that increasing the number of replicas reduces fault rate. For example, when $\mu = 0.001$ and $t = 1000$, running one additional replica (increasing $r = 3$ to $r = 4$) reduces

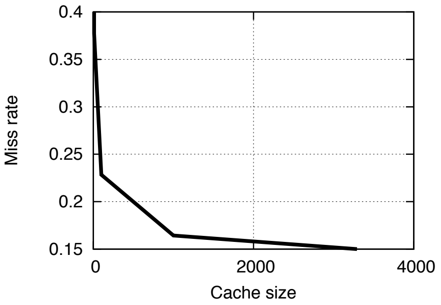


Figure 5: Amount of memory required to achieve desired hit rate.

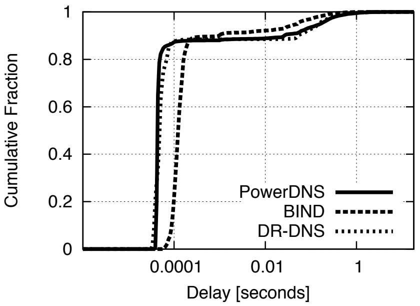


Figure 6: Amount of delay required to process requests.

fault rate by a factor of eight.

5.2 Costs

First, DNS implementations are often configured with large caches to reduce request traffic. Our system increases request traffic even further, as it runs multiple replicas, which do not share their cache contents. To evaluate this, we measured the amount of memory required to achieve a certain desired hit rate in Figure 5. Interestingly, we found that reducing cache size to a third of its original size (which would be necessary to run three replicas) did not substantially reduce hit rate. To offset this further, we implemented a *shared cache* in DR-DNS's DNS hypervisor. To improve resilience to faulty results returned by replicas, DR-DNS's cache periodically evicts cached entries. While this increases hypervisor complexity slightly (adds an additional 52 lines of code), it maintains the same hit rate as a standalone DNS server.

Second, our design imposes additional delay on servicing requests, as it must wait for the multiple replicas to arrive at their result before proceeding. To evaluate this, we measured the amount of time it took for a request to be satisfied (the round trip time from a client machine back to that originating client). Figure 6 plots the amount of time to service a request. We compare a standalone DNS server running BIND with DR-DNS running $r = 3$ copies (BIND, PowerDNS, and djbdns). We find that BIND runs more quickly than PowerDNS, and DR-DNS runs slightly more slowly than PowerDNS. This is because in its default configuration, DR-DNS runs at the speed of the slowest copy, as it waits for all copies to respond before proceeding. To mitigate this, we found that increasing the cache size can offset any additional delays incurred by processing.

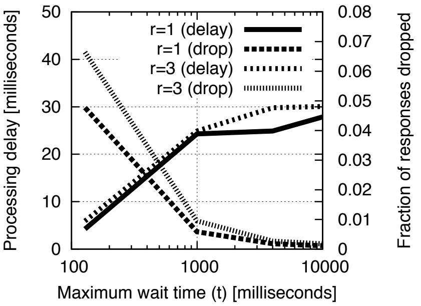


Figure 7: Effect of timeout on reducing delay.

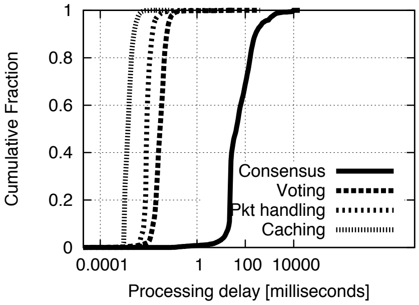


Figure 8: Microbenchmarks showing most of delay is spent waiting for replicas to reach consensus.

An alternate way to reduce delay is to vary t (to bound the maximum amount of time the voter will wait for a replica to respond) or k (to allow the voter to proceed when the first k replicas finish processing). As one might expect, we found that increasing k or increasing t both produce a similar effect: increasing them reduces fault rate, but increases delay. However, we found that manipulating t provided a way to bound worst-case delay (e.g., to make sure a request would be serviced within a certain time bound), while manipulating k provided a worst-case resilience against bugs (e.g., to make sure a response would be voted upon by at least k replicas). Also, as shown in Figure 7, we found that making t too small increased the number of dropped requests. This happens because, if no responses from replicas are received before the timeout, DR-DNS drops the request (we also considered a scheme where we wait for at least one copy to respond, and achieved a reduced drop rate at the expense of increased delay).

To investigate the source of delays in DR-DNS, we performed microbenchmarking. Here, we instrument DR-DNS with timing code to measure how much time is spent handling/parsing DNS packets, performing voting, checking the local cache, and waiting for responses from remote DNS servers. Figure 8 shows that the vast majority of request processing time is spent on waiting for the replicas to finish communicating with remote servers and to achieve consensus. This motivates our use of k and t: since these parameters control the amount of time required to achieve consensus, they provide knobs that allow us to effectively control delay (or to trade it off against fault rate).

Under heavy loads, we found that DR-DNS dropped a slightly larger number of requests than a standalone DNS server (0.31% vs. 0.1%). Under moderate and light loads, we found DR-DNS dropped fewer requests than a standalone DNS server (0.004% vs. 0.036%). This happens because there is some small amount of loss between DR-DNS and the remote root servers, and since like other schemes that replicate queries [23], our design sends multiple copies of a request, it can recover from some of these losses at the added expense of additional packet overhead.

6. REPLICATION ACROSS NODES

Our work so far has focused on *internal replication* – running multiple DNS replicas within a single host. However, the distributed nature of the DNS hierarchy means that there are often multiple remote DNS servers that can respond to a request. This provides the opportunity for DR-DNS to leverage *external replication* as well. Hence, in order to increase the reliability of the whole DNS query resolution process, we use the existing DNS hierarchy and redundancy as another form of diversity. In particular, we extend the DR-DNS design to allow its internal DNS replicas to send queries to multiple diverse upstream DNS servers and apply voting for the final answer. *Path diversity*, the selection of the diverse upstream DNS servers, can be considered as leveraging diversity across upstream DNS servers. While this approach presents some practical challenges, we present results to indicate the benefits of maintaining and increasing diversity in the existing DNS hierarchy. The rest of the section is organized as follows. Section 6.1 provides the design extensions of DR-DNS to support path diversity. Section 6.2 presents the benefits and costs of path diversity. Finally, Section 6.3 discusses the path diversity in the existence of CDNs and DNS load balancing.

6.1 Leveraging path diversity in DR-DNS

We extend the DR-DNS design to leverage path diversity in the DNS hierarchy. In the extended DR-DNS design each internal DNS replica (1) sends replicated queries to multiple diverse upstream DNS servers and (2) applies voting on the received answers. Hence, we extended each internal DNS replica with a *replica hypervisor*, i.e. a DNS hypervisor without a cache. The DNS hypervisor already has a Multicast module (MCast) to replicate the queries and Voter module to apply majority voting on the received answers. In this case, we disabled the caches of replica hypervisors since DNS replicas include their own caches. Whenever a DNS replica wants to send a query to upstream DNS servers, it simply sends the query to its replica hypervisor. Then, the multicast module in the replica hypervisor replicates the query and forwards copies to selected upstream DNS servers. Upon receiving answers, the voter module simply applies majority voting on the answers and replies to its DNS replica with the final answer.

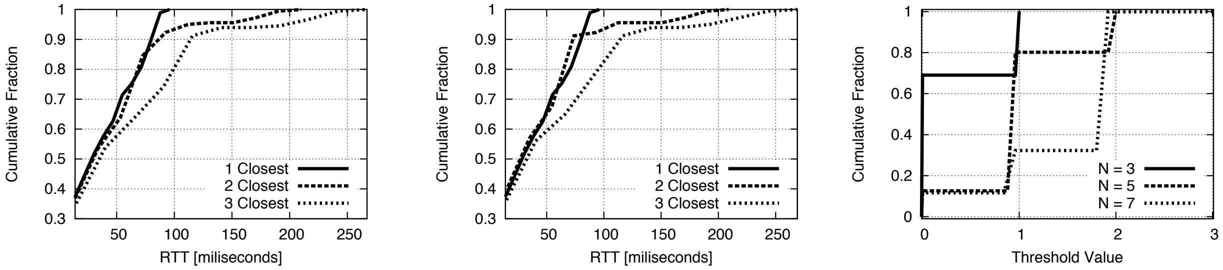
6.2 Benefits and Costs

The primary benefit of our design extension is in improving resilience to errors that can occur in any DNS servers involved in the query resolution. However, the amount of exact benefit gained depends on the level of diversity achieved across upstream DNS servers. To increase the reliability of DNS query resolution process, one needs to avoid sending queries to upstream DNS servers that share software vulnerabilities. Hence, we select the upstream DNS servers with either different software implementations (e.g., BIND and PowerDNS) or the same software implementation with major version changes (e.g., BIND 8.4.7 and BIND 9.6.0). One can also select upstream DNS servers running different operating systems (e.g., Windows or Linux).

To measure diversity of the existing DNS infrastructure, we used two open-source fingerprinting tools: (1) *fpdns*, a DNS software fingerprinting tool [5], and (2) *nmap*, an OS fingerprinting tool [6]. fpdns is based on borderline DNS protocol behavior. It benefits from the fact that some DNS implementations do not offer the full set of features of DNS protocol. Furthermore, some implementations offer extra features outside the protocol set, and even some implementations do not conform to standards. Given these differences among implementations, fpdns sends a series of borderline queries and compares the responses against its database to identify the vendor, product and version of the DNS software on the remote server. The nmap tool, on the other hand, contains a massive database of heuristics for identifying different operating systems based on how they respond to a selection of TCP/IP probes. It sends TCP packets to the hosts with different packet sequences or packet contents that produce known distinct behaviors associated with specific OS TCP/IP implementations.

First, we collected a list of 3,000 DNS servers from the DNS root traces [4] on December 2008 and probed these DNS servers to check their availability from a client within the UIUC campus network. Then, we eliminated the non-responding servers. Second, we identified the DNS software and OS version of each available server with fpdns and nmap tools. This gives us a list of available DNS servers with corresponding DNS software and OS versions. One can easily select diverse upstream DNS servers from this list. However, careless selection comes with major cost: increased delay due to forwarding queries to distant upstream DNS servers compared to closest local upstream DNS server. Hence, one needs to select diverse upstream DNS servers that are close to the given host to minimize the additional delay. Here, we propose a simple selection heuristic: for a given host, we first find the top k diverse DNS servers which have the longest prefix matches with the host IP address. This results in k available DNS servers topologically very close to the host. Then, we use the King delay estimation methodology [15] to order these DNS servers according to their computed distance from the host. For practical purposes, we have used $k = 5$ in our experiments. Finally, to evaluate the additional delay, we first collected a list of 1000 hosts from [3]. Then, for each host in this list we measured the amount of extra time needed to use multiple diverse upstream DNS servers. Figures 9a (DNS software diversity) and 9b (OS diversity) plot the amount of total time to service the queries as additional diverse upstream DNS servers are accessed.

The results show that BIND is the most common DNS software among DNS servers we analyzed (69.8% BIND v9.x, 10% BIND v8.x). We also found that OS distribution among DNS servers is more balanced: 54% Linux and 46% Windows. Even though the software diversity among public DNS servers should be improved, the results indicate that current degree of diversity is sufficient for our reliability purposes. However, there is a delay cost in using multiple upstream DNS servers since we have to wait for all answers of the upstream DNS servers. This extra delay is shown in Figures 9a and 9b. We found that with an average of 26ms delay increase, we can use additional upstream DNS servers with



(a) (b) (c)

Figure 9: (a) Achieving diversity may require sending requests to more distant (higher-latency) DNS servers. Effect of DNS software diversity on latency inflation. (b) Effect of OS software diversity on latency inflation. (c) Number of failures that can be masked with N, the number of upstream DNS servers.

diverse DNS software to increase the reliability. Similarly, upstream DNS servers with diverse OS software can be used with an average of 19ms extra delay. We found that we can use OS diversity with a smaller overhead since OS distribution among DNS servers is more balanced. We conclude that DR-DNS extensions to use path diversity improves the reliability and protects the end users from software bugs and failures of upstream DNS servers. Moreover, the average delay cost is small and can be tolerated by the end users. Finally, our design increases the traffic load on upstream DNS servers, and this component of DR-DNS may be disabled if needed. However, we believe that the increasing severity of DNS vulnerabilities and software errors, coupled with the reduced costs of multicore technologies making computational and processing capabilities cheaper, will make this a worthwhile tradeoff.

6.3 Effect of Content Distribution Networks

Content distribution networks (CDNs) deliver content to end-hosts from geographically distributed servers using the following procedure. First, a *content provider* provide the content to a CDN. Next, the CDN replicates the content in replicas, multiple geographically distributed servers. Finally, an end-host requesting the content is redirected to one of the replicas instead of the original content provider. There are numerous advantages of CDNs: scalability, load balancing, high performance, etc. Some CDNs use *DNS redirection* technique to redirect the end-hosts to the best available CDN server for content delivery. Therefore, the CDN replica providing the content to the end-host may change dynamically depending on a few parameters including the geographic location of the end-host, network conditions, the time of the day and the load on the CDN replicas [1, 26]. As a result, a specific end-host may receive different DNS answers to the same query in subsequent requests. Hence, one might ask the question: *How does the existence of CDNs affect DR-DNS?*

DR-DNS applies majority voting to multiple DNS answers where each DNS answer includes a set of ordered IP addresses. In the existence of CDNs, DNS answers include IP addresses of CDN replicas which can deliver the content efficiently. Therefore, two DNS answers to the same query may not have any common IP addresses. This results in no winning IP set after the majority voting in DR-DNS. However, in this case DR-DNS cannot make any final decision and simply returns all IP addresses to the end-host. As a rule, DR-DNS returns all IP addresses from the DNS answers if it fails to find the majority set. Note that this approach still works *correctly* since *any* of the returned IP addresses will direct the client to a valid CDN server, and DR-DNS ensures that one of those IP addresses is always returned. However, DR-DNS heavily relies on the results of majority voting to improve the reliability. To evaluate how CDNs affect the reliability of DR-DNS, we measured the variation in DNS answers from Akamai, a well known CDN.

6.3.1 Effect of geographic location

CDNs use DNS redirection technique to redirect the end-hosts to the best available replicas. In DNS redirection, the end-host's query is handled by the DNS server authoritative for the requested domain, which is controlled by the CDN to return IP addresses of CDN replicas from which the content can be delivered most efficiently. CDN replicas for content delivery is chosen dynamically depending on the location of the end-host. For instance, an end-host located in New York may be more likely to be redirected to a replica in New Jersey rather than a replica in Seattle. Hence, in the existence of CDNs, DNS answers heavily depend on the location of the upstream DNS server. Two geographically distant upstream DNS servers will be likely to return different IP sets in the DNS answers to the same query. However, DR-DNS relies on the majority voting that elevates the common IP addresses in the returned DNS answers to improve reliability. To understand how often DR-DNS cannot do majority voting in the existence of CDNs, we carried out the following experiment. First, we selected the top 1000 most popular worldwide domains from [2] to use as queries since many content providers are in this list. Even though using top domains as queries results in biased measurements, it helps us to get an upper bound for the worst case. Next, for each query we randomly selected $N = 3, 5, 7$ upstream DNS servers from (1) the same state (Louisiana), (2) same country (USA) and (3) different countries. For the third experiment, we selected the countries from distinct continents (USA, Brazil, UK, Turkey, Japan, Australia, South Africa) to again evaluate the worst case. Table 1 shows the ratio of top domain queries where DR-DNS cannot find the majority set.

We found that CDNs affect the majority voting more if the selected upstream DNS servers are geographically distributed around the world. The results also show that CDN effects can be minimized in DR-DNS by selecting upstream DNS servers from a smaller region. For instance, selecting upstream DNS servers from the same state guarantees

	$N = 3$	$N = 5$	$N = 7$
State (Louisiana)	0.3%	0.7%	0.8%
Country (USA)	1.0%	2.0%	1.7%
World	1.6%	2.4%	2.0%

Table 1: The ratio of top domain queries where majority voting fails. N is the number of upstream DNS servers.

that DR-DNS improves the reliability of more than 99% of the queries. The main conclusion is that one should choose upstream DNS servers close to end-hosts for better reliability. Moreover, the heuristic that we developed in the previous section for path diversity chooses diverse upstream DNS servers close to the end-host, so DR-DNS already minimizes CDN effects.

	$N = 3$	$N = 5$	$N = 7$
USA - Top Domains	1.0%	2.0%	1.7%
USA - UIUC Trace	0.6%	0.9%	0.7%

Table 2: The ratio of top domain queries that majority voting fails, for USA-located hosts. The UIUC trace contains less queries to CDN clients. N is the number of upstream DNS servers.

Next, to obtain more realistic results, we repeated the same experiment with 1000 queries randomly selected from the UIUC primary DNS server trace. Table 2 shows that DR-DNS is less affected from CDNs in the UIUC trace.

6.3.2 Effect of number of upstream DNS servers

Next, we studied how the control overhead and the resilience in DR-DNS changes as we increase the number of upstream DNS servers. We found that control overhead increased linearly with the number of simultaneous requests, as expected. To evaluate the resilience, we performed the following experiment: we repeatedly send a random DNS query to multiple servers, and look at their answers. In some cases, the IP addresses in DNS answers may differ due to CDNs. If the majority voting fails, then DR-DNS doesn't improve the reliability. To evaluate performance, we then count these cases. Majority voting finds a winning IP set if more than half of the upstream DNS servers agree on at least one IP address. Let N be the number of upstream DNS servers DR-DNS queries simultaneously. Then, the minimum number of upstream servers that need to agree for the majority result is $N_{min} = \lfloor \frac{N}{2} \rfloor + 1$. For a given query, let C be the number of upstream DNS servers that agrees on the winning IP set (majority voting succeeds). Since there is a winning IP set, $C >= N_{min}$. Now, we define the threshold $T = C - N_{min}$ to measure how many extra upstream DNS servers agreed on the majority set. Note that if $T = 0$, then the majority result is agreed upon by N_{min} number of upstream DNS servers. In this case, if one server that contributes to the majority result becomes buggy, then majority voting fails. However, if $T = N - N_{min}$ is at maximum value (all upstream DNS servers agree on the winning IP set), then to fail in majority voting, $N - N_{min} + 1$ upstream DNS servers need to become buggy simultaneously. Hence, to evaluate resilience, we measure the threshold T for every query. The reliability of the majority answer is directly proportional to threshold value T. Figure 9c shows the increase in reliability as we increase the number of upstream DNS servers.

Overall, we found that for most queries, DR-DNS enabled with our external replication techniques could perform majority voting to mask the bug, thereby increasing reliability. DR-DNS was unable to do majority voting for only 0.3% of the top domain queries if three upstream DNS servers are selected from the same state. While for these small number of queries it does not mask the fault, it is important to note that it performs no worse than a normal (uninstrumented) baseline DNS system even in these cases. Finally, the reliability of the majority answer can be increased by sending queries to more upstream DNS servers.

7. RELATED WORK

DNS suffers from a wide variety of problems. Reliability of DNS can be harmed through a number of ways. Physical outages such as server failures or dropped lookup packets may prevent request processing. The DNS also suffers from performance issues, which can delay responses or increase loads on servers [27]. DNS servers may be misconfigured, which may lead to cyclic dependencies between zones, or cause servers to respond incorrectly to requests [22]. Also, implementation errors in DNS code can make servers prone to attack, and can lead to faulty responses [7, 25].

Dealing with failures in DNS is certainly not a new problem. For example, DNS root zones are comprised of hundreds of geographically distributed servers, and anycast addressing is used to direct requests to servers, reducing proneness to physical failures. Redundant lookups and cooperative caching can substantially reduce lookup latencies and resilience to fail-stop failures [23, 24]. Troubleshooting tools that actively probe via monitoring points can detect large classes of misconfigurations [22]. Our work does not aim to address fail-stop failures, and instead we leverage these previous techniques, which work well for such problems.

However, these techniques do not aim to improve resilience to problems arising from implementation errors in DNS code. A vulnerability in a single DNS root server affects hundreds of thousands of unique hosts per hour of compromise [11, 27], and a single DNS name depends on 46 servers on average, whose compromise can lead to domain hijacks [25]. The DNS has experienced several recent high-profile implementation errors and vulnerabilities. As techniques dealing with fail-stop failures become more widely deployed, we expect that implementation errors may make up a larger source of DNS outages. While there has been work on securing DNS (e.g., DNSSEC), these techniques focus on authenticating the source of DNS information and checking its integrity, rather than masking incorrect lookup results. In this work, we aim to address this problem at its root, by increasing the software diversity of the DNS infrastructure.

Software diversity techniques have been used to prevent attacks on large scale networks in multiple studies. It has been shown that reliability of single-machine servers to software bugs or attacks can be increased with diverse replication [13]. In another work, diverse replication is used to protect large scale distributed systems from Internet catastrophes [18]. Similarly, to limit malicious nodes to compromise its neighbors in the Internet, software diversity is used to assign nodes diverse software packages [21]. In another work, to increase the defense capabilities of a network, the authors suggest increasing the diversity of nodes to make the network more heterogeneous [29]. To the best of our knowledge, our work is the first to directly address the root cause of implementation errors in DNS software, via the use of diverse replication. However, our work is only an early first step in this direction, and we are currently investigating a wider array of practical issues as part of future work.

8. CONCLUSIONS

Today's DNS infrastructure is subject to implementation errors, leading to vulnerabilities and buggy behavior. In this work, we take an early step towards addressing these problems with *diverse replication*. Our results show that available DNS software packages have sufficient diversity in code resulting in a minimal number of shared bugs. However, DNS software with minor version changes share most of the code base resulting in less diversity. We have also found that the number of bugs is not reduced in later versions of the same software since usually new functionality is added to software introducing new bugs. We also find that our system masks buggy behavior with diverse replication, reducing the fault rate by an order of magnitude. Increasing the number of replicas further decreases the fault rate. Our results indicate that DR-DNS runs quickly enough to keep up with the loads of a large university's DNS queries. In addition, DR-DNS can leverage redundancy in the current DNS server hierarchy (replicated DNS servers, public DNS servers, etc.). We can use this redundancy to select diverse upstream DNS servers to protect the end-host from possible errors existing in the upstream servers. Selecting a different upstream DNS server may increase response time, but our results show that a slight increase in response time enables a significant improvement in reliability. CDNs and DNS-level load balancing may result in DNS queries being resolved to different sets of IP addresses, which can limit ability of DR-DNS to mask bugs across remote servers. However, our results indicate that performance is reduced only minimally in practice, and correctness of operation is not affected.

While our results are promising, much more work remains to be done. First, we plan to design a *server-side* voting strategy, to protect the DNS root from bogus queries [27]. Also, we plan to investigate whether porting our Java-based implementation to C++ will speed request processing further. We are also currently in the process of deploying our system for use within the campus network of a large university, to investigate practical issues in a live operational network. Finally, we plan to extend our study to include many other protocols to investigate how diversity changes among protocols. This helps us to generalize our method for other protocols.

9. REFERENCES

[1] Akamai. http://www.akamai.com.

[2] Alexa. http://www.alexa.com.

[3] CAIDA. http://www.caida.org/data/.

[4] DNS-OARC. Domain name system operations, analysis, and research center. http://www.dns-oarc.net.

[5] fpdns - DNS fingerprinting tool. http://code.google.com/p/fpdns.

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

[7] Securityfocus: Bugtraq mailing list. http://www.securityfocus.com/ vulnerabilities.

[8] Root nameserver (Wikipedia article). http://en.wikipedia.org/wiki/Root_nameserver.

[9] Bent, L., and Voelker, G. Whole page performance. In *The 7th International Web Caching Workshop (WCW)* (August 2002).

[10] Berger, E., and Zorn, B. Diehard: Probabilistic memory safety for unsafe languages. In *Programming Languages Design and Implementation* (June 2006).

[11] Brownlee, N., kc claffy, and Nemeth, E. DNS measurements at a root server. In *IEEE GLOBECOM* (November 2001).

[12] Castro, M., and Liskov, B. Practical byzantine fault tolerance. In *OSDI* (February 1999).

[13] Chun, B.-G., Maniatis, P., and Shenker, S. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX ATC* (June 2008).

[14] Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A. A sense of self for unix processes. In *IEEE Symposium on Security and Privacy* (1996), pp. 120–128.

[15] Gummadi, K. P., Saroiu, S., and Gribble, S. D. King: Estimating latency between arbitrary Internet end hosts. In *SIGCOMM Internet Measurement Workshop* (2002).

[16] Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A., Vahdat, A., Varghese, G., and Voelker, G. Difference engine: Harnessing memory redundancy in virtual machines. In *OSDI* (December 2008).

[17] Jung, J., Sit, E., Balakrishnan, H., and Morris, R. DNS performance and the effectiveness of caching. In *ACM SIGCOMM* (October 2002).

[18] Junqueira, F., Bhagwan, R., Hevia, A., Marzullo, K., and Voelker, G. Surviving Internet catastrophes. In *USENIX ATC* (April 2005).

[19] Keller, E., Yu, M., Caesar, M., and Rexford, J. Virtually eliminating router bugs. In *CoNEXT* (December 2009).

[20] Markopoulou, A., Iannaccone, G., Bhattacharyya, S., Chuah, C.-N., and Diot, C. Characterization of failures in an IP backbone. In *IEEE INFOCOM* (March 2004).

[21] O'Donnell, A. J., and Sethu, H. On achieving software diversity for improved network security using distributed coloring algorithms. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security* (New York, NY, USA, 2004), ACM, pp. 121–131.

[22] Pappas, V., Faltstrom, P., Massey, D., and Zhang, L. Distributed DNS troubleshooting. In *ACM SIGCOMM Workshop on Network Troubleshooting* (August 2004).

[23] Park, K., Pai, V. S., Peterson, L., and Wang, Z. CoDNS: Improving DNS performance and reliability via cooperative lookups. In *OSDI* (December 2004).

[24] Ramasubramanian, V., and Sirer, E. G. The design and implementation of a next generation name service for the Internet. In *ACM SIGCOMM* (August 2004).

[25] Ramasubramanian, V., and Sirer, E. G. Perils of transitive trust in the domain name system. In *Internet Measurement Conference* (October 2005).

[26] Su, A.-J., Choffnes, D. R., Kuzmanovic, A., and án E. Bustamante, F. Drafting behind Akamai (Travelocity-based detouring). In *ACM SIGCOMM* (2006).

[27] Wessels, D., and Fomenkov, M. Wow, that's a lot of packets. In *Passive and Active Measurement* (April 2003).

[28] Yumerefendi, A., Mickle, B., and Cox, L. Tightlip: Keeping applications from spilling the beans. In *NSDI* (April 2007).

[29] Zhang, Y., Vin, H., Alvisi, L., Lee, W., and Dao, S. K. Heterogeneous networking: A new survivability paradigm. In *NSPW '01: Proceedings of the 2001 workshop on New security paradigms* (New York, NY, USA, 2001), ACM, pp. 33–39.

[30] Zhou, Y., Marinov, D., Sanders, W., Zilles, C., d'Amorim, M., Lauterburg, S., and Lefever, R. Delta execution for software reliability. In *Hot Topics in System Dependability* (June 2007).