Virtually Eliminating Router Bugs

Eric Keller* Minlan Yu* Matthew Caesar† Jennifer Rexford*

* Princeton University, Princeton, NJ, USA † UIUC, Urbana, IL, USA

ekeller@princeton.edu {minlanyu, jrex}@cs.princeton.edu caesar@cs.uiuc.edu

ABSTRACT

Software bugs in routers lead to network outages, security vulnerabilities, and other unexpected behavior. Rather than simply crashing the router, bugs can violate protocol semantics, rendering traditional failure detection and recovery techniques ineffective. Handling router bugs is an increasingly important problem as new applications demand higher availability, and networks become better at dealing with traditional failures. In this paper, we tailor software and data diversity (SDD) to the unique properties of routing protocols, so as to avoid buggy behavior at run time. Our bug-tolerant router executes multiple diverse instances of routing software, and uses voting to determine the output to publish to the forwarding table, or to advertise to neighbors. We design and implement a router hypervisor that makes this parallelism transparent to other routers, handles fault detection and booting of new router instances, and performs voting in the presence of routing-protocol dynamics, without needing to modify software of the diverse instances. Experiments with BGP message traces and open-source software running on our Linux-based router hypervisor demonstrate that our solution scales to large networks and efficiently masks buggy behavior.

Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*; C.4 [**Performance of Systems**]: [Fault tolerance, Reliability, availability and serviceability]

General Terms

Design, Reliability

Keywords

Routers, Bugs, Reliability, BGP

CoNEXT'09, December 1–4, 2009, Rome, Italy.

1. INTRODUCTION

The Internet is an extremely large and complicated distributed system. Selecting routes involves computations across millions of routers spread over vast distances, multiple routing protocols, and highly customizable routing policies. Most of the complexity in Internet routing exists in protocols implemented as *software* running on routers. These routers typically run an operating system, and a collection of protocol daemons which implement the various tasks associated with protocol operation. Like any complex software, routing software is prone to implementation errors, or *bugs*.

1.1 Challenges in dealing with router bugs

The fact that bugs can produce incorrect and unpredictable behavior, coupled with the mission-critical nature of Internet routers, can produce disastrous results. This can be seen from the recent spate of high-profile vulnerabilities, outages, and huge spikes in global routing instability [40, 39, 16, 22, 21, 13, 31]. Making matters worse, ISPs often run the same protocols and use equipment from the same vendor network-wide, increasing the probability that a bug causes simultaneous failures or a network-wide crash. While automated systems can *prevent* misconfigurations from occurring [23, 24], these techniques do not work for router bugs, and in fact the state-of-the-art solution today for dealing with router bugs involves heavy manual labor—testing, debugging, and fixing code. Unfortunately operators must wait for vendors to implement and release a patch for the bug, or find an intermediate work around on their own, leaving their networks vulnerable in the meantime.

Worse still, bugs are often discovered only *after* they cause serious outages. While there has been work on dealing with failures in networks [35, 33, 27], router bugs differ from traditional "fail-stop" failures (failures that cause the router to halt in some easily-detectable way) in that they violate the semantics of protocol operation. Hence a router can keep running, but behave incorrectly – by advertising incorrect information in routing updates, or by distributing the wrong forwarding-table entries to the data plane, which can trigger persistent loops, oscillations, packet loss, session failure, as well as new kinds of anomalies that can't happen in correctly behaving protocols. This fact, coupled with the high complexity and distributed nature of Internet routing, makes router bugs notoriously difficult to detect, localize, and contain.

As networks become better at dealing with traditional failures, and as systems that automate configuration become more widely deployed, we expect bugs to become a major

roadblock in improving network availability. While we acknowledge the long-standing debate in the software engineering community on whether it is possible to completely prevent software errors, we believe unforeseen interactions across protocols, the potential to misinterpret RFCs, the increasing functionality of Internet routing, and the ossification of legacy code and protocols will make router bugs a "fact-of-life" for the foreseeable future and we proceed under that assumption.

1.2 The case for diverse replication in routers

Unlike fail-stop failures, router bugs can cause *Byzantine* faults, i.e., they cause routers to not only behave incorrectly, but violate protocol specification. Hence, we are forced to take a somewhat heavy-handed approach in dealing with them (yet as we will find, one that appears to be necessary, and one that our results indicate is practical). In particular, our design uses a simple replication-based approach: instead of running one instance of routing software, our design uses a *router hypervisor*[1] to run multiple *virtual* instances of routing software in parallel. The instances are made *diverse* to decrease the likelihood they all simultaneously fail due to a bug. We leverage *data diversity* (to manipulate the inputs to the router, for example by jittering arrival time of updates, or changing the layout of the executable in memory) and *software diversity* (given multiple implementations of routing protocols already exist, running several of them in parallel). We then rely on Byzantine-fault tolerant (BFT) techniques to select the "correct" route to send to the forwarding table (FIB), or advertise to a neighbor [2].

The use of BFT combined with *diverse replication* (running multiple diverse instances) has proven to be a great success in the context of traditional software, for example in terms of building robust operating systems and runtime environments [18, 28, 36, 44, 12]. These techniques are widely used since heterogeneous replicas are unlikely to share the same set of bugs [18, 28, 44]. In this paper, we adapt diverse replication to build router software that is tolerant of bugs.

A common objection of this approach is performance overheads, as running multiple replicas requires more processing capacity. However, BFT-based techniques provide a simple (and low-cost) way to leverage the increasingly parallel nature of multicore router processors to improve availability without requiring changes to router code. Network operators also commonly run separate *hardware* instances for resilience, across multiple network paths (e.g., multihoming), or multiple routers (e.g., VRRP [27]). Some vendors also protect against fail-stop failures by running a hot-standby redundant control plane either on multiple blades within a single router or even on a single processor with the use of virtual machines [19], in which case little or no additional router resources are required. Since router workloads have long periods with low load [9], redundant copies may be run during idle cycles. Recent breakthroughs vastly reduce computational overhead [45] and memory usage [26], by skipping redundancy across instances.

[1]We use the term *router hypervisor* to refer to a software layer that maintains arbitrates between outputs from multiple software replicas. However, our approach does not require true virtualization to operate, and may instead take advantage of lighter-weight containerization techniques [4].

[2]For BGP, sources of non-determinism such as age-based tie-breaking and non-deterministic MED must be disabled. This is often done by operators anyway because they lead to unpredictable network behavior (making it hard to engineer traffic, provision network capacity, and predict link loads).

1.3 Designing a Bug-Tolerant Router

In this paper, we describe how to eliminate router bugs "virtually" (with use of virtualization technologies). We design a *bug-tolerant* router (BTR), which masks buggy behavior, and avoids letting it affect correctness of the network layer, by applying software and data diversity to routing. Doing so, however, presents new challenges that are not present in traditional software. For example, (i) wide-area routing protocols undergo a rich array of dynamics, and hence we develop BFT-based techniques that react quickly to buggy behavior without over-reacting to transient inconsistencies arising from routing convergence, and (ii) our design must interoperate with existing routers, and not require extra configuration efforts from operators, and hence we develop a *router hypervisor* that masks parallelism and churn (*e.g.,* killing a faulty instance and bootstrapping a new instance).

At the same time we leverage new opportunities made available by the nature of routing to build custom solutions and extend techniques previously developed for traditional software. For example, (i) routers are typically built in a modular fashion with well-defined interfaces, allowing us to adapt BFT with relatively low complexity, and implement it in the hypervisor with just a few hundred lines of code, (ii) using mechanisms that change transient behavior without changing steady-state outcomes are acceptable in routing, which we leverage to achieve diversity across instances, and (iii) routing has limited dependence on past history, as the effects of a bad FIB update or BGP message can be undone simply by overwriting the FIB or announcing a new route, which we leverage to speed reaction by selecting a route early, when only a subset of instances have responded, and updating the route as more instances finish computing. Moreover, router outputs are independent of the precise ordering and timing of updates, which simplifies recovery and bootstrapping new instances.

The next section discusses how diversity can be achieved and how effective it is, followed by a description of our design (Section 3) and implementation (Section 4). We then give performance results in Section 5, consider possible deployment scenarios in Section 6, contrast with related work in Section 7, and conclude in Section 8.

2. SOFTWARE AND DATA DIVERSITY IN ROUTERS

The ability to achieve diverse instances is essential for our bug-tolerant router architecture. Additionally, for performance reasons, it is important that the number of instances that need to be run concurrently is minimal. Fortunately, the nature of routing and the current state of routing software lead to a situation where we are able to achieve enough diversity and that it is effective enough that only a small number of instances are needed (*e.g.,* 3-5, as discussed below). In this section we discuss the various types of diversity mechanisms, in what deployment scenario they are likely to be used, and how effective they can be in avoiding bugs.

Unfortunately, directly evaluating the benefits of diversity across large numbers of bugs is extremely challenging, as it requires substantial manual labor to reproduce bugs. Hence, to gain some rough insights, we studied the bug re-

ports from the XORP and Quagga Bugzilla databases [8, 5], and taxonomized each into what type of diversity would likely avoid the bug and experimented with a small subset, some of which are described in Table 1.[3]

2.1 Diversity in the software environment

Code base diversity : The most effective, and commonly thought of, type of diversity is where the routing software comes from different code bases. While often dismissed as being impractical because a company would never deploy multiple teams to develop the same software, we argue that diverse software bases are already available and that router vendors do not need to start from scratch and deploy multiple teams.

First, consider that there are already several open-source router software packages available (*e.g.,* XORP, Quagga, BIRD). Their availability has spawned the formation of a new type of router vendor based on building a router around open-source software [7, 8].

Additionally, the traditional (closed-source) vendors can make use of open-source software, something they have done in the past (*e.g.,* Cisco IOS is based on BSD Unix), and hence may run existing open-source software as a "fallback" in case their main routing code crashes or begins behaving improperly. Router vendors that do not wish to use open-source software have other alternatives for code diversity, for example, router vendors commonly maintain code acquired from the purchase of other companies [38].

As a final possibility, consider that ISPs often deploy routers from multiple vendors. While it is possible to run our bug-tolerant router across physical instances, it is most practical to run in a single, virtualized, device. Even without access to the source code, this is still a possibility with the use of publicly available router emulators [1, 3]. This way, network operators can run commercial code along with our hypervisor directly on routers or server infrastructure without direct support from vendors. While intellectual property restrictions arising from their intense competition makes vendors reticent to share source code with one another, this also makes it likely that different code bases from different vendors are unlikely to share code (and hence unlikely to share bugs).

We base our claim that this is the most effective approach partially from previous results which found that software implementations written by different programmers are unlikely to share the vast majority of implementation errors in code [30]. This result can be clearly seen in two popular open-source router software packages: Quagga and XORP differ in terms of update processing (timer-driven vs. event-driven), programming language (C vs. C++), and configuration language, leading to different sorts of bugs, which are triggered on differing inputs. As such, code-base diversity is very effective and requires only three instances to be run concurrently.

However, effectively evaluating this is challenging, as bug reports typically do not contain information about whether inputs triggering the bug would cause other code bases to fail. Hence we only performed a simple sanity-check: we selected 9 bugs from the XORP Bugzilla database, determined the router inputs which triggered the bug, verified that the bug occurred in the appropriate branch of XORP code, and then replayed the same inputs to Quagga to see if it would simultaneously fail. We then repeated this process to see if Quagga's bugs existed in XORP. In this small check, we did not find any cases where a bug in one code base existed in the other, mirroring the previous findings.

Version diversity : Another source of diversity lies in the different versions of the same router software itself. One main reason for releasing a new version of software is to fix bugs. Unfortunately, operators are hesitant to upgrade to the latest version until it has been well tested, as it is unknown whether their particular configuration, which has worked so far (possibly by chance), will work in the latest version. This hesitation comes with good reason, as often times when fixing bugs or adding features, new bugs are introduced into code that was previously working (i.e., not just in new features). This can be seen in some of the example bugs described in Table 1. With our bug-tolerant router, we can capitalize on this diversity.

For router vendors that fully rely on open-source software, version diversity will add little over the effectiveness of code-base diversity (assuming they use routers from three code bases). Instead, version diversity makes the most sense for router vendors that do not fully utilize code-base diversity. In this case, running the old version in parallel is protection against any newly introduced bugs, while still being able to take advantage of the bug fixes that were applied.

Evaluating this is also a challenge as bug reports rarely contain the necessary information. Because of this, to evaluate the fraction of bugs shared across versions (and thus, the effectiveness), we ran static analysis tools (splint, uno, and its4) over several versions of Quagga, and investigated overlap across versions. For each tool, we ran it against each of the earlier versions, and then manually checked to see how many bugs appear in both the earlier version as well as the most recent version. We found that overlap decreases quickly, with 30% of newly-introduced bugs in 0.99.9 avoided by using 0.99.1, and only 25% of bugs shared across the two versions. As it is not 100% effective, this will most likely be used in combination with other forms of diversity (*e.g.,* diversity in the execution environment, described next).

2.2 Execution environment diversity

Data diversity through manipulation of the execution environment has been shown to automatically recover from a wide variety of faults [12]. In addition, routing software specific techniques exist, two of which are discussed below. As closed-source vendors do not get the full benefit from running from multiple code bases, they will need to rely on data diversity, most likely as a complement to version diversity. In that case, around five instances will be needed depending on the amount of difference between the different versions. This comes from the result of our study which showed version diversity to be 75% effective, so we assume that two versions will be run, each with two or three instances of that version (each diversified in terms of execution environment, which as we discuss below can be fairly effective).

Update timing diversity: Router code is heavily concurrent, with multiple threads of execution and multiple processes on a single router, as well as multiple routers simultaneously running, and hence it is not surprising that this creates the potential for concurrency problems. Luckily, we

[3]To compare with closed-source software, we also studied publicly available Cisco IOS bug reports, though since we do not have access to IOS source code we did not run our system on them.

Bug	Description	Effective Diversity
XORP 814	The asynchronous event handler did not fairly allocate its resources when processing events from the various file descriptors. Because of this, a single peer sending a long burst of updates could cause other sessions to time out due to missed keepalives.	Version (worked in 1.5, but not 1.6)
Quagga 370	The BGP default-originate command in the configuration file does not work properly, preventing some policies from being correctly realized.	Version (worked in 0.99.5, but not 0.99.7)
XORP 814	(See above)	Update (randomly delay delivery)
Quagga XX (see note)	A race condition exists such that when a prefix that is withdrawn and immediately re-advertised, the router only propagates to peers the withdraw message, and not the subsequent advertisement.	Update (randomly delay delivery)
XORP 31	a peer that initiates a TCP connection and then immediately disconnects causes the BGP process to stop listening for incoming connections.	Connection (can delay disconnect)
Quagga 418	Static routes that have an unreachable next hop are correctly considered inactive. However, the route remains inactive even when the address of network device is changed to something that would make the next hop reachable (e.g., a next hop of 10.0.0.1 and an device address that changed from 9.0.0.2/24 to 10.0.0.2/24).	Connection (can interpret change as reset as well)

Table 1: Example bugs and the diversity that can be used to avoid them. Note for the bug listed as Quagga XX, it was reported on the mailing list titled "quick route flap gets mistaken for duplicate, route is then ignored," but never filed in Bugzilla.

can take advantage of the asynchronous nature of the routing system to increase diversity, for example, by introducing delays to alter the timing/ordering of routing updates received at different instances without affecting the correctness of the router (preserving any ordering required by the dependencies created by the protocol, *e.g.,* announcements for the same prefix from a given peer router must be kept in order, but announcements from different peer routers can be processed in any order). We were able to avoid two of the example bugs described in Table 1 with a simple tool to introduce a randomized short delay (1-10ms) when delivering messages to the given instance. Further, by manually examining the bug databases, we found that approximately 39% of bugs could be avoided by manipulating the timing/ordering of routing updates.

Connection diversity: Many bugs are triggered by changes to the router's network interfaces and routing sessions with neighbors. From this, we can see that another source of diversity involves manipulating the timing/order of events that occur from changes in the state or properties of the links/interfaces or routing session. As our architecture (discussed in Section 3) introduces a layer between the router software and the sessions to the peer routers, we can modify the timing and ordering of connection arrivals or status changes in network interfaces. For the two example bugs in Table 1, we found they could be avoided by simple forms of connection diversity, by randomly delaying and restarting connections for certain instances. By manually examining the bug database, we found that approximately 12% of bugs could be avoided with this type of diversity.

2.3 Protocol diversity

As network operators have the power to perform configuration modifications, something the router vendors have limited ability to do, there are additional forms of diversity that they can make use of. Here, we discuss one in particular. The process of routing can be accomplished by a variety of different techniques, leading to multiple different routing *protocols* and algorithms, including IS-IS, OSPF, RIP, etc. While these implementations differ in terms of the precise mechanisms they use to compute routes, they all perform a functionally-equivalent procedure of determining a FIB that can be used to forward packets along a shortest path to a destination. Hence router vendors may run multiple different routing protocols in parallel, voting on their outputs as they reach the FIB. To get some rough sense of this approach, we manually checked bugs in the Quagga and XORP Bugzilla databases to determine the fraction that resided in code that was shared between protocols (e.g., the *zebra* daemon in Quagga), or code that was protocol independent. From our analysis, we estimate that at least 60% of bugs could be avoided by switching to a different protocol.

3. BUG TOLERANT ROUTER (BTR)

Our design works by running multiple diverse router instances in parallel. To do this, we need some way of allowing multiple router software instances to simultaneously execute on the same router hardware. This problem has been widely studied in the context of operating systems, through the use of *virtual machine* (VM) technologies, which provide isolation and arbitrate sharing of the underlying physical machine resources. However, our design must deal with two new key challenges: (i) replication should be transparent and hidden from network operators and neighboring routers (Section 3.1), and (ii) reaching consensus must handle the transient behavior of routing protocols, yet must happen quickly enough to avoid slowing reaction to failures (Section 3.2).

3.1 Making replication transparent

First, our design should hide replication from neighboring routers. This is necessary to ensure deployability (to maintain sessions with legacy routers), efficiency (to avoid requiring multiple sessions and streams of updates between

peers), and ease of maintenance (to avoid the need for operators to perform additional configuration work). To achieve this, our design consists of a *router hypervisor*, as shown in Figure 1. The router hypervisor performs four key functions:

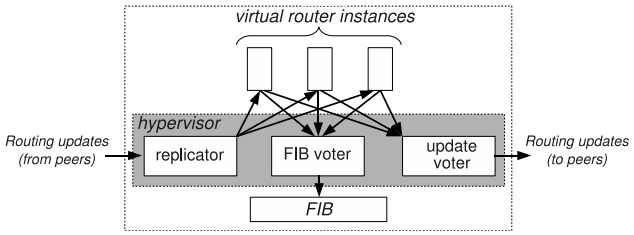

Figure 1: Architecture of a bug-tolerant router.

Sharing network state amongst replicas: Traditional routing software receives routing updates from neighbors, and uses information contained within those updates to select and compute paths to destinations. In our design, multiple instances of router software run in parallel, and somehow all these multiple router instances need to learn about routes advertised by neighbors. To compute routes, each internal instance needs to be aware of routing information received on peering sessions. However, this must happen without having instances directly maintain sessions with neighboring routers. To achieve this, we use a *replicator* component, which acts as a replica coordinator to send a copy of all received data on the session to each router instance within the system. Note that there may be multiple sessions with a given peer router (*e.g.,* in the case of protocol diversity), in which case the replicator sends received data to the appropriate subset of instances (*e.g.,* those running the same protocol). The replicator does *not* need to parse update messages, as it simply forwards all data it receives at the transport layer to each instance.

Advertising a single route per prefix: To protect against buggy results, which may allow the router to keep running but may cause it to output an incorrect route, we should select the majority result when deciding what information to publish to the FIB, or to advertise to neighbors. To do this, we run a *voter* module that monitors advertisements from the router instances, and determines the route the router should use (e.g., the majority result).[4] Our design contains two instances of the voter: an *update voter* that determines which routing updates should be sent to neighbors, and a *FIB voter* that determines which updates should be sent to the router's FIB (forwarding table). As with the replicator, the update voter may vote among a subset of instances, for example, those belonging to the same protocol. The FIB voter will vote among all instances, as all instances must come to the same decisions with regard to the FIB. To ensure advertisements are consistent with FIB contents, the update voter and FIB voter must select the same routes. To handle this, the same voting algorithm must be used on both updates and FIB changes.

To avoid introducing bugs, the voter should be as simple as possible (our voter implementation, containing multiple alternative voting strategies, is 514 lines of code). We assume the voter is trusted (since it is much simpler than router code, we expect it to have significantly fewer bugs and therefore the fact that it is a single point-of-failure is only a slight concern), and that replication is asynchronous (we do not assume all instances respond equally fast, as instances may be slow or mute due to bugs), and transparent (external routers do not interact directly with the multiple instances, so as to simplify deployment).

Maintaining a set of running replicas: BFT-based techniques rely on having a sufficient number of correctly-behaving replicas in order to achieve consensus. Hence, if an instance crashes or begins producing buggy output, we may wish to replace it with a new copy. To achieve this, our hypervisor is responsible for *bootstrapping* the new instance when it begins running. For traditional routers, bootstrapping involves establishing a session with a neighboring router, which causes the neighboring router to send out update messages for each of the prefixes it has an entry for in its RIB. To avoid introducing externally visible churn, the hypervisor keeps a history of the last update peers have sent for each prefix, and replays this for any new instance upon startup of that instance.

Presenting a common configuration interface: As there is no standardization of the configuration interface in routers, each router has ended up with its own interface. In the case where instances from different code bases are used, to keep the network operator from needing to configure each instance separately, a mechanism is needed to hide the differences in each configuration interface. Fortunately, this is not unlike today's situation where ISPs use routers from multiple vendors. To cope with this, ISPs often run configuration management tools which automate the process of targeting each interface with a common one. As such, we can rely on these same techniques to hide the configuration differences.

3.2 Dealing with the transient and real-time nature of routers

The voter's job is to arbitrate amongst the "outputs" (modifications to the FIB, outbound updates sent to neighbors) of individual router instances. This is more complex than simply selecting the majority result – during convergence, the different instances may temporarily have different outputs without violating correctness. At the same time, routers must react quickly enough to avoid slowing convergence. Here, we investigate several alternative voting strategies to address this problem, along with their tradeoffs.

Handling transience with *wait-for-consensus*: The extreme size of the Internet, coupled with the fact that routing events are propagated globally and individual events trigger multiple routing updates, results in very high update rates at routers. With the use of replication, this problem is potentially worsened, as different instances may respond at different times, and during convergence they may temporarily (and legitimately) produce different outputs. To deal with this, we use *wait-for-consensus* voting, in which the voter waits for all instances to compute their results before determining the majority vote. Because all non-buggy routers output the same correct result in steady-state, this approach can guarantee that if k or fewer instances are faulty with at least $2k + 1$ instances running, no buggy result will reach the FIB or be propagated to a peer.

Note that in practice, waiting for consensus may also re-

[4]Since voting also reveals the set of misbehaving instances, our approach also simplifies diagnosis, as the hypervisor can explicitly report the set of buggy outputs it observes.

duce instability, as it has an effect similar to the MRAI (Minimum Route Advertisement Interval) timer (routers with MRAI send updates to their neighbors only when a timer expires, which eliminate multiple updates to a prefix that occur between timer expiries). Namely, forcing the voter to wait for all instances to agree eliminates the need to advertise changes that happen multiple times while it is waiting (e.g., in the presence of unstable prefixes). However, the downside of this is that reaction to events may be slowed in some cases, as the voter must wait for the $k+1$th slowest instance to finish computing the result before making a decision.

Speeding reaction time with *master/slave*: Routers must react quickly to failures (including non-buggy events) to ensure fast convergence and avoid outages. At the same time, the effects of a bad FIB update or BGP message can be undone simply by overwriting the FIB or announcing a new route. To speed reaction time, we hence consider an approach where we allow outputs to temporarily be faulty. Here, we mark one instance as the *master*, and the other instances as slaves. The voter operates by always outputting the master's result. The slaves' results are used to cross-check against the master after the update is sent or during idle cycles. The benefit of this approach is that it speeds convergence to the running time of the master's computation. In addition, convergence is no worse than the convergence of the master, and hence at most one routing update is sent for each received update. However, the downside of this approach is that if the master becomes buggy, we may temporarily output an incorrect route. To address this, when failing over to a slave, the voter readvertises any differences between the slaves' routing tables and the routing table computed by the master. Hence, temporarily outputting an incorrect route may not be a problem, as it only leads to a transient problem that is fixed when the slaves overthrow the master.

Finally, we consider a hybrid scheme which we refer to as *continuous-majority*. This approach is similar to wait-for-consensus in that the majority result is selected to be used for advertisement or for population into the FIB. However, it is also similar to master/slave in that it does not wait for all instances to compute results before selecting the result. Instead, every time an instance sends an update, the voter reruns its voting procedure, and updates are only sent when the majority result changes. The benefit of this approach is it may speed reaction to failure, and the majority result may be reached before the slowest instance finishes computing. The downside of this approach is that convergence may be worsened, as the majority result may change several times for a single advertised update. Another downside of this approach is that voting needs to be performed more often, though, as we show in our experiments (Section 5) this overhead is negligible under typical workloads.

4. ROUTER HYPERVISOR PROTOTYPE

Our implementation had three key design goals: (i) not requiring modifications to routing software, (ii) being able to automatically detect and recover from faults, and (iii) low complexity, to not be a source of new bugs. Most of our design is agnostic to the particular routing protocol being used. For locations where protocol-specific logic was needed, we were able to treat messages mostly as opaque strings. This section describes our implementation, which consists of a set of extensions built on top of Linux. Our implementation was tested with XORP versions 1.5 and 1.6, Quagga versions 0.98.6 and 0.99.10, and BIRD version 1.0.14. We focused our efforts on supporting BGP, due to its complexity and propensity for bugs. Section 4.1 describes how we provide a *wrapper* around the routing software, in order for unmodified routing software to be used, and Section 4.2 describes the various faults that can occur and how our prototype detects and recovers from them.

4.1 Wrapping the routing software

To eliminate the need to modify existing router software, our hypervisor acts as a wrapper to hide from the routing software the fact that it is a part of a bug-tolerant router, and allows the routing instances to share resources such as ports, and access to the FIB. Our design (Figure 2) takes advantage of the fact that sockets are used for communicating with peer routers, and for communicating forwarding table (FIB) updates to the kernel. Hence, our implementation intercepts socket calls from the router instances using the LD_PRELOAD environment variable and uses a modified libc library, called *hv-libc*, to redirect messages to a user-space module, called *virtd*, which manages all communication.

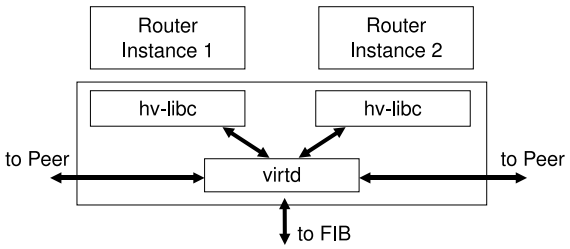

Figure 2: Implementation architecture.

The two key functions the hypervisor then needs to manage are discussed below:

Socket-based communications: To connect to peer routers (with TCP) and for writing to the common FIB (with Netlink), the multiple routers need to share access to a common identifier space (*e.g.,* port 179 in BGP). We handle this by intercepting socket system calls in hv-libc, performing address translation in hv-libc, and using virtd as a proxy (*e.g.,* when a router instance listens on port 179, instead they are made to listen on a random port and virtd will listen on 179 and connect to each of the random ports when receiving an incoming connection).

Bootstrapping new connections: When the BTR initially starts up, the routing instances start with empty routing tables. In BGP, a session with a peer is established by creating a TCP connection, exchanging OPEN messages, and acknowledging the OPEN message with a KEEPALIVE message. After the session is established, the peers exchange routing information. However, when replacing a failed instance, we need to bootstrap it locally, to prevent the failure from being externally visible (*e.g.,* sending a *route-refresh* to a peer). Additionally, we need to bootstrap it independently, to prevent the new instance starting in a faulty state (*e.g.,* bootstrapping off another router instance). Since a router's state only depends on the last received RIB advertised by its neighbors, we add some additional logic to the hypervisor

to store the last-received update for each (prefix,neighbor) pair. Then when a new instance is started, the hypervisor replays its stored updates. To lower complexity, the hypervisor treats the (prefix, neighbor) fields and other attributes in the packets as opaque strings, and does not implement protocol logic such as route selection.

4.2 Detecting and recovering from faults

To deal with bugs, our hypervisor must *detect* which outputs are buggy (e.g., with voting), and *recover* from the buggy output (by advertising the voting result, and if necessary restarting/replacing the buggy instance).

Detection: One of our main goals is that the BTR should be able to automatically detect and recover from bugs affecting correctness of the router's control or data planes.[5] Since our design fundamentally relies on detecting differences in *outputs* of different instances, we need to handle every possible way their outputs could differ. All faults can be generalized to four categories: (i) an instance sending a message when it should not, (ii) an instance not sending a message when it should, (iii) an instance sending a message with incorrect contents, and (iv) bugs that cause a detectable faulty system event, such as process crashing or socket error. The first three categories are detected by using voting (the fourth category is easily detectable, so no further discussion is given). If an instance has a different output from the majority, we consider it a fault. For example, in case (i) above, the winning update will be the NULL update, in cases (ii) and (iii) the winning update will be the most-commonly advertised one. To avoid reacting to transient changes, voting is only performed across *steady-state* instance outputs, which have been stable for a threshold period of time. We then mark instances whose steady-state outputs differ from those of the majority or those that are not yet stable as being faulty (including in schemes like master/slave, which perform this step after advertising).[6]

Recovery: In the common case, recovering from a buggy router simply involves using the output from the voting procedure. However, to deal with cases where the router is persistently buggy, or crashes, we need some way to kill and restart the router. As a heuristic, we modified our hypervisor with a *fault threshold* timeout. If an instance continues to produce buggy output for longer than the threshold, or if the router undergoes a faulty system event, the router is killed. To maintain a quorum of instances on which voting can be performed, the BTR can restart the failed instance, or replace it with an alternate diverse copy. In addition, to support the master/slave voting scheme, we need some way to overwrite previously-advertised buggy updates. To deal with this, our implementation maintains a history of previously-advertised updates when running this voting scheme. When the hypervisor switches to a new master, all updates in that history that differ from the currently advertised routes are sent out immediately.

4.3 Reducing complexity

It is worth discussing here the role the hypervisor plays in the overall reliability of the system. As we are adding software, this can increase the possibility of bugs in the overall system. In particular, our goals for the design are that (i) the design is *simple*, implementing only a minimal set of functionality, reducing the set of components that may contain bugs, and (ii) the design is *small*, opening the possibility of formal verification of the hypervisor – a more realistic task than verifying an entire routing software implementation. To achieve these goals, our design only requires the hypervisor to perform two functions: (i) acting as a TCP proxy, and (ii) bootstrapping new instances. Below, we described how these functions are performed with low complexity.

Acting as a TCP proxy: To act as a TCP proxy simply involves accepting connections from one end point (remote or local) and connecting to the other. When there is a TCP connection already, the hypervisor simply needs to accept the connection. Then, upon any exchange of messages (in or out) the hypervisor simply passes data from one port to another. In addition, our design uses voting to make replication transparent to neighboring routers. Here, the update messages are voted upon before being sent to the adjacent router. However, this is simply comparing opaque strings (the attributes) and does not involve understanding the values in the strings.

Overall, our implementation included multiple algorithms and still was only 514 lines of code. These code changes occur only in the hypervisor, reducing potential for new bugs by increasing modularity and reducing need to understand and work with existing router code. From this, we can see that the hypervisor design is simple in terms of functionality and much of the functionality is not in the critical section of code that will act as a single point of failure.

Bootstrapping new instances: To bootstrap new instances requires maintaining some additional state. However, bugs in this part of the code only affect the ability to bootstrap new instances, and do not affect the "critical path" of voting code. One can think of this code as a parallel routing instance which is used to initialize the state of a new instance. Of course, if this instance's RIB is faulty, the new instance will be started in an incorrect state. However, this faulty state would either be automatically corrected (*e.g.,* if the adjacent router sends a new route update that overwrites the local faulty copy) or it would be determined to be faulty (*e.g.,* when the faulty route is advertised), in which case a new instances is started. Additionally, the RIB that needs to be kept is simply a history of messages received from the adjacent router and therefore is simple. Bootstrapping a new instance also requires intercepting BGP session establishment. Here, the hypervisor simply needs to observe the first instance starting a session (an OPEN message followed by a KEEPALIVE) and subsequent instances simply get the two received messages replayed.

5. EVALUATION

We evaluate the three key assumptions in our work:

It is possible to perform voting in the presence of dynamic churn (Section 5.1): Voting is simple to do on fixed inputs, but Internet routes are transient by nature. To distinguish between instances that are still converging to the correct output from those that are sending buggy outputs, our system delays voting until routes become stable, introducing a tradeoff between false positives (incorrectly believing an unstable route is buggy) and detection time (during which

[5]We do not address, for example, faults in logging.

[6]We consider legitimate route-flapping due to persistent failures and protocol oscillations to be rare. However, we can detect this is occurring as the majority of instances will not be stable and we can act accordingly.

time a buggy route may be used). Since these factors are independent of the precise nature of bugs but depend on update dynamics, we inject synthetic faults, and replay real BGP routing traces.

It is possible for routers to handle the additional overhead of running multiple instances (Section 5.2): Internet routers face stringent performance requirements, and hence our design must have low processing overhead. We evaluate this by measuring the *pass-through time* for routing updates to reach the FIB or neighboring routers after traversing our system. To characterize performance under different operating conditions, we vary the routing update playback rate, the source of updates (edge vs. tier-1 ISP), and the number of peers.

Running multiple router replicas does not substantially worsen convergence (Section 5.3): Routing dynamics are highly dependent on the particular sequence of steps taken to arrive at the correct route – choosing the wrong sequence can vastly increase processing time and control overhead. To ensure our design does not harm convergence, we simulate update propagation in a network of BTRs, and measure convergence time and overhead. For completeness, we also cross-validate these against our implementation.

5.1 Voting in the presence of churn

To evaluate the ability to perform voting in the presence of routing churn, we replayed BGP routing updates collected from Route Views [6] against our implementation. In particular, we configure a BGP trace replayer to play back a 100 hour long trace starting on March 1st 2007 at 12:02am UTC. The replayer plays back multiple streams of updates, each from a single vantage point, and we collect information on the amount of time it takes the system to select a route. Since performance is dependent only on whether the bug is detected by voting or not, and independent of the particular characteristics of the bug being injected, here we use a simplified model of bugs (based on the model presented in Section 4.2), where bugs add/remove updates and change the next-hop attribute for a randomly-selected prefix, and have two parameters: (i) *duration*, or the length of time an instance's output for a particular prefix is buggy, (ii) *interarrival time*, or the length of time between buggy outputs. As a starting point for our baseline experiments, we assume the length of time a bug affects a router, and their interarrival times, are similar to traditional failures, with duration of 600 seconds, and interarrival time of 1.2 million seconds [34].

5.1.1 Comparison of voting strategies

There is a very wide space of voting strategies that could be used in our system. To explore tradeoffs in this space, we investigated performance under a variety of alternative voting strategies and parameter settings. We focus on several metrics: the *fault rate* (the fraction of time the voter output a buggy route), *waiting time* (the amount of time the voter waits before outputting the correct route) and *update overhead* (the number of updates the voter output).

Fault rate: We investigate the fault rate of the voting strategies by injecting synthetic faults and varying their properties. First, we varied the mean duration and interarrival times of synthetic faults (Figures 3 and 4). We found that for very high bug rates, wait-3 (waiting for $K = 3$ out of $R = 3$ copies to agree before selecting the majority result) outperformed master/slave. This happened because wait-3 is more robust to simultaneous bugs than master/slave, as master/slave takes some short time to detect the fault, potentially outputting an incorrect route in the meantime. In addition, unless the bug rate is extremely high, continuous-majority performs nearly as well as wait-3, with similar robustness and update overhead.

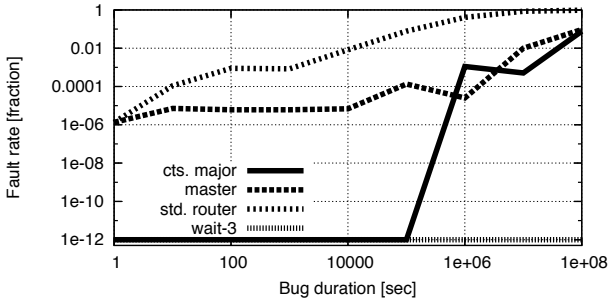

Figure 3: Effect of bug duration on fault rate, holding bug interarrival times fixed at 1.2 million seconds.

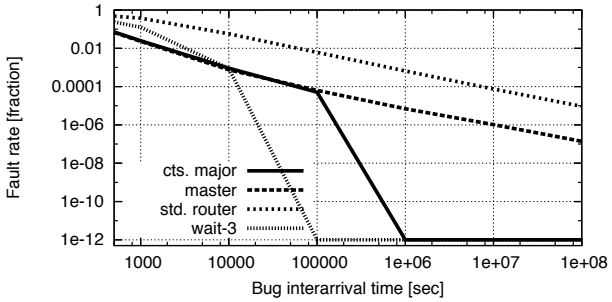

Figure 4: Effect of bug interval on fault rate, holding bug duration fixed at 600 seconds.

Overall, we found that recovery almost always took place within one second. Increasing the number of instances running in parallel (R) makes the router even more tolerant of faults, but incurs additional overheads. Also, wait-for-consensus and continuous-majority gain more from larger values of R than the master/slave strategy. For example, when moving from $R = 3$ to $R = 4$ instances, the fault rate decreases from 0.088% to 0.003% with wait-for-consensus, while with master/slave the fault rate only decreases from 0.089% to 0.06%.

However, there may be practical limits on the amount of diversity achievable (for example, if there is a limited number of diverse code instances, or a bound on the ability to randomize update timings). This leads to the question—if we have a fixed number of diverse instances, how many should be run, and how many should be kept as standbys (not running, but started up on demand)? We found that standby routers were less effective than increasing R, but only for small values of R, indicating that for large numbers of diverse instances, most instances could be set aside as standbys to decrease runtime overhead. For example, if $R = 3$, under the continuous-majority strategy we attain a fault rate of 0.02%. Increasing R to 4 reduced the fault rate to 0.0006%, while instead using a standby router with $R = 3$ reduced the fault rate to 0.0008%. This happens because buggy outputs are detected quickly enough that failing over to a standby is nearly as effective as having it participate

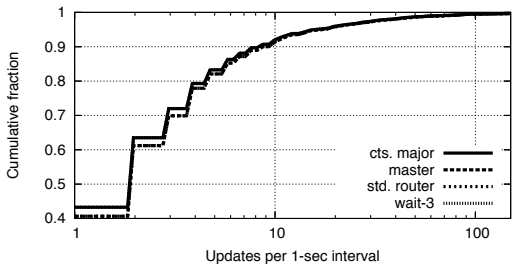

Figure 5: Effect of voting on update overhead.

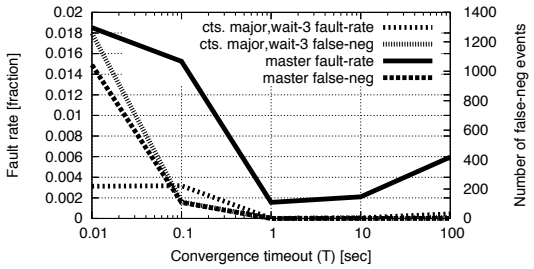

Figure 6: Effect of convergence time threshold.

in voting at every time step. Because of this, operators can achieve much of the benefits of a larger number of instances, even if these additional instances are run as lower-priority (e.g., only updated during idle periods) standbys.

Waiting time: Different voting algorithms provide different tradeoffs between waiting time (time from when a new best-route arrives, to when it is output by the voter) and the fault rate. The master/slave strategy provides the smallest waiting time (0.02 sec on average), but incurs a higher fault rate (0.0006% on average), as incorrect routes are advertised for a short period whenever the master becomes buggy. Continuous-majority has longer wait times (0.035 sec on average), but lower fault rate (less than 0.00001% on average), as routes are not output until multiple instances converge to the same result. The wait-for-consensus strategy's performance is a function of the parameter K—larger values of K increase wait time but decreases fault rate. However, we found that increasing K to moderate sizes incurred less delay than the pass-through time for a single instance, and hence setting $K = R$ offered a low fault rate with only minor increases in waiting time.

Update overhead: Finally, we compare the voting strategies in terms of their effect on update overhead (number of routing updates they generate), and compare them against a standard router (*std. router*). Intuitively, running multiple voters within a router might seem to increase update overhead, as the voter may change its result multiple times for a single routing update. However, in practice, we find no substantial increase, as shown in Figure 5, which plots a CDF of the number of updates (measured over one second intervals). For the master/slave strategy this is expected, since a single master almost always drives computation. In wait-for-consensus, no updates are generated until all instances arrive at an answer, and hence no more than one outbound update is generated per inbound update, as in a standard router. Interestingly, the continuous-majority strategy also does not significantly increase update overhead. This happens because when an update enters the system, the voter's output will only change when the majority result changes, which can only happen once per update.

5.1.2 Performance of fault detection

Protocols today often incorporate thresholds (such as BGP's MRAI timer) to rate-limit updates. To evaluate the level of protection our scheme provides against unstable instances, as well as the ability to distinguish steady-state from transient behavior, we incorporated a configurable timeout parameter (T) in fault detection to identify when a route becomes stable. Figure 6 shows the tradeoff as this parameter varies between the *false negative rate* (the number of times a non-buggy instance is treated as buggy), and the *fault rate* (i.e., the false positive rate of the voter, or the fraction of time a buggy route is treated as non-buggy). We found that as T increases, the false negative rate decreases, as larger values of T reduce the probability that transient changes will be considered when voting. The false negative rate does not vary among different voting strategies, as fault detection is only performed on steady-state outputs, and the algorithmic differences between the strategies disappear when performed on outputs that are not dynamically changing. The fault rate increases with T, as when a bug does occur, it takes longer to detect it. Interestingly, the fault rate initially decreases with T; this happens because for low values of T, more instances are treated as buggy, giving fewer inputs to the voter and increasing the probability of an incorrect decision. Overall, we found that it was possible to tune T to simultaneously achieve a low fault rate, low false negative, and low detection time.

5.2 Processing overhead

We evaluate the overhead of running multiple instances using our hypervisor with both XORP- and Quagga-based instances running on single-core 3 Ghz Intel Xeon machines with 2 GB RAM. We measure the *update pass-through time* as the amount of time from when the BGP replayer sends a routing update to when a resulting routing update is received at the monitor. However, some updates may not trigger routing updates to be sent to neighbors, if the router decides to continue using the same route. To deal with this case, we instrument the software router's source code to determine the point in time when it decides to retain the same route. We also instrument the kernel to measure the *FIB pass-through time*, as the amount of time from when the BGP replayer sends an update to the time the new route is reflected in the router's FIB (which is stored as the routing table in the Linux kernel).

Figure 7 shows the *pass-through* time required for a routing change to reach the FIB. We replayed a Routeviews update trace and varied the number of Quagga instances from 1 to 31, running atop our router hypervisor on a single-core machine. We found the router hypervisor increases FIB pass-through time by 0.08% on average, to 0.06 seconds. Our router hypervisor implementation runs in user space, instead of directly in the kernel, and with a kernel-based implementation this overhead would be further reduced. Increasing the number of instances to 3 incurred an additional 1.7% increase, and to 5 incurred a 4.6% increase. This happens because the multiple instances contend for CPU resources (we found that with multicore CPUs this overhead was substantially lower under heavy loads). To

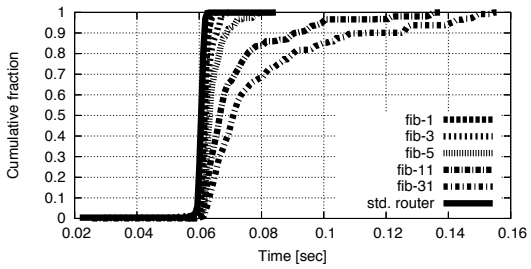

Figure 7: BTR pass-through time.

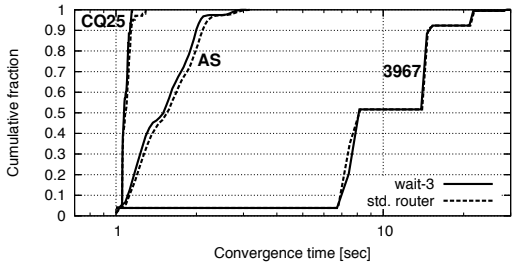

Figure 8: Network-wide simulations, per-router convergence delay distribution.

evaluate performance under heavier loads, we increased the rate at which the replayer played back routing updates by a factor of 3000x. Under this heavy load, FIB pass-through times slow for both the standard router and BTR due to increased queuing delays. However, even under these heavy loads, the BTR incurs a delay penalty of less than 23%. To estimate effects on convergence, we also measured the *update pass-through time* as the time required for a received routing change to be sent to neighboring routers. We found this time to be nearly identical to the FIB pass-through time when the MRAI timer was disabled. as updates are sent immediately after updating the FIB. When MRAI was enabled (even when set to 1 second, the lowest possible setting for Quagga), the variation in delay across instances was dwarfed by delay incurred by MRAI. Finally, we found that switching to the master/slave voting strategy reduces pass-through delay, though it slightly increases the fault rate, as discussed previously in Section 5.1.

5.3 Effect on convergence

Next, we study the effect of our design on network-wide convergence. We do this by simulating a network of BTRs (each with eight virtual router instances) across three network-level graphs: the entire AS-level topology (labeled *AS* in Figure 8) sampled on Jan 20 2008, AS 3967's internal network topology as collected from Rocketfuel (labeled *3967*), and cliques (labeled *CQ*) of varying sizes (since a clique contains the "worst case" for routing, allowing potential to explore all $n!$ possible paths in a clique of size n). To determine ordering of when BTRs respond, we run our implementation over routing updates, record pass-through times, and replay them within our simulation framework. Since for the master/slave approach there is no effect on network operation unless a bug is triggered (since the slaves only operate as standbys), we focus our evaluation on the other strategies.

We found several key results. First, as shown in Figure 8, the voting schemes do not produce any significant change in convergence beyond the delay penalty described in previous sections, as compared to a network only containing standard routers. We found this delay penalty to be much smaller than propagation delays across the network, and to be reduced further when MRAI is activated. As the number of instances increases (up to the number of processor cores), continuous-majority's delay decreases, because it becomes increasingly likely that one will finish early. The opposite is true for wait-for-consensus, as the delay of the slowest instances becomes increasingly large. Next, while we have thus far considered a *virtual router* level deployment, where voting is performed at each router, we also considered a *virtual network* deployment, where voting is performed at the edges of the network. In our experiments we ran eight virtual networks and found that this speeds up convergence, as routers do not have to wait for multiple instances to complete processing before forwarding updates. Hence, for small numbers of diverse instances, voting per-router has smaller convergence delay. However, virtual-network approaches require substantially more control overhead than the virtual-router voting schemes. To address this, we found that simple compression schemes [11] that eliminate redundancy across updates could reduce the vast majority of this overhead. Finally, to validate our simulations, we set up small topologies on Emulab [2], injected routing events, and compared with simulations of the same topology. We found no statistically significant difference.

6. DISCUSSION

For simplicity, this paper discusses the one particular design point. However, our architecture is amenable to deployment on varying levels of granularity:

Server-based operation: Instead of running the diverse instances within a single router, their computations may be offloaded to a set of dedicated servers running in the network (*e.g.,* an RCP-like platform [15]). These servers run the router software in virtualized environments, and cross-check the results of routers running within the network. When a buggy result is detected, virtual router instances may be migrated into the network to replace the buggy instance. Alternatively, the servers may be configured to operate in *read-only mode*, such that they may signal alarms to network operators, rather than participate directly in routing.

Network-wide deployment: Instead of running instances of individual router software in parallel, ensembles of routers may collectively run entire virtual networks in parallel. Here, the outputs of a router are not merged into a single FIB, or as a single stream of updates sent to its neighbors. Instead, each router maintains a separate FIB for each virtual network, and voting is used at border routers to determine which virtual network data packets should be sent on. The advantage of this approach is it allows different routing protocols to be used within each virtual network, making it simpler to achieve diversity. For example, OSPF may be run in one network and IS-IS in another. In addition, convergence speed may be improved, as individual physical routers do not have to wait for their instances to reach a majority before sending a routing update.

Process-level deployment: Our design runs multiple instances of routing software in parallel, and hence incurs some memory overhead. On many Internet routers this is not an issue, due to low DRAM costs, and the fact that

DRAM capacity growth has far exceeded that of routing table growth. That said, if it is still desirable to decrease memory usage, router software may be modified to vote on a shared RIB instead of a FIB. We found the RIB is by far the largest source of memory usage in both Quagga and XORP, incurring 99.3% of total memory usage. Voting on a shared RIB would reduce this overhead by eliminating the need to store separate copies of the RIB across router instances. Here, voting could be performed across multiple routing daemons (e.g., multiple BGP processes within a single instance of Cisco IOS) to construct a single shared RIB. In addition to reducing memory usage, finer-grained diversity may speed reaction (by only cloning and restarting individual processes or threads), and finer-grained control (during times of load, only mission-critical components may be cloned to reduce resource usage). However, code development may become more challenging, since this approach relies on knowing which parts of code are functionally equivalent. To address this, router software could be written to a common API, to allow replication and composition of modules from different code bases while sharing state.

Leveraging existing redundancy: Instead of running multiple instances in parallel, a router may be able to leverage redundant executions taking place at other routers in the network. For example, networks often provision redundant network equipment to protect against physical failures. For example, the VRRP [27] protocol allows multiple routers to act collectively as a single router. Our architecture is amenable to leveraging physical redundancy, as the multiple instances may be deployed across the redundant router instances. In addition, all routers in the ISP compute the same *egress set* of BGP routes that are "equal" according to the first few steps of the decision process that deal with BGP attributes [24, 15]. To leverage this redundancy, it may be possible to extend our architecture to support voting across multiple router's egress sets.

7. RELATED WORK

Software and data diversity has been widely applied in other areas of computing, including increasing server reliability [18], improving resilience to worm propagation [36], building survivable Internet services [28], making systems secure against vulnerabilities [20], building survivable overlay networks [44], building fault tolerant networked file systems [17], protecting private information [43], and recovering from memory errors [12]. Techniques have also been developed to minimize computational overhead by eliminating redundant executions and redundant memory usage across parallel instances [45, 26].

However as discussed in Section 1.3, routing software presents new challenges for SDD (e.g., routers must react quickly to network changes, have vast configuration spaces and execution paths, rely on distributed operations), as well as new opportunities to customize SDD (routers have small dependence on past history, can achieve the same objectives in different ways, have well-defined interfaces). We address these challenges and opportunities in our design. There has also been work studying router bugs and their effects [42, 32], and our design is inspired by these measurement studies. Also, [14] used a graph-theoretic treatment to study the potential benefits of diversity across physical routers (as opposed to diversity within a router). As work dealing with misconfigurations [23, 24] and traditional fail-stop failures [10, 35, 33, 27] becomes deployed we envision router bugs will make up an increasingly significant roadblock in improving network availability.

Our work can be contrasted to techniques which attempt to prevent bugs by formally verifying the code. These techniques are typically limited to small codebases, and often require manual efforts to create models of program behavior. For example, with manual intervention, a small operating system kernel was formally verified [29]. For routing, work has been done on languages to model protocol behavior (*e.g.,* [25]), however the focus of this work is on algorithmic behaviors of the protocol, as opposed to other possible places where a bug can be introduced. In contrast, our approach leverages a small and low-complexity hypervisor, which we envision being possible to formally verify.

Our design leverages router virtualization to maintain multiple diverse instances. Router virtualization is an emerging trend gaining increased attention, as well as support in commercial routers. Our design builds on the high-level ideas outlined in [16] by providing a complete design, several algorithms for detecting and recovering from bugs, and an implementation and evaluation. In addition, our design is complementary to use of models of router behavior [23, 24] and control-plane consistency checks [41, 37], as these models/checks can be run in place of one or more of the router virtual instances. Finally, systems such as MARE (Multiple Almost-Redundant Executions) [45] and the Difference Engine [26] focus on reducing overheads from replication. MARE runs a single instruction stream most of the time, and only runs redundant instruction streams when necessary. The Difference Engine attains substantial savings in memory usage across VMs, through use of sub-page level sharing and in-core memory compression. These techniques may be used to further reduce overheads of our design.

8. CONCLUSIONS

Implementation errors in routing software harm availability, security, and correctness of network operation. In this paper, we described how to improve resilience of networks to bugs by applying Software and Data Diversity (SDD) techniques to router design. Although these techniques have been widely used in other areas of computing, applying them to routing introduces new challenges and opportunities, which we address in our design. This paper takes an important first step towards addressing these problems by demonstrating diverse replication is both viable and effective in building robust Internet routers. An implementation of our design shows improved robustness to router bugs with some tolerable additional delay.

9. REFERENCES

[1] Cisco 7200 simulator. (software to run Cisco IOS images on desktop PCs) `www.ipflow.utc.fr/index.php/Cisco_7200_Simulator`.

[2] Emulab.net. `www.emulab.net`.

[3] Olive. (software to run Juniper OS images on desktop PCs) `juniper.cluepon.net/index.php/Olive`.

[4] OpenVZ. `http://www.openvz.org`.

[5] Quagga software routing suite. `www.quagga.net`.

[6] Route views project. `www.routeviews.org`.

[7] Vyatta (open-source router vendor). `www.vyatta.com`.

[8] Xorp, inc. `www.xorp.net`.

[9] S. Agarwal, C. Chuah, S. Bhattacharyya, and C. Diot. Impact of BGP dynamics on router CPU utilization. In *Passive and Active Measurement*, April 2004.

[10] C. Alaettinoglu, V. Jacobson, and H. Yu. Towards millisecond IGP convergence. In *IETF Draft*, November 2000.

[11] R. Alimi, Y. Wang, and Y. R. Yang. Shadow configuration as a network management primitive. In *SIGCOMM*, August 2008.

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

[13] B. Brenner. Cisco IOS flaw prompts symantec to raise threat level. In *Information Security Magazine*, Sept. 2005.

[14] J. Caballero, T. Kampouris, D. Song, and J. Wang. Would diversity really increase the robustness of the routing infrastructure against software defects? In *NDSS*, Feb. 2008.

[15] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and K. van der Merwe. Design and implementation of a routing control platform. In *NSDI*, April 2005.

[16] M. Caesar and J. Rexford. Building bug-tolerant routers with virtualization. In *PRESTO*, August 2008.

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

[18] B.-G. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX Annual Technical Conference*, June 2008.

[19] Cisco ASR 1000 series aggregation services router high availability: Delivering carrier-class services to midrange router. `http://www.cisco.com/en/US/prod/collateral/routers/ps9343/solution_over%view_c22-450809_ps9343_Product_Solution_Overview.html`.

[20] B. Cox, D. Evans, A. Filip, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *Usenix Security*, August 2006.

[21] J. Duffy. BGP bug bites Juniper software. In *Network World*, December 2007.

[22] J. Evers. Trio of Cisco flaws may threaten networks. In *CNET News*, January 2007.

[23] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, May 2005.

[24] N. Feamster and J. Rexford. Network-wide prediction of BGP routes. In *IEEE/ACM Trans. Networking*, April 2007.

[25] T. G. Griffin and J. L. Sobrinho. Metarouting. In *SIGCOMM*, August 2005.

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

[27] R. Hinden. Virtual router redundancy protocol (VRRP). RFC 3768, April 2004.

[28] F. Junqueira, R. Bhgwan, A. Hevia, K. Marzullo, and G. Voelker. Surviving Internet catastrophes. In *HotOS*, May 2003.

[29] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[30] J. Knight and N. Leveson. A reply to the criticisms of the Knight & Leveson experiment. *ACM SIGSOFT Software Engineering Notes*, January 1990.

[31] W. Knight. Router bug threatens 'Internet backbone'. In *New Scientist Magazine*, July 2003.

[32] A. Kuatse, R. Teixeira, and M. Meulle. Characterizing network events and their impact on routing. In *CoNEXT (Student Poster)*, December 2007.

[33] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica. Achieving convergence-free routing using failure-carrying packets. In *SIGCOMM*, August 2007.

[34] A. Markopoulou, G. Iannaconne, S. Bhattacharrya, C.-N. Chuah, and C. Diot. Characterization of failures in an IP backbone. In *IEEE/ACM Trans. Networking*, Oct. 2008.

[35] M. Motiwala, M. Elmore, N. Feamster, and S. Vempala. Path splicing. In *SIGCOMM*, 2008.

[36] A. O'Donnell and H. Sethu. On achieving software diversity for improved network security using distributed coloring algorithms. In *ACM CCS*, October 2004.

[37] R. Rajendran, V. Misra, and D. Rubenstein. Theoretical bounds on control-plane self-monitoring in routing protocols. In *SIGMETRICS*, June 2007.

[38] M. Reardon. Cisco offers justification for procket deal. June 2004. `http://news.cnet.com/Cisco-offers-justification-for-Procket-deal/2100-1%033_3-5237818.html`.

[39] Renesys. AfNOG takes byte out of Internet. `http://www.renesys.com/blog/2009/05/byte-me.shtml`.

[40] Renesys. Longer is not always better. `http://www.renesys.com/blog/2009/02/longer-is-not-better.shtml`.

[41] L. Wang, D. Massey, K. Patel, and L. Zhang. FRTR: A scalable mechanism to restore routing table consistency. In *DSN*, June 2004.

[42] Z. Yin, M. Caesar, and Y. Zhou. Towards understanding bugs in router software. In *Technical report, UIUC*, January 2009. `www.cs.uiuc.edu/homes/caesar/bugs.pdf`.

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

[44] Y. Zhang, S. Dao, H. Vin, L. Alvisi, and W. Lee. Heterogeneous networking: A new survivability paradigm. In *New Security Paradigms Workshop*, September 2008.

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