Design Guidelines for Robust Internet Protocols

Tom Anderson* Scott Shenker† Ion Stoica‡ David Wetherall*

Abstract

Robustness has long been a central design goal of the Internet. Much of the initial effort towards robustness focused on the "fail-stop" model, where node failures are complete and easily detectable by other nodes. The Internet is quite robust against such failures, routinely surviving various catastrophes with only limited outages. This robustness is largely due to the widespread belief in a set of guidelines for critical design decisions such as where to initiate recovery and how to maintain state.

However, the Internet remains extremely vulnerable to more arbitrary failures where, through either error or malice, a node issues syntactically correct responses that are not semantically correct. Such failures, some as simple as misconfigured routing state, can seriously undermine the functioning of the Internet. With the Internet playing such a central role in the global telecommunications infrastructure, this level of vulnerability is no longer acceptable.

In this paper we propose a set of six design guidelines for improving the robustness of network protocols to these kinds of arbitrary failures. These guidelines emerged from a study of past examples of failures, and determining what could have been done to prevent the problem from occurring in the first place. The unifying theme behind the various guidelines is that we need to design protocols more defensively, expecting malicious attack, misimplementation, and misconfiguration at every turn.

1 Introduction

Robustness has been, from the very beginning, one of the central design goals of the Internet [8]. As a result, the Internet architecture has been highly resilient to various kinds of catastrophes; the Internet has survived hurricanes, earthquakes, tunnel fires, and terrorist attacks with only temporary and partial loss of end to end connectivity [20]. To a large extent, this success story can be attributed to Internet protocol designers following a set of guidelines for robust design: e.g., end hosts should be responsible for error recovery, failures should be assumed to be the common case, and all critical state should be refreshed periodically.

However, the vaunted robustness of the Internet does not apply to all varieties of failures. An implicit but fundamental assumption underlying the early notion of robustness, and hence the design of many Internet protocols, is that systems are "fail-stop"—when systems fail they completely and detectably stop working. Thus, Internet designers have long assumed that nodes can recognize that another node has failed by either absent or syntactically incorrect (malformed) protocol responses.

Unfortunately, not all failures are so cleanly defined and easily detected; either by malicious intent or by accident, systems obeying the syntax of a protocol may in fact be behaving incorrectly. These more *arbitrary* failures [9] occur with surprising regularity, in part because the increasing scale and heterogeneity of the Internet makes subtle inconsistencies more likely. Many different protocol implementations, operational practices, and user motivations must somehow safely coexist for the Internet to be highly robust.

Moreover, these arbitrary failures can often cascade through the Internet with serious consequences to parties not at fault. In one recent example, widespread outages were triggered because one vendor's BGP implementation ignores but propagates incorrect route announcements, while another vendor's routers terminate any BGP session propagating an obviously incorrect announcement [14]. To put this incident in perspective, it arguably caused a more serious Internet outage than the September 11 terrorist attacks in which a large amount of connectivity through New York City was severed by the collapse of the World Trade Center.

We proceed by considering some past examples where protocols failed to be robust to arbitrary failures, and attempt to draw lessons from those examples as to how we should better design protocols in the future. Fortunately, some of these lessons have already begun to be applied in an ad hoc manner to individual protocols; we argue that we should be applying them more systematically to the design of all protocols that need to survive arbitrary failures.

Our overarching recommendation is that we should design protocols *defensively*, keeping in mind that other nodes are not to be trusted. We identify six guidelines within this over-

*CSE Department, University of Washington, e-mails:{tom,djw}@cs.washington.edu. This work was supported in part by DARPA Contract Number F30602-00-2-0565.

†ICSI Center for Internet Research, e-mail: shenker@icsi.berkeley.edu.

‡EECS Department, Univeristy of California, Berkeley, e-mail:istoica@cs.berkeley.edu. This work was supported in part by NSF Contract Number NSF-ITR 0085879.

all theme, and demonstrate by example that robustness could have been improved had those guidelines been applied systematically in the past. Of course, the need for defensive design is not unique to networks—software engineers apply defensive programming to reduce bug rates [15], and operating systems apply defensive programming to isolate errors to individual applications. However networks do pose some significant challenges not found in other disciplines. For example, network protocols often survive for decades, they are typically implemented by numerous separate organizations, and often must work across organizational boundaries.

2 Related Approaches

There already exist some approaches that help protect against arbitrary failures. This is because robustness in its many guises has been a longstanding design goal for distributed systems. In particular, cryptographic authentication, fault-tolerance via consensus, and formal (and informal) protocol specification are extremely useful, and should be used whenever appropriate. However, in this section we argue that by themselves they do not suffice. We do so by exploring a motivating example, an early ARPANET routing incident. We first present the failure and then consider the utility of each of the above approaches in preventing it.

Routing in the early ARPANET was based on a link-state design. In link-state protocols, the advertisements sent by a node carry increasing sequence numbers that are used to distinguish a new advertisement from old ones; routers forward new advertisements to all of their neighbors, and silently discard old ones. In the ARPANET scheme, a "modulo" sequence number space was used to avoid the problem of becoming stuck when the maximum sequence number was reached. That is, the sequence number counter simply wrapped, and the test to determine whether one sequence number was larger or smaller than another was whether it was a shorter distance forward or backward in sequence number space, respectively.

This scheme worked well when all the nodes were functioning correctly. However, a network-wide storm was caused by a malfunctioning router that injected a series of messages with incorrect sequence numbers before it crashed. The faulty messages had sequence numbers that formed a cycle of progressively "larger" advertisements. That is, the sequence numbers increased by jumps, wrapping as necessary, with each growing "larger", yet the first in the sequence was "larger" than the last. This led to an infinite sequence of updates, forwarded in a cycle to all nodes in the network, without any further input on behalf of the malfunctioning node. To address this problem, the network had to be entirely purged of the faulty messages—not a pleasant thought were a similar self-sustaining chain of events to occur in the Internet today. This is precisely the kind of unanticipated critical failure that we would like to preclude by more robust design.

How could this failure have been prevented? Clearly, cryptographic security mechanisms would have been of no help in this incident. In general, encryption-based authentication is extremely important for helping prevent malicious attackers from hijacking protocol communication. Passwords and encryption can be used to validate that only authorized users or machines are allowed to participate in a protocol, eliminating a whole class of potential problems. By itself, however, strong authentication is not sufficient for completely eliminating arbitrary failures. Authentication can demonstrate that the speaker is who she says she is, but it cannot validate the semantic content of the protocol information—the speaker could have been compromised or could be simply behaving incorrectly. In the ARPANET case, authentication would not have prevented the malfunctioning router from sending (authentic) messages with improper sequence numbers.

Approaches for achieving fault tolerance via consensus are also not a complete solution. There is a vast literature on the theory of algorithms for achieving Byzantine agreement [13] as well as recent work on more practical systems [5]. However, these solutions typically depend on the ability to replicate a computation and then run an agreement algorithm to determine the correct answer. In some cases, it can make sense to replicate state and computation, and hence apply consensus techniques. However, in many protocols it does not appear feasible to do so while preserving efficiency. For example, the computation that causes a node to issue new routing advertisements depends on the state of the attached links, state which is available only localy.

Finally, we consider whether better or more precise specifications could have helped with the ARPANET incident. Unfortunately, the fact is that there is nothing wrong with the protocol when it is implemented and operated as specified—it does converge to the available routes as expected. It was only when a malfunctioning node injected corrupt data that an unanticipated, cascading error was induced. Our point is that despite being correct, the protocol is fragile or vulnerable to arbitrary failures, which makes it an undesirable design. Thus, even if we posit better tools for checking the correctness of implementations before they are applied to live systems [3], we also need to check whether the protocol itself is vulnerable if some nodes are misimplemented or malicious. Formal methods that automatically check specifications for vulnerabilities— while a promising avenue for future research—are not yet in widespread use beyond security protocols [4].

As an aside, the actual solution adopted in the ARPANET routing protocol was to revise the design to remove modulo sequence numbers; modern routing protocols such as OSPF and IS-IS follow this lead. Linear sequence numbers were used instead, with a separate timeout-based mechanism (that is, aging) for resetting when the maximum sequence number is reached. With this design, no advertisement can be propagated unless it was recently injected by some router.

In sum, the traditional robustness techniques of authentication, consensus, and specification are invaluable in modern distributed systems, and Internet protocols should incorporate them whenever applicable. However, these techniques are not sufficient by themselves to make Internet protocols robust against arbitrary failures. We believe that we must complement these techniques with additional guidelines for protocol design to protect against arbitrary failures.

3 Design Guidelines for Robust Protocols

The Internet is an immense and diverse distributed system. It spans many different administrative domains, incorporates a wide variety of underlying technologies, and supports a vast and rapidly evolving array of applications. One cannot approach such a system with a rigid set of narrow rules without killing the vitality that has been at the heart of the Internet's success. This was recognized by the early designers of the Internet, who, instead of enforcing rigid rules, worked with a set of much more general and imprecise guidelines that offer advice on basic design decisions such as how and where to keep state or initiate recovery. Examples include endpoint error recovery, critical state should be refreshed periodically, and implementations should be engineered to interoperate with non-conforming peers.

The main purpose of this paper is to call for the identification of a similar set of guidelines to improve robustness against arbitrary failures. The overarching philosophy of the guidelines we propose is that one should *design defensively*. That is, everpresent in the design process should be a recognition that incoming information might be incorrect, and that nodes might be malicious or broken. While one wants to stop short of a paralyzing paranoia, protocols should always adopt a cautious skepticism. Of course, we are not the first to propose defensive design, and many protocols already exhibit this, albeit in an ad hoc fashion. Our goal is to synthesize and generalize the underlying ideas behind some of the better protocols that have been recently proposed, in the hope that making these guidelines explicit will help the design of future protocols.

Our first guideline is perhaps the least controversial but one of the hardest to achieve in practice: use very clean and simple interfaces that do not overload mechanism with multiple functions, have clean functional (not procedural) semantics, and do not embed performance optimizations in the protocol definition.

Guideline #1: *Value conceptual simplicity*

The natural tendency is for interfaces to become increasingly complex over time, as designers evolve systems by adding features in a backwardly compatible way. But this can lead to problems! It is much harder for multiple parties to correctly implement complex designs, and unforeseen interactions between components can hide potentially devastating vulnerabilities.

One illustration of the effect of complex semantics is the persistent route oscillations that can arise in BGP [18]. In BGP, a router selects its best route among possible alternatives according to an ordered decision criteria, e.g., the highest "local preference" before the shortest AS path length. One of these criteria is the Multi-Exit Discriminator (MED), which is used by one ISP to specify ingress preferences to another. This provides a traffic engineering mechanism for overriding "hot-potato" or "early-exit" routing. Unfortunately, MED has semantics that differ from the other attributes. The other attributes are straightforward in that they can be used to order and directly compare routes, so that the best route may be selected by simply selecting amongst best candidates. MED, on the other hand, cannot be used to order routes for selection directly because MED values are only comparable when learned from the same neighboring AS. This can result in, for example, a route with a MED causing the current best route to be dropped, yet without the MED route becoming the new current best route. This means that with MEDs, all candidate routes must be gathered and then selected in a single pass.

This dependence of routes on each other can manifest itself as oscillations when route reflectors [1] are used. Here, we discuss only the overall problem and refer readers to [18] for details. To understand the problem, note that BGP routers within an AS must coordinate with each other to make consistent route selections. This was originally achieved by having every BGP router peer with every other router, a configuration known as a BGP mesh in which all routers receive all information and run compatible decision procedures in parallel. However, meshes are inherently unscalable to large ASes. A route reflector is a more scalable alternative in which a central location, the route reflector, receives information from and redistributes best route information to all BGP routers. Unfortunately the premise behind route reflectors—scalability via information hiding—conflicts with the semantics of MEDs because all information is required at each router to ensure consistency. The result can be that as routes propagate through the system they cause other routes to change in a vicious cycle. This is handled today by operational guidelines that discourage vulnerable configurations.

Our second guideline is less obvious: if other nodes are potentially untrustworthy because they could be misimplemented or even malicious, then protocols should be explicitly designed to reduce each node's circle of trust to the minimal set of information required to achieve the protocol's purpose. Caution in trusting other nodes is merely prudent in a large and heterogeneous distributed system like the Internet.

Guideline #2: *Minimize your dependencies*

When protocols are being initially designed, it is often more convenient to assume the other nodes participating in a protocol are trustworthy, but that trust is often misplaced. The problems often manifest themselves not when the protocol is first being deployed, but later as the protocol is used by a wider population with differing motivations and security policies. For example, the TCP congestion control mechanism relies on information provided by the receiver as to which packets arrived correctly. Although it seems natural that the TCP sender and receiver need to cooperate to transfer data, the protocol is defined in such a way that a sender must simply trust the congestion control information being provided by a receiver. This allows malicious receivers to trick today's Web servers into sending at rates far above TCP-friendly rates [21]. Specifically, the TCP fast recovery mechanism uses duplicate acknowledgements to first trigger a retransmission of the missing packet and then to trigger additional sends, reasoning that each duplicate acknowledgment implies that another packet has left the network. This opens the door for an attack—an unscrupulous receiver can send an infinite stream of duplicate acknowledgements! Indeed, since the IP packet delivery model allows duplication, this is an undetectable error in the existing TCP protocol specification. A simple fix is to use selective acknowledgements (SACK [17]). These are more robust than duplicate acknowledgements because they can be designed to be unambiguous and idempotent in their effects.

Of course, one cannot avoid all dependencies in protocols. In cases where a node has to rely on information from another node, the protocol designer should add mechanisms into the protocol to allow for verification whenever possible by, for example, actively testing the node's responses or by comparing the data to the information provided by other nodes.

Guideline #3: *Verify when possible*

Note that explicitly adding redundancy into protocols conflicts with the common practice of stripping out all redundancy as superfluous. At times the desire to verify information may conflict with the desire to reduce complexity; that is, verification may require information that is not necessary for the basic functioning of the protocol. How to best balance these two competing goals–reducing complexity and collecting enough information to reliably verify—will depend on the context.

As a concrete example, consider Explicit Congestion Notification (ECN) [19]. With ECN, routers mark packets by setting an ECN bit in the packet header when they become congested. The marks are then returned from receiver to sender via acknowledgement packets. If the ECN bit is set, the server interprets it as a congestion indication and reacts accordingly by reducing the congestion window. This mechanism has the advantage of signaling the congestion early without causing loss.

However, it was observed in [11] that signaling congestion with marks is not as robust as signaling with drops. Once a packet is dropped, it cannot be "undropped", at least not in a manner that achieves reliable transport. This makes it highly likely that drops will be observed by the sender. Yet with marks, the receiver could fail to return the mark signal to the sender, or the signal could be stripped off by devices along the network path between the marking router and original sender (e.g., firewalls that normalize IP header bits or VPN boxes that strip off an outer IP header). This kind of arbitrary failure can result in a flow obtaining much more than its fair share of the bandwidth. And the fact that a receiver that fails to return mark signals might receive significantly more bandwidth encourages this form of failure.

A solution to this problem (described in [11]) is to revise the design to use nonces, in a manner that generalizes the TCP acknowledgement example above. Packets carry a single-bit nonce that is erased by marking routers to signal congestion. Receivers echo the nonce sum in acknowledgements to the sender along with their indication of whether the packet was marked. (The sum is used to cover sequences of packets because acknowledgements can be lost.) When marks are not being signaled, this sum gives the sender confidence that the packets were in fact received and that they were not marked, *i.e.*, that congestion has not been signaled. This is a strong result because it depends on no assumptions about receiver behavior for its correctness.

An important observation in our context is that this verification mechanism is lightweight. It uses few header bits and little state and computation at end systems to detect misbehavior probabilistically. The chance that any incident of erasing a mark will be detected is 50%, as only one bit of check information is used. However, because nonces are random, each loss represents an independent trial. This means that repeated misbehavior will be detected quickly.

Our fourth guideline concerns unsolicited requests by unauthenticated parties; this can leave systems vulnerable to intentional or accidental resource exhaustion. Recent denial of service attacks are just one example of this. Careful planning is needed to keep the resources of one node from being at the mercy of other nodes:

Guideline #4: *Protect your resources*

Although this may seem intractable, the potential for resource exhaustion can be greatly reduced through careful protocol design. Consider the example of the TCP connection setup. Connection state is established on receipt of an initial SYN packet and must be maintained until the three-way handshake and subsequent connection completes, or until the connection attempt is timed out. This behavior has been exploited by "SYN flood" denial-of-service attacks that send a flood of initial SYN packets (typically with spoofed source addresses) and never complete the handshake. A

SYN flood can exhaust connection resources at a server and thereby cause legitimate traffic to be shut out [6]. This problem is not readily addressed by verification because the end-system receiving a TCP connection request cannot determine, at the time a SYN packet arrives, whether the packet is part of a legitimate connection establishment sequence. The vulnerability arises because the server allocates resources to connection attempts that are not yet known to be valid.

Fortunately, a simple re-design of the connection setup protocol can protect the server's resources by shifting the resource burden. Instead of keeping state on the server, return the state to the initiating client, since they are the party for whom the state is being maintained. This can be achieved by returning the state as part of the SYN-ACK, and later have the client re-supply the server with missing state when the handshake is completed. Assuming there is an inexpensive way to verify the integrity of the re-supplied state, the server will not be left holding incomplete or incorrect connection state. SYN cookies is a backwards-compatible Linux implementation of this more robust connection setup protocol [2]. In SYN cookies, the initial sequence number (ISN) is used to encode and return connection state to the client, and the encoding is based on a secure hash that is efficient to verify.

The above philosophy can be applied to protect the local resources (computation, buffer space, connection state, bandwidth, and so forth) that can be consumed in response to messages received from parties whose behavior is uncertain. In fact, the cookie technique is quite general, being originally used in the design of Photuris [12], a protocol used for establishing shared session keys, to avoid accumulating state during the key exchange.

Fifth, we observe that since errors cannot always be prevented or caught, we must also design protocols to limit the possible damage resulting from incorrect behavior. While we would like to prevent instabilities from occurring in the first place, but if they do occur, we also want to make sure the effects do not cascade out of control.

Guideline #5: *Limit the scope of vulnerability*

Again, this can seem intractable, yet oftentimes very simple techniques can protect systems against broad classes of unforeseen errors. Consider the example of route flapping. Until fairly recently, router and link instabilities could cause serious load issues for BGP because routes were "flapping". The problem was that, at the BGP level, route announcements and withdrawals were essentially propagated throughout the world, so that every location saw the cumulative sum of all route flaps in the entire Internet. To decrease the extent to which local link and router instabilities effect distant networks, a Route Flap Damping [10] mechanism has been added to BGP. It works by holding down routes that are repeatedly withdrawn and then announced, with infrequent changes resulting in no hold-down, and oscillation causing the greatest amount of hold-down. Every router that implements damping thus creates a barrier separating oscillations from the rest of the network.

Another recent example is BGP error processing. The BGP specification requires that an incorrect announcement causes the entire session to be dropped and restarted, since error checking is not provided at the announcement level (as is done in OSPF and IS-IS). However, the result is that one bad announcement can cause the entire routing session (that is roughly 100K announcements today) to be dropped and restarted. This was compounded when one vendor chose to pass bad announcements rather than drop the entire session, while other vendors dropped sessions as required [7]. In June 2001, a single operator's misconfiguration tickled this bug: it was propagated to a substantial fraction of the Internet, where it caused many sessions to be dropped, resulting in Internet-wide outages.

Finally, we observe that, as with the previous example, some robustness problems can lie dormant, only being triggered by other errors that occur in the system. Thus, making systems more robust may also require that system designers and operators seek out and fix errors before they begin to cascade. Unfortunately, there is a tension here. If systems can continue operating in the face of failures, errors can persist indefinitely without anyone having the visibility or motivation to fix the underlying problem. If left untreated, however, failures can combine with severe unforeseen consequences. Thus, we end with our sixth guideline:

Guideline #6: *Expose errors*

As an example, the recent investigation of TCP checksum failures resulted in the discovery that data was being corrupted at hosts or routers [22]. Because these failures were discovered at the receiver and then ignored (once the packet was discarded) the sending host received no indication that anything was amiss, other than a negligible increase in the rate of retransmissions. As another example, researchers have recently found a flaw in a common BGP configuration strategy by which ISPs connect to their customers [16]. ISPs often filter routes advertised by their customers, to prevent them from using the ISP to provide transit to other customers. The flawed configuration filters the customer routes based on their prefixes. This works correctly when there are no additional failures. However, if the customer is multihomed, and the link to the customer fails, the filtering rules may result in an ISP providing unintended transit to the customer's other ISP. This can be avoided by repairing the latent misconfiguration, filtering on AS-PATH rather than prefixes.

4 Discussion

In this paper, we have argued that it is time to seriously treat and systematically defend against arbitrary failures – implementation bugs and other failures that, through error or malice, are not fail-stop as are simple link and node failures. We have discussed and categorized several examples of problems and solutions to argue three main points. First, arbitrary failures occur surprisingly often and can result in significant disruptions to the Internet; consider the DNS and TCP bugs that have proved to be pervasive and BGP vulnerabilities that have caused widespread outages. Hence they must be tackled if we are to significantly improve the dependability of the Internet. Second, arbitrary failures are not sufficiently well addressed by existing bodies of work. We observed, for example, that authentication determines which party sent a message but not whether the contents of the message make sense. Third, the key component of a long-term solution that we feel must be developed is design guidelines that help protocol implementations defend against arbitrary failures. Design guidelines for robustness have been extremely successful at providing a solid foundation for tolerating fail-stop failures. Fail-stop failures are regularly considered during the design stage and concepts such as soft-state are applied systematically. In contrast, arbitrary failures tend to be viewed as isolated incidents and dealt with in an ad hoc fashion.

To work towards defensive protocols, we have put forward a candidate set of guidelines, as an opening proposal for further correction and refinement. The examples we presented provide an existence proof that there do exist designs that can withstand arbitrary failures, and that these designs are essentially as efficient and scalable as designs that do not tolerate arbitrary failures. If we are to make the leap from ad hoc treatment of arbitrary failures to defensive design, then we must be able to extract lessons from one protocol setting and apply them in another.

Finally, all our effort will be for naught unless the guidelines that are ultimately developed are used in practice. Thus we hope to raise the level of consciousness of arbitrary failures during the period that protocols are being designed. One suggestion we would like to make is that every RFC contain a “Robustness Considerations” section that answers a simple question: What would go awry if the protocol messages are corrupted? We hope that such a ”Robustness Considerations” section, with its focus on a simple and direct question, will be of more use than ”Security Considerations” sections have been in the past.

References

[1] T. Bates, R. Chandra, and E. Chen. BGP route reflection - an alternative to full mesh IBGP. RFC 2796, IETF, Apr. 2000.

[2] D. J. Bernstein. SYN cookies. `http://cr.yp.to/syncookies.html`, 1996.

[3] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Safety and performance in the SPIN operating system. In *Symposium on Operating System Principles (SOSP)*, Dec. 1995.

[4] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems (TOCS)*, 8(1), Feb. 1990.

[5] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Operating Systems Design and Implementation (OSDI)*, 1999.

[6] CERT advisory ca-1996-21 TCP SYN flooding and IP spoofing attacks. `http://www.cert.org/advisories/CA-1996-21.html`, Sep. 1996.

[7] Cisco security advisory: Cisco IOS BGP attribute corruption vulnerability. `http://www.cisco.com/warp/public/707/ios-bgp-attr-corruption-pub.shtml`, May 2001.

[8] D. D. Clark. The design philosophy of the DARPA Internet protocols. In *ACM SIGCOMM*, Aug. 1988.

[9] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2), 1991.

[10] R. G. Curtis Villamizar, Ravi Chandra. BGP route flap damping. RFC 2439, IETF, Nov. 1998.

[11] D. Ely, N. Spring, D. Wetherall, S. Savage, and T. Anderson. Robust congestion signaling. In *9th International Conference on Network Protocols (ICNP)*, Nov. 2001.

[12] P. Karn and W. A. Simpson. Photuris: Session-key management protocol. RFC 2522, IETF, March 1999.

[13] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *TOPLAS*, 4(3), July 1982.

[14] M. Levine. BGP noise tonight? NANOG mail archives, Oct. 2001. `http://www.merit.edu/mail.archives/nanog/2001-10/msg00221.html`.

[15] S. Maguire. *Writing Solid Code : Microsoft's Techniques for Developing Bug-Free C Programs*. Microsoft Press, 1993.

[16] R. Mahajan. A study of BGP misconfiguration. Talk at ISMA winter workshop, Dec. 2001.

[17] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgement options. RFC 2018, IETF, April 1996.

[18] D. McPherson, V. Gill, D. Walton, and A. Retana. BGP persistent route oscillation condition. Internet Draft draft-mcpherson-bgp-route-oscillation-01.txt, IETF, March 2001.

[19] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ECN) to IP. RFC 3168, IETF, Sep. 2001.

[20] P. Salus. The Internet under stress. Talk at NANOG 23, Oct. 2001.

[21] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP congestion control with a misbehaving receiver. *Computer Communication Review*, 29(5), 1999.

[22] J. Stone and C. Partridge. When the checksum and the data disagree. In *ACM SIGCOMM*, Aug. 2000.