Netalyzr: Illuminating Edge Network Neutrality, Security, and Performance

Christian Kreibich[†], Nicholas Weaver[†],
Boris Nechaev*, and Vern Paxson[†]

TR-10-006

May 2010

Abstract

In this paper we present Netalyzr, a network measurement and debugging service that evaluates the functionality provided by people's Internet connectivity. The design aims to prove both comprehensive in terms of the properties we measure and easy to employ and understand for users with little technical background. We structure Netalyzr as a signed Java applet (which users access via their Web browser) that communicates with a suite of measurement-specific servers. Traffic between the two then probes for a diverse set of network properties, including outbound port filtering, hidden in-network HTTP caches, DNS manipulations, NAT behavior, path MTU issues, IPv6 support, and access-modem buffer capacity. In addition to reporting results to the user, Netalyzr also forms the foundation for an extensive measurement of edge-network properties. To this end, along with describing Netalyzr's architecture and system implementation, we present a detailed study of 112,000 measurement sessions that the service has recorded since we made it publicly available in June 2009.

†International Computer Science Institute, 1947 Center St., Ste. 600, Berkeley, CA 94704
* Helsinki Institute for Information Technology, PO Box 19800, 00076 Aalto, Finland

This work was partially supported by funding provided to ICSI through National Science Foundation grants CNS: 0722035 ("Architectural Support for Network Trouble-Shooting"), NSF: 0433702 ("CCIED: Center for Internet Epidemiology and Defenses"), and CNS: 0905631 ("Invigorating Empirical Network Research via Mediated Trace Analysis"), and also by funding provided by Amazon.com. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators and do not necessarily reflect the views of the National Science Foundation. We are deeply grateful to the Netalyzr users for enabling this study and to our beta testers for the insightful comments and feedback. We would particularly like to thank Mark Allman, Paul Barford, Scott Bradner, John Brzozowski, Randy Bush, Niels Bakker, Richard Clayton, Chris Cowart, Keith Dawson, Adrian Dimcev, Holger Dreger, Brandon Enright, Kevin Fall, Carrie Gates, Andrei Gurtov, Mark Handley, Theodore Hong, Kelly Kane, Matthew Kogan, Keith Medcalf, Thomas Narten, Michael Ross, Wooter Wijngaards, and Richard Woundy.

1. INTRODUCTION

For most Internet users, their network experience—perceived service availability, connectivity constraints, responsiveness, and reliability—is largely determined by the configuration and management of their *edge network*, i.e., the specifics of what their Internet Service Provider (ISP) gives them in terms of Internet access. While conceptually we often think of users receiving a straight-forward "bit pipe" service that transports traffic transparently, in reality a myriad of factors affect the fate of their traffic.

It then comes as no surprise that this proliferation of complexity constantly leads to troubleshooting headaches for novice users and technical experts alike, leaving providers of web-based services uncertain regarding what caliber of connectivity their clients possess. Only a few tools exist to analyze even specific facets of these problems, and fewer still that people with limited technical understanding of the Internet will find usable. Similarly, the lack of such tools has resulted in the literature containing few measurement studies that characterize in a comprehensive fashion the prevalence and nature of such problems in the Internet.

In this work we seek to close this gap. We present the design, implementation, and evaluation of *Netalyzr*,[1] a publicly available service that lets any Internet user obtain a detailed analysis of the operational envelope of their Internet connectivity, serving both as a source of information for the curious as well as an extensive troubleshooting diagnostic should users find anything amiss with their network experience. *Netalyzr* tests a wide array of properties of users' Internet connections, starting at the network layer, including IP address use and translation, IPv6 support, DNS resolver fidelity and security, TCP and UDP service reachability, proxying and firewalling, anti-virus intervention, content-based download restrictions, content manipulation, HTTP caching prevalence and correctness, latencies, and access-link buffering.

We believe the breadth and depth of analysis *Netalyzr* provides is unique among tools available for such measurement. In addition, as of this writing we have recorded 112,000 runs of the system from 86,000 different public IP addresses, allowing us to construct a large-scale picture of many facets of Internet edge behavior. The measurements have found a wide range of behavior, on occasion even revealing traffic manipulation that the network operators themselves did not know about. More broadly, we find chronic over-buffering of links, a significant inability to handle fragmentation, numerous incorrectly operating HTTP caches, common NXDOMAIN wildcarding, impediments to DNSSEC deployment, poor DNS performance, and deliberate manipulation of DNS results.

We begin by presenting *Netalyzr*'s architecture and implementation (§ 2) and the specifics of the different types of measurements it conducts (§ 3). We have been operating *Netalyzr* publicly and continuously since June 2009, and in § 4 report on the resulting data collection, including flash crowds, their resulting measurement biases, and our extensive calibration tests to assess the correct operation of *Netalyzr*'s test suite. In § 5 we present a detailed analysis of the resulting dataset and some consequences of our findings. We defer our main discussion of related work to § 6 in order to have the context of the details of our measurement analysis to compare against. Finally, we summarize in § 7.

[1]http://netalyzr.icsi.berkeley.edu

2. SYSTEM DESIGN

When designing *Netalyzr* we had to strike a balance between a tool with sufficient flexibility to conduct a wide range of measurement tests, yet with a simple enough interface that unsophisticated users would run it—giving us access to a much larger (and less biased towards "techies") end-system population than possible if the measurements required the user to install privileged software. To this end, we decided to base our approach on using a Java applet to drive the bulk of the tests, since (*i*) Java applets run automatically within most major web browsers, (*ii*) applets can engage in raw TCP and UDP flows to arbitrary ports (though not with altered IP headers), and, if the user approves trusting the applet, contact hosts outside the same-origin policy, (*iii*) Java applets come with intrinsic security guarantees for users (e.g., no host-level file system access allowed by default runtime policies), and (*iv*) Java's fine-grained permissions model allows us to adapt gracefully if a user declines to fully trust our applet.

The resulting system includes about 5,000 lines of Java for the applet (as well as some JavaScript to implement the client side of some test connections) and 12,000 lines of Python for the different servers. Figure 1 shows the conceptual *Netalyzr* architecture, whose components we now discuss in turn.

Application Flow. Users initiate a test session by visiting the *Netalyzr* website and clicking **Start Analysis** on the webpage with the embedded Java test applet. Once loaded, the applet conducts a large set of measurements probes, indicating test progress to the user. When testing completes, the applet redirects to a summary page that shows the results of the tests in detail and with explanations (Figure 2). The users can later revisit a session's results via a permanent link associated with each session. We also save the session state (and server-side packet traces) for subsequent analysis.

Front- and Back-end Hosts. The *Netalyzr* system involves three distinct locations: (*i*) the user's machine running the test applet in a browser, (*ii*) the *front-end* machine responsible for dispatching users and providing DNS service, and (*iii*) multiple *back-end* machines that each host both a copy of the applet and a full set of test servers. All back-end machines run identical configurations and *Netalyzr* conducts all tests in a given client's session using the same back-end machine.

The front-end machine runs Linux 2.6 on a 2.5 GHz Intel

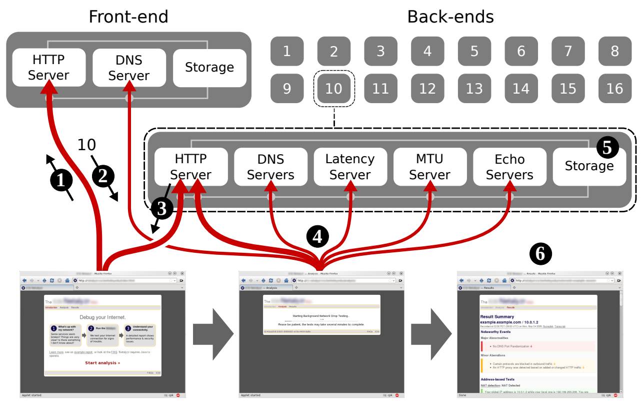
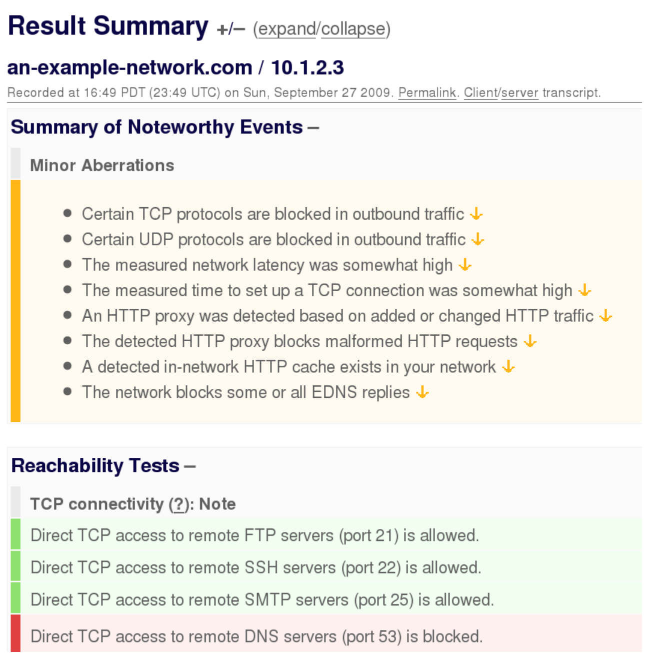


Figure 1: *Netalyzr*'s conceptual architecture. ❶ The user visits the *Netalyzr* website. ❷ When starting the test, the front-end redirects the session to a randomly selected back-end node. ❸ The browser downloads and executes the applet. ❹ The applet conducts test connections to various *Netalyzr* servers on the back-end, as well as DNS requests which are eventually received by the main *Netalyzr* DNS server on the front-end. ❺ We store the test results and raw network traffic for later analysis. ❻ *Netalyzr* presents a summary of the test results to the user.



Result Summary +/– (expand/collapse)

an-example-network.com / 10.1.2.3

Recorded at 16:49 PDT (23:49 UTC) on Sun, September 27 2009. Permalink. Client/server transcript.

Summary of Noteworthy Events –

Minor Aberrations

- Certain TCP protocols are blocked in outbound traffic ↓
- Certain UDP protocols are blocked in outbound traffic ↓
- The measured network latency was somewhat high ↓
- The measured time to set up a TCP connection was somewhat high ↓
- An HTTP proxy was detected based on added or changed HTTP traffic ↓
- The detected HTTP proxy blocks malformed HTTP requests ↓
- A detected in-network HTTP cache exists in your network ↓
- The network blocks some or all EDNS replies ↓

Reachability Tests –

TCP connectivity (?): Note

Direct TCP access to remote FTP servers (port 21) is allowed.

Direct TCP access to remote SSH servers (port 22) is allowed.

Direct TCP access to remote SMTP servers (port 25) is allowed.

Direct TCP access to remote DNS servers (port 53) is blocked.

Figure 2: A partial screen capture of *Netalyzr*'s results page as seen by the user upon completion of all tests. The full report is 4–10 times this size, depending on whether the user expands the different sections.

Xeon machine with 8 GB of memory, physically located at our institute. We manage the back-end machines using Amazon's EC2 service [1] to facilitate scalability. These hosts are virtual 2.6 GHz AMD Opteron machines with 1.8 GB of memory and run Linux 2.6. At peak load times we employ 20 back-end hosts.

Front-end Web Server. Running on the front-end machine, this server provides the main website, including a landing/dispatch page, documentation, FAQs, an example report, and access to reports from previous sessions. The server employs a pre-forked pool of multithreaded child processes. The front page also includes a Java dispatch applet that ensures that the user has Java installed and then directs the user to a randomly selected back-end server to load-balance the actual testing process. Finally, the front page rate-limits visitors to a fixed number of measurements per minute per back-end server.

Back-end Web Servers. The back-end web servers host the actual measurement applet (so that its probe connections to the server accord with the same-origin policy) and perform HTTP testing and overall session management. When sending the measurement applet, the server includes a set of configuration parameters, including a globally unique session ID.

Measurement Applet. The Java applet implements 38 types of tests, some with a number of subtests. We describe them in detail in Section 3. The applet conducts the test cases sequentially, but also employs multithreading to ensure that test sessions cannot stall the entire process, and to speed up some parallelizable tasks. As tests complete, the applet transmits detailed test results to the back-end server; it also sends a continuously recorded client-side transcript of the session. Finally, we sign our applet with a certificate from a trusted authority so that browsers indicate a valid signature.

DNS Servers. An instance of this server runs on the front-end as well as the back-end machines. On the front-end, it acts as the authoritative resolver for two subdomains, `.`*n*`.na.edu` and `.`*n*`.na.org`, while on the back-ends it receives DNS test queries generated directly from the applet rather than through the user's DNS resolver library. The server interprets queries for specific names as commands, generating replies that encode values in A and CNAME records. For example, requesting `has_edns.`*n*`.na.edu` will return an A record reflecting whether the query message indicated EDNS support. The server also accepts names with arbitrary interior padding to act as a cache-busting nonce, ensuring that queries reach our server.

Echo Servers. An array of simple TCP and UDP echo servers allow us to test service-level reachability and content modification of traffic on various ports. The servers mostly run on well-known ports but do not implement the associated application protocol. Rather, they use their own simple payload schema to convey timing, sequencing, and the requester's IP address and source port back to the client. An additional server can direct a DNS request to the user's public address to check if the user's NAT or gateway acts as a proxy for external DNS requests.

Bandwidth Measurement Servers. To assess bandwidth, latency, buffer sizing, and packet dynamics (loss, reordering, duplication) we employ dedicated UDP-based measurement servers. Like the echo servers, these use a custom payload schema that includes timing information, sequence numbers, instructions regarding future sending, and aggregate counters.

Path MTU Measurement Server. To measure directional path MTUs, we use a server that can capture and transmit raw packets, giving us full access to and control over all packet headers.

Storage. To maintain a complete record of server-side session activity, we record all relevant network traffic on the front- and back-end machines, except for the relatively high-volume bandwidth tests. Since Java applets do not have the ability to record packets, we cannot record such traces on the client side.

Session Management. The back-end web servers establish and maintain session state as test sessions progress, identifying sessions via RFC 4122 UUIDs. We serialize completed session state to disk on the back-end hosts and periodically archive it on the front-end. When viewing a session summary, the front-end web server redirects the request to the appropriate back-end (encoded in the session ID) if it does not have the state locally, and the back-end web server does the opposite, with cycle detection to avoid looping.

3. MEASUREMENTS CONDUCTED

We now describe the types of measurements *Netalyzr* conducts and the particular methodology used. We begin with layer 3 measurements (addressing, fragmentation, MTU, raw performance, IPv6 support) and then progress to higher layers (general service reachability, DNS, HTTP), finishing with a discussion of user feedback and tests we chose to omit.

3.1 Network-layer Information

Addressing. We obtain the client's local IP address via the Java API, and use a set of raw TCP connections and UDP flows to our echo servers to learn the client's public address. From this set of connections we can identify the presence of NAT, and if so how it renumbers addresses and ports. If across multiple flows we observe more than one public address, then we assess whether the address flipped from one to another—indicating the client changed networks while the test was in progress—or alternates back and forth. This latter implies either the use of load-balancing, or that the NAT does not attempt to associate local systems with a single consistent public address but simply assigns new flows out of a public address block as convenient. (Only 1% of sessions included an address change from any source.)

IP Fragmentation. We test for proper support of IP fragmentation (and also for MTU measurement; see below) by sending UDP payloads to our test servers. We first check for the ability to send and receive fragmented UDP datagrams. In the applet → server direction, we send a 2 KB datagram which, if received, generates a small confirmation response. Due to the prevalence of Ethernet framing, we would expect most clients to send this packet in fragments, but it will always be fragmented by the time it reaches the server. We likewise test the server → applet direction by our server transmitting (in response to a small query from the client) a 2 KB message to the client. This direction will definitely fragment, as the back-end nodes have an interface MTU of 1500 bytes.

If either of the directional tests fails, the applet performs binary search to find the maximum packet size that it can successfully send/receive unfragmented.

The applet also tries to send and receive packets with 1471 bytes of UDP payload (normally yielding a 1499-byte IP packet) which would maximize the payload on an Ethernet network without fragmentation. This checks for the existence of an "MTU hole", where packets can be sent unfragmented by the endpoint but cannot be refragmented properly when passing through a path MTU bottleneck, either because the bottleneck is functioning incorrectly or the host sent the packet with `DF` set.

Path MTU. A related set of tests conducts path MTU probing. The back-end server for this test supports two modes, one for each direction. In the applet → server direction, the applet sends a large UDP datagram, resulting in fragmentation. The server monitors arriving packets and reports the IP datagram size of the entire original message (if received unfragmented) or of the original message's initial resulting fragment. This represents a lower bound on MTU in the applet → server direction, since the first fragment's size is not necessarily the full path MTU. (Such "runts" occurred in only a handful of sessions).

In the server → applet direction, the applet conducts a binary search beginning with a request for 1500 bytes. The server responds by sending datagrams of the requested size with `DF` set. In each iteration one of three cases occurs. First, if the applet receives the `DF`-enabled response, its size is no more than the path MTU. Second, if the response exceeds the path MTU, the server processes any resulting ICMP "fragmentation required" messages and sends to the applet the attempted message size, the offending location's IP address, and the next-hop MTU conveyed in the ICMP message. Finally, if no messages arrive at the client, the applet infers that the ICMP "fragmentation required" message was not generated or did not reach the server, and thus a path MTU problem exists.

Latency, Bandwidth, and Buffering. We measure packet delivery performance in terms of round-trip latencies, directional bandwidth limits, and buffer sizing. With these, our primary goal is not to measure capacity itself (which numerous test sites already address [31]), but as a means to measure the sizing of bottleneck buffers, which can significantly affect user-perceived latency. We do so by measuring the increase in latency between quiescence and that experienced during the bandwidth test, which in most cases will briefly saturate the path capacity in one direction and thus fill the buffer at the bottleneck.

Netalyzr conducts these measurements in two basic ways. First, early in the measurement process it starts sending in the background small packets at a rate of 5 Hz. We use this test to detect transient outages, such as those due to a poor wireless signal.

Second, it conducts an explicit latency and bandwidth test. The test begins with a 10 Hz train of 200 small UDP packets, for which the back-end's responses provide the baseline mean latency used when estimating buffer sizing effects. The test next sends a train of small UDP packets that elicit 1000-byte replies, with exponentially ramping up (over 10 seconds) the size in slow-start fashion: for each packet received, the applet sends two more. In the second half of the interval, the applet measures the sustained rate at which it receives packets, as well as the average latency. (It also notes duplicated and reordered packets over the entire run.) After waiting 5 seconds for queues to drain, it repeats with sizes reversed, sending large packets to the server that trigger small responses. Note that most Java implementations will throttle sending rates to $\leq$ 20 Mbps, imposing an upper bound on the speed we can measure.

IPv6 Adoption. To measure IPv6 connectivity we have to rely on an approximation because neither our institution nor Amazon EC2 supports IPv6. However, on JavaScript-enabled hosts the analysis page requests a small logo from `ipv6.google.com`, reachable only over IPv6. We report the outcome of this request to our HTTP server. Since we cannot prevent this test from possibly fetching a cached image, we could overcount IPv6 connectivity if the user's system earlier requested the same resource (perhaps due to a previous *Netalyzr* run from an IPv6-enabled network).

3.2 Service Reachability

To assess any restrictions the user's connectivity may impose on the types of services they can access, we attempt to connect to 25 well-known services along with a few additional ports on the back-end. For `80/tcp` and `53/udp` connectivity, the applet speaks proper HTTP and DNS, respectively. We test all other services using our echo server protocol as described in Section 2.

In addition to detecting static blocking, these probes also allow us to measure the prevalence of proxying. In the absence of a proxy, our traffic will flow unaltered and the response will include our public IP address as expected. On the other hand, protocol-specific proxies will often transform this non-protocol-compliant response into an error, or simply abort the connection. Such proxies can reside on the end host (e.g., as part of an AV system) or in the network, with additional protocol information such as banners or headers often suggesting the source.

3.3 DNS Measurements

Netalyzr performs extensive measurements of DNS behavior, since DNS manipulations and subtle errors can have a major impact on a user's network experience. We implement two levels of measurement, *restricted* and *unrestricted*. Restricted measurements comply with Java's default same-origin policy, which for most JVMs allows the lookup of arbitrary names but only ever returns the IP address of the origin server, or throws an exception if the result is not the origin server's address. If however the user trusts the applet, then we can look up arbitrary names through the system's DNS resolver unrestrictedly, allowing us to conduct substantially more comprehensive testing. We refer to names corresponding to *Netalyzr*'s actual domain as *internal*, and any others as *external*. We can only look up the latter if unrestricted.

As mentioned earlier, our DNS authority server interprets requests for specific names as commands telling it what sort of response to generate. We encode Boolean results by returning the IP address of the back-end service for *true* and the address of an unrelated host in our institution for *false*. For results that return names, we indicate failure with the hostname `return_false`.

In our discussion, we abbreviate the fully qualified hostname of the back-end node as follows. First, *n*`.na.edu` stands for *node*`.netalyzr.icsi.berkeley.edu` (likewise *n*`.na.org` stands for *node*`.netalyzr.icir.org`). Second, if we give only a hostname `name`, it stands for `name.`*node*`.netalyzr.icsi.berkeley.edu`. Finally, we indicate the presence of a pseudo-random nonce value (to ensure cache penetration) using "*nonce*" in the name.

Glue Policy. One important but subtle aspect of the DNS resolution process concerns the acceptance and promotion of response data in the Authoritative or Additional records of a response, commonly referred to as *"glue"* records. Acceptance of such records can boost performance by avoiding future lookups, but also risk cache poisoning attacks [6]. Assessing the acceptance of these records is commonly referred to as "bailiwick checking," but the guidelines on the procedure allow latitude in how to conduct it [11]. *Netalyzr* leverages glue acceptance to enable tests of the DNS resolver itself.

We first check acceptance of arbitrary A records in the Additional section by sending lookups of special names (made distinct with nonces) that return particular additional A records. We then look up those additional names directly to see whether the resolver issues new queries for the names (which would return *false* when those names are queried directly) or answers them from its cache (returning *true*), indicating that the resolver accepted the glue. We then likewise check for caching of Authority A records. Finally, we check whether the server will automatically follow CNAME aliases. In this test, the response provides an Answer of a CNAME for `return_false`, with an Additional record encoding `return_false` as *true*. Thus, the query evaluates as *true* only if the resolver accepts the A record associated with the CNAME.

DNS Server Identification and Properties. We next probe more general DNS properties, including resolver identity, IPv6 support, `0x20` support [8], respect for short TTLs, port randomization for DNS requests, and whether the user's NAT, if present, acts as a DNS proxy on its external IP address.

When able to conduct unrestricted DNS measurements, we identify the resolver's IP address (as seen by our server) by returning it in an A record in response to a query for `server.`*nonce*`.`*n*`.na.edu`. This represents the address of the final server sending the request, not necessarily the one the client uses to generate the request. During our beta-testing we changed the applet code to conduct this query multiple times because we observed that some hosts will shift between DNS resolvers, and some DNS resolvers actually operate as clusters.

We test IPv6 AAAA support by resolving `ipv6_set.`*nonce*. We expect the resolver to request at least an A record for this name, and if it supports IPv6 then also a AAAA record. We discard the server's reply for the A record and then then resolve `ipv6_check.`*nonce*. When the A record request for this name arrives, the server checks whether it saw a AAAA request for the previous name (which might have arrived after the original A request, and thus could not have been reported initially), which it indicates by whether it returns *true* for the second A request. By proceeding in this fashion, we can assess resolver support for IPv6 even if the client itself does not support it.

Queries for the name `0x20` return *true* if the capitalization in a mix-cased request retains the original mix of casing. This detects non-`0x20`-compliant resolvers that change the capitalization of requested names.

If the DNS resolver accepts glue records for nameservers (NS responses in Authority or Additional), we leverage this to check whether the resolver respects short TTLs. Responses to the name `ttl0` or `ttl1` place a glue record for `return_false` in the Authoritative section with a TTL of 0 or 1 seconds, respectively. A subsequent fetch of `return_false` reveals whether the short TTLs were respected. (We can't simply use A records for this test because both the browser and end host may cache these records independently.)

We also use lookups of `glue_ns.`*nonce* to measure request latency. If the DNS resolver accepts glue records, it then also looks up `return_false.`*nonce* to check the latency for a cached lookup. We repeat this process ten times and report the mean value to the server, and also validate that `return_false.`*nonce* was fetched from the resolver's cache rather than generating a new lookup.

Finally, we test DNS port randomization. For unrestricted measurements, we perform queries for `port.`*nonce*, which the server answers by encoding in an A record the source port of the UDP datagram that delivered the request. For restricted measurements, the applet sends several queries for `dns_rand_set` and then checks the result by a query for `dns_rand_check`, with the latter resolving as *true* if the ports seen by our DNS server appeared non-monotone.

EDNS, DNSSEC, and actual DNS MTU. DNS resolvers can advertise the ability to receive large responses using EDNS [29], though they might not actually be capable of doing so. For example, some firewalls will not pass IP fragments, creating a de-facto DNS MTU of 1478 bytes for Ethernet framing. Other firewall devices may block all DNS replies greater than 512 bytes under the out-of-date assumption that DNS replies cannot be larger. While today small replies predominate, a lack of support for large replies poses a significant concern for DNSSEC deployment, as it will result in unpredictable performance degradation when DNS replies exceed unstated and hidden limits.

We measure the prevalence of this limitation by issuing lookups (*i*) to determine whether requests arrive indicating EDNS support, (*ii*) to measure the DNS MTU (for unrestricted measurements), and (*iii*) to check whether the resolver requests DNSSEC records. For the first, we look up `has_edns`, which returns *true* if the request contained an EDNS `OPT` pseudo-record. Responses for `edns_mtu` encode the advertised EDNS MTU in the lower 16 bits of an A record, and `wants_dnssec` returns *true* if the `DO` ("use DNSSEC") flag is set in an EDNS pseudo-record.

That a DNS resolver advertises (via EDNS) the ability to receive large responses does not guarantee that it actually can. We test its ability by requesting names `edns_medium` and `edns_large`, padded to 1300 and 1700 bytes, respectively. (We pad the replies to those sizes by adding Additional CNAME records.) Their arrival at the client indicates the resolver an indeed receive larger DNS replies.

During beta-testing we made this test more precise: the server answers requests for `ednspadding_`*X* with a response padded to exactly *X* bytes of DNS payload. We use this mechanism and binary search to determine the actual maximum supported by the resolver (whether or not it advertises EDNS).

NXDOMAIN Wildcarding. Some DNS operators configure their resolvers to perform "NXDOMAIN wildcarding", where they rewrite hostname lookups that fail with a "no such domain" error to instead return an A record for the IP address of a web server. The presumption of such blanket rewriting is that the original lookup reflected web surfing, and therefore returning the impostor address will lead to the subsequent HTTP traffic coming to the operator's web server, which then typically offers suggestions related to the presumed intended name. Such rewriting—often motivated by selling advertisements on the landing page—corrupts the web browsers' URL auto-complete features, and, worse, breaks protocol semantics for any non-HTTP application looking a hostname.

If unrestricted, the applet checks for this behavior by querying for a series of names in our own domain namespace, and which do not exist. We first look up `www.`*nonce*`.com`. If this yields an IP address, we have detected NXDOMAIN wildcarding, and proceed to probe the behavior in more detail, including simple transpositions (`www.yahoo.cmo`), other top-level domains (`www.`*nonce*`.org`), non-web domains (`fubar.`*nonce*`.com`), and domain internal to our site (`nxdomain.`*n*`.na.edu`). The applet also attempts to contact the host returned for `www.`*nonce*`.com` on `80/tcp` to obtain the imposed web content, which we log.

DNS proxies, NATs, and Firewalls. Another set of DNS problems arise not due to ISP interference but misconfigured or misguided NATs and firewalls. If the applet operates unrestricted, it conducts the following tests to probe for these behaviors. First, it measures DNS awareness and proxying. Our servers answer requests for `entropy.`*n*`.na.edu` with a CNAME encoding the response's parameters, including the public address, UDP port, DNS transaction ID, and presence of `0x20` encoding. The applet sends such DNS requests directly to the back-end server, bypassing the configured resolver. If it observes any change in the response (e.g., a different transaction ID or public address), then we have found in-path DNS proxying. The applet makes another request directly to the back-end server, now with deliberately invalid format, to which our server generates a similarly broken reply. If blocked, we have detected a DNS-aware middlebox that prohibits non-DNS traffic on `53/udp`. The applet then issues direct queries for the names `edns_large` and `edns_medium` (discussed above), and now also `edns_small` (a 400-byte response with EDNS), to check whether the NAT or firewall has problems handling either EDNS replies or large DNS responses.

During beta-testing we added a series of tests for the presence of DNS proxies in NAT devices. NATs often include such a proxy, returning via DHCP its local address to clients as the DNS resolver location if the NAT has not yet itself acquired an external DNS resolver.[2] Upon detecting the presence of a NAT, the applet assumes the gateway's local address is the *a.b.c.*`1` address in the same `/24` as the local IP address[3] and sends it a query for `entropy.`*n*`.na.edu`. Any reply indicates with high probability that the NAT implements a DNS proxy. In addition, we can observe to where it forwards the request based on the client IP address seen by our server.

During our beta-testing we became aware of the possibility that some in-gateway DNS resolvers act as open relays *for the outside* (i.e., for queries coming from external sources), enabling amplification attacks [22] and other mischief. We thus added a test in which the the applet instructs the back-end DNS server to send a UDP datagram containing a DNS request for `entropy.`*n*`.na.edu` to the public IP address of the client to see if it elicits a resulting response at our DNS server.

[2]Once the NAT obtains its external DHCP lease, it then forwards all DNS requests to the remote resolver.

[3] We assume this is the address, rather than probe for it, to avoid creating any apparent scanning activity.

Name Lookup Test. Finally, if unrestricted the applet looks up a list of 70+ common names, including major search engines, advertisement providers, financial institutions, email providers, and e-commerce sites. It uploads the results to our server, which then performs reverse lookups to test the forward lookups for consistency. This testing unearthed numerous aberrations, as discussed below.

3.4 HTTP Proxying and Caching

For analyzing HTTP behavior, the applet employs two different methods: using Java's *high-level API*, or its *low-level TCP sockets* (for which we implement our own HTTP logic). The first allows us to assess behavior imposed on the user by their browser (such as proxy settings), while the latter reflects behavior imposed by their access connectivity. (For the latter we take care to achieve the same HTTP "personality" as the browser by having our server mirror the browser's HTTP request headers to the applet so it can emulate them in subsequent low-level requests.) In general, the applet co-ordinates measurement tasks with the server using URL-encoded commands that instruct the server to deliver specific kinds of content (such as cache-sensitive images), report on properties of the request (e.g., specific header values), and establish and store session state.

Proxy Detection. We detect proxy configuration settings by monitoring request and result headers, as well as the server-perceived client address of a test connection. Differences when using the high-level API versus the socket API indicate the presence of a configured proxy. We first send a low level message with specific headers to the web server. The server mirrors the headers back to the applet, allowing the applet to conduct a comparison. Added, deleted, or modified headers flag the presence of an in-path proxy. To improve the detectability of such proxies, we use eccentric capitalization of header names (e.g. `User-AgEnt`) and observe whether these arrive with the same casing. A second test relies on sending an invalid request method (as opposed to `GET` or `POST`). This can confuse proxies and cause them to terminate the connection. A final test sets the `Host` request header to `www.google.com` instead of *Netalyzr*'s domain. Some proxies use this header's value to direct the outgoing connection [13]. The applet monitors for unexpected content—either Google's HTML banner, or a 302 redirect to a country-specific Google page. If seen, this represents a significant security vulnerability, as such proxies will allow Java and Flash to violate same-origin policies arbitrarily. However, we saw only a handful of instances of such behavior.

Caching policies, Content Transcoding, and File-type Blocking. We next test for in-network HTTP caching. For this testing, our server provides two test images of identical size (67 KB) and dimensions (512·512 pixels), but each the color-inverse of the other. Consecutive requests for the image result in alternating images returned to the applet. We can thus reliably infer when the applet receives a cached image based on the unchanged contents (or an HTTP 304 status code, "Not Modified"). We conduct four such request pairs, varying the cacheability of the images via various request and response headers, and including a unique identifier in each request URL to ensure each session starts uncached.

The applet can also identify image transcoding or blocking by comparing the received image's size to the expected one. In the post-beta codebase, the applet uploads any changed content for off-line analysis.

Finally, we test for content-based filtering. The applet downloads (*i*) an innocuous Windows PE executable (notepad.exe), (*ii*) a small MP3 file, (*iii*) a bencoded BitTorrent download file (for a Linux distribution's DVD image), and (*iv*) the EICAR test "virus",[4] a benign file that AV vendors recognize as malicious for testing purposes.

3.5 User Feedback

Because we cannot readily measure the physical context in which the user runs *Netalyzr*, we include a small, optional questionnaire in the results page. Some 19% of the users provided feedback. Of those, 57% reported using a wired rather than a wireless network; 17% reported running *Netalyzr* at work, 79% from home, 2% on public networks, and 2% on "other" networks.

3.6 Intentional Omissions

We considered several tests for inclusion but decided not to do so for one of two reasons. First, some tests can result in potentially destructive or abusive effects, particularly if run frequently or by multiple users. In this regard we decided against tests to measure the NAT's connection table size (which could disrupt unrelated network connections purged from the table), fingerprint NATs by connecting to its internal web-administration interface (which might expose sensitive information), general scanning either locally or remotely, and sustained high-bandwidth tests (such as BitTorrent throttling, for which alternative, bandwidth-intensive tests exist [10]). Another reason to omit a test concerns potential long-term side-effects for the users themselves. These could occur for technical reasons (e.g., we contribute towards possible upload/download volume caps) or legal/political ones (e.g., tests that attempt to determine whether access to certain sites suffers from censorship). Finally, we do not store tracking cookies in the user's browsers, since we do not aim to collect mobility profiles and can manage sessions using state on our servers.

4. DATA COLLECTION

We began running *Netalyzr* publicly in June 2009 and have kept it available continuously. We initially offered the service as a "beta" release (termed BETA), and for the most part did not change the operational codebase until January 2010, when we rolled out a substantial set of adjustments and additional tests (RELEASE). These comprise about 68% and 32% of the measurements, respectively. Unless otherwise specified, discussion refers to the combination of both datsets.

Website Operation. To date we have collected 112,239 sessions from 86,252 public IP addresses. The peak rate of data acquisition occurred during the June roll-out, with a maximum of 1,452 sessions in one hour. This spike resulted from mention of our service on several web sites. A similar but smaller spike occurred during the January relaunch, resulting in a peak load of 373 sessions in one hour.

Calibration. We undertook extensive calibration of the measurement results to build up confidence in the coherence and meaningfulness of our data. A particular challenge in realizing *Netalyzr* has been that it must operate correctly in the presence of a wide range of failure modes. While we put extensive effort into anticipating these problems during development, subsequent calibration served as a key technique to validate our assumptions and learn how the tests actually work on a large scale. In addition, it proved highly beneficial to employ someone for this task who was not involved in developing the tests, as doing so avoided incorporating numerous assumptions implicitly present in the code. Finally, we emphasize the importance of capturing subtle flaws in the data and uncovering inconsistencies that would otherwise skew the analysis results or deflate the scientific value of the data.

We based our calibration efforts on the BETA dataset, using it to identify and remedy sources of errors before beginning the RELEASE data collection. To do so, we assessed data consistency individually for each of the tests mentioned in § 3. We emphasized finding missing or ambiguous values in test results, checking value ranges, investigating outliers, confirming that each test's set of result variables exhibited consistency (e.g., examining that mutual exclusiveness was honored, or that fractions added up to a correct total), ensuring that particular variable values complied with corresponding preconditions (e.g., availability of raw UDP capability reliably enabling certain DNS tests), and searching for systematic errors in the data.

To our relief, this process did not uncover any major flaws in the codebase or the data. The most common problems we uncovered were ambiguity (for example, in distinguishing silent test failures from cases when a test was not executed at all) and inaccuracies in the process of importing the data into our session database. The RELEASE version of the codebase only differs from BETA in the presence of more unambiguous and extensive result reporting (along with the addition of new tests).

Identified Measurement Biases. A disadvantage of website-driven data collection is vulnerability to sudden referral surges from specific websites—in particular if these entail a technologically biased user population that can skew our dataset. In addition, our Java runtime requirement could

[4] `http://www.eicar.org/anti_virus_test_file.htm`

discourage non-technical users whose systems do not have the runtime installed by default. It also precludes the use of *Netalyzr* on many smartphone platforms. We now analyze the extent to which our dataset contains such bias.

The five sites referring the most users to *Netalyzr* are: stumbleupon.com (25%), lifehacker.com (14%), slashdot.org (13%), google.com (7%), and heise.de (7%). The context of these referrals affects the number of sessions we record for various ISPs. For example, most users arriving from slashdot.org did so in the context of an article on alleged misbehavior by Comcast's DNS servers, likely contributing to making their customers the biggest share of our users (10.9% of our sessions originate from Comcast's IP address ranges). Coverage in Germany via heise.de likely drove visits from customers of Deutsche Telekom, accounting for 2.6% of the sessions. We show a summary of the dominant ISPs in our dataset in Table 3 below.

The technical nature of our service introduced a "geek bias" in our dataset, which we can partially assess by using the `User-Agent` HTTP request headers of our users to infer browser type and operating system. Here we compare against published "typical" numbers [33, 34], which we give in parentheses. 39.8% (90%) of our users ran Windows, 8.1% (1.0%) used Linux, and 14.3% (5.9%) used MacOS. We find Firefox over-represented with 60.9% (28.3%) of sessions, followed by 18.8% (59.2%) for Internet Explorer, 15.6% (4.5%) for Safari, and 2.9% (1.7%) for Opera. This bias also extends to the choice of DNS resolver, with 12% of users selecting OpenDNS as their DNS provider.

While such bias is undesirable, it can be difficult to avoid in a study that requires user participation. We can at least ameliorate distortions from it because we can identify its presence. Its primary effect concerns our characterizations across ISPs, where we endeavor to normalize accordingly, as discussed below. We also note that technically savvy users may be more likely to select ISPs with fewer connectivity deficiencies, which would mean the prevalence of problems we observe may reflect underestimates.

5. DATA ANALYSIS

We now turn to an assessment of the data gathered from *Netalyzr* measurements to date. In our discussion we follow the presentation of the different types of tests above, beginning with layer 3 measurements and then progressing to general service reachability and specifics regarding DNS and HTTP behavior.

5.1 ISP and Geographic Diversity

We estimate the ISP and location of *Netalyzr* users by inspecting reverse (PTR) lookups of their public IP address, if available; or else the final Start-of-Authority record in the DNS when attempting the PTR lookup. We found these results available for 97% of our sessions.

To extract a meaningful organizational name, we started with a database of "effective TLDs," i.e., domains for which the parent is a broad, undifferentiated domain such as `gouv.fr` [19], to identify the relevant name preceding these TLDs. Given this approach, our dataset consists of sessions from 6,868 organizations (see Table 3 below for the 15 most frequent) across 182 countries, as shown in Figure 3. Activity however was dominated by users in the USA (48.2%), the EU (31.3%, with Germany accounting for 9.7% and Great Britain for 7.9%), and Canada (5.4%). 10 countries contributed sessions from more than 1,000 addresses, 46 from more than 100, and 97 from more than 10.

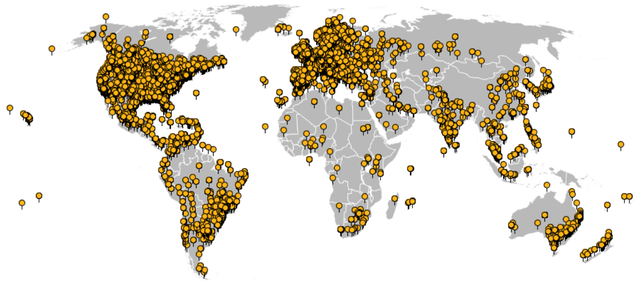


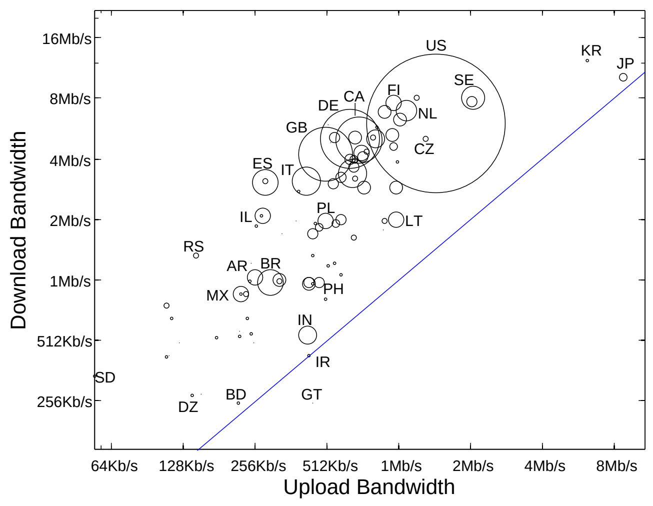
Figure 3: Global locations of *Netalyzr* runs.

5.2 Network-Layer Information

Network Address Translation. Unsurprisingly, we find NATs very prevalent among *Netalyzr* users (90% of all sessions). 79% of these sessions used the `192.168/16` address range, 16% used `10/8`, and 4% used `172.16/12`. 2% of the address-translated sessions employed some form of non-private address (either public or not allocated for private use). We did not discern any particular pattern in these sessions or their addresses; some were quite bizarre.

Port sequencing behavior. For more recent *Netalyzr* runs we have tracked potential NAT port renumbering explicitly, recording port numbers as seen by both the client and the server for a batch of 10 TCP connections. Of 19,510 sessions, 33% exhibit port renumbering. Of these, 8.9% appear random,[5] while 89.0% renumber in a strictly monotone-increasing fashion. We find a median "spread" for this sequence (range from smallest port to largest, inclusively) of 10, indicating renumbering that exactly reflects the tests we generated. A number of sessions have much higher spread, however (with a mean of 102). For these we have ruled out little-endian increments (i.e., by 256 rather than by 1) for other than a handful of sessions, but have not at this point assessed whether sessions with higher means contain significant forward jumps. Such jumps could occur due to effects other than the NAT concurrently processing additional connections separate from our measurements. Identifying and removing these would then enable us to estimate the level of multiplexing apparently present in the user's access link.

[5]We use a Wald-Wolfowitz test with sequence threshold 4 to measure randomness.



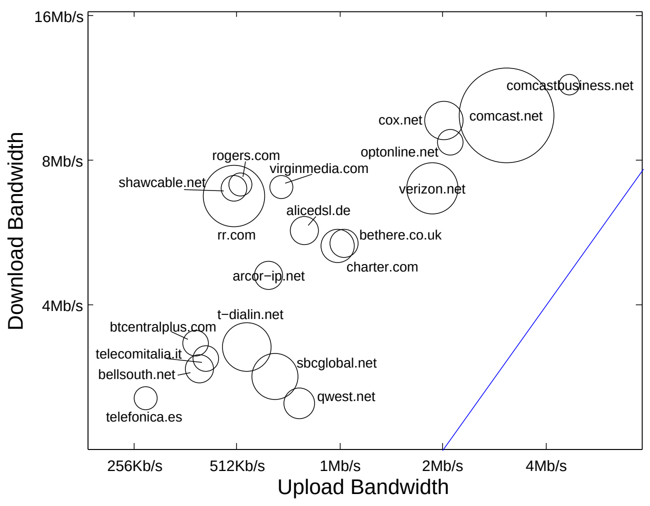


Figure 4: Average up/downstream bandwidths for countries with $\geq$ 10 sessions (left) and the 20 most prevalent ISPs (right). Circle areas are proportional to prevalence in the dataset, and diagonal lines mark symmetric upload and download capacity.

IPv6. We found IPv6 support to be rare but non-negligible: 4.5% of sessions fetched the logo from `ipv6.google.com`. As discussed above, this represents an upper bound due to possible caching effects.

Fragmentation. Overall, we find that fragmentation is not as reliable as desired [16, 26]. The RELEASE included a significant evaluation of UDP fragmentation behavior, for which we found 8% of the sessions unable to send 2 KB UDP packets, and likewise 8% unable to receive them.

We also found that 3% of the sessions which *could* send 2 KB packets could *not* send 1500 B packets. We find that 88% of these sessions come from Linux systems, strongly suggesting the likely cause to be Linux's arguably incorrect application of Path MTU discovery to UDP traffic, sending unfragmented UDP with `DF` set unless the system previously received an ICMP "fragmentation required" message from the recipient's path. Java, likewise, does not appear to re-transmit in the face of such ICMP feedback, instead raising an exception which *Netalyzr* reports as a failure.

Regarding the path MTU from our server to the client, 80% of the sessions exhibited a path MTU of 1500 B, followed by 1492 B (15%) which suggests a prevalence of PPP over Ethernet (PPPoE). We also observe small clusters at 1480 B, 1476 B, 1460 B, and 1458 B, but these are rare. Only 1% reported an MTU less than 1450 bytes.

For sessions with an MTU $<$ 1500 B, only 58% had a path that successfully sent a proper "fragmentation required" ICMP message back to our server. This finding reinforces that systems should avoid PMTU for UDP, and for TCP should provide robustness in the presence of MTU black holes [18].

Latency and Bandwidth. Figure 4 illustrates the balance of upstream vs. downstream capacities for countries and ISPs. Figure 5 shows the distribution of download bandwidths for particularly prominent ISPs. Two years after the study by Dischinger et al. [9] our results still partially match theirs, particularly for RoadRunner.

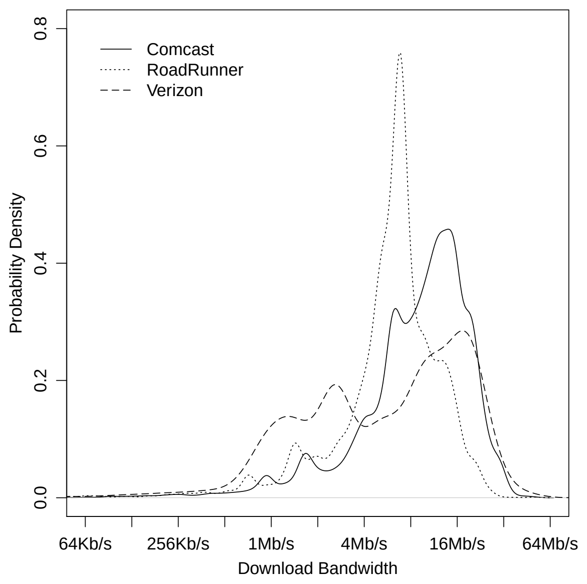


Figure 5: PDF of download bandwidths for the three most prominent ISPs in our dataset.

From the most aggregated perspective, we observed an average download bandwidth of 6.7 Mbps and for upload 2.7 Mbps. We find far more symmetric bandwidths for sessions that users self-reported as at work (10 Mbps/8.2 Mbps), and reported home connections exhibited far more asymmetry and lower bandwidth (6.3 Mbps/1.6 Mbps). Public networks exhibited less download bandwidth but more symmetry (3.4 Mbps/2.3 Mbps).

We saw less variation in the aggregate perspective for quiescent latency. Sessions reported as run at work had an average latency of 100 ms, while home networks experienced 120 ms and public networks 180 ms of latency.

Network Uplink Buffering. A known problem [9] confirmed by *Netalyzr* concerns the substantial over-buffering present in the network, especially in end-user access devices. *Netalyzr* attempts to measure this by recording the amount of delay induced by the high-bandwidth burst of traffic once it exceeds the actual bandwidth obtained. We then infer the buffer capacity as equal to the sustained sending rate multiplied by the additional delay induced by this test. Since the test uses UDP, no back-off comes into play to keep the buffer from completely filling, though we note that *Netalyzr* cannot determine whether the buffer did indeed actually fill to capacity.

When plotting measured upload bandwidth vs. inferred upload buffer capacity (Figure 6), several features stand out. First, we note that because we keep the test short in order to not induce excessive load on the user's link, sometimes *Netalyzr* cannot completely fill the buffer, leading to noise, which also occurs when the bandwidth is quite small (so we do not have a good "quiescence" baseline). Next, horizontal banding in the figure reflects commonly provided levels of service.

Most strikingly, we observe frequent instances of very large buffers. Vertical bands reflect common buffer sizes, which we find fall into powers of two, with many sessions exhibiting buffers of 128 KB or 256 KB in size. Even with a relatively fast 8 Mbps uplink, such buffers can easily induce 250 ms of additional latency during file transfers. For a not atypical 1 Mbps uplink, such buffers translate into well over 1 sec queueing delays.

We can leverage the biases in our data to partially validate these results. By examining only Comcast customers, we would naturally expect only one or two buffer sizes to predominate, due to more homogeneous hardware deployments—and indeed the Ruthann figure for just Comcast manifests sizes mainly at 128 KB and 256 KB. In this figure, another more subtle feature stands out with the small cluster that lies along a diagonal. Its presence suggests that a small number of customer have access modems that size their buffers directly in terms of time, rather than memory.

In both plots, the scattered values above 256 KB that lack any particular power-of-two alignment suggest the possible existence of other buffering processes in effect for large UDP transfers. For example, we have observed that some of our notebook wireless connections occasionally experience larger delays during this test apparently because the notebook buffers packets at the wireless interface (perhaps due to use of ARQ) to recover from wireless congestion.

Yet even given noise introduced by other sources, the conclusion is inescapable: over-buffering is endemic in access devices, and they would significantly benefit from dynamically sized buffers that introduce only a fixed delay before dropping packets.

Packet Duplication, Reordering, Outages, and Corruption. The bandwidth tests deliberately stress the network, not only to test the buffer capacity but to induce duplication or reordering. For these tests, the bottleneck point receives 1000 B packets at up to 2x the maximum rate of the bottleneck. Only 1% of the uplink tests exhibited packet duplication, while 16% included some reordering. For downlink tests, 2% exhibited duplication and 33% included reordering. The prevalence of reordering qualitatively matches considerably older results [2]; more direct comparisons are difficult because the inter-packet spacing in our tests varies, and reordering rates fundamentally depend on this spacing.

In addition, the RELEASE data includes the background monitoring process that enables us to check for transient outages. We define an outage as a period with a loss of ≥ 3 background test packets (sent at 5 Hz) in a row. We find fairly frequent outages, with 9% of sessions experiencing one or more such events (45% of these reflect single loss bursts, while 28% included ≥ 5 bursts). These burst are generally short, with 48% of sessions with losses having outages ≤ 1 sec.

We also find a significant correlation between such bursts and whether the user reported use of a wireless vs. wired network, with 10% of the former sessions exhibiting at least one outage, versus only 5% of the wired sessions.

Finally, analysis of the server-side packet traces finds no instances of TCP or IP checksum errors. We do see UDP checksum errors at an overall rate of about $1.6 \cdot 10^{-5}$, but these are heavily dominated by bursts experienced by just a few systems. The presence of UDP errors but not TCP might suggest use of selective link-layer checksum schemes such as UDP Lite.

5.3 Service Reachability

Table 1 summarizes the prevalence of service reachability for the application ports *Netalyzr* measures. As explained above, for TCP services we can distinguish between blocking (no successful connection), application-aware connectivity (established connection terminated when our server's reply violates the protocol), and proxying (we observe altered requests/responses). For UDP services we cannot in general distinguish the second case due to the lack of explicit connection establishment.

The first four entries likely reflect ISP security policies in terms of limiting exposure to services well-known for vulnerabilities and not significantly used across the wide-area (first three) or to thwart some forms of email spam (SMTP). For this latter, the fraction of blocking in fact appears lower than expected, suggests that many ISPs may employ dynamic blocking for SMTP or other methods to fight bot infections, rather than wholesale blocking of all SMTP.

The prevalence of blocking and termination ("BLOCKED") for FTP, however, likely arises as an artifact of NAT usage: because FTP uses a separate data

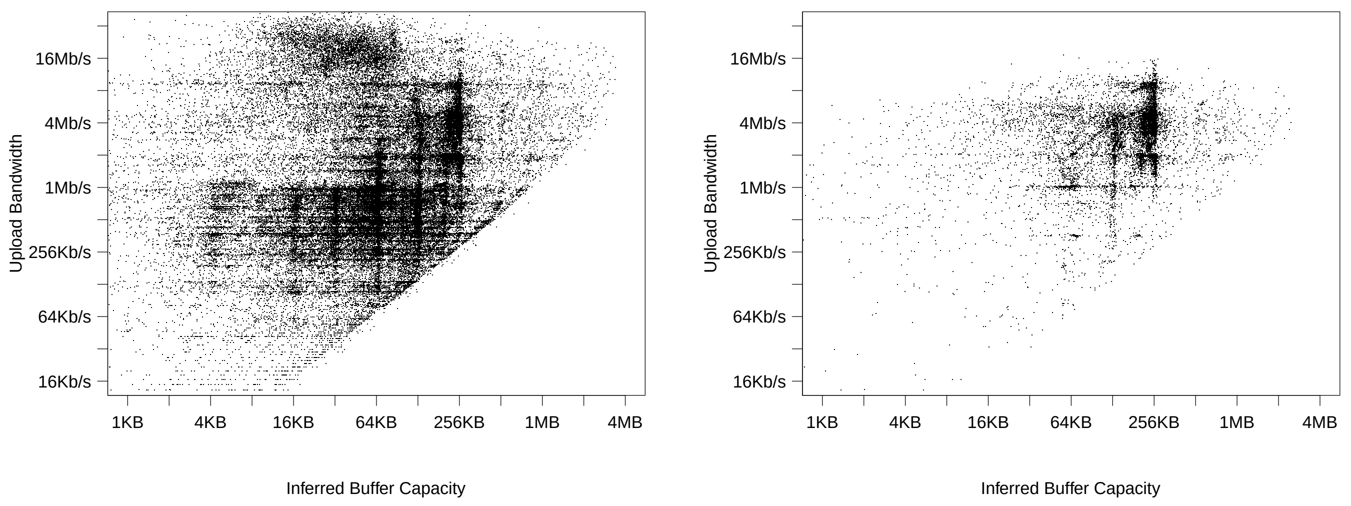


Figure 6: Inferred upload packet-buffer capacity (x-axis) vs. bandwidth (y-axis), for all sessions (left) and Comcast (right).

channel, many NATs implement FTP proxies, which presumably terminate our FTP probing when observing a protocol violation in the response from our server.

Somewhat surprising is the prevalence of blocking for `1434/udp`, used by the Slammer worm of 2003. Likely these blocks reflect legacy countermeasures that have remained in place for years even though Slammer no longer poses a significant threat.

The large fraction of terminated or proxied POP3 connections appears due to in-host anti-virus software that attempts to relay all email requests. In particular, we can identify almost all of the proxying as due to AVG anti-virus because it alters the banner in the POP3 dialog. We expect that the large number of terminated IMAP connections has a similar explanation.

We found the prevalence of terminated SIP connections surprising. Apparently numerous NATs and Firewalls are SIP-aware and take umbrage out our echo server's protocol violation. We learned that this blocking can even occur without the knowledge of the network administrators—a *Netalyzr* run at a large university flagged the blockage, which came as a surprise to the operators, who removed the restriction once we reported it.

Finally, services over TLS (particularly HTTPS, `443/tcp`) are generally unmolested in the network, as expected given the end-to-end security properties that TLS provides. Thus clearly if one wishes to construct a network service resistant to network disruption, tunneling it over over HTTPS should prove effective.

5.4 DNS Measurements

Selected DNS Server Properties. We measured several DNS server properties of interest, including glue policy, IPv6 queries, EDNS, and MTU. Regarding the first, most resolvers behave conservatively, with only 22% accepting any glue records present in the Additional field, and those only doing so for records for subdomains of the authoritative server. Similarly, only 25% accept A records corresponding to CNAMEs contained in the reply. On the other hand, resolvers much more readily (63%) accept glue records when the glue records refer to authoritative nameservers.

We find `0x20` usage scarce amongst resolvers (1.8% of sessions). However, only 4% removed capitalizations from requests, which bodes well for `0x20`'s deployability. Similarly, only a minuscule number of sessions incorrectly cached a 0-TTL record, and none cached a 1 sec TTL record for two seconds.

We quite commonly observe requests for AAAA (IPv6) records (12% of sessions), perhaps largely due to a common Linux default rather than a resolver property, as 38% of sessions with a Linux-related User-Agent requested AAAA records.

The prevalence of EDNS and DNSSEC in requests is significant but not universal, due to BIND's default behavior of requesting DNSSEC data in replies even in the absence of a configured root of trust. 51% of sessions used EDNS-aware DNS resolvers, with 48% of sessions DNSSEC-enabled. Most cases where we observe an advertised MTU show the BIND default of 4096 B (94%), but some other MTUs also occur, notably 512 B (3.4%), 2048 B (1.7%) and 1280 B (0.3%)

The prevalence of DNSSEC-enabled resolvers does not mean transition to broad use of DNSSEC will prove painless, however. For EDNS sessions with an advertised MTU of $\geq$ 1800 B, 14% failed to fetch the large EDNS-enabled reply and 2.1% for the medium-sized one. This finding suggests a common failure where the DNS resolver is connected through a network that either won't carry fragmented UDP traffic or assumes that DNS replies never exceed 1500 B (since `edns_medium` is unlikely to be fragmented). Since

Service	Port	Interference (%) Blocked	Closed	Proxied
NetBIOS	139 T	51.0	1.0	
SMB	445 T	50.3	1.0	
RPC	135 T	46.2	1.2	
SMTP	25 T	25.7	8.2	1.0
FTP	21 T	20.0	3.7	0.1
MSSQL	1434 U	11.1		
SNMP	161 T	7.4	0.2	
BitTorrent	6881 T	6.7	0.5	
AuthSMTP	587 T	6.6	0.2	0.7
SecureIMAP	585 T	6.2	0.2	
Netalyzr Echo	1947 T	6.1		
SIP	5060 T	5.8	4.9	
SecureSMTP	465 T	5.7	0.3	<0.1
PPTP Control	1723 T	5.5	5.1	<0.1
OpenVPN	1194 T	5.3	0.2	
DNS	53 T	5.2	0.8	
IMAP/SSL	993 T	5.1	0.2	<0.1
TOR	9001 T	5.0	0.2	
POP3/SSL	995 T	5.0	0.3	<0.1
IMAP	143 T	5.0	6.7	0.2
POP3	110 T	4.0	7.4	6.1
SSH	22 T	3.6	0.1	<0.1
HTTPS	443 T	2.3	0.4	<0.1
HTTP	80 T		3.8	5.3

Table 1: Reachability for services examined by *Netalyzr*, for all attempted connections. "Blocked" reflects failure to connect to the servers, "Closed" are cases where an in-path proxy or firewall terminated the established connection after the request was sent, likely due to a protocol violation. "Proxied" indicates cases where a proxy revealed its presence through its response, excluding the "closed" cases. Omitted values reflect zero occurrences.

DNSSEC replies will likely exceed 1500 B, the prevalence of this problem suggests a potentially serious deployment issue that will require changes to the resolver logic.

The Release data includes a full validation of DNS MTU up to 4 KB. We find that despite not advertising a large MTU, almost all sessions (95%) used a resolver capable of receiving messages over 512 B. However, a significant number of sessions (16%) exhibited a measured DNS MTU of 1472 B, suggesting an inability to receive fragmented traffic. This even occurred for 11% of sessions that explicitly advertised an explicit EDNS MTU $>$ 1472 B.

A similar problem exists in the clients themselves, but often due to a different cause. When the client directly requests `edns_large`, `edns_medium`, and `edns_small` from the server, 14.5%/4.5%/1.3% failed, respectively. This suggest two additional difficulties: network devices assuming DNS replies do not exceed 512 B (both `edns_large` and `edns_medium` fail) or networks that do not handle EDNS at all (all three fail).[6] We find this high failure rate quite problematic, as the experiences with NXDOMAIN wildcarding and DNS lookups (§ 5.4) clearly demonstrate that DNS resolvers can behave in an adversarial manner. Thus, sound DNSSEC validation requires implementation on the end host's stub resolver to achieve end-to-end security, which requires that end hosts can receive large, EDNS-enabled DNS messages.

Another concern comes from the continued lack of DNS port randomization [6]. This widely publicized vulnerability was over a year old when we first released *Netalyzr*, but 5% of sessions used monotone or fixed ports in DNS requests, Manual examination suggests that these cases mostly reflect small resolvers run by individuals or institutions— no major ISP showed significant problems with this test.

In terms of DNS performance, it appears that DNS resolvers may constitute a bottleneck for many users. 9% of the sessions required 300 ms more time to look up an name within our domain versus the base round-trip time to our server, and 4.6% required more than 600 ms. (We can account for likely at most 100 ms of the increase due to our DNS server residing at a different location than the back-end servers.)

When the user's resolver accepted glue records (53% of sessions), we could directly measure the performance of DNS requests answered from the resolver's cache. Surprisingly, 10% of such sessions required over 200 ms to look up *cached* items, and 3.7% required over 500 ms. Such high latency suggests a considerable distance between the client and the resolver, and for example we found 15% of sessions that used OpenDNS required over 200 ms for cached answers compared to 9% for non-OpenDNS sessions.

Finally, we note that numerous resolvers reflect BIND implementations: 32% of the sessions used resolvers that match a BIND fingerprint in terms of glue policy, CNAME processing, and request options.

NXDOMAIN Wildcarding. We find NXDOMAIN wildcarding quite prevalent among *Netalyzr* users. 28% performing this test found NXDOMAIN wildcarding for `www.`*nonce*`.com`. Even excluding users of both OpenDNS (which wildcards by default) and Comcast (which started wildcarding during the course of our measurements), 21% show NXDOMAIN wildcarding. This wildcarding will disrupt features such as Firefox's address bar, which prepends `www.` onto failed DNS lookups before defaulting to a Google search.

Of further concern is the number of users affected by NXDOMAIN wildcarding that causes broader collateral damage. Excluding Comcast and OpenDNS users[7], 44% of sessions with NXDOMAIN wildcarding also showed wildcarding for non-`www` names. Wildcarding all addresses mistakenly assumes that only web browsers will generate name lookups.

DNS Proxies, NATs, and Firewalls. Many NATs and

[6] We note that the failures we observe could instead be due to heavy packet loss. However, each failure would require five consecutive losses just after a successful non-EDNS query. Furthermore, such failures should not particularly favor one type of query over another, yet we observe only 0.09% of sessions for which `edns_medium` succeeded while `edns_small` failed.

[7] Comcast only wildcards names beginning with `www.`, while the default OpenDNS behavior wildcards all invalid names.

	All Lookups (%)		OpenDNS (%)	
Domain	Failed	Blocked	Failed	Changed
www.nationwide.co.uk	2.3	<0.01	1.6	0.01
ad.doubleclick.net	1.5	1.99	1.6	1.27
www.citibank.com	1.3	<0.01	0.9	0.03
windowsupdate.microsoft.com	0.7	0.02	0.5	0.01
www.microsoft.com	0.7	<0.01	0.4	0.01
mail.yahoo.com	0.6	0.02	0.4	0.17
mail.google.com	0.4	0.02	0.3	0.13
www.paypal.com	0.4	0.04	0.1	0.03
www.google.com	0.3	0.01	0.2	76.71
www.meebo.com	0.3	0.03	0.2	0.79

Table 2: Reliability of DNS lookups for 10 selected names (reflecting 107,000 sessions, 11,000 of which used OpenDNS).

firewalls are DNS-aware and may act as DNS proxies. Although we find 99% able to perform direct DNS queries, 11% of these sessions show evidence of a DNS-aware network device, where a non-DNS test message destined for `53/udp` failed (but proper DNS messages succeeded). Far fewer networks contain mandatory DNS proxies, with only 1.2% of DNS-capable sessions indicating such in the form of changed DNS transaction ID.

Although most NATs don't automatically proxy DNS, most contain DNS proxies. We found 67% of the NATs would forward a DNS request to the server (with this measurement restricted to the cases where *Netalyzr* correctly guessed the gateway IP address). Of these, only 1.8% of the sessions contained their own recursive resolver, rather than forwarding the request to a different recursive resolver. Finally, although rare the number of NATs providing open DNS resolution *externally* accessible is still significant. When queried by our server, 4.4% of the NATed sessions forwarded the query to our DNS servers. Such systems can be used both for DNS amplification attacks and to probe the ISP's resolver.

DNS Reliability of Important Names. DNS lookups can fail for a variety of reasons, including an unreliable local network, problems in the DNS resolver infrastructure, and failures in the DNS authorities or paths between the resolver and authority. Table 2 characterizes some failure modes for 10 common domain names. For general lookups, "failure" reflects a negative result or an exception returned to the applet by `InetAddress.getByName()`, or a 20 sec timeout expiring. "Blocked" denotes the return of an obviously invalid address (such as a loopback address).

We explored reliability for OpenDNS users in more detail. OpenDNS not only performs NXDOMAIN wildcarding, but also wildcards SERVFAIL (for the latter, returning the IP address of `hit-servfail.opendns.com`). Thus for queries generated by OpenDNS users we can distinguish between failures which occur between the client and OpenDNS, and server failures due to problems between OpenDNS and the DNS authority for the domain. OpenDNS also includes powerful features to change other names. For `www.google.com`, OpenDNS will act as a proxy by default, redirecting users transparently through an OpenDNS server. For other domains, OpenDNS allows users or domain administrators to block "undesirable" names, with OpenDNS instead returning the address of various blocking servers.

Some behavior immediately stands out. First, regardless of resolver, we observe significant unreliability of DNS to the client, due to packet loss and other issues. Caching also helps, as highly popular names have a failure rate substantially less than that for less common names. For example, compare the failure rate of `www.nationwide.co.uk` to that of `mail.google.com`, for which we presume resolvers will have the latter cached significantly more often.

Second, we observe high reliability for the DNS authorities of the names we tested. Only 14 sessions had OpenDNS returning the SERVFAIL wildcard in response to a legitimate query. (One such session showed many names failing to resolve, obviously due to a problem with OpenDNS's resolver rather than the authority servers.)

Third, we can see the acceptance of DNS as a tool for network management and control. All but the `www.google.com` case for OpenDNS represent user or site-admin configured redirections. For domains like `mail.yahoo`, the common change is to return a private Internet address, most likely configured in the institution's DNS server, while blocking of `ad.doubleclick` commonly uses nonsense addresses (such as `0.0.0.0`), which may reflect resolution directly from the user's hosts file (as suggested on some forum discussions on blocking `ad.doubleclick`).

The DNS results also included two strains of maliciousness. The first concerns an ISP (Wide Open West) that commonly returned their own proxy as an answer for www.google.com or search.yahoo.com (but not sites such as mail.google.com or www.yahoo.com). Deliberately invalid requests to these proxies return a reference to "phishing-warning-site.com", a domain parked with GoDaddy. We also observed similar behavior for customers of sigecom.net, cavtel.net, rcn.net, fuse.net, and o1.com.

Second, in a few dozen sessions we observed malicious DNS resolvers due to malcode having reconfigured an infected user's system settings. These servers exhibit two signatures: (*i*) malicious resolution for windowsupdate.microsoft.com, which instead returns an arbitrary Google server to disable Windows Update, and (*ii*) sometimes a malicious result for ad.doubleclick.net. In these latter (less frequent) instances, these ad servers insert malicious advertisements that replace the normal ads a user sees with ones for items like "ViMax Male Enhancement" [12].

5.5 HTTP Proxying and Caching

8.6% of all sessions show evidence of HTTP proxying. Of these, 32.4% had the browser explicitly configured to use an

HTTP proxy, as the server recorded a different client-side IP address only for HTTP connections made via Java's HTTP API. More interestingly, 90.8% of proxied sessions showed evidence of a mandatory in-path proxy for all HTTP traffic. (These are not mutually exclusive—the overlap is explained by users that are double-proxied.) We detect such proxies by several mechanisms, including changes to headers or expected content, requests from a different IP address, or in-network caching. A proxy may announce its location through `Via` or `X-Cache-Lookup` response headers. The applet follows such clues by attempting a direct connection to such potential proxies with instructions to connect to our back-end server, which succeeded in 11.0% of proxied sessions. The reported names can be net-local hostnames (such as "`CLT-PRXY-04`" or "`Bastion`") or fully qualified domain names. Of the announced proxies, 25.2% used a domain matching that of the client's PTR record.

We rarely observed caching of our 67 KB image (5.3% of sessions cached at least one version of it). Manual examination reveals that such caching most commonly occurred in wireless hotspots and corporate networks. Two South African ISPs used in-path caching throughout, presumably to reduce bandwidth costs and improve latency.

The infrequency of such caches perhaps represents a blessing in disguise, as they often get it wrong. A minor instance concerns the 55.8% of caches that cached the image we specified it as weakly uncacheable (no cache-specific HTTP headers). More problematic are the 37.8% that cached the image despite strong uncacheability (use of headers such as `Cache-control: no-cache,no-store`, a fresh `Last-Modified` timestamp expiring immediately). Finally, 5.3% of these broken caches failed to cache a highly cacheable version of the image (those with `Last-Modified` well in the past and `Expires` well into the future, or with an `ETag` identifier). Considering that 41.5% of all HTTP-proxied connections did not gain the benefits of caching legitimately cacheable content, we identify considerable unrealized savings.

Network proxies seldom transcode the raw images during this test, but it does occur. 0.05% of the sessions showed transcoding of one or more of the fetched images, detected as a returned result smaller than the expected length but > 10 KB. Manual examination of a few cases verified that the applet received a proper HTTP response for the image with a reduced `Content-Length` header, and thus the network did indeed change the image rather than merely truncate.

In-path processes also only rarely interrupt file transfers. Only 0.7% of all sessions failed to correctly fetch the `.mp3` file and 1.0% for the `.exe`. Slightly more, 1.3%, failed to fetch the `.torrent` file, suggesting that some networks filter on file type. However, 10% filtered the EICAR test "virus", suggesting significant deployment of either in-network or host-based AV. As only 0.36% failed to fetch all four test-files, these results do not reflect proxies that block all of the tests.

5.6 ISP Profiles

Table 3 illustrates some of the policies that *Netalyzr* observed for the 15 most common ISPs. As mentioned above, the relative lack of SMTP blocking amongst several major ISPs could reflect that some IPS perform dynamic response to block spam-bots in their network. Likewise, a few ISPs do not appear to filter Windows traffic outbound from customer connections. They might however block these ports inbound, which we cannot determine since *Netalyzr* does not perform inbound scanning. cannot determine if these ports are unblocked on inbound traffic.

Another characteristic we see reflects early design decisions still in place today. Although DSL always offered the ability to provide direct Ethernet connections, many DSL providers initially offered PPPoE connections rather than IP over Ethernet [32]. DOCSIS-based cable-modems, however, always used IP-over-Ethernet. We can see the effects of this transition for Verizon customers, as only 9% of Verizon customers whose reverse name suggests they are FiOS (fiber to the home) customers manifest the PPPoE MTU, while 69% of the others do.

A final trend concerns the growth of NXDOMAIN wildcarding, especially ISPs wildcarding all names rather than just `www` names. During *Netalyzr*'s initial release, Comcast had yet to implement NXDOMAIN wildcarding, but began wildcarding during Fall 2009.

We also confirmed that the observed policies for Comcast match their stated policies. Comcast has publicly stated that they will block outbound traffic on the Windows ports, and may block outbound SMTP with dynamic techniques [7]. When they began widespread deployment of their wildcarding, they also stated that they would only wildcard `www` addresses, but we did observe the results of an early test deployment that wildcarded all addresses for a short period of time.

6. RELATED WORK

There is a substantial existing body of work on approaches for measuring various aspects of the Internet. Here we focus on those related to our study in the nature of the measurements conducted or how data collection occurred.

Network performance. Dischinger et al. studied network-level performance characteristics, including link capacities, latencies, jitter, loss, and packet queue management [9]. They used measurement packet trains similar to ours, but picked the client machines by scanning ISP address ranges for responding hosts, subsequently probing 1,894 such hosts autonomously. In 2002 Saroiu et al. studied similar access link properties as well as P2P-specific aspects of 17,000 Napster file-sharing nodes and 7,000 Gnutella peers [24]. They identified probe targets by crawling the P2P overlays, and identified a large diversity in bandwidth (only 35% of hosts exceeded an upload bandwidth of 100Kb/s,

						DNS			
			BLOCKED (%)			WILDCARDING		PPPOE	
ISP	SESSIONS	COUNTRY	WIN	SMTP	MSSQL	TYPE	%	(%)	MEDIUM
Comcast	13,403	US	99	8		www	33		Cable
RoadRunner	5,544	US				www	63		Cable
Verizon	3,854	US	7	14		www	83	33	DSL/Fiber
SBC	2,938	US	51	73					DSL
Deutsche Telekom	2,523	DE	76			all	48	56	DSL
Cox Cable	2,187	US	92	77	88				Cable
Charter Comm.	1,665	US	95	23	32	all	62		Cable
Qwest	1,334	US	18	6		all	51	70	DSL
BE Un Limited	1,276	UK		49					DSL
Arcor	1,139	DE	33					6	DSL
BellSouth	1,080	US	62	69	96			16	DSL
Alice DSL	1,032	DE	30			www	62	69	DSL
Shaw Cable	1,018	US	6	61					Cable
telecomitalia.it	918		7		14	all	64	65	
Optimum Online	866	US	97	78		www	77		Cable

Table 3: Policies detected for the top 15 ISPs. We indicate blocking when $>$ 5% of sessions manifested outbound filtering, particularly for Windows services (TCP 135/139/445). We infer PPPoE from path MTUs of 1492 B.

8% exceeded 10Mbps, between 8% and 25% used dial-up modems, and at least 30% had more than 3Mb/s downstream bandwidth) and latency (the fastest 20% of hosts exhibited latencies under 70ms, the slowest 20% exceeded 280ms). Maier et al. analyzed residential broadband traffic of a major European ISP [17], finding that round-trip latencies between users and the ISP's border gateway often exceed that between the gateway and the remote destination (due to DSL interleaving), and that most of the observed DSL lines used only a small fraction of the available bandwidth. Various techniques have been developed for measuring bandwidth or latency directly, including Sting [25], IGI [14], YAZ [27], and Iperf [20]. In addition, numerous websites offer throughput tests aimed at home users [31]. We could in principle incorporate the techniques underlying some of these tools into our measurements, but have not at this point in order to keep our main focus on ways in which users have their connectivity restricted or shaped, rather than end-to-end performance.

Network neutrality. Several studies have looked at the degree to which network operators provide different service to different types of traffic. Dischinger et al. provided a downloadable tool enabling users detect whether their ISP imposes restrictions on BitTorrent traffic. They studied 47,000 sessions conducted using the tool, finding that around 8% of the users experienced BitTorrent blocking [10]. Bin Tariq et al. devised NANO, a distributed measurement platform, to detect statistically and policy-agnostically whether a given ISP intentionally or accidentally causes degraded performance for specific classes of service [28]. They evaluate their system in Emulab, using Click configurations to synthesize "ISP" discrimination, and source synthetic traffic from PlanetLab nodes. Beverly et al. leveraged the "referral" feature of Gnutella to conduct TCP port reachability tests from 72,000 unique Gnutella clients, finding that Microsoft's network filesharing ports are frequently blocked, and that email-related ports are more than twice as likely to be blocked as other ports [3]. Reis et al. used JavaScript-based "web tripwires" to detect modifications to HTTP-borne HTML documents [23]. Of the 50,000 unique IP addresses from which users visited their test website, approximately 0.1% experienced content modifications. Weaver et al. examined properties of TCP RST packets observed in the network traffic of four sites [30]. They identified the operational specifics and apparent policy goals underlying a set of reset injection products, including filtering implemented by the "Great Firewall of China". Nottingham provided a cache fidelity test for XMLHttpRequest implementations [21], analyzing a large variety of caching properties including HTTP header values, content validation and freshness, caching freshness, and variant treatment. NetPolice [35] measured traffic differentiation in 18 large ISPs for several popular services in terms of packet loss, using multiple end points inside a given ISP to transmit application-layer traffic to destinations using the same ISP egress points. They found clear indications of preferential treatments for different kinds of service. Finally, subsequent to *Netalyzr*'s release, Huang et al. released a network tester for smartphones to detect hidden proxies and service blocks using methodology inspired by *Netalyzr* [15].

Address fidelity. Casado and Freeman investigated the reliability of using a client's IP address—as seen by a public server—in order to identify the client [5]. Their basic methodology somewhat resembles ours in that they used active web content to record and measure various connection properties, but also differs significantly with regard to the process of users running the measurements. They instrumented several websites to serve an `iframe` "web bug", leading to narrow data collection—users had to coincidentally visit those sites, and remained oblivious to the fact that measurement was occurring opportunistically. They found that 60% of the observed clients reside behind NATs, which typically translated no more than seven clients, while 15% of the clients arrived via HTTP proxies, often originating from a diverse geographical region. Maier et al. [17] found that DHCP-based address reuse is frequent, with 50% of all addresses being assigned at least twice per day. Finally, Beverly and Bauer's Spoofer Project [4] employed a downloadable measurement client to measure the extent to which end systems can spoof IP source addresses. They analyzed an extensive longitudinal dataset, finding that through the period of study a significant minority of clients could perform arbitrary spoofing.

7. SUMMARY

The *Netalyzr* system demonstrates the possibility of developing a browser-based tool that provides detailed diagnostics, discovery, and debugging for end-user network connectivity. Visitors who ran the *Netalyzr* applet conducted 112,000 measurement sessions from 86,000 public IP Addresses. *Netalyzr* both reveals specific problems to individual users and forms the foundation for a broad survey of edge-network behavior. Some systemic problems revealed include difficulties with fragmentation, the unreliability of path MTU discovery, restrictions on DNSSEC deployment, legacy network blocks, frequent over-buffering of access devices, poor DNS performance for many clients, and deliberate manipulations of DNS results. The tool remains in active use and we aim to support it indefinitely as an ongoing service for illuminating edge network neutrality, security, and performance.

8. ACKNOWLEDGEMENTS

We are deeply grateful to the Netalyzr users for enabling this study and to our beta testers for the insightful comments and feedback. We would particularly like to thank Mark Allman, Paul Barford, Scott Bradner, John Brzozowski, Randy Bush, Niels Bakker, Richard Clayton, Chris Cowart, Keith Dawson, Adrian Dimcev, Holger Dreger, Brandon Enright, Kevin Fall, Carrie Gates, Andrei Gurtov, Mark Handley, Theodore Hong, Kelly Kane, Matthew Kogan, Keith Medcalf, Thomas Narten, Michael Ross, Wooter Wijngaards, and Richard Woundy. We thank Amazon.com for supporting our EC2 deployment. This work was supported by the National Science Foundation under grants NSF CNS-0722035, NSF-0433702, and CNS-0905631.

9. REFERENCES

[1] Amazon.com. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/, August 2006.
[2] J. Bennett, C. Partridge, and N. Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM Transactions on Networking (TON)*, 7:789–798, 1999.
[3] R. Beverly, S. Bauer, and A. Berger. The Internet's Not a Big Truck: Toward Quantifying Network Neutrality. In *Proc. PAM*, 2007.
[4] R. Beverly, A. Berger, Y. Hyun, and k claffy. Understanding the efficacy of deployed internet source address validation filtering. In *Proc. IMC*, 2009.
[5] M. Casado and M. Freedman. Peering through the Shroud: The Effect of Edge Opacity on IP-based Client Identification. In *Proc. NSDI*, 2007.
[6] Chad R. Dougherty. CERT Vulnerability Note VU 800113: Multiple DNS implementations vulnerable to cache poisoning, July 2008.
[7] What ports are blocked by Comcast High-Speed Internet? http://lite.help.comcast.net/content/faq/What-ports-are-blocked-by-Comcast-High-Speed-Internet.
[8] D. Dagon, M. Antonakakis, P. Vixie, T. Jinmei, and W. Lee. Increased DNS Forgery Resistance Through 0x20-bit Encoding. In *Proc. CCS*, 2008.
[9] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu. Characterizing Residential Broadband Networks. In *Proc. IMC*, 2007.
[10] M. Dischinger, A. Mislove, A. Haeberlen, and K. Gummadi. Detecting BitTorrent Blocking. In *Proc. IMC*.
[11] R. Erzs and R. Bush. Clarifications to the DNS Specification. RFC 2181, IETF, July 1997.
[12] M. Fauenfelder. How to get rid of Vimax ads. http://boingboing.net/2009/01/16/how-to-get-rid-of-vi.html, January 2009.
[13] R. Giobbi. CERT Vulnerability Note VU 435052: Intercepting proxy servers may incorrectly rely on HTTP headers to make connections, February 2009.
[14] N. Hu and P. Steenkiste. Evaluation and characterization of available bandwidth probing techniques. *IEEE Journal on Selected Areas in Communications*, 21(6):879–894, 2003.
[15] J. Huang, Q. Xu, B. Tiwana, and M. Mao. The UMich Smartphone 3G Test. http://www.eecs.umich.edu/3gtest/.
[16] C. Kent and J. Mogul. Fragmentation considered harmful. *ACM SIGCOMM Computer Communication Review*, 25(1):87, 1995.
[17] G. Maier, A. Feldmann, V. Paxson, and M. Allman. On dominant characteristics of residential broadband internet traffic. In *Proc. IMC*, 2009.
[18] M. Mathis and J. Heffner. Packetization Layer Path MTU Discovery. RFC 4821, IETF, March 2007.
[19] Mozilla. Effective TLD names. http://mxr.mozilla.org/mozilla-central/source/netwerk/dns/src/effective_tld_names.dat.
[20] NLANR/DAST. Iperf. http://iperf.sourceforge.net/, March 2008.
[21] M. Nottingham. XMLHttpRequest Caching Tests. http://www.mnot.net/javascript/xmlhttprequest/cache.html, December 2008.
[22] V. Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *ACM SIGCOMM*

Computer Communication Review, 31(3):38–47, 2001.

[23] C. Reis, S. Gribble, T. Kohno, and N. Weaver. Detecting In-Flight Page Changes with Web Tripwires. In *Proc. NSDI*, 2008.

[24] S. Saroiu, P. Gummadi, S. Gribble, et al. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking*, volume 2002, page 152, 2002.

[25] S. Savage. Sting: a TCP-based network measurement tool. In *Proc. USITS*, 1999.

[26] P. Savola. MTU and Fragmentation Issues with In-the-Network Tunneling. RFC 4459, 2006.

[27] J. Sommers, P. Barford, and W. Willinger. Laboratory-based calibration of available bandwidth estimation tools. *Microprocessors and Microsystems*, 31(4):222–235, 2007.

[28] M. Tariq, M. Motiwala, N. Feamster, and M. Ammar. Detecting network neutrality violations with causal inference. In *Proc. Emerging Networking Experiments and Technologies*, 2009.

[29] P. Vixie. Extension Mechanisms for DNS (EDNS0). RFC 2671, IETF, August 1999.

[30] N. Weaver, R. Sommer, and V. Paxson. Detecting Forged TCP Reset Packets. In *Proc. NDSS*, 2009.

[31] Web resources. Bandwidth/latency measurement sites. http://www.speedtest.net, http://helpme.att.net/dsl/speedtest/, http://www2.verizon.net/micro/speedtest/hsi/, http://www.dslreports.com/stest.

[32] Wikipedia. Point-to-point protocol over ethernet. http://en.wikipedia.org/wiki/Point-to-Point_Protocol_over_Ethernet, January 2010.

[33] Wikipedia. Usage share of operating systems. http://en.wikipedia.org/wiki/Usage_share_of_operating_systems, January 2010.

[34] Wikipedia. Usage share of web browsers. http://en.wikipedia.org/wiki/Usage_share_of_web_browsers, January 2010.

[35] Y. Zhang, Z. M. Mao, and M. Zhang. Detecting traffic differentiation in backbone ISPs with NetPolice. In *Proc. IMC*, 2009.