TorFlow: Tor Network Analysis

Mike Perry
mikeperry@fscked.org

Tor Tech Report 2009-08-003*
August 7, 2009

Abstract

The Tor Network is a low-latency anonymity, privacy, and censorship resistance network whose servers are run by volunteers around the Internet. This distribution of trust creates resilience in the face of compromise and censorship; but it also creates performance, security, and usability issues. The TorFlow suite attempts to address this by providing a library and associated tools for measuring Tor nodes for reliability, capacity and integrity, with the ultimate goal of feeding these measurements back into the Tor directory authorities.

1 Introduction

The Tor [7] Network is a TCP overlay network whose infrastructure is run entirely by volunteers. It is the largest public anonymity network in the world, consisting of approximately 1500 nodes with a total capacity of approximately 3Gbps, and almost 1Gbps of total exit throughput. In over 6 years of operation, it has never gone down, and has never had a "flag day".

Clients that use the Tor network construct paths called circuits that consist of 3 nodes (guard, middle, and exit) upon which they route multiple TCP streams. The nodes in each circuit are chosen probabilistically according to the maximum bandwidth they claim to observe themselves transmit over a 24 hour period [4]. The Tor client software ensures that no two nodes run by the same (cooperating) operator nor any two nodes from the same /16 netmask are ever chosen in the same circuit [5].

However, the same distributed and heterogeneous nature of the network that gives Tor its strength is also a source of security, performance, and usability issues. The largest barrier to widespread use of the network is performance [10], and the biggest user-visible performance issue is not actually total capacity, but the high variance in circuit performance and non-uniform distribution of network load. [11]

Moreover, there are a non-trivial number of nodes that alter content due to misconfiguration, or much less often, due to malice. This most frequently comes in the form of truncating TCP

*This report was presented at 2nd Hot Topics in Privacy Enhancing Technologies (HotPETs 2009), Seattle, WA, USA, August 2009.

streams or failing DNS, but occasionally presents itself as SSL spoofing or interception by the upstream ISP. On rare occasion, SSH hijacking and web content injection have also been observed.[1].

2 Overview

The TorFlow suite attempts to address these issues with 4 major components. At the core is the TorCtl control port and path selection library, which provides the ability to create paths subject to arbitrary restrictions, measure various aspects of node reliability and performance, and store results in-memory or in a SQL database of your choice.

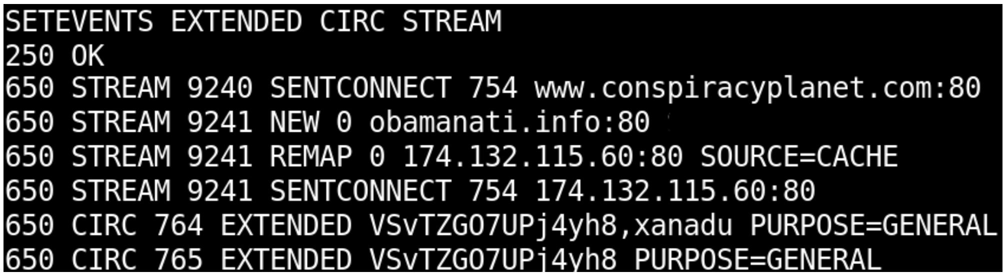
In addition, three network scanners are built on top of this library: SpeedRacer, BuildTimes, and SoaT (Snakes on a Tor). SpeedRacer measures average stream capacity. BuildTimes measures circuit construction speeds and failure rates. SoaT is a multi-protocol Tor exit node scanner, for detecting misconfigured or malicious exit nodes.

3 Path Selection and Measurement Library

The Python-based TorCtl library was initially authored by Nick Mathewson to provide programmatic access to the Tor Control Port protocol [3]. It has since been extended to provide the ability to use modular components to build up custom path selection algorithms and gather statistics on their characteristics.

3.1 Tor Control Port Protocol

The Tor Control Port protocol used by TorCtl is a plaintext TCP-based protocol that provides well-formed information on Tor client status and events and optionally enables control over circuit construction and association of SOCKS streams to individual circuits.



```
SETEVENTS EXTENDED CIRC STREAM
250 OK
650 STREAM 9240 SENTCONNECT 754 www.conspiracyplanet.com:80
650 STREAM 9241 NEW 0 obamanati.info:80
650 STREAM 9241 REMAP 0 174.132.115.60:80 SOURCE=CACHE
650 STREAM 9241 SENTCONNECT 754 174.132.115.60:80
650 CIRC 764 EXTENDED VSvTZGO7UPj4yh8,xanadu PURPOSE=GENERAL
650 CIRC 765 EXTENDED VSvTZGO7UPj4yh8 PURPOSE=GENERAL
```

Figure 1: Example Tor Control Port connection with representative Tor Traffic.

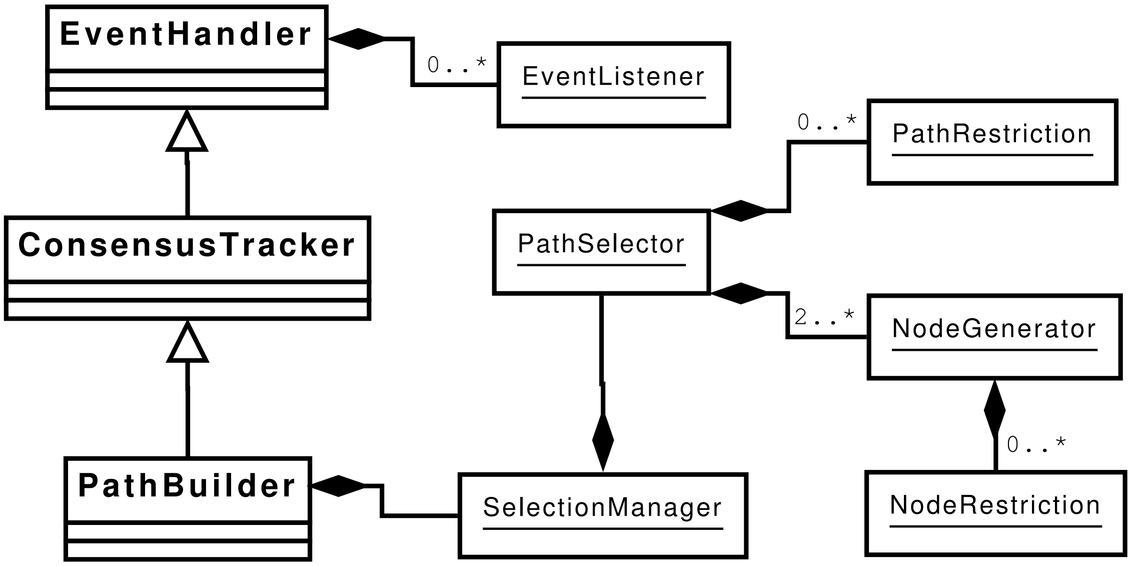


Figure 2: TorCtl Core Class Diagram

3.2 TorCtl Organization

TorCtl provides access to the Tor Control Port at several different layers of abstraction. At the lowest layer is the TorCtl Connection object (not shown), which is used to send commands to the control port and get responses. TorCtl Connections are often associated with a TorCtl EventHandler instance, which is passed objects representing parsed Tor Events.

The EventHandler interface can be implemented directly, but is also extended by the library to provide a ConsensusTracker, which maintains a Python-based representation of the current Tor directory consensus as seen by the Tor client.

If a higher level of control over pathing is desired, programmers can choose to utilize the Path Support routines by using a PathBuilder instance. The PathBuilder is an EventHandler that tracks incoming stream events and builds circuits for them according to the direction of a SelectionManager, which is queried whenever a new path is needed.

The provided SelectionManager breaks this construction down into NodeGenerators managed by a PathSelector. NodeGenerators govern the underlying probability distribution from which nodes are drawn. Uniform, ordered, and Tor-compatible bandwidth-weighted node generators are provided. One NodeGenerator instance exists for each hop in a path of length 2 up to arbitrary N.

Each NodeGenerator has a collection of zero to arbitrary N NodeRestrictions that are used to restrict which nodes are eligible for generation. Restrictions based on node exit policy, directory flags, identity key, country, bandwidth, operating system, Tor version, and percentile rank are provided. Meta-restrictions such as And, Or, Not, and N of M are also provided.

After a candidate path is generated using the NodeGenerators, the PathSelector checks this path for validity against any supplied PathRestrictions. These include Tor's Subnet16, Node-

[1] https://lists.torproject.org/pipermail/tor-talk/2007-January/007352.html

Family, and UniqueNode restrictions. Additionally, single continent, single country, continent changing, country changing, unique country, and unique continent restrictions were contributed by Johannes Renner for his research into geolocational path selection [12].

3.3 EventListener Statistics Support

The TorCtl library provides two main mechanisms for gathering statistics. These are both implemented as EventListeners which can be attached to arbitrary EventHandlers regardless of what is managing path construction and stream servicing.

The first is a hand-coded StatsHandler that computes a series of statistics on circuit creation time and failure reason, and stream capacity and failure reason.

The second is a SQL-based system that stores circuit and/or stream events in SQL tables. The SQL system uses Elixir[2] and SQLAlchemy[3], so the backend database can be any that is supported by SQLAlchemy (which includes just about every modern database backend).

4 Performance Measurements

TorFlow currently utilizes the TorCtl support library towards two main classes of performance measurement: circuit-based and stream-based. The circuit-based measurements are concerned with construction time and reliability. The stream-based measurements are concerned primarily with stream capacity.

4.1 Circuit Construction Speed and Reliability

As stated previously, one of the major issues with Tor is the high variance of performance of nodes and paths. One way to avoid overloaded paths is to simply give up on circuits that take too long to build. The problem is that how long to wait before giving up is heavily dependent on the local link, as it must be traversed three times for each circuit construction due to the telescope-style circuit construction Tor employs. [6].

With this in mind, we created a Google Summer of Code project to determine a probability distribution that could be used to model Tor circuit construction times and then to implement code in the Tor client itself to estimate a particular client's distribution parameters and determine a proper circuit timeout cutoff for that client that would cut out the slowest X% of paths from usage.

Fallon Chen took this project, and started out by creating a utility called BuildTimes using TorCtl. It builds up to N circuits in parallel, subject to configurable TorCtl NodeRestrictions. In addition to collecting TorCtl aggregate statistics, it also logs circuit extend times to plaintext files that can be easily post-processed.

Based on her measurements, we were able to determine that the circuit build times can be loosely modeled as a Pareto distribution, as shown in Figure 3 below. They can also be more tightly modeled with a Fréchet distribution, but the estimators for this distribution are not as

[2] http://elixir.ematia.de/trac/wiki
[3] http://www.sqlalchemy.org/

straightforward as Pareto, and for our purposes the only significant portion of the distribution is the tail, which matches well enough.

Moreover, the major irregularity occurring at 6000ms (and also to a lesser degree at every second) in the histogram has been identified by Karsten Loesing as being a result of our rate limiting algorithm emptying its token buckets in sync across the network at the top of each second as opposed to continuously. When this is addressed, the Pareto fit should improve.

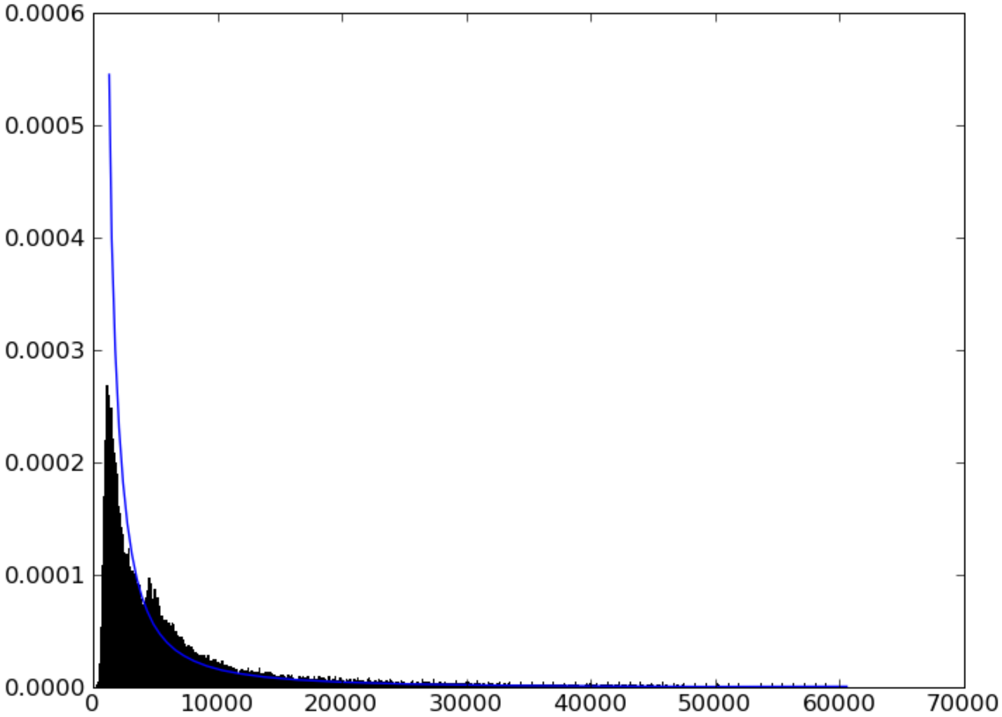


Figure 3: Network-wide bandwidth-weighted circuit build time distribution (ms).

Unfortunately, Fallon's research obligations over the summer prevented her from completing the second half of the project and we have not yet recalibrated Tor's circuit timeout in the client.

4.2 Guard Node Rebalancing

In 2007, an early TorFlow implementation was used to measure circuit responsiveness and reliability of 5% slices of the network (lower percentiles indicate higher advertised bandwidth). Repeated measurement showed that nodes became progressively less responsive and more failure prone as they got slower, up until the 50% mark, at which point the pattern suddenly stopped. This pattern can be seen in the left side of Figures 4 and 5.

This 50% mark was the same point where nodes ceased to be considered for 'guard' status.

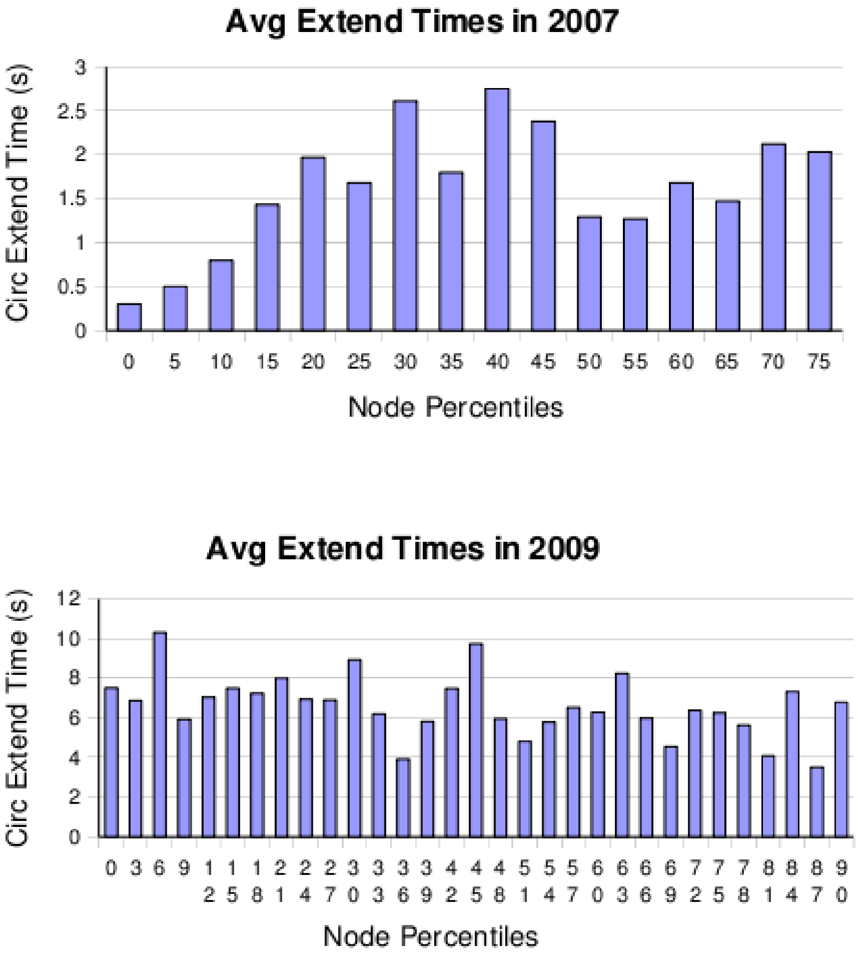


Figure 4: Circuit construction times before and after guard rebalancing.

We eventually discovered that client guard node selection, instead of being weighted based on bandwidth, was actually uniform. We developed a new algorithm to fix this, as well as to properly account for weighting both guards and exits according to their scarcity when being selected for other positions in the network.[4] ,[5]

Without an autoupdater, it took over a year for enough clients to upgrade for the results to be visible in our scans, but it appears that at least among guards, the load is now considerably more uniform.

However, it is obvious that irregularities still remain. Interestingly, the points of very low failure rates in Figure 5 correspond to the periods between 01:00 and 03:00 PST, when most of the US is asleep, and consistently appeared at that time in numerous scans. This seems to suggest we should avoid capacity scans during those hours. It also suggests that circuit reliability may be dependent on load in a non-linear fashion. One potential source for this is CPU load: when Tor detects that is not able to complete crypto operations fast enough, it begins

[4] https://trac.torproject.org/projects/tor/ticket/440
[5] https://lists.torproject.org/pipermail/tor-dev/2007-July/001255.html

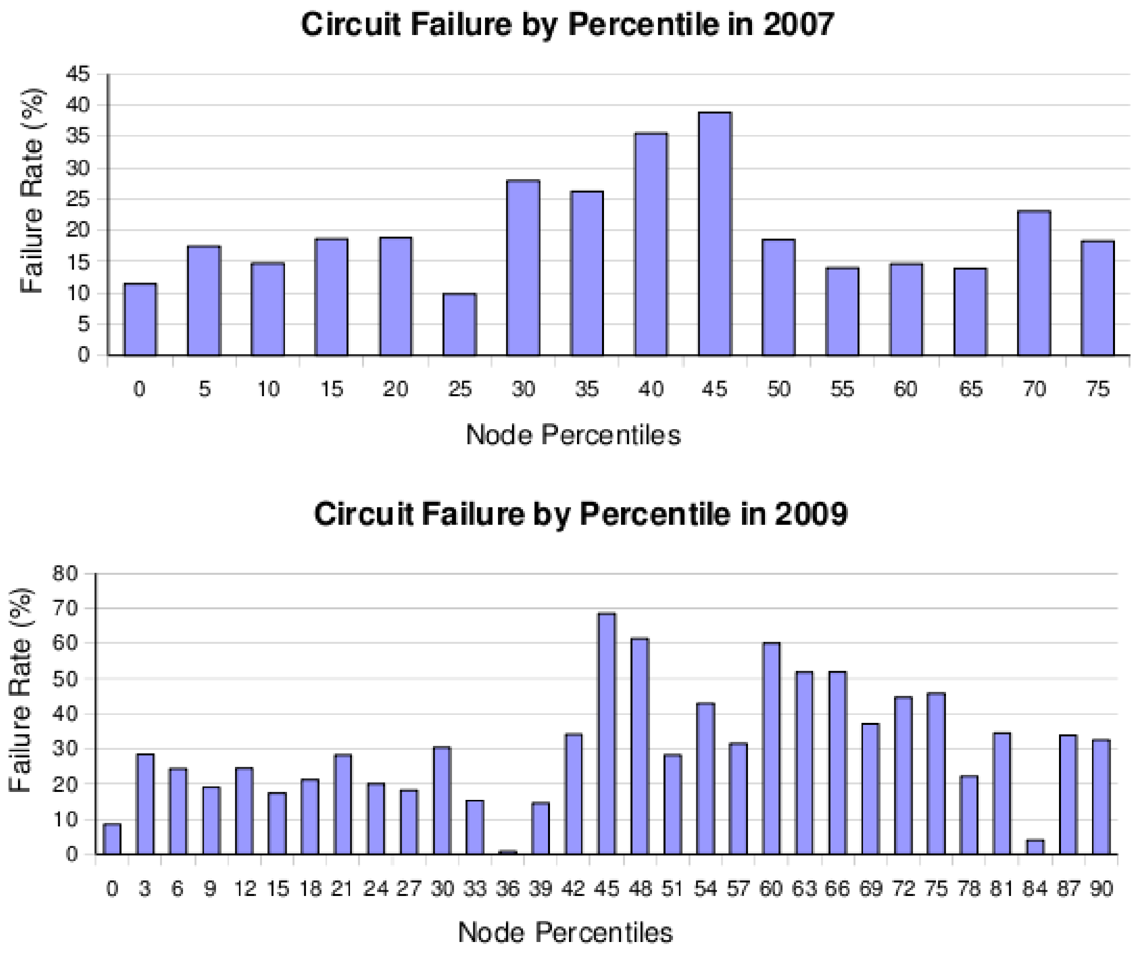


Figure 5: Circuit Failure Rates before and after guard rebalancing

dropping circuit creation cells. This could explain the sharp difference in high load vs low load conditions for circuit failure, but not for stream capacity.

Furthermore, it appears in the right side of Figure 5 as though the slower 50% of the network is now exhibiting significantly higher failure percentages than the first 50%. In order to explore this, we ran a number of additional circuit failure scans utilizing TorFlow's Node Restriction capabilities.

4.3 Drilling Down with Restrictions

Our first guess was that middle nodes were bearing more than their proportion of directory requests due to changes in client node selection in the guard rebalancing fixes mentioned above.

However, circuit construction scans showed this not to be the case. In fact, it showed the exact opposite. Nodes with their directory port open did not exhibit consistent increase in either circuit failure or extend time over nodes without their directory port open, and middle nodes exhibited much less circuit failure than guard and exit nodes.

After more investigation and many scans, two consistent failure classes emerged: Windows nodes, and non-bandwidth limited nodes, each of which seemed to perform a bit worse as Guard and Exit nodes than as the middle nodes.

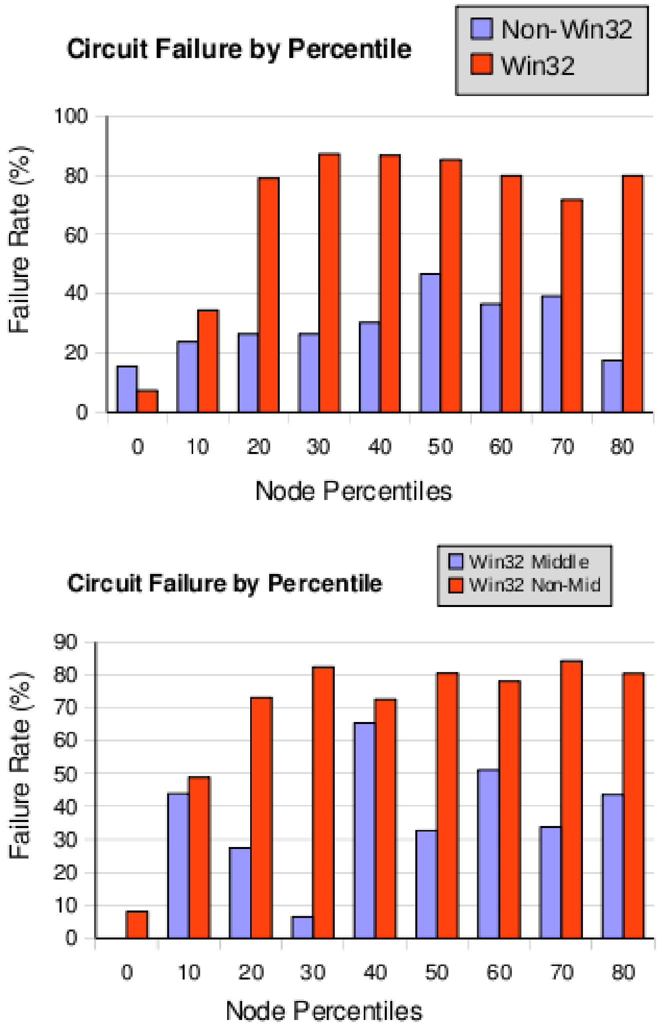


Figure 6: Circuit failure of Windows nodes

The Windows node result is not entirely surprising, as it is known that these nodes will have difficulty servicing large numbers of sockets using normal WinSock [9]. As can be seen in Figure 6, these nodes exhibit significantly higher circuit failure rates than non-Windows nodes, and also predictably fare worse in either the Guard or Exit position, where they have to maintain significantly more TCP sockets for clients and exit streams, respectively.

There are some aberrations. In particular, the high-end Windows nodes seem to be on par with their peers. This is likely due to the higher socket limits of server editions of Windows as compared to desktop.

Interestingly, as can be seen in the left side of Figure 7, nodes that have configured a specific bandwidth rate limit are considerably more reliable than those that set no limit and just fill their upstream to the max. One potential reason for this could be that due to Tor's multiplexing

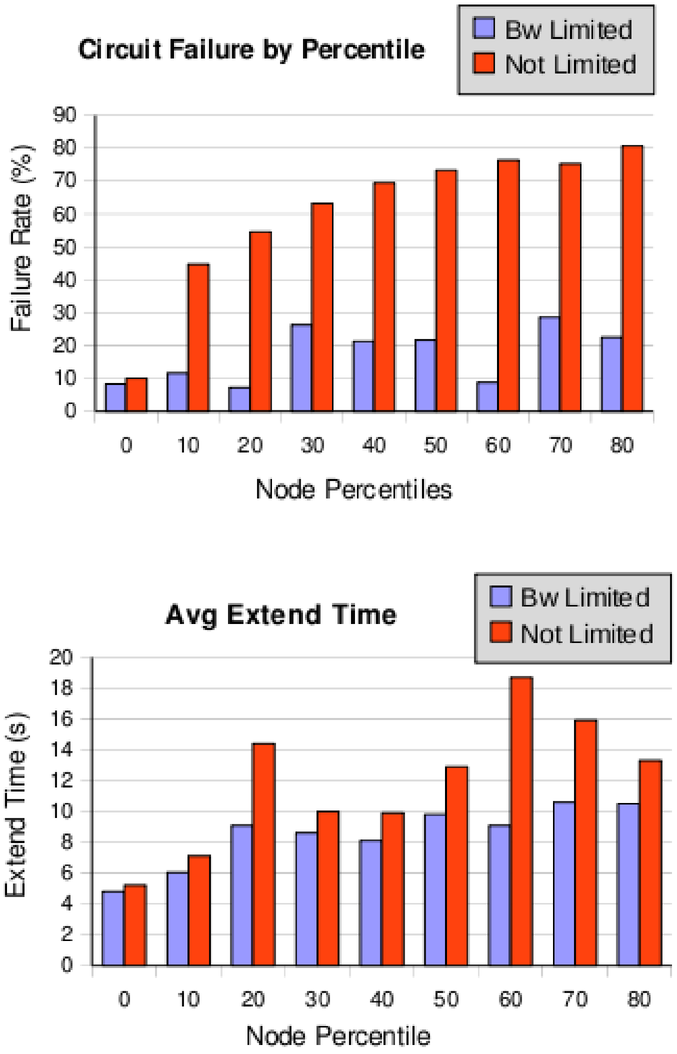


Figure 7: Circuit failure and extend times of bandwidth limited vs non-limited nodes

of streams inside TCP, the backoff properties of TCP are really damaging to the ability of circuit creation cells to get through in time. Other possibilities include asymmetric bandwidth limits and OS and CPU limits being easier to hit, causing failure as opposed to smooth throttling.

Also of interest from the right side of Figure 7 is the fact that despite only emptying their queues once per second, the circuit extend latencies of bandwidth-limited nodes are still typically less than their non-limited neighbors. This indicates that most of these rate limited nodes have more than one second worth of data queued up. It also again hints at the possibility that the normal fairness properties of TCP are not functioning properly for the non-limited nodes.

Additional scans have also shown that unlike the Windows nodes, non-limited nodes exhibit the same failure characteristics in both middle and Guard/Exit positions. Furthermore, more failures are caused by timeout as opposed to closed connections for the non-limited nodes than for the Windows nodes. These two facts seem to indicate that the circuit failure is independent of the ability to make a TCP connection for non-limited nodes, and is possibly tied to the ability to transfer a create cell through the network and also implicate TCP flow control issues.

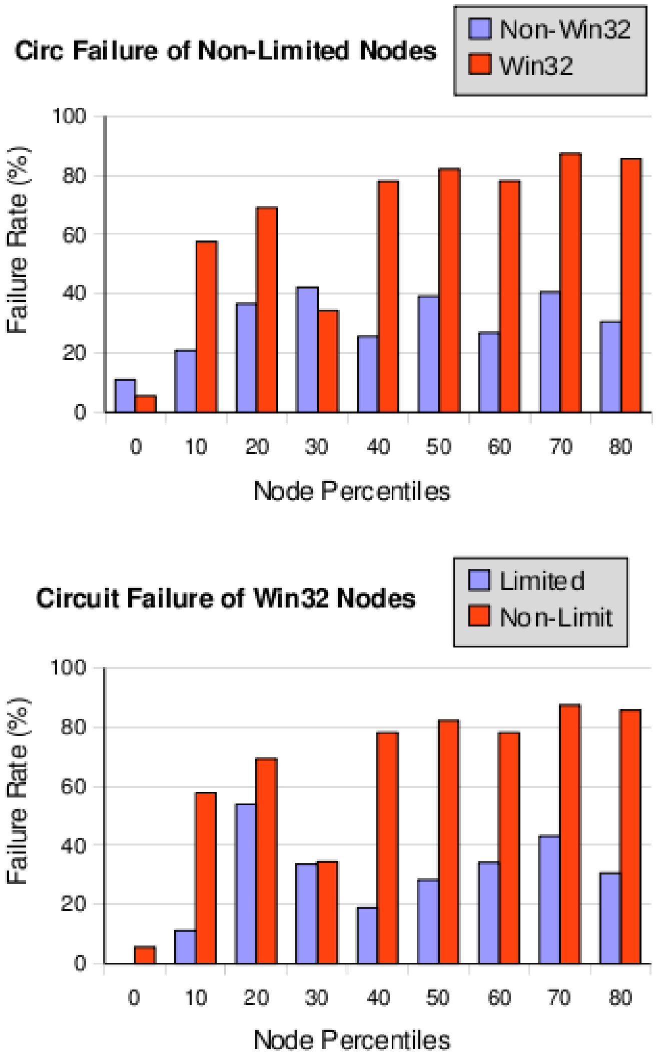


Figure 8: Circuit failure of Windows versus Non-Limited nodes

It is also the case that many of the Windows nodes also do not set bandwidth limits for themselves. This led us to perform four scans to compare the effect of Windows, the results of which are shown in Figure 8.

On the left side of Figure 8, it can be seen that while non-Win32 non-limited nodes do exhibit higher failure rates than the limited nodes in Figure 7, the bulk of the failure is due to the Windows non-limited nodes. Furthermore, on the right of Figure 8, it can be seen that Limited Windows nodes perform significantly better than non-limited, though again not quite on par with limited nodes from Figure 7. This could be due to the limited nodes' operators ensuring that they set their bandwidth rate below the point at which Tor begins to experience performance problems or otherwise slow down their system.

This seems to indicate that we need to encourage node operators to set bandwidth limits below their connection's capacity, and that we need to ensure that the Vidalia UI is clear enough for Windows users to be able to set limits properly, and understand the importance of doing so.

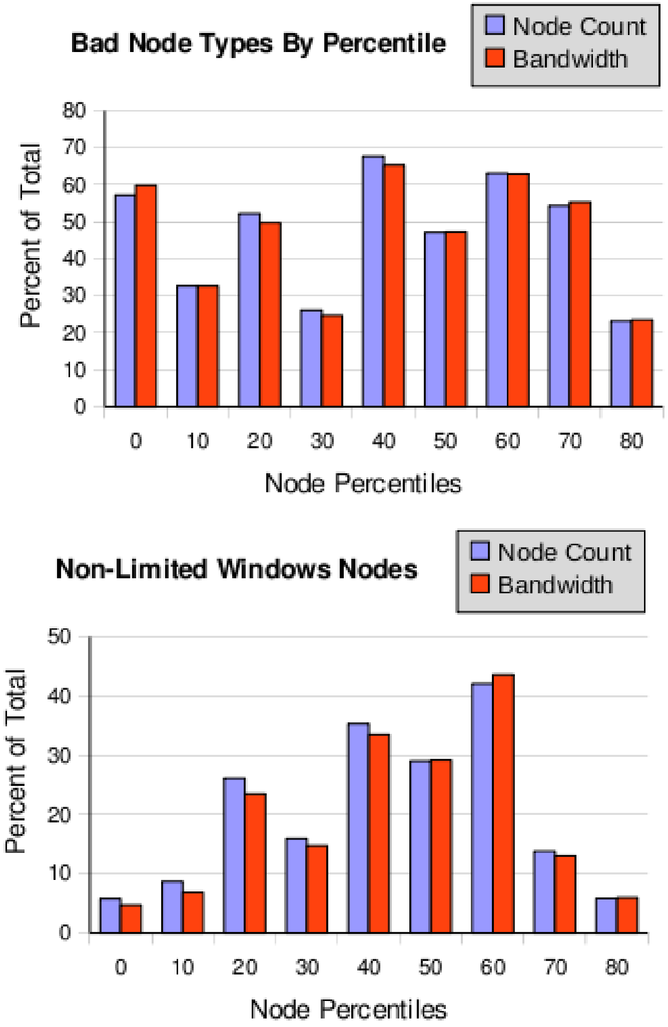


Figure 9: Density of non-limited and Windows nodes per percentile

To help understand the effects on the network as a whole, we broke down the proportion of non-limited and Windows nodes in each percentile slice in Figure 9. On the left is the combination of non-limited and Windows nodes, and on the right is Windows nodes that are non-limited. It can be seen that the amount of Windows non-limited nodes roughly correspond to the level of failure rates from the right side of Figure 5. Of course, correlation does not imply causation, but it does give us a starting point to work with.

4.4 Directory Feedback

Trying to determine the source of unbalancing by guesswork, measurement, and experiment is time consuming. Correcting for these discovered issues algorithmically is even harder, and is also fragile to other changes in the network.

It would be much better if we could use a balancing metric or metrics, and use them directly to alter client load allocation to correct for arbitrary unbalancing.

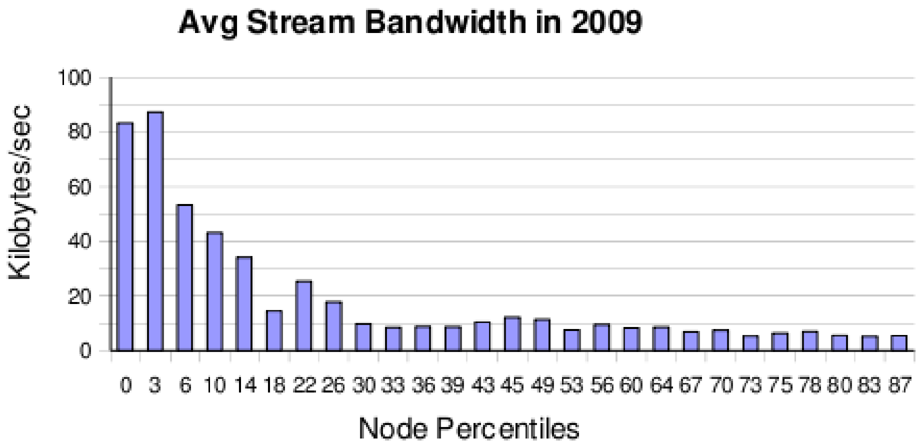


Figure 10: Stream bandwidth capacity as measured by recent feedback scan

In a well-balanced network, all streams should receive the same bandwidth. It can clearly be seen from Figure 10 that this is not the case, and that some segments of the network are providing clients with much better capacities than others.

Based on this, a good metric to gauge the load of a node relative to its peers should be the ratio of its average stream capacity to that of the rest of the network. This ratio should represent the ratio of extra load the node is carrying over its peers.

The TorCtl SpeedRacer utility produces these ratios. It divides the network into 50 node slices based on advertised bandwidth rankings and repeatedly fetches a large file through 2-hop Tor circuits created inside this range.

It then uses the SQL support of TorCtl to produce a preliminary ratio for each node. To prevent potential sabotage and to preserve fairness, nodes with ratios significantly less than 1.0 are filtered from the results of other nodes, as are any other slow outlier streams, and the ratios are recomputed.

These ratios can then be used to form a feedback loop with the directory authorities: a set of scanning authorities will use the ratios to adjust node bandwidths in their consensus document while retaining the original values in the node descriptors. The authorities will then vote on the official values by taking the median of the scanning authority values.

Newer Tor clients (v0.2.1 and above) have already been modified to accept these new values as part of work done by Peter Palfrader to reduce descriptor size and directory server overhead. The weighted values will cause these new clients to choose these nodes less often, due to the probabilistic weighting of Tor's node selection algorithm [5].

Subsequent passes of the scanner will then use the published value when computing new ratios, and their computed ratios should eventually converge to 1.0.

Because we keep the original values on-hand, we can implement algorithmic balancing improvements in tandem with this system without jeopardizing the network. All we need to do is see if the new algorithmic changes alter the ratios that are being computed for better or for worse.

It should be noted that it is also possible to similarly compute circuit failure ratios that can be multiplied with the stream capacity ratios to deal with nodes experiencing forms of load other than bandwidth constraint, such as those in the preceding section. Our current plan is to perform the stream adjustments first, and note what effect, if any, this has on circuit reliability, and then introduce those ratio adjustments if needed.

5 Exit Node Scanning

The Snakes on a Tor exit scanner performs a series of actions via various exit nodes. The current implementation was initially written by Google Summer of Code student Aleksei Gorny. It supports scanning exit nodes for modifications to SSL, HTML, JavaScript, HTTP Headers, and arbitrary HTTP content.

Target sites are obtained by querying Google and Yahoo for keywords from a keyword file and pulling a random selection from those results.

All aspects of the scan are continually checkpointed and serialized as pickled Python objects, so that scans can be resumed at any time in the event of code modification or failure. A Snake Inspector script examines the pickled results and is able to display error classification and differences in a human readable format, as well as re-evaluate failures based on changing criteria.

5.1 Goals

We face a similar problem scope to antivirus software. It is unreasonable to expect to be able to detect all malicious exit nodes on the network. Nodes can target specific sites in languages the scanners do not speak, or they can target only specific users.

Instead, our goals are essentially twofold: Firstly, we aim for the more modest goal of removing exit status of misconfigured or egregiously censored nodes. Second, we aim to prevent dragnet user exploitation and unmasking via Tor. We want to make it such that there is a high level of risk and effort associated with using exploits against large sections of the Tor userbase for purposes of creating botnets or mining account credentials.

5.2 General Methodology

In general, all scans follow the pattern in the pseudocode in Figure 11: First they perform an operation without Tor. They then perform that same operation through Tor. If the relevant content matches, it is a success. Otherwise, they perform the operation again from a new Non-Tor IP but using the same credentials (such as cookies) that were used during Tor. Optionally, they perform a "Difference Pruning" step that removes items that have changed between the two Non-Tor operations from consideration from the second Non-Tor fetch. Finally, they compare the Tor fetch to this new Non-Tor fetch. If there are no relevant differences, then it is a success. Otherwise, the node is marked as a failure.

In reality, we have many more failure types than just truncation and modification. There are also DNS failures, various types of network errors, timeout errors, HTTP errors, SSL errors, Tor errors, and also subclassifications of these.

```
TorResult = PerformFetch(Tor, URL, TorAuthSet)
NonTorResult1 = PerformFetch(NonTorIP1, URL, NonTorAuthSet)
if IsPrefix(TorResult, NonTorResult1):
  return FAIL_TRUNCATION

TorResult = StripIrrelevant(TorResult)
NonTorResult1 = StripIrrelevant(NonTorResult1)
if TorResult == NonTorResult1:
  return OK

NonTorResult2 = PerformFetch(NonTorIP2, URL, TorAuthSet)
NonTorResult2 = StripIrrelevant(NonTorResult2)

if DifferencePruner: # Difference Pruning Step (optional)
  Diffs |= Diff(NonTorResult1, NonTorResult2)
  NonTorResult2 = Prune(NonTorResult2, Diffs)
  TorResult = Prune(TorResult, Diffs)

if NonTorResult2 == TorResult:
  return OK
return FAIL_MODIFICATION
```

Figure 11: Pseudocode that all scans follow

5.3 HTML and JavaScript Scanning

In the case of HTML scanning, we use Beautiful Soup[6] to strip content down to only tags that can contain plugins, script, or CSS in order to eliminate localization and content changes from comparison and to obtain a set of page script, iframe, object, and link tags for recursion.

The Difference Pruning step is done by first building sets of HTML tags and their attributes seen for a particular URL. If any new tags or attributes appear during successive Non-Tor fetches, they are added to the set of differences, elements of which then set-subtracted from the Tor fetches.

The HTML Difference Pruner objects persist for the duration of the scan and are also serialized, so that differences can continue to be accumulated and filtered out, and that initial failures can be corrected by the Snake Inspector script post-scan.

If the HTML Difference Pruner finds no new Tor differences after pruning, we rerun the unpruned fetches through a JavaScript Difference Pruner that uses a Javascript parser from the Antlr Project[7] to prune differences from an AST. This is done to ensure we haven't pruned a tag or attribute that varies because of minor Javascript differences (such as unique identifiers embedded in script). If no Tor differences remain, the node has passed.

The Javascript parser and Javascript Difference Pruner are also used for pure Javascript files during page element recursion.

[6] http://www.crummy.com/software/BeautifulSoup/

[7] http://www.antlr.org/grammar/1206736738015/JavaScript.g

5.4 Filtering False Positives

Despite the above efforts, false positives inevitably arise. To deal with them, we have implemented URL-based filters such that if a particular URL causes more than a set percentage of exit nodes to fail a scan, it is automatically removed from consideration and the nodes are rescanned with fresh URLs.

Additionally, SoaT provides the ability to rescan nodes that were marked as failed during any of its previous scans.

5.5 Results

By far the most common result are nodes that simply routinely timeout instead of completing streams. A close second to this are nodes that truncate streams part of the way through. Less common, but still prevalent, are nodes that cannot perform DNS resolution. There's also usually one or two nodes injecting Javascript that hooks window.open, presumably as part of their antivirus software's popup blocking mechanism. On occasion, we'll hit a node that has some sort of payment portal up, possibly due to people doing things like attempting to run Tor nodes out of fee-based Internet services in hotels, coffeshops, or airports.

On the more malicious end of the spectrum, we've seen a couple of nodes that spoof SSL, typically with self-signed SSL certificates. Usually these are in China, but we have seen one in Europe, and another in Atlanta, Georgia. We've also seen one instance of a node spoofing SSH keys.

6 Future Work

There are several directions for immediate and long-term future work.

6.1 Exit Scanning

Despite our efforts, exit scanning still has a number of weaknesses against diligent adversaries.

Firstly, we currently have no ability to recursively fetch URLs constructed by Javascript, which means that if an adversary were able to detect any fetches generated through them, they would be able to modify the resulting content in only these URLs without fear of us noticing. The best way to gain full visibility of these URLs is to actually have a real Firefox instance controlled by SoaT that gives us a list of URLs to recurse from a given source URL. We could also extend this to use Firefox to perform additional Tor fetches from an instrumented virtual machine that actually watches for signs of proxy bypass, unauthorized disk access, or new executable mappings in the Firefox process itself.

Second, we have no visibility behind POST requests and forms. We need to figure out a way to either randomly post at forms, or use fixed logins for a set of non-SSL webapps.

Third, we currently have very limited visibility into websites that are highly dynamic. In particular, websites that produce completely different layouts for different countries are often removed from consideration by our URL based filters. One thing we can do about this is have country-specific scanners that use GeoIP path restrictions to only scan exits in their own country.

Another might be to delegate trusted exit nodes in each location that can be used to fetch content where we normally used the local IP.

Another more serious issue along these lines are the ad networks. Some of them exhibit too many changes for our difference pruner to be left with much useful content, and thus may be potentially replaced with malicious content and still be undetected by our scans. Shipping something like Adblock in the default configuration of Tor would essentially eliminate this from our threat model (and make our users safer in general, as the ad networks themselves are often vectors for malware [13]). However, this may introduce political complications. Many sites are already trigger-happy to ban the entire Tor network from contributing content, and a loss of revenue stream may be all the reason they need to ban readers as well.

Fourth, it would be interesting to set up some honeypot accounts that log in to POP3 and IMAP accounts only from specific exit IPs, to determine if these accounts are ever used outside of the scan.

6.2 Reliability and Node-Based Directory Feedback

As we've seen, stream capacity is not the whole story when it comes to node load. Nodes can become CPU or file descriptor constrained, both of which will cause them to fail circuits but still have reasonable capacity for those that get through. In fact, many of the faster nodes on the network max out CPU before network capacity, which will cause them to drop create cells and close circuits. However, the circuits and streams that do manage to get through will have good capacity through these nodes. Does this mean we should weight them higher, or should we possibly multiply their stream ratio by an additional circuit failure ratio?

Along these lines, there is another class of adversary that we'd like to be able to detect as well: Those that fail circuits that don't involve their colluding peers [8]. Such adversaries can lie about their bandwidth capacity by an amount that is proportional to the amount of circuits that they fail [1, 2], and would able to ensure that every byte they devote to Tor is devoted to surveiling a compromised path. Worse, their inflated bandwidth statistics would not be noticed by our capacity scanners because they would be able to lie just enough to compensate for the circuits they fail that are not compromised.

It is our intuition that our reliability statistics can help detect these attacks, but to have a truly resilient defense, we'd need node based monitoring of circuit reliability as opposed to client based, which is prone to fingerprinting and detection.

6.3 We Need More Data!

We've only just begun to leverage the new TorCtl framework, and for every question it answers, there are often two more uncovered as a result. More scans need to be done to test various hypotheses to attempt to answer these questions.

In particular, the circuit construction scans that shed insight into particular trouble-spots in the network should also be repeated as stream capacity scans. Because stream capacity scans are considerably slower and more taxing on the network, it was much easier to find the trouble spots with circuit scans first. However, certain characteristics may affect circuit failure and not stream capacity and vice-versa, so all of the pertinent results should ideally be repeated with stream bandwidth scans.

Acknowledgments

We'd like to thank all of our Google Summer of Code students who have contributed various features to TorFlow: Johannes Renner for his GeoIP-based restrictions and his work in building latency maps of the Tor network, Fallon Chen for her work on the BuildTimes utility, and Aleksei Gorny for his work on an initial Python rewrite of SoaT. We'd also like to thank Google for sponsoring Tor and providing a generous allocation of students year after year.

We'd also like to thank Karsten Loesing for his thorough analysis of our circuit BuildTimes data and for his Python rewrite of the URL fetching piece of SpeedRacer.

Lastly, we'd like to thank Roger and Nick for having the foresight to design such a flexible control mechanism for Tor, for their thorough efforts at documenting it and the rest of Tor, and for Tor in general.

References

[1] Kevin Bauer, Damon McCoy, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. Low-resource routing attacks against Tor. In *WPES '07: Proceedings of the 2007 ACM workshop on Privacy in electronic society*, pages 11–20, New York, NY, USA, 2007. ACM.

[2] Nikita Borisov, George Danezis, Prateek Mittal, and Parisa Tabriz. Denial of service or denial of security? In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 92–102, New York, NY, USA, 2007. ACM.

[3] Roger Dingledine and Nick Mathewson. Tor control protocol specifications. https://gitweb.torproject.org/torspec.git/blob/HEAD:/control-spec.txt.

[4] Roger Dingledine and Nick Mathewson. Tor directory protocol specifications. https://gitweb.torproject.org/torspec.git/blob/HEAD:/dir-spec.txt.

[5] Roger Dingledine and Nick Mathewson. Tor Path Specifications. https://gitweb.torproject.org/torspec.git/blob/HEAD:/path-spec.txt.

[6] Roger Dingledine and Nick Mathewson. Tor Protocol Specifications. https://gitweb.torproject.org/torspec.git/blob/HEAD:/tor-spec.txt.

[7] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[8] Xinwen Fu and Zhen Ling. One Cell is Enough to Break Tor's Anonymity. http://www.blackhat.com/presentations/bh-dc-09/Fu/BlackHat-DC-09-Fu-Break-Tors-Anonymity.pdf.

[9] Nick Mathewson. Some Notes on Progress with IOCP and Libevent. https://blog.torproject.org/blog/some-notes-progress-iocp-and-libevent.

[10] Steven J. Murdoch. Economics of Tor performance. http://www.lightbluetouchpaper.org/2007/07/18/economics-of-tor-performance/.

[11] Mike Perry. Securing the Tor Network. http://www.blackhat.com/presentations/bh-usa-07/Perry/Presentation/bh-usa-07-perry.pdf.

[12] Johannes Renner. Performance-Improved Onion Routing. Master's thesis, Aachen Univiversity, September 2007.

[13] Betsy Schiffman. Hackers Use Banner Ads on Major Sites. Wired Magazine, Nov 2007. http://www.wired.com/techbiz/media/news/2007/11/doubleclick.