Circuit-level padding

The circuit padding system in Tor is an extension of the WTF-PAD event-driven state machine design[15]. At a high level, this design places one or more padding state machines at the client, and one or more padding state machines at a relay, on each circuit.

State transition and histogram generation has been generalized to be fully programmable, and probability distribution support was added to support more compact representations like APE[16]. Additionally, packet count limits, rate limiting, and circuit application conditions have been added.

At present, Tor uses this system to deploy two pairs of circuit padding machines, to obscure differences between the setup phase of client-side onion service circuits, up to the first 10 relay cells.

This specification covers only the resulting behavior of these padding machines, and thus does not cover the state machine implementation details or operation. For full details on using the circuit padding system to develop future padding defenses, see the research developer documentation[17].

Circuit Padding Negotiation

Circuit padding machines are advertised as "Padding" subprotocols. The onion service circuit padding machines are advertised as "Padding=2" (`PADDING_MACHINES_CIRC_SETUP`).

Because circuit padding machines only become active at certain points in circuit lifetime, and because more than one padding machine may be active at any given point in circuit lifetime, there is also a PADDING_NEGOTIATE message and a PADDING_NEGOTIATED message. These are relay commands 41 and 42 respectively, with relay headers as per section 6.1 of tor-spec.txt.

The fields in the body of a PADDING_NEGOTIATE message are as follows:

Acknowledgements

Thanks to everyone who has contributed to this design with feedback and discussion.

Thanks go to arma, ioerror, kernelcorn, nickm, s7r, Sebastian, teor, weasel and everyone else!

References:

```
[RANDOM-REFS]:
   http://projectbullrun.org/dual-ec/ext-rand.html
   https://lists.torproject.org/pipermail/tor-dev/2015-November/
009954.html

[RNGMESSAGING]:
   https://moderncrypto.org/mail-archive/messaging/2015/002032.html

[HOPPER]:
   https://lists.torproject.org/pipermail/tor-dev/2014-January/
006053.html

[UNICORN]:
   https://eprint.iacr.org/2015/366.pdf

[VDFS]:
   https://eprint.iacr.org/2018/601.pdf
```

Tor Path Specification

Roger Dingledine
Nick Mathewson

Note: This is an attempt to specify Tor as currently implemented. Future versions of Tor will implement improved algorithms.

This document tries to cover how Tor chooses to build circuits and assign streams to circuits. Other implementations MAY take other approaches, but implementors should be aware of the anonymity and load-balancing implications of their choices.

THIS SPEC ISN'T DONE YET.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

```
        until application data begins.
      - Default: 1

    * nf_pad_relays
      - If set to 1, we also pad inactive relay-to-relay connections
      - Default: 0

    * nf_conntimeout_relays
      - The number of seconds that idle relay-to-relay connections are
kept
        open.
      - Default: 3600
```

```
    * nf_ito_low
      - The low end of the range to send padding when inactive, in ms.
      - Default: 1500

    * nf_ito_high
      - The high end of the range to send padding, in ms.
      - Default: 9500
      - If nf_ito_low == nf_ito_high == 0, padding will be disabled.

    * nf_ito_low_reduced
      - For reduced padding clients: the low end of the range to send
padding
        when inactive, in ms.
      - Default: 9000

    * nf_ito_high_reduced
      - For reduced padding clients: the high end of the range to send
padding,
        in ms.
      - Default: 14000

    * nf_conntimeout_clients
      - The number of seconds to keep never-used circuits opened and
        available for clients to use. Note that the actual client
timeout is
        randomized uniformly from this value to twice this value.
      - The number of seconds to keep idle (not currently used)
canonical
        channels are open and available. (We do this to ensure a
sufficient
        time duration of padding, which is the ultimate goal.)
      - This value is also used to determine how long, after a port
has been
        used, we should attempt to keep building predicted circuits
for that
        port. (See path-spec.txt section 2.1.1.)  This behavior was
        originally added to work around implementation limitations,
but it
        serves as a reasonable default regardless of implementation.
      - For all use cases, reduced padding clients use half the
consensus
        value.
      - Implementations MAY mark circuits held open past the reduced
padding
        quantity (half the consensus value) as "not to be used for
streams",
        to prevent their use from becoming a distinguisher.
      - Default: 1800

    * nf_pad_before_usage
      - If set to 1, OR connections are padded before the client uses
them
        for any application traffic. If 0, OR connections are not
padded
```

General operation

Tor begins building circuits as soon as it has enough directory information to do so. Some circuits are built preemptively because we expect to need them later (for user traffic), and some are built because of immediate need (for user traffic that no current circuit can handle, for testing the network or our reachability, and so on).

```
   [Newer versions of Tor (0.2.6.2-alpha and later):
    If the consensus contains Exits (the typical case), Tor will build
both
    exit and internal circuits. When bootstrap completes, Tor will be
ready
    to handle an application requesting an exit circuit to services
like the
    World Wide Web.
```

If the consensus does not contain Exits, Tor will only build internal circuits. In this case, earlier statuses will have included “internal” as indicated above. When bootstrap completes, Tor will be ready to handle an application requesting an internal circuit to hidden services at “.onion” addresses.

If a future consensus contains Exits, exit circuits may become available.]

When a client application creates a new stream (by opening a SOCKS connection or launching a resolve request), we attach it to an appropriate open circuit if one exists, or wait if an appropriate circuit is in-progress. We launch a new circuit only if no current circuit can handle the request. We rotate circuits over time to avoid some profiling attacks.

To build a circuit, we choose all the nodes we want to use, and then construct the circuit. Sometimes, when we want a circuit that ends at a given hop, and we have an appropriate unused circuit, we “cannibalize” the existing circuit and extend it to the new terminus.

These processes are described in more detail below.

This document describes Tor’s automatic path selection logic only; path selection can be overridden by a controller (with the EXTENDCIRCUIT and ATTACHSTREAM commands). Paths constructed through these means may violate some constraints given below.

Terminology

A “path” is an ordered sequence of nodes, not yet built as a circuit.

A “clean” circuit is one that has not yet been used for any traffic.

A “fast” or “stable” or “valid” node is one that has the ‘Fast’ or ‘Stable’ or ‘Valid’ flag set respectively, based on our current directory information. A “fast” or “stable” circuit is one consisting only of “fast” or “stable” nodes.

In an “exit” circuit, the final node is chosen based on waiting stream requests if any, and in any case it avoids nodes with exit policy of “reject *:*”. An “internal” circuit, on the other hand, is one where the final node is chosen just like a middle node (ignoring its exit policy).

A “request” is a client-side stream or DNS resolve that needs to be served by a circuit.

A “pending” circuit is one that we have started to build, but which has not yet completed.

A circuit or path “supports” a request if it is okay to use the circuit/path to fulfill the request, according to the rules given below. A circuit or path “might support” a request if some aspect of the request is unknown (usually its target IP), but we believe the path probably supports the request according to the rules given below.

A relay’s bandwidth

Old versions of Tor did not report bandwidths in network status documents, so clients had to learn them from the routers’ advertised relay descriptors.

For versions of Tor prior to 0.2.1.17-rc, everywhere below where we refer to a relay’s “bandwidth”, we mean its clipped advertised bandwidth, computed by taking the smaller of the ‘rate’ and ‘observed’ arguments to the “bandwidth” element in the relay’s descriptor. If a router’s advertised bandwidth is greater than MAX_BELIEVABLE_BANDWIDTH (currently 10 MB/s), we clipped to that value.

For more recent versions of Tor, we take the bandwidth value declared in the consensus, and fall back to the clipped advertised bandwidth only if the consensus does not have bandwidths listed.

If the client has opted to use reduced padding, it continues to send padding cells sampled from the range [9000,14000] milliseconds (subject to consensus parameter alteration as per Section 2.6), still using the Y=max(X,X) distribution. Since the padding is now unidirectional, the expected frequency of padding cells is now governed by the Y distribution above as opposed to Z. For a range of 5000ms, we can see that we expect to send a padding packet every 9000+3332.8 = 12332.8ms. We also half the circuit available timeout from ~50min down to ~25min, which causes the client’s OR connections to be closed shortly there after when it is idle, thus reducing overhead.

These two changes cause the padding overhead to go from 309KB per one-time-use Tor connection down to 69KB per one-time-use Tor connection. For continual usage, the maximum overhead goes from 103 bytes/sec down to 46 bytes/sec.

If a client opts to completely disable padding, it sends a PADDING_NEGOTIATE to instruct the relay not to pad, and then does not send any further padding itself.

Currently, clients negotiate padding only when a channel is created, immediately after sending their NETINFO cell. Recipients SHOULD, however, accept padding negotiation messages at any time.

If a client which previously negotiated reduced, or disabled, padding, and wishes to re-enable default padding (ie padding according to the consensus parameters), it SHOULD send PADDING_NEGOTIATE START with zero in the ito_low_ms and ito_high_ms fields. (It therefore SHOULD NOT copy the values from its own established consensus into the PADDING_NEGOTIATE cell.) This avoids the client needing to send updated padding negotiations if the consensus parameters should change. The recipient’s clamping of the timing parameters will cause the recipient to use its notion of the consensus parameters.

Clients and bridges MUST reject padding negotiation messages from relays, and close the channel if they receive one.

Consensus Parameters Governing Behavior

Connection-level padding is controlled by the following consensus parameters:

If X is a random variable uniform from 0..R-1 (where R=high-low), then the random variable Y = max(X,X) has Prob(Y == i) = (2.0*i* + *1)/(R*R).

Then, when both sides apply timeouts sampled from Y, the resulting bidirectional padding packet rate is now a third random variable: Z = min(Y,Y).

The distribution of Z is slightly bell-shaped, but mostly flat around the mean. It also turns out that Exp[Z] ~= Exp[X]. Here's a table of average values for each random variable:

R	Exp[X]	Exp[Z]	Exp[min(X,X)]	Exp[Y=max(X,X)]
2000	999.5	1066	666.2	1332.8
3000	1499.5	1599.5	999.5	1999.5
5000	2499.5	2666	1666.2	3332.8
6000	2999.5	3199.5	1999.5	3999.5
7000	3499.5	3732.8	2332.8	4666.2
8000	3999.5	4266.2	2666.2	5332.8
10000	4999.5	5328	3332.8	6666.2
15000	7499.5	7995	4999.5	9999.5
20000	9900.5	10661	6666.2	13332.8

Maximum overhead bounds

With the default parameters and the above distribution, we expect a padded connection to send one padding cell every 5.5 seconds. This averages to 103 bytes per second full duplex (~52 bytes/sec in each direction), assuming a 512 byte cell and 55 bytes of TLS+TCP+IP headers. For a client connection that remains otherwise idle for its expected ~50 minute lifespan (governed by the circuit available timeout plus a small additional connection timeout), this is about 154.5KB of overhead in each direction (309KB total).

With 2.5M completely idle clients connected simultaneously, 52 bytes per second amounts to 130MB/second in each direction network-wide, which is roughly the current amount of Tor directory traffic[11]. Of course, our 2.5M daily users will neither be connected simultaneously, nor entirely idle, so we expect the actual overhead to be much lower than this.

Reducing or Disabling Padding via Negotiation

To allow mobile clients to either disable or reduce their padding overhead, the PADDING_NEGOTIATE cell (tor-spec.txt section 7.2) may be sent from clients to relays. This cell is used to instruct relays to cease sending padding.

Building circuits

Here we describe a number of rules for building circuits: under what circumstances we do so, how we choose the paths for them, when we give up on an in-progress circuits, and what we do when circuit construction fails.

When we build

We don't build circuits until we have enough directory info

There's a class of possible attacks where our directory servers only give us information about the relays that they would like us to use. To prevent this attack, we don't build multi-hop circuits (including preemptive circuits, [on-demand circuits(#on-demand), onion-service circuits] or self-testing testing circuits) for real traffic until we have enough directory information to be reasonably confident this attack isn't being done to us.

Here, "enough" directory information is defined as:

```
      * Having a consensus that's been valid at some point in the
        last REASONABLY_LIVE_TIME interval (24 hours).

      * Having enough descriptors that we could build at least some
        fraction F of all bandwidth-weighted paths, without taking
        ExitNodes/EntryNodes/etc into account.

        (F is set by the PathsNeededToBuildCircuits option,
        defaulting to the 'min_paths_for_circs_pct' consensus
        parameter, with a final default value of 60%.)

      * Having enough descriptors that we could build at least some
        fraction F of all bandwidth-weighted paths, _while_ taking
        ExitNodes/EntryNodes/etc into account.

        (F is as above.)

      * Having a descriptor for every one of the first
        NUM_USABLE_PRIMARY_GUARDS guards among our primary guards.
(see
        guard-spec.txt)
```

We define the "fraction of bandwidth-weighted paths" as the product of these three fractions.

Implementation

Tor clients currently maintain one TLS connection to their Guard node to carry actual application traffic, and make up to 3 additional connections to other nodes to retrieve directory information.

We pad only the client's connection to the Guard node, and not any other connection. We treat Bridge node connections to the Tor network as client connections, and pad them, but otherwise not pad between normal relays.

Both clients and Guards will maintain a timer for all application (ie: non-directory) TLS connections. Every time a padding packet sent by an endpoint, that endpoint will sample a timeout value from the max(X,X) distribution described in Section 2.3. The default range is from 1.5 seconds to 9.5 seconds time range, subject to consensus parameters as specified in Section 2.6.

(The timing is randomized to avoid making it obvious which cells are padding.)

If another cell is sent for any reason before this timer expires, the timer is reset to a new random value.

If the connection remains inactive until the timer expires, a single PADDING cell will be sent on that connection (which will also start a new timer).

In this way, the connection will only be padded in a given direction in the event that it is idle in that direction, and will always transmit a packet before the minimum 10 second inactive timeout.

(In practice, an implementation may not be able to determine when, exactly, a cell is sent on a given channel. For example, even though the cell has been given to the kernel via a call to `send(2)`, the kernel may still be buffering that cell. In cases such as these, implementations should use a reasonable proxy for the time at which a cell is sent: for example, when the cell is queued. If this strategy is used, implementations should try to observe the innermost (closest to the wire) queue that they practically can, and if this queue is already nonempty, padding should not be scheduled until after the queue does become empty.)

Padding Cell Timeout Distribution Statistics

To limit the amount of padding sent, instead of sampling each endpoint timeout uniformly, we instead sample it from max(X,X), where X is uniformly distributed.

For reference, here are default values and ranges (in parenthesis when known) for common routers, along with citations to their manuals.

Some routers speak other collection protocols than Netflow, and in the case of Juniper, use different timeouts for these protocols. Where this is known to happen, it has been noted.

	Inactive Timeout	Active
Cisco IOS[3]	15s (10-600s)	30min (1-60min)
Cisco Catalyst[4]	5min	32min
Juniper (jFlow)[5]	15s (10-600s)	30min (1-60min)
Juniper (Netflow)[6,7]	60s (10-600s)	30min (1-30min)
H3C (Netstream)[8]	60s (60-600s)	30min (1-60min)
Fortinet[9]	15s	30min
MicroTik[10]	15s	30min
nProbe[14]	30s	120s
Alcatel-Lucent[2]	15s (10-600s)	30min (1-600min)

The combination of the active and inactive netflow record timeouts allow us to devise a low-cost padding defense that causes what would otherwise be split records to “collapse” at the router even before they are exported to the collector for storage. So long as a connection transmits data before the “inactive flow timeout” expires, then the router will continue to count the total bytes on that flow before finally emitting a record at the “active flow timeout”.

This means that for a minimal amount of padding that prevents the “inactive flow timeout” from expiring, it is possible to reduce the resolution of raw per-flow netflow data to the total amount of bytes send and received in a 30 minute window. This is a vast reduction in resolution for HTTP, IRC, XMPP, SSH, and other intermittent interactive traffic, especially when all user traffic in that time period is multiplexed over a single connection (as it is with Tor).

Though flow measurement in principle can be bidirectional (counting cells sent in both directions between a pair of IPs) or unidirectional (counting only cells sent from one IP to another), we assume for safety that all measurement is unidirectional, and so traffic must be sent by both parties in order to prevent record splitting.

```
      * The fraction of descriptors that we have for nodes with the
Guard
        flag, weighted by their bandwidth for the guard position.
      * The fraction of descriptors that we have for all nodes,
        weighted by their bandwidth for the middle position.
      * The fraction of descriptors that we have for nodes with the
Exit
        flag, weighted by their bandwidth for the exit position.
```

If the consensus has zero weighted bandwidth for a given kind of relay (Guard, Middle, or Exit), Tor instead uses the fraction of relays for which it has the descriptor (not weighted by bandwidth at all).

If the consensus lists zero exit-flagged relays, Tor instead uses the fraction of middle relays.

Clients build circuits preemptively

When running as a client, Tor tries to maintain at least a certain number of clean circuits, so that new streams can be handled quickly. To increase the likelihood of success, Tor tries to predict what circuits will be useful by choosing from among nodes that support the ports we have used in the recent past (by default one hour). Specifically, on startup Tor tries to maintain one clean fast exit circuit that allows connections to port 80, and at least two fast clean stable internal circuits in case we get a resolve request or hidden service request (at least three if we *run* a hidden service).

After that, Tor will adapt the circuits that it preemptively builds based on the requests it sees from the user: it tries to have two fast clean exit circuits available for every port seen within the past hour (each circuit can be adequate for many predicted ports – it doesn’t need two separate circuits for each port), and it tries to have the above internal circuits available if we’ve seen resolves or hidden service activity within the past hour. If there are 12 or more clean circuits open, it doesn’t open more even if it has more predictions.

Only stable circuits can “cover” a port that is listed in the LongLivedPorts config option. Similarly, hidden service requests to ports listed in LongLivedPorts make us create stable internal circuits.

Note that if there are no requests from the user for an hour, Tor will predict no use and build no preemptive circuits.

The Tor client SHOULD NOT store its list of predicted requests to a persistent medium.

Clients build circuits on demand

Additionally, when a client request exists that no circuit (built or pending) might support, we create a new circuit to support the request. For exit connections, we pick an exit node that will handle the most pending requests (choosing arbitrarily among ties), launch a circuit to end there, and repeat until every unattached request might be supported by a pending or built circuit. For internal circuits, we pick an arbitrary acceptable path, repeating as needed.

Clients consider a circuit to become "dirty" as soon as a stream is attached to it, or some other request is performed over the circuit. If a circuit has been "dirty" for at least MaxCircuitDirtiness seconds, new circuits may not be attached to it.

In some cases we can reuse an already established circuit if it's clean; see "cannibalizing circuits"

for details.

Relays build circuits for testing reachability and bandwidth

Tor relays test reachability of their ORPort once they have successfully built a circuit (on startup and whenever their IP address changes). They build an ordinary fast internal circuit with themselves as the last hop. As soon as any testing circuit succeeds, the Tor relay decides it's reachable and is willing to publish a descriptor.

We launch multiple testing circuits (one at a time), until we have NUM_PARALLEL_TESTING_CIRC (4) such circuits open. Then we do a "bandwidth test" by sending a certain number of relay drop cells down each circuit: BandwidthRate * 10 / CELL_NETWORK_SIZE total cells divided across the four circuits, but never more than CIRCWINDOW_START (1000) cells total. This exercises both outgoing and incoming bandwidth, and helps to jumpstart the observed bandwidth (see dir-spec.txt).

Tor relays also test reachability of their DirPort once they have established a circuit, but they use an ordinary exit circuit for this purpose.

Hidden-service circuits

See section 4 below.

Connection-level padding

Background

Tor clients and relays make use of PADDING to reduce the resolution of connection-level metadata retention by ISPs and surveillance infrastructure.

Such metadata retention is implemented by Internet routers in the form of Netflow, jFlow, Netstream, or IPFIX records. These records are emitted by gateway routers in a raw form and then exported (often over plaintext) to a "collector" that either records them verbatim, or reduces their granularity further[1].

Netflow records and the associated data collection and retention tools are very configurable, and have many modes of operation, especially when configured to handle high throughput. However, at ISP scale, per-flow records are very likely to be employed, since they are the default, and also provide very high resolution in terms of endpoint activity, second only to full packet and/or header capture.

Per-flow records record the endpoint connection 5-tuple, as well as the total number of bytes sent and received by that 5-tuple during a particular time period. They can store additional fields as well, but it is primarily timing and bytecount information that concern us.

When configured to provide per-flow data, routers emit these raw flow records periodically for all active connections passing through them based on two parameters: the "active flow timeout" and the "inactive flow timeout".

The "active flow timeout" causes the router to emit a new record periodically for every active TCP session that continuously sends data. The default active flow timeout for most routers is 30 minutes, meaning that a new record is created for every TCP session at least every 30 minutes, no matter what. This value can be configured from 1 minute to 60 minutes on major routers.

The "inactive flow timeout" is used by routers to create a new record if a TCP session is inactive for some number of seconds. It allows routers to avoid the need to track a large number of idle connections in memory, and instead emit a separate record only when there is activity. This value ranges from 10 seconds to 600 seconds on common routers. It appears as though no routers support a value lower than 10 seconds.

Overview

Tor supports two classes of cover traffic: connection-level padding, and circuit-level padding.

Connection-level padding uses the PADDING cell command for cover traffic, where as circuit-level padding uses the DROP relay command. PADDING cells are single-hop only and can be differentiated from normal traffic by Tor relays (“internal” observers), but not by entities monitoring Tor OR connections (“external” observers).

DROP relay messages are multi-hop, and is not visible to intermediate Tor relays, because the relay command field is covered by circuit layer encryption. Moreover, Tor’s ‘recognized’ field allows DROP messages to be sent to any intermediate node in a circuit (as per Section 6.1 of tor-spec.txt).

Tor uses both connection level and circuit level padding. Connection level padding is described in section 2. Circuit level padding is described in section 3.

The circuit-level padding system is completely orthogonal to the connection-level padding. The connection-level padding system regards circuit-level padding as normal data traffic, and hence the connection-level padding system will not add any additional overhead while the circuit-level padding system is actively padding.

Rate limiting of failed circuits

If we fail to build a circuit N times in a X second period (see “Handling failure” for how this works), we stop building circuits until the X seconds have elapsed. XXXX

When to tear down circuits

Clients should tear down circuits (in general) only when those circuits have no streams on them. Additionally, clients should tear-down stream-less circuits only under one of the following conditions:

```
      - The circuit has never had a stream attached, and it was created
too
        long in the past (based on CircuitsAvailableTimeout or
        cbtlearntimeout, depending on timeout estimate status).

      - The circuit is dirty (has had a stream attached), and it has
been
        dirty for at least MaxCircuitDirtiness.
```

Path selection and constraints

We choose the path for each new circuit before we build it, based on our current directory information. (Clients and relays use the latest directory information they have; directory authorities use their own opinions.)

We choose the exit node first, followed by the other nodes in the circuit, front to back. (In other words, for a 3-hop circuit, we first pick hop 3, then hop 1, then hop 2.)

Universal constraints

~~All~~ Most paths we generate obey the following constraints:

- We do not choose relays without the Fast flag for any non-testing circuit.
- We do not choose the same router twice for the same path.
- We do not choose any router in the same family as another in the same path.
- We do not choose more than one router in a given network range, which defaults to /16 for IPv4 and /32 for IPv6. (C Tor overrides this with `EnforceDistinctSubnets`; Arti overrides this with `ipv[46]_subnet_family_prefix`.)
- The first node must be a Guard (see discussion below and in the guard specification).
- XXXX Choosing the length

> Note: There are exceptions to some of these rules, in order to balance the risk of traffic confirmation attacks and the risk of guard inference attacks. As of this writing (Feb 2025) we are revisiting these tradeoffs; see torspec#307.

Determining family membership

There are two mechanisms for determining whether two relays belong to the same family.

First, two relays belong to the same family if each relay lists the other relay in its *family list*. (A *family list* is the contents of a `family` entry in a router descriptor or a `family` entry in a microdescriptor.) This rule is only enforced when the `use-family-lists` parameter is set to 1.

Tor Padding Specification

Mike Perry, George Kadianakis

Note: This is an attempt to specify Tor as currently implemented. Future versions of Tor will implement improved algorithms.

This document tries to cover how Tor chooses to use cover traffic to obscure various traffic patterns from external and internal observers. Other implementations MAY take other approaches, but implementors should be aware of the anonymity and load-balancing implications of their choices.

```
def ProbR(N, r, ProbFunc=ProbMaxXX):
   return ProbFunc(N, r)*r/ExpFn(N, ProbFunc)
```

For the full CDF, we simply sum up the fractional probability density for all rotation durations. For rotation durations less than t days, we add the entire probability mass for that period to the density function. For durations d greater than t days, we take the fraction of that rotation period's selection probability and multiply it by t/d and add it to the density. In other words:

```
def FullCDF(N, t, ProbFunc=ProbR):
  density = 0.0
  for d in range(N):
    if t >= d: density += ProbFunc(N, d)
    # The +1's below compensate for 0-indexed arrays:
    else: density += ProbFunc(N, d)*(float(t+1))/(d+1)
  return density
```

Computing this yields the following distribution for our current parameters:

```
 t          P(SECOND_ROTATION <= t)
 1               0.03247
 2               0.06494
 3               0.09738
 4               0.12977
 5               0.16207
10               0.32111
15               0.47298
20               0.61353
25               0.73856
30               0.84391
35               0.92539
40               0.97882
45               1.00000
```

This CDF tells us that for the second-level Guard rotation, the adversary can expect that 3.3% of the time, their third-level Sybil attack will provide them with a second-level guard node that has only 1 day remaining before it rotates. 6.5% of the time, there will be only 2 day or less remaining, and 9.7% of the time, 3 days or less.

Note that this distribution is still a day-resolution approximation.

Second, two relays belong to the same family if they have at least one *family ID* in common. (A relay has a family ID if that ID is listed in the `family-ids` entry in its microdescriptor, or if that ID corresponds to the signing key of a well-formed `family-cert` entry in its router descriptor.) This rule is only enforced when the `use-family-ids` parameter is set to 1.

Special-purpose constraints

Additionally, we may be building circuits with one or more requests in mind. Each kind of request puts certain constraints on paths.

Similarly, some circuits need to be "Stable". For these, we only choose nodes with the Stable flag.

- All service-side introduction circuits and all rendezvous paths should be Stable, and their endpoints should not be flagged MiddleOnly.
- All connection requests for connections that we think will need to stay open a long time require Stable circuits. Currently, Tor decides this by examining the request's target port, and comparing it to a list of "long-lived" ports. (Default: 21, 22, 706, 1863, 5050, 5190, 5222, 5223, 6667, 6697, 8300.)

Weighting node selection

For all circuits, we weight node selection according to router bandwidth.

We also weight the bandwidth of Exit and Guard flagged nodes depending on the fraction of total bandwidth that they make up and depending upon the position they are being selected for.

These weights are published in the consensus, and are computed as described in "Computing Bandwidth Weights" in the directory specification. They are:

```
Wgg - Weight for Guard-flagged nodes in the guard position
Wgm - Weight for non-flagged nodes in the guard Position
Wgd - Weight for Guard+Exit-flagged nodes in the guard Position

Wmg - Weight for Guard-flagged nodes in the middle Position
Wmm - Weight for non-flagged nodes in the middle Position
Wme - Weight for Exit-flagged nodes in the middle Position
Wmd - Weight for Guard+Exit flagged nodes in the middle Position

Weg - Weight for Guard flagged nodes in the exit Position
Wem - Weight for non-flagged nodes in the exit Position
Wee - Weight for Exit-flagged nodes in the exit Position
Wed - Weight for Guard+Exit-flagged nodes in the exit Position

Wgb - Weight for BEGIN_DIR-supporting Guard-flagged nodes
Wmb - Weight for BEGIN_DIR-supporting non-flagged nodes
Web - Weight for BEGIN_DIR-supporting Exit-flagged nodes
Wdb - Weight for BEGIN_DIR-supporting Guard+Exit-flagged nodes

Wbg - Weight for Guard+Exit-flagged nodes for BEGIN_DIR requests
Wbm - Weight for Guard+Exit-flagged nodes for BEGIN_DIR requests
Wbe - Weight for Guard+Exit-flagged nodes for BEGIN_DIR requests
Wbd - Weight for Guard+Exit-flagged nodes for BEGIN_DIR requests
```

If any of those weights is malformed or not present in a consensus, clients proceed with the regular path selection algorithm setting the weights to the default value of 10000.

Choosing an exit

If we know what IP address we want to connect to, we can trivially tell whether a given router will support it by simulating its declared exit policy.

(DNS resolve requests are only sent to relays whose exit policy is not equivalent to "reject *:".)

Because we often connect to addresses of the form hostname:port, we do not always know the target IP address when we select an exit node. In these cases, we need to pick an exit node that "might support" connections to a given address port with an unknown address. An exit node "might support" such a connection if any clause that accepts any connections to that port precedes all clauses (if any) that reject all connections to that port.

Unless requested to do so by the user, we never choose an exit node flagged as "BadExit" by more than half of the authorities who advertise themselves as listing bad exits.

```
Range  Min(X,X)   Max(X,X)
22      6.84        14.16**
23      7.17        14.83
24      7.51        15.49
25      7.84        16.16
26      8.17        16.83
27      8.51        17.49
28      8.84        18.16
29      9.17        18.83
30      9.51        19.49
31      9.84        20.16
32      10.17       20.83
33      10.51       21.49
34      10.84       22.16
35      11.17       22.83
36      11.50       23.50
37      11.84       24.16
38      12.17       24.83
39      12.50       25.50
40      12.84       26.16
40      12.84       26.16
41      13.17       26.83
42      13.50       27.50
43      13.84       28.16
44      14.17       28.83
45      14.50       29.50
46      14.84       30.16
47      15.17       30.83
48      15.50       31.50***
```

The Cumulative Density Function (CDF) tells us the probability that a guard will no longer be in use after a given number of time units have passed.

Because the Sybil attack on the third node is expected to complete at any point in the second node's rotation period with uniform probability, if we want to know the probability that a second-level Guard node will still be in use after t days, we first need to compute the probability distribution of the rotation duration of the second-level guard at a uniformly random point in time. Let's call this P(R=r).

For P(R=r), the probability of the rotation duration depends on the selection probability of a rotation duration, and the fraction of total time that rotation is likely to be in use. This can be written as:

```
P(R=r) = ProbMaxXX(X=r)*r / \sum_{i=1}^N ProbMaxXX(X=i)*i
```

or in Python:

```
1     1       1
    25%             3     2     1     1     1     1     1     1
1     1       1
    50%             7     4     3     2     2     2     1     1
1     1       1
    60%             9     5     3     3     2     2     2     1
1     1       1
    75%            14     7     5     4     3     3     2     2
2     2       1
    85%            19    10     7     5     4     4     3     3
2     2       2
    90%            22    11     8     6     5     4     3     3
3     2       2
    95%            29    15    10     8     6     5     4     4
3     3       2
    99%            44    22    15    11     9     8     6     5
5     4       3
```

The rotation counts in these tables were generated with:

Skewed Rotation Distribution

In order to skew the distribution of the third layer guard towards higher values, we use max(X,X) for the distribution, where X is a random variable that takes on values from the uniform distribution.

Here's a table of expectation (arithmetic means) for relevant ranges of X (sampled from 0..N-1). The table was generated with the following python functions:

```
def ProbMinXX(N, i): return (2.0*(N-i)-1)/(N*N)
def ProbMaxXX(N, i): return (2.0*i+1)/(N*N)

def ExpFn(N, ProbFunc):
  exp = 0.0
  for i in range(N): exp += i*ProbFunc(N, i)
  return exp
```

The current choice for second-layer Vanguards-Lite guards is noted with **, and the current choice for third-layer Full Vanguards is noted with ***.

User configuration

Users can alter the default behavior for path selection with configuration options.

```
   - If "ExitNodes" is provided, then every request requires an exit
node on
     the ExitNodes list.  (If a request is supported by no nodes on
that list,
     and StrictExitNodes is false, then Tor treats that request as if
     ExitNodes were not provided.)

   - "EntryNodes" and "StrictEntryNodes" behave analogously.

   - If a user tries to connect to or resolve a hostname of the form
     <target>.<servername>.exit, the request is rewritten to a request
for
     <target>, and the request is only supported by the exit whose
nickname
     or fingerprint is <servername>.

   - When set, "HSLayer2Nodes" and "HSLayer3Nodes" relax Tor's path
     restrictions to allow nodes in the same /16 and node family to
reappear
     in the path. They also allow the guard node to be chosen as the
RP, IP,
     and HSDIR, and as the hop before those positions.
```

Cannibalizing circuits

If we need a circuit and have a clean one already established, in some cases we can adapt the clean circuit for our new purpose. Specifically,

For hidden service interactions, we can “cannibalize” a clean internal circuit if one is available, so we don’t need to build those circuits from scratch on demand.

We can also cannibalize clean circuits when the client asks to exit at a given node – either via the “.exit” notation or because the destination is running at the same location as an exit node.

```
  1.0% Network Compromise:
   Sybil Success   One   Two  Three  Four  Five  Six  Eight  Nine
Ten  Twelve  Sixteen
    10%            11     6     4     3     3     2     2     2
2     1       1
    15%            17     9     6     5     4     3     3     2
2     2       2
    25%            29    15    10     8     6     5     4     4
3     3       2
    50%            69    35    23    18    14    12     9     8
7     6       5
    60%            92    46    31    23    19    16    12    11
10     8       6
    75%           138    69    46    35    28    23    18    16
14    12       9
    85%           189    95    63    48    38    32    24    21
19    16      12
    90%           230   115    77    58    46    39    29    26
23    20      15
    95%           299   150   100    75    60    50    38    34
30    25      19
    99%           459   230   153   115    92    77    58    51
46    39      29

  5.0% Network Compromise:
   Sybil Success   One   Two  Three  Four  Five  Six  Eight  Nine
Ten  Twelve  Sixteen
    10%             3     2     1     1     1     1     1     1
1     1       1
    15%             4     2     2     1     1     1     1     1
1     1       1
    25%             6     3     2     2     2     1     1     1
1     1       1
    50%            14     7     5     4     3     3     2     2
2     2       1
    60%            18     9     6     5     4     3     3     2
2     2       2
    75%            28    14    10     7     6     5     4     4
3     3       2
    85%            37    19    13    10     8     7     5     5
4     4       3
    90%            45    23    15    12     9     8     6     5
5     4       3
    95%            59    30    20    15    12    10     8     7
6     5       4
    99%            90    45    30    23    18    15    12    10
9     8       6

  10.0% Network Compromise:
   Sybil Success   One   Two  Three  Four  Five  Six  Eight  Nine
Ten  Twelve  Sixteen
    10%             2     1     1     1     1     1     1     1
1     1       1
    15%             2     1     1     1     1     1     1     1
```

Vanguard Rotation Statistics

Sybil rotation counts for a given number of Guards

The probability of Sybil success for Guard discovery can be modeled as the probability of choosing 1 or more malicious middle nodes for a sensitive circuit over some period of time.

```
P(At least 1 bad middle) = 1 - P(All Good Middles)
                         = 1 - P(One Good middle)^(num_middles)
                         = 1 - (1 - c/n)^(num_middles)
```

c/n is the adversary compromise percentage

In the case of Vanguards, num_middles is the number of Guards you rotate through in a given time period. This is a function of the number of vanguards in that position (v), as well as the number of rotations (r).

```
P(At least one bad middle) = 1 - (1 - c/n)^(v*r)
```

Here's detailed tables in terms of the number of rotations required for a given Sybil success rate for certain number of guards.

Learning when to give up ("timeout") on circuit construction

Since version 0.2.2.8-alpha, Tor clients attempt to learn when to give up on circuits based on network conditions.

Distribution choice

Based on studies of build times, we found that the distribution of circuit build times appears to be a Frechet distribution (and a multi-modal Frechet distribution, if more than one guard or bridge is used). However, estimators and quantile functions of the Frechet distribution are difficult to work with and slow to converge. So instead, since we are only interested in the accuracy of the tail, clients approximate the tail of the multi-modal distribution with a single Pareto curve.

How much data to record

From our observations, the minimum number of circuit build times for a reasonable fit appears to be on the order of 100. However, to keep a good fit over the long term, clients store 1000 most recent circuit build times in a circular array.

These build times only include the times required to build three-hop circuits, and the times required to build the first three hops of circuits with more than three hops. Circuits of fewer than three hops are not recorded, and hops past the third are not recorded.

The Tor client should build test circuits at a rate of one every 'cbttestfreq' (10 seconds) until 'cbtmincircs' (100 circuits) are built, with a maximum of 'cbtmaxopencircs' (default: 10) circuits open at once. This allows a fresh Tor to have a CircuitBuildTimeout estimated within 30 minutes after install or network change (see Detecting Changing Network Conditions below.)

Timeouts are stored on disk in a histogram of 10ms bin width, the same width used to calculate the Xm value above. The timeouts recorded in the histogram must be shuffled after being read from disk, to preserve a proper expiration of old values after restart.

Thus, some build time resolution is lost during restart. Implementations may choose a different persistence mechanism than this histogram, but be aware that build time binning is still needed for parameter estimation.

Parameter estimation

Once 'cbtmincircs' build times are recorded, Tor clients update the distribution parameters and recompute the timeout every circuit completion (though see below for when to pause and reset timeout due to too many circuits timing out).

Tor clients calculate the parameters for a Pareto distribution fitting the data using the maximum likelihood estimator. For derivation, see: https://en.wikipedia.org/wiki/Pareto_distribution#Estimation_of_parameters

Because build times are not a true Pareto distribution, we alter how Xm is computed. In a max likelihood estimator, the mode of the distribution is used directly as Xm.

Instead of using the mode of discrete build times directly, Tor clients compute the Xm parameter using the weighted average of the midpoints of the 'cbtnummodes' (10) most frequently occurring 10ms histogram bins. Ties are broken in favor of earlier bins (that is, in favor of bins corresponding to shorter build times).

(The use of 10 modes was found to minimize error from the selected cbtquantile, with 10ms bins for quantiles 60-80, compared to many other heuristics).

To avoid ln(1.0+epsilon) precision issues, use log laws to rewrite the estimator for 'alpha' as the sum of logs followed by subtraction, rather than multiplication and division:

```
alpha = n/(Sum_n{ln(MAX(Xm, x_i))} - n\*ln(Xm))
```

In this, n is the total number of build times that have completed, x_i is the ith recorded build time, and Xm is the modes of x_i as above.

All times below Xm are counted as having the Xm value via the MAX(), because in Pareto estimators, Xm is supposed to be the lowest value. However, since clients use mode averaging to estimate Xm, there can be values below our Xm. Effectively, the Pareto estimator then treats that everything smaller than Xm happened at Xm. One can also see that if clients did not do this, alpha could underflow to become negative, which results in an exponential curve, not a Pareto probability distribution.

could ensure that it does not consist solely of the same family or /16, but this property cannot easily apply to conflux circuits).

The second change prevents an adversary from forcing the use of a different entry guard by enumerating all guard-flagged nodes as the RP. This change is important once onion services support conflux.

The third change prevents an adversary from learning the guard node by way of noticing which nodes were not chosen for the hop before it. This change is important once onion services support conflux.

Path Construction

Both vanguards systems use a mesh topology: this means that circuits select a hop from each layer independently, allowing paths from any relay in a layer to any relay in the next layer.

Selecting Relays

Vanguards relays are selected from relays with the Stable and Fast flags.

Tor replaces a vanguard whenever it is no longer listed in the most recent consensus, with the goal that we will always have the right number of vanguards ready to be used.

For implementation reasons, we also replace a vanguard if it loses the Fast or Stable flag, because the path selection logic wants middle nodes to have those flags when it's building preemptive vanguard-using circuits.

The design doesn't have to be this way: we might instead have chosen to keep vanguards in our list as long as possible, and continue to use them even if they have lost some flags. This tradeoff is similar to the one in Bug #17773, about whether to continue using Entry Guards if they lose the Guard flag – and Tor's current choice is "no, rotate" for that case too.

Path Restriction Changes

Path restrictions, as well as the ordering of their application, are currently extremely problematic, resulting in information leaks with this topology. Until they are reworked, we disable many of them for onion service circuits.

In particular, we allow the following:

1. Nodes from the same /16 and same family for any/all hops in a circuit
2. Guard nodes can be chosen for RP/IP/HSDIR
3. Guard nodes can be chosen for hop before RP/IP/HSDIR.

The first change prevents the situation where paths cannot be built if two layers all share the same subnet and/or node family, or if a layer consists of only one family and that family is the same as the RP/IP/HSDIR. It also prevents the the use of a different entry guard based on the family or subnet of the IP, HSDIR, or RP. (Alternatives of this permissive behavior are possible: For example, each layer

The timeout itself is calculated by using the Pareto Quantile function (the inverted CDF) to give us the value on the CDF such that 80% of the mass of the distribution is below the timeout value (parameter 'cbtquantile').

The Pareto Quantile Function (inverse CDF) is:

```
F(q) = Xm/((1.0-q)^(1.0/alpha))
```

Thus, clients obtain the circuit build timeout for 3-hop circuits by computing:

```
timeout_ms = F(0.8)    # 'cbtquantile' == 0.8
```

With this, we expect that the Tor client will accept the fastest 80% of the total number of paths on the network.

Clients obtain the circuit close time to completely abandon circuits as:

```
close_ms = F(0.99)     # 'cbtclosequantile' == 0.99
```

To avoid waiting an unreasonably long period of time for circuits that simply have relays that are down, Tor clients cap timeout_ms at the max build time actually observed so far, and cap close_ms at twice this max, but at least 60 seconds:

```
     timeout_ms = MIN(timeout_ms, max_observed_timeout)
     close_ms = MAX(MIN(close_ms, 2*max_observed_timeout),
'cbtinitialtimeout')
```

Calculating timeouts thresholds for circuits of different lengths

The timeout_ms and close_ms estimates above are good only for 3-hop circuits, since only 3-hop circuits are recorded in the list of build times.

To calculate the appropriate timeouts and close timeouts for circuits of other lengths, the client multiples the timeout_ms and close_ms values by a scaling factor determined by the number of communication hops needed to build their circuits:

```
timeout_ms\[hops=n\] = timeout_ms * Actions(N) / Actions(3)

close_ms\[hops=n\] = close_ms * Actions(N) / Actions(3)
```

where `Actions(N) = N * (N + 1) / 2.`

To calculate timeouts for operations other than circuit building, the client should add X to Actions(N) for every round-trip communication required with the Xth hop.

How to record timeouts

Pareto estimators begin to lose their accuracy if the tail is omitted. Hence, Tor clients actually calculate two timeouts: a usage timeout, and a close timeout.

Circuits that pass the usage timeout are marked as measurement circuits, and are allowed to continue to build until the close timeout corresponding to the point 'cbtclosequantile' (default 99) on the Pareto curve, or 60 seconds, whichever is greater.

The actual completion times for these measurement circuits should be recorded.

Implementations should completely abandon a circuit and ignore the circuit if the total build time exceeds the close threshold. Such closed circuits should be ignored, as this typically means one of the relays in the path is offline.

Detecting Changing Network Conditions

Tor clients attempt to detect both network connectivity loss and drastic changes in the timeout characteristics.

To detect changing network conditions, clients keep a history of the timeout or non-timeout status of the past 'cbtrecentcount' circuits (20 circuits) that successfully completed at least one hop. If more than 90% of these circuits timeout, the client discards all buildtimes history, resets the timeout to 'cbtinitialtimeout' (60 seconds), and then begins recomputing the timeout.

If the timeout was already at least `cbtinitialtimeout`, the client doubles the timeout.

The records here (of how many circuits succeeded or failed among the most recent 'cbrrecentcount') are not stored as persistent state. On reload, we start with a new, empty state.

Consensus parameters governing behavior

Clients that implement circuit build timeout learning should obey the following consensus parameters that govern behavior, in order to allow us to handle bugs

Vanguards-Lite

Vanguards-Lite is meant for short-lived onion services such as OnionShare, as well as for onion client activity.

It is designed for clients and services that expect to have an uptime of no longer than 1 month.

Design

Because it is for short-lived activity, its rotation times are chosen with the Sybil adversary in mind, using the max(X,X) skewed distribution.

We let NUM_LAYER2_GUARDS=4. We also introduce a consensus parameter `guard-hs-l2-number` that controls the number of layer2 guards (with a maximum of 19 layer2 guards).

No third layer of guards is used.

We don't write guards on disk. This means that the guard topology resets when tor restarts.

Rotation Period

The Layer2 lifetime uses the max(x,x) distribution with a minimum of one day and maximum of 22 days. This makes the average lifetime of approximately two weeks. Significant extensions of this lifetime are not recommended, as they will shift the balance in favor of coercive attacks.

From the Sybil Rotation Table, with NUM_LAYER2_GUARDS=4 it can be seen that this means that the Sybil attack on Layer2 will complete with 50% chance in 18*14 days (252 days) for the 1% adversary, 4*14 days (two months) for the 5% adversary, and 2*14 days (one month) for the 10% adversary.

50% chance in 9*31.5 hours (15.75 days) for the 1% adversary, ~4 days for the 5% adversary, and 2.62 days for the 10% adversary.

See the statistical analysis for more analysis on these constants.

or other emergent behaviors due to client circuit construction. If these parameters are not present in the consensus, the listed default values should be used instead.

```
      cbtdisabled
        Default: 0
        Min: 0
        Max: 1
        Effect: If 1, all CircuitBuildTime learning code should be
                disabled and history should be discarded. For use in
                emergency situations only.

      cbtnummodes
        Default: 10
        Min: 1
        Max: 20
        Effect: This value governs how many modes to use in the
weighted
        average calculation of Pareto parameter Xm. Selecting Xm as
the
        average of multiple modes improves accuracy of the Pareto tail
        for quantile cutoffs from 60-80% (see cbtquantile).

      cbtrecentcount
        Default: 20
        Min: 3
        Max: 1000
        Effect: This is the number of circuit build outcomes (success
vs
                timeout) to keep track of for the following option.

      cbtmaxtimeouts
        Default: 18
        Min: 3
        Max: 10000
        Effect: When this many timeouts happen in the last
'cbtrecentcount'
                circuit attempts, the client should discard all of its
                history and begin learning a fresh timeout value.

                Note that if this parameter's value is greater than
the value
                of 'cbtrecentcount', then the history will never be
                discarded because of this feature.

      cbtmincircs
        Default: 100
        Min: 1
        Max: 10000
        Effect: This is the minimum number of circuits to build before
                computing a timeout.

                Note that if this parameter's value is higher than
1000 (the
                number of time observations that a client keeps in its
                circular buffer), circuit build timeout calculation is
                effectively disabled, and the default timeouts are
used
```

favor of a somewhat lengthy Sybil attack. For this reason, users SHOULD NOT be encouraged to pin this layer.

> Note that in practice, our implementation uses the MAX(X,X) distribution in both cases (for both the `second_guard_set` and the 'third_guard_set') for the sake of simplicity.

Each relay's rotation time is tracked independently, to avoid disclosing the rotation times of the primary and second-level guards.

The selected vanguards and their rotation timestamp MUST be persisted to disk.

Parameterization

We set NUM_LAYER2_GUARDS to 4 relays and NUM_LAYER3_GUARDS to 6 relays.

We set MIN_SECOND_GUARD_LIFETIME to 30 days, and MAX_SECOND_GUARD_LIFETIME to 60 days inclusive, for an average rotation rate of 45 days, using a uniform distribution. This range was chosen to average out to half of the Guard rotation period; there is no strong motivation for it otherwise, other than to be long. In fact, it could be set as long as the Guard rotation, and longer periods MAY be provided as a configuration parameter.

From the Sybil rotation table in statistical analysis, with NUM_LAYER2_GUARDS=4, it can be seen that this means that the Sybil attack on layer3 will complete with 50% chance in 18*45 days (2.2 years) for the 1% adversary, 180 days for the 5% adversary, and 90 days for the 10% adversary, with a 45 day average rotation period.

If this range is set equal to the Guard rotation period (90 days), the 50% probability of Sybil success requires 18*90 days (4.4 years) for the 1% adversary, 4*90 days (1 year) for the 5% adversary, and 2*90 days (6 months) for the 10% adversary.

We set MIN_THIRD_GUARD_LIFETIME to 1 hour, and MAX_THIRD_GUARD_LIFETIME to 48 hours inclusive, for an average rotation rate of 31.5 hours, using the max(X,X) distribution. (Again, this wide range and bias is used to discourage the adversary from exclusively performing coercive attacks, as opposed to mounting the Sybil attack, so increasing it substantially is not recommended).

From the Sybil rotation table in statistical analysis, with NUM_LAYER3_GUARDS=6, it can be seen that this means that the Sybil attack on layer3 will complete with

relay that the adversary is attempting to discover. When Tor's internal protocol side channels are dealt with, this will be both observable and controllable at the Application Layer, by operators.

3. The adversary is strongly disincentivized from compromising additional relays that may prove useless, as active compromise attempts are even more risky for the adversary than a Sybil attack in terms of being noticed. In other words, the adversary is unlikely to attempt to compromise or coerce additional relays that are in use for only a short period of time.

Given this threat model, our security parameters were selected so that the first two layers of guards should take a very long period of time to attack using a Sybil guard discovery attack and hence require a relay compromise attack.

On the other hand, the outermost layer of guards (the third layer) should rotate fast enough to *require* a Sybil attack. If the adversary were to attempt to compromise or coerce these relays after they are discovered, their rotation times should be fast enough that the adversary has a very high probability of them no longer being in use.

Design

When a hidden service picks its guard relays, it also picks an additional NUM_LAYER2_GUARDS-sized set of middle relays for its `second_guard_set`, as well as a NUM_LAYER3_GUARDS-sized set of middle relays for its `third_guard_set`.

When a hidden service needs to establish a circuit to an HSDir, introduction point or a rendezvous point, it uses relays from `second_guard_set` as the second hop of the circuit and relays from `third_guard_set` as third hop of the circuit.

A hidden service rotates relays from the 'second_guard_set' at a uniformly random time between MIN_SECOND_GUARD_LIFETIME hours and MAX_SECOND_GUARD_LIFETIME hours, chosen for each layer 2 relay. Implementations MAY expose this layer as an explicit configuration option to pin specific relays, similar to how a user is allowed to pin specific Guards.

A hidden service rotates relays from the 'third_guard_set' at a random time between MIN_THIRD_GUARD_LIFETIME and MAX_THIRD_GUARD_LIFETIME hours, as weighted by the max(X,X) distribution, chosen for each relay. This skewed distribution was chosen so that there is some probability of a very short rotation period, to deter compromise/coercion, but biased towards the longer periods, in

```
                indefinitely.

      cbtquantile
        Default: 80
        Min: 10
        Max: 99
        Effect: This is the position on the quantile curve to use to
set the
                timeout value. It is a percent (10-99).

      cbtclosequantile
        Default: 99
        Min: Value of cbtquantile parameter
        Max: 99
        Effect: This is the position on the quantile curve to use to
set the
                timeout value to use to actually close circuits. It is
a
                percent (0-99).

      cbttestfreq
        Default: 10
        Min: 1
        Max: 2147483647 (INT32_MAX)
        Effect: Describes how often in seconds to build a test circuit
to
                gather timeout values. Only applies if less than
'cbtmincircs'
                have been recorded.

      cbtmintimeout
        Default: 10
        Min: 10
        Max: 2147483647 (INT32_MAX)
        Effect: This is the minimum allowed timeout value in
milliseconds.

      cbtinitialtimeout
        Default: 60000
        Min: Value of cbtmintimeout
        Max: 2147483647 (INT32_MAX)
        Effect: This is the timeout value to use before we have enough
data
                to compute a timeout, in milliseconds.  If we do not
have
                enough data to compute a timeout estimate (see
cbtmincircs),
                then we use this interval both for the close timeout
and the
                abandon timeout.

      cbtlearntimeout
        Default: 180
        Min: 10
        Max: 60000
```

```
        Effect: This is how long idle circuits will be kept open while
cbt is
                learning a new timeout value.

      cbtmaxopencircs
        Default: 10
        Min: 0
        Max: 14
        Effect: This is the maximum number of circuits that can be
open at
                at the same time during the circuit build time
learning phase.
```

Full Vanguards

Full Vanguards is intended for use by long-lived onion services, which intend to remain in operation for longer than one month.

Full Vanguards achieves this longer expected duration by having two layers of additional fixed relays, of different rotation periods.

The rotation period of the first vanguard layer (layer 2 guards) is chosen such that it requires an extremely long and persistent Sybil attack, or a coercion/ compromise attack.

The rotation period of the second vanguard layer (layer 3 guards) is chosen to be small enough to force the adversary to perform a Sybil attack against this layer, rather than attempting to coerce these relays.

Threat model, Assumptions, and Goals

Consider an adversary with the following powers:

```
     - Can launch a Sybil guard discovery attack against any position
of a
       rendezvous circuit. The slower the rotation period of their
target
       position, the longer the attack takes. Similarly, the higher
the
       percentage of the network is controlled by them, the faster the
       attack runs.

     - Can compromise additional relays on the network, but this
compromise
       takes time and potentially even coercive action, and also
carries risk
       of discovery.
```

We also make the following assumptions about the types of attacks:

1. A Sybil attack is observable by both people monitoring the network for large numbers of new relays, as well as vigilant hidden service operators. It will require large amounts of traffic sent towards the hidden service over many many test circuits.

2. A Sybil attack requires either a protocol side channel or an application-layer timing side channel in order to determine successful placement next to the

Alternatives

An alternative to vanguards for client activity is to restrict the number of onion services that a Tor client is allowed to connect to, in a certain period of time. This defense was explored in Onion Not Found.

We have opted not to deploy this defense, for three reasons:

1. It does not generalize to the service-side of onion services
2. Setting appropriate rate limits on the number of onion service content elements on a page for Tor Browser is difficult. Sites like Facebook use multiple onion domains for various content elements on a single page.
3. It is even more difficult to limit the number of service connections for arbitrary applications, such as cryptocurrency wallets, mining, and other distributed apps deployed on top of onion services that connect to multiple services (such as Ricochet).

Handling failure

If an attempt to extend a circuit fails (either because the first create failed or a subsequent extend failed) then the circuit is torn down and is no longer pending. (XXXX really?) Requests that might have been supported by the pending circuit thus become unsupported, and a new circuit needs to be constructed.

If a stream “begin” attempt fails with an EXITPOLICY error, we decide that the exit node’s exit policy is not correctly advertised, so we treat the exit node as if it were a non-exit until we retrieve a fresh descriptor for it.

Excessive amounts of either type of failure can indicate an attack on anonymity. See discussion of path bias detection for how excessive failure is handled.

Attaching streams to circuits

When a circuit that might support a request is built, Tor tries to attach the request's stream to the circuit and sends a BEGIN, BEGIN_DIR, or RESOLVE relay cell as appropriate. If the request completes unsuccessfully, Tor considers the reason given in the CLOSE relay cell. [XXX yes, and?]

After a request has remained unattached for SocksTimeout (2 minutes by default), Tor abandons the attempt and signals an error to the client as appropriate (e.g., by closing the SOCKS connection).

XXX Timeouts and when Tor auto-retries.

- What stream-end-reasons are appropriate for retrying.

If no reply to BEGIN/RESOLVE, then the stream will timeout and fail.

Visualizing Full Vanguards

Full Vanguards pins these two middle positions into a mesh topology, where any relay in a layer can be used in that position in a circuit, as follows:

```
                  -> vanguard_2A
                                  -> vanguard_3A
   -> guard_1A  -> vanguard_2B -> vanguard_3B
HS                                -> vanguard_3C
   -> guard_1B  -> vanguard_2C -> vanguard_3D
                                  -> vanguard_3E
                  -> vanguard_2D -> vanguard_3F
```

Additionally, to avoid trivial discovery of the third layer, and to minimize linkability, we insert an extra middle relay after the third layer guard for client side intro and hsdir circuits, and service-side rendezvous circuits. This means that the set of paths for Client (C) and Service (S) side look like this:

```
Client hsdir:  C - G - L2 - L3 - M - HSDIR
Client intro:  C - G - L2 - L3 - M - I
Client rend:   C - G - L2 - L3 - R
Service hsdir: S - G - L2 - L3 - HSDIR
Service intro: S - G - L2 - L3 - I
Service rend:  S - G - L2 - L3 - M - R
```

Visualizing Vanguards-Lite

Vanguards-Lite uses only one layer of vanguards:

```
                  -> vanguard_2A

   -> guard_1A  -> vanguard_2B
HS
   -> guard_1B  -> vanguard_2C

                  -> vanguard_2D
```

This yields shorter path lengths, of the following form:

```
Client hsdir:  C -> G -> L2 -> M -> HSDir
Client intro:  C -> G -> L2 -> M -> Intro
Client rend:   C -> G -> L2 -> Rend
Service hsdir: C -> G -> L2 -> M -> HSDir
Service intro: C -> G -> L2 -> M -> Intro
Service rend:  C -> G -> L2 -> M -> Rend
```

Overview

In this specification, we specify two forms of a multi-layered Guard system: one for long-lived services, called Full Vanguards, and one for onion clients and short-lived services, called Vanguards-Lite.

Both approaches use a mesh topology, where circuits can be created from any relay in a preceding layer to any relay in a subsequent layer.

The core difference between these two mechanisms is that Full Vanguards has two additional layers of fixed vanguard relays, which results in longer path lengths and higher latency. In contrast, Vanguards-Lite has only one additional layer of fixed vanguard relays, and preserves the original path lengths in use by onion services. Thus, Full Vanguards comes with a performance cost, where as Vanguards-Lite does not. The rotation periods of the two approaches also differ.

Vanguards-Lite MUST be the default for all onion service and onion client activity; Full Vanguards SHOULD be available as an optional configuration option for services.

Neither system applies to Exit activity.

Terminology

Tor's original guards are called First Layer Guards.

The first layer of vanguards is at the second hop, and is called the Second Layer Guards.

The second layer of vanguards is at the third hop, and is called the Third Layer Guards.

A circuit stem is the beginning portion of a hidden service circuit that is common to all hidden service circuit types, under a given vanguards behavior option (none, lite, or full). Abstracting this common portion is useful for circuit prebuilding, to maintain a pool of circuit prefixes (stems) that can be used for any hidden service circuit.

> Note: In Arti, this circuit stem prefix is further abstracted, to handle the fact that some hidden circuit types require an additional random middle node, where as others do not (see diagrams below). Other than easing circuit pre-building, this distinction has no consequence on paths. The below diagrams are canonical behavior for each given circuit type.

Hidden-service related circuits

XXX Tracking expected hidden service use (client-side and hidserv-side)

Guard nodes

We use Guard nodes (also called “helper nodes” in the research literature) to prevent certain profiling attacks. For an overview of our Guard selection algorithm – which has grown rather complex – see guard-spec.txt.

How consensus bandwidth weights factor into entry guard selection

When weighting a list of routers for choosing an entry guard, the following consensus parameters (from the “bandwidth-weights” line) apply:

```
Wgg - Weight for Guard-flagged nodes in the guard position
Wgm - Weight for non-flagged nodes in the guard Position
Wgd - Weight for Guard+Exit-flagged nodes in the guard Position
Wgb - Weight for BEGIN_DIR-supporting Guard-flagged nodes
Wmb - Weight for BEGIN_DIR-supporting non-flagged nodes
Web - Weight for BEGIN_DIR-supporting Exit-flagged nodes
Wdb - Weight for BEGIN_DIR-supporting Guard+Exit-flagged nodes
```

Please see “bandwidth-weights” in §3.4.1 of dir-spec.txt for more in depth descriptions of these parameters.

If a router has been marked as both an entry guard and an exit, then we prefer to use it more, with our preference for doing so (roughly) linearly increasing w.r.t. the router’s non-guard bandwidth and bandwidth weight (calculated without taking the guard flag into account). From proposal 236:

| | Let Wpf denote the weight from the ‘bandwidth-weights’ line a | client would apply to N for position p if it had the guard | flag, Wpn the weight if it did not have the guard flag, and B the | measured bandwidth of N in the consensus. Then instead of choosing | N for position p proportionally to Wpf*B or Wpn*B, clients should | choose N proportionally to F*Wpf*B + (1-F)*Wpn*B.

where F is the weight as calculated using the above parameters.

Tor Vanguards Specification

Introduction and motivation

A guard discovery attack allows attackers to determine the guard relay of a Tor client. The hidden service protocol provides an attack vector for a guard discovery attack since anyone can force an HS to construct a 3-hop circuit to a relay, and repeat this process until one of the adversary’s middle relays eventually ends up chosen in a circuit. These attacks are also possible to perform against clients, by causing an application to make repeated connections to multiple unique onion services.

The adversary must use a protocol side channel to confirm that their relay was chosen in this position (see Proposal #344), and then learns the guard relay of a client, or service.

When a successful guard discovery attack is followed with compromise or coercion of the guard relay, the onion service or onion client can be deanonymized. Alternatively, Netflow analytics data purchase can be (and has been) used for this deanonymization, without interacting with the Guard relay directly (see again Proposal #344).

This specification assumes that Tor protocol side channels have 100% accuracy and are undetectable, for simplicity in reasoning about expected attack times. Presently, such 100% accurate side channels exist in silent form, in the Tor Protocol itself.

As work on addressing Tor’s protocol side channels progresses, these attacks will require application-layer activity that can be monitored and managed by service operators, as opposed to silent and unobservable side channel activity via the Tor Protocol itself. Application-layer side channels are also expected to have less accuracy than native Tor protocol side channels, due to the possibility of false positives caused by similar application activity elsewhere on the Tor network. Despite this, we will preserve the original assumption of 100% accuracy, for simplicity of explanation.

```
        "in" -- the name of the guard state instance that this
        sampled guard is in.  If a sampled guard is in two guard
        states instances, it appears twice, with a different "in"
        field each time. Required.

        "rsa_id" -- the RSA id digest for this guard, encoded in
        hex. Required.

        "bridge_addr" -- If the guard is a bridge, its configured
address and
        port (this can be the ORPort or a pluggable transport port).
Optional.

        "nickname" -- the guard's nickname, if any. Optional.

        "sampled_on" -- the date when the guard was sampled. Required.

        "sampled_by" -- the Tor version that sampled this guard.
        Optional.

        "unlisted_since" -- the date since which the guard has been
        unlisted. Optional.

        "listed" -- 0 if the guard is not listed; 1 if it is.
Required.

        "confirmed_on" -- date when the guard was
        confirmed. Optional.

        "confirmed_idx" -- position of the guard in the confirmed
        list. Optional.

        "pb_use_attempts", "pb_use_successes", "pb_circ_attempts",
        "pb_circ_successes", "pb_successful_circuits_closed",
        "pb_collapsed_circuits", "pb_unusable_circuits",
        "pb_timeouts" -- state for the circuit path bias algorithm,
        given in decimal fractions. Optional.
```

All dates here are given as a (spaceless) ISO8601 combined date and time in UTC (e.g., 2016-11-29T19:39:31).

Server descriptor purposes

There are currently three “purposes” supported for server descriptors: general, controller, and bridge. Most descriptors are of type general – these are the ones listed in the consensus, and the ones fetched and used in normal cases.

Controller-purpose descriptors are those delivered by the controller and labelled as such: they will be kept around (and expire like normal descriptors), and they can be used by the controller in its CIRCUITEXTEND commands. Otherwise they are ignored by Tor when it chooses paths.

Bridge-purpose descriptors are for routers that are used as bridges. See doc/design-paper/blocking.pdf for more design explanation, or proposal 125 for specific details. Currently bridge descriptors are used in place of normal entry guards, for Tor clients that have UseBridges enabled.

Detecting route manipulation by Guard nodes (Path Bias)

The Path Bias defense is designed to defend against a type of route capture where malicious Guard nodes deliberately fail or choke circuits that extend to non-colluding Exit nodes to maximize their network utilization in favor of carrying only compromised traffic.

In the extreme, the attack allows an adversary that carries c/n of the network capacity to deanonymize c/n of the network connections, breaking the O((c/n)^2) property of Tor's original threat model. It also allows targeted attacks aimed at monitoring the activity of specific users, bridges, or Guard nodes.

There are two points where path selection can be manipulated: during construction, and during usage. Circuit construction can be manipulated by inducing circuit failures during circuit extend steps, which causes the Tor client to transparently retry the circuit construction with a new path. Circuit usage can be manipulated by abusing the stream retry features of Tor (for example by withholding stream attempt responses from the client until the stream timeout has expired), at which point the tor client will also transparently retry the stream on a new path.

The defense as deployed therefore makes two independent sets of measurements of successful path use: one during circuit construction, and one during circuit usage.

The intended behavior is for clients to ultimately disable the use of Guards responsible for excessive circuit failure of either type (for the parameters to do this, see "Parameterization" below); however known issues with the Tor network currently restrict the defense to being informational only at this stage (see "Known barriers to enforcement").

Measuring path construction success rates

Clients maintain two counts for each of their guards: a count of the number of times a circuit was extended to at least two hops through that guard, and a count of the number of circuits that successfully complete through that guard. The ratio of these two numbers is used to determine a circuit success rate for that Guard.

Random values

Frequently, we want to randomize the expiration time of something so that it's not easy for an observer to match it to its start time. We do this by randomizing its start date a little, so that we only need to remember a fixed expiration interval.

By RAND(now, INTERVAL) we mean a time between now and INTERVAL in the past, chosen uniformly at random.

Persistent state format

The persistent state format doesn't need to be part of this specification, since different implementations can do it differently. Nonetheless, here's the one Tor uses:

The "state" file contains one Guard entry for each sampled guard in each instance of the guard state (see section 2). The value of this Guard entry is a set of space-separated K=V entries, where K contains any nonspace character except =, and V contains any nonspace characters.

Implementations must retain any unrecognized K=V entries for a sampled guard when they regenerate the state file.

The order of K=V entries is not allowed to matter.

Recognized fields (values of K) are:

```
{param:REMOVE_UNLISTED_GUARDS_AFTER} -- 20 days
  [previously ENTRY_GUARD_REMOVE_AFTER]

{param:MIN_FILTERED_SAMPLE} -- 20

{param:N_PRIMARY_GUARDS} -- 3

{param:PRIMARY_GUARDS_RETRY_SCHED}

   We recommend the following schedule, which is the one
   used in Arti:

   -- Use the "decorrelated-jitter" algorithm from "dir-spec.txt"
      section 5.5 where `base_delay` is 30 seconds and `cap`
      is 6 hours.

   This legacy schedule is the one used in C tor:

   -- every 10 minutes for the first six hours,
   -- every 90 minutes for the next 90 hours,
   -- every 4 hours for the next 3 days,
   -- every 9 hours thereafter.

{param:GUARDS_RETRY_SCHED} --

   We recommend the following schedule, which is the one
   used in Arti:

   -- Use the "decorrelated-jitter" algorithm from "dir-spec.txt"
      section 5.5 where `base_delay` is 10 minutes and `cap`
      is 36 hours.

   This legacy schedule is the one used in C tor:

   -- every hour for the first six hours,
   -- every 4 hours for the 90 hours,
   -- every 18 hours for the next 3 days,
   -- every 36 hours thereafter.

{param:INTERNET_LIKELY_DOWN_INTERVAL} -- 10 minutes

{param:NONPRIMARY_GUARD_CONNECT_TIMEOUT} -- 15 seconds

{param:NONPRIMARY_GUARD_IDLE_TIMEOUT} -- 10 minutes

{param:MEANINGFUL_RESTRICTION_FRAC} -- .2

{param:EXTREME_RESTRICTION_FRAC} -- .01

{param:GUARD_CONFIRMED_MIN_LIFETIME} -- 60 days

{param:NUM_USABLE_PRIMARY_GUARDS} -- 1

{param:NUM_USABLE_PRIMARY_DIRECTORY_GUARDS} -- 3
```

Circuit build timeouts are counted as construction failures if the circuit fails to complete before the 95% “right-censored” timeout interval, not the 80% timeout condition.

If a circuit closes prematurely after construction but before being requested to close by the client, this is counted as a failure.

Measuring path usage success rates

Clients maintain two usage counts for each of their guards: a count of the number of usage attempts, and a count of the number of successful usages.

A usage attempt means any attempt to attach a stream to a circuit.

Usage success status is temporarily recorded by state flags on circuits. Guard usage success counts are not incremented until circuit close. A circuit is marked as successfully used if we receive a properly recognized RELAY cell on that circuit that was expected for the current circuit purpose.

If subsequent stream attachments fail or time out, the successfully used state of the circuit is cleared, causing it once again to be regarded as a usage attempt only.

Upon close by the client, all circuits that are still marked as usage attempts are probed using a RELAY_BEGIN cell constructed with a destination of the form 0.a.b.c:25, where a.b.c is a 24 bit random nonce. If we get a RELAY_COMMAND_END in response matching our nonce, the circuit is counted as successfully used.

If any unrecognized RELAY cells arrive after the probe has been sent, the circuit is counted as a usage failure.

If the stream failure reason codes DESTROY, TORPROTOCOL, or INTERNAL are received in response to any stream attempt, such circuits are not probed and are declared usage failures.

Prematurely closed circuits are not probed, and are counted as usage failures.

Scaling success counts

To provide a moving average of recent Guard activity while still preserving the ability to verify correctness, we periodically “scale” the success counts by multiplying them by a scale factor between 0 and 1.0.

Scaling is performed when either usage or construction attempt counts exceed a parametrized value.

To avoid error due to scaling during circuit construction and use, currently open circuits are subtracted from the usage counts before scaling, and added back after scaling.

Parametrization

The following consensus parameters tune various aspects of the defense.

Appendices

Acknowledgements

This research was supported in part by NSF grants CNS-1111539, CNS-1314637, CNS-1526306, CNS-1619454, and CNS-1640548.

Parameters with suggested values.

(All suggested values chosen arbitrarily)

{param:MAX_SAMPLE_THRESHOLD} – 20%

{param:MAX_SAMPLE_SIZE} – 60

{param:GUARD_LIFETIME} – 120 days

We generate a new circuit when we don't have enough circuits either built or in-progress to handle a given stream, or an expected stream.

For the purpose of this rule, we say that <waiting_for_better_guard> circuits are neither built nor in-progress; that <complete> circuits are built; and that the other states are in-progress.

4.12. When we are missing descriptors. [Section:MISSING_DESCRIPTORS]

We need either a router descriptor or a microdescriptor in order to build a circuit through a guard. If we do not have such a descriptor for a guard, we can still use the guard for one-hop directory fetches, but not for longer circuits.

(Also, when we are missing descriptors for our first {NUM_USABLE_PRIMARY_GUARDS} primary guards, we don't build circuits at all until we have fetched them.)

```
     pb_mincircs
       Default: 150
       Min: 5
       Effect: This is the minimum number of circuits that must
complete
               at least 2 hops before we begin evaluating construction
rates.

     pb_noticepct
       Default: 70
       Min: 0
       Max: 100
       Effect: If the circuit success rate falls below this
percentage,
               we emit a notice log message.

     pb_warnpct
       Default: 50
       Min: 0
       Max: 100
       Effect: If the circuit success rate falls below this
percentage,
               we emit a warn log message.

     pb_extremepct
       Default: 30
       Min: 0
       Max: 100
       Effect: If the circuit success rate falls below this
percentage,
               we emit a more alarmist warning log message. If
               pb_dropguard is set to 1, we also disable the use of
the
               guard.

     pb_dropguards
       Default: 0
       Min: 0
       Max: 1
       Effect: If the circuit success rate falls below pb_extremepct,
               when pb_dropguard is set to 1, we disable use of that
               guard.

     pb_scalecircs
       Default: 300
       Min: 10
       Effect: After this many circuits have completed at least two
hops,
               Tor performs the scaling described in
               ["Scaling success counts"](#scaling).

     pb_multfactor and pb_scalefactor
       Default: 1/2
       Min: 0.0
```

```
       Max: 1.0
       Effect: The double-precision result obtained from
               pb_multfactor/pb_scalefactor is multiplied by our
current
               counts to scale them.

     pb_minuse
       Default: 20
       Min: 3
       Effect: This is the minimum number of circuits that we must
attempt to
               use before we begin evaluating construction rates.

     pb_noticeusepct
       Default: 80
       Min: 3
       Effect: If the circuit usage success rate falls below this
percentage,
               we emit a notice log message.

     pb_extremeusepct
       Default: 60
       Min: 3
       Effect: If the circuit usage success rate falls below this
percentage,
               we emit a warning log message. We also disable the use
of the
               guard if pb_dropguards is set.

     pb_scaleuse
       Default: 100
       Min: 10
       Effect: After we have attempted to use this many circuits,
               Tor performs the scaling described in
               ["Scaling success counts"](#scaling).
```

Known barriers to enforcement

Due to intermittent CPU overload at relays, the normal rate of successful circuit completion is highly variable. The Guard-dropping version of the defense is unlikely to be deployed until the ntor circuit handshake is enabled, or the nature of CPU overload induced failure is better understood.

until the attempt to use it succeeds or fails. We remember when the guard became Pending with the {pending_since variable}.

After completing a circuit, the implementation must check whether its guard is usable. A guard's usability status may be "usable", "unusable", or "unknown". A guard is usable according to these rules:

1. Primary guards are always usable.

2. Non-primary guards are usable *for a given circuit* if every guard earlier in the preference list is either unsuitable for that circuit (e.g. because of family restrictions), or marked as Unreachable, or has been pending for at least `{NONPRIMARY_GUARD_CONNECT_TIMEOUT}`.

   Non-primary guards are not usable *for a given circuit* if some guard earlier in the preference list is suitable for the circuit *and* Reachable.

   Non-primary guards are unusable if they have not become usable after `{NONPRIMARY_GUARD_IDLE_TIMEOUT}` seconds.

3. If a circuit's guard is not usable or unusable immediately, the circuit is not discarded; instead, it is kept (but not used) until the guard becomes usable or unusable.

Whenever we get a new consensus.

We update {GUARDS}.

For every guard in {SAMPLED_GUARDS}, we update {IS_LISTED} and {FIRST_UNLISTED_AT}.

[**] We remove entries from {SAMPLED_GUARDS} if appropriate, according to the sampled-guards expiration rules. If they were in {CONFIRMED_GUARDS}, we also remove them from {CONFIRMED_GUARDS}.

We recompute {FILTERED_GUARDS}, and everything that derives from it, including {USABLE_FILTERED_GUARDS}, and {PRIMARY_GUARDS}.

(Whenever one of the configuration options that affects the filter is updated, we repeat the process above, starting at the [**] line.)

```
4.11. Deciding whether to generate a new circuit.
  [Section:NEW_CIRCUIT_NEEDED]
```

Updating the list of waiting circuits

We run this procedure whenever it's possible that a <waiting_for_better_guard> circuit might be ready to be called <complete>.

```
* If any circuit C1 is <waiting_for_better_guard>, AND:
    * All primary guards have reachable status of <no>.
    * There is no circuit C2 that "blocks" C1.
  Then, upgrade C1 to <complete>.

Definition: In the algorithm above, C2 "blocks" C1 if:
    * C2 obeys all the restrictions that C1 had to obey, AND
    * C2 has higher priority than C1, AND
    * Either C2 is <complete>, or C2 is <waiting_for_better_guard>,
      or C2 has been <usable_if_no_better_guard> for no more than
      {NONPRIMARY_GUARD_CONNECT_TIMEOUT} seconds.

We run this procedure periodically:

* If any circuit stays in <waiting_for_better_guard>
  for more than {NONPRIMARY_GUARD_IDLE_TIMEOUT} seconds,
  time it out.

   **Rationale**
```

If we open a connection to a guard, we might want to use it immediately (if we're sure that it's the best we can do), or we might want to wait a little while to see if some other circuit which we like better will finish.

When we mark a circuit <complete>, we don't close the lower-priority circuits immediately: we might decide to use them after all if the <complete> circuit goes down before {NONPRIMARY_GUARD_IDLE_TIMEOUT} seconds.

Without a list of waiting circuits

As an alternative to the section [SECTION:UPDATE_WAITING] above, this section presents a new way to maintain guard status independently of tracking individual circuit status. This formulation gives a result equivalent or similar to the approach above, but simplifies the necessary communications between the guard and circuit subsystems.

As before, when all primary guards are Unreachable, we need to try non-primary guards. We select the first such guard (in preference order) that is neither Unreachable nor Pending. Whenever we give out such a guard, if the guard's status is Unknown, then we call that guard "Pending" with its {is_pending} flag,

Tor Guard Specification

Introduction and motivation

Tor uses entry guards to prevent an attacker who controls some fraction of the network from observing a fraction of every user's traffic. If users chose their entries and exits uniformly at random from the list of servers every time they build a circuit, then an adversary who had (k/N) of the network would deanonymize F=(k/N)^2 of all circuits... and after a given user had built C circuits, the attacker would see them at least once with probability 1-(1-F)^C. With large C, the attacker would get a sample of every user's traffic with probability 1.

To prevent this from happening, Tor clients choose a small number of guard nodes (e.g. 3). These guard nodes are the only nodes that the client will connect to directly. If they are not compromised, the user's paths are not compromised.

This specification outlines Tor's guard housekeeping algorithm, which tries to meet the following goals:

- Heuristics and algorithms for determining how and which guards are chosen should be kept as simple and easy to understand as possible.
- Clients in censored regions or who are behind a fascist firewall who connect to the Tor network should not experience any significant disadvantage in terms of reachability or usability.
- Tor should make a best attempt at discovering the most appropriate behavior, with as little user input and configuration as possible.
- Tor clients should discover usable guards without too much delay.
- Tor clients should resist (to the extent possible) attacks that try to force them onto compromised guards.
- Should maintain the load-balancing offered by the path selection algorithm.

State instances

In the algorithm below, we describe a set of persistent and non-persistent state variables. These variables should be treated as an object, of which multiple instances can exist.

In particular, we specify the use of three particular instances:

A. UseBridges

```
       If UseBridges is set, then we replace the {GUARDS} set in
       [Sec:GUARDS] below with the list of configured
       bridges.  We maintain a separate persistent instance of
       {SAMPLED_GUARDS} and {CONFIRMED_GUARDS} and other derived
       values for the UseBridges case.

       In this case, we impose no upper limit on the sample size.

    B. EntryNodes / ExcludeNodes / Reachable*Addresses /
        FascistFirewall / ClientUseIPv4=0

       If one of the above options is set, and UseBridges is not,
       then we compare the fraction of usable guards in the consensus
       to the total number of guards in the consensus.

       If this fraction is less than {MEANINGFUL_RESTRICTION_FRAC},
       we use a separate instance of the state.

       (While Tor is running, we do not change back and forth between
       the separate instance of the state and the default instance
       unless the fraction of usable guards is 5% higher than, or 5%
       lower than, {MEANINGFUL_RESTRICTION_FRAC}.  This prevents us
       from flapping back and forth between instances if we happen to
       hit {MEANINGFUL_RESTRICTION_FRAC} exactly.

       If this fraction is less than {EXTREME_RESTRICTION_FRAC}, we use
a
       separate instance of the state, and warn the user.

       [TODO: should we have a different instance for each set of
heavily
       restricted options?]

   C. Default

       If neither of the above variant-state instances is used,
       we use a default instance.
```

consensus and we are told “404”. In this case, we mark the appropriate directory-specific `{is_reachable}` instance for that guard to `<no>`.]

> C tor implements the above “note 2” by treating requests for directory guards for as if they had an extra type of restriction, rather than a separate instance of `{is_reachable}`. (For more on restrictions, see “Selecting Guards” above.) This requires the C tor impementation to special-case this restriction type, so that it is treated the same way as an `{is_reachable}` variable.

Rationale

See [SELECTING] above for rationale.

When a circuit succeeds

When a circuit succeeds in a way that makes us conclude that a guard *was* reachable, we take these steps:

```
       * We set its {is_reachable} status to <yes>.
       * We set its {failing_since} to "never".
       * If the guard was {is_pending}, we clear the {is_pending} flag
         and set {pending_since} to false.
       * If the guard was not a member of {CONFIRMED_GUARDS}, we add
         it to the end of {CONFIRMED_GUARDS}.

       * If this circuit was <usable_on_completion>, this circuit is
         now <complete>. You may attach streams to this circuit,
         and use it for hidden services.

       * If this circuit was <usable_if_no_better_guard>, it is now
         <waiting_for_better_guard>.  You may not yet attach streams to
it.

         Then check whether the {last_time_on_internet} is more than
         {INTERNET_LIKELY_DOWN_INTERVAL} seconds ago:

            * If it is, then mark all {PRIMARY_GUARDS} as "maybe"
              reachable.

            * If it is not, update the list of waiting circuits. (See
              [UPDATE_WAITING] below)
```

[Note: the existing Tor logic will cause us to create more circuits in response to some of these steps; and see [ON_CONSENSUS].]

Rationale

See [SELECTING] above for rationale.

In some cases (for example, when we need a certain directory feature, or when we need to avoid using a certain exit as a guard), we need to restrict the guards that we use for a single circuit. When this happens, we remember the restrictions that applied when choosing the guard for that circuit, since we will need them later (see [UPDATE_WAITING].).

Rationale

We're getting to the core of the algorithm here. Our main goals are to make sure that

1. If it's possible to use a primary guard, we do.
2. We probably use the first primary guard.

So we only try non-primary guards if we're pretty sure that all the primary guards are down, and we only try a given primary guard if the earlier primary guards seem down.

When we *do* try non-primary guards, however, we only build one circuit through each, to give it a chance to succeed or fail. If ever such a circuit succeeds, we don't use it until we're pretty sure that it's the best guard we're getting. (see below).

[XXX timeout.]

When a circuit fails.

When a circuit fails in a way that makes us conclude that a guard is not reachable, we take the following steps:

```
* Set the guard's {is_reachable} status to <no>.  If it had
  {is_pending} set to true, we make it non-pending and clear
  {pending_since}.

* Close the circuit, of course.  (This removes it from
  consideration by the algorithm in [UPDATE_WAITING].)

* Update the list of waiting circuits.  (See [UPDATE_WAITING]
  below.)
```

[Note: the existing Tor logic will cause us to create more circuits in response to some of these steps; and also see [ON_CONSENSUS].]

[Note 2: In the case of a one-hop circuit made for a directory request, it is possible for the request to fail *after* the circuit is built: for example, if we ask for the latest

Circuit Creation, Entry Guard Selection (1000 foot view)

A circuit in Tor is a path through the network connecting a client to its destination. At a high-level, a three-hop exit circuit will look like this:

Client <-> Entry Guard <-> Middle Node <-> Exit Node <-> Destination

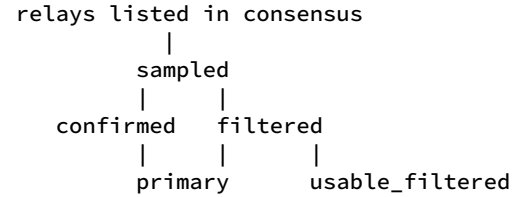
Entry guards are the only nodes which a client will connect to directly. Exit relays are the nodes by which traffic exits the Tor network in order to connect to an external destination.

Path selection

For any multi-hop circuit, at least one entry guard and middle node(s) are required. An exit node is required if traffic will exit the Tor network. Depending on its configuration, a relay listed in a consensus could be used for any of these roles. However, this specification defines how entry guards specifically should be selected and managed, as opposed to middle or exit nodes.

Managing entry guards

At a high level, a relay listed in a consensus will move through the following states in the process from initial selection to eventual usage as an entry guard:



Relays listed in the latest consensus can be sampled for guard usage if they have the "Guard" flag. Sampling is random but weighted by a measured bandwidth multiplied by bandwidth-weights (Wgg if guard only, Wgd if guard+exit flagged).

Once a path is built and a circuit established using this guard, it is marked as confirmed. Until this point, guards are first sampled and then filtered based on information such as our current configuration (see SAMPLED and FILTERED

sections) and later marked as usable_filtered if the guard is not primary but can be reached.

It is always preferable to use a primary guard when building a new circuit in order to reduce guard churn; only on failure to connect to existing primary guards will new guards be used.

Middle and exit node selection

Middle nodes are selected at random from relays listed in the latest consensus, weighted by bandwidth and bandwidth-weights. Exit nodes are chosen similarly but restricted to relays with a sufficiently permissive exit policy.

Circuit Building

Once a path is chosen, Tor will use this path to build a new circuit.

If the circuit is built successfully, Tor will either use it immediately, or Tor will wait for a circuit with a more preferred guard if there's a good chance that it will be able to make one.

If the circuit fails in a way that makes us conclude that a guard is not reachable, the guard is marked as unreachable, the circuit is closed, and waiting circuits are updated.

Here is the algorithm. It is given as a series of sub-algorithms, in decreasing order of preference from best case to worst case. When we want to build a circuit, and we need to pick a guard:

- In the base case, if any entry in PRIMARY_GUARDS has {is_reachable} status of `<maybe>` or `<yes>`, consider only such guards. Call them the "reachable primary guards".

  Start by considering the the first {NUM_USABLE_PRIMARY_GUARDS} (or {NUM_USABLE_PRIMARY_DIRECTORY_GUARDS}) reachable primary guards. Remove any guards that do not follow our path selection restrictions, to build a temporary list. If that temporary list contains at least one guard, choose a guard from that list uniformly at random.

  If the temporary list contains no guards, return the first guard from our reachable primary guard that does obey the path restriction.

  When selecting a guard according to this approach, its circuit is `<usable_on_completion>`.

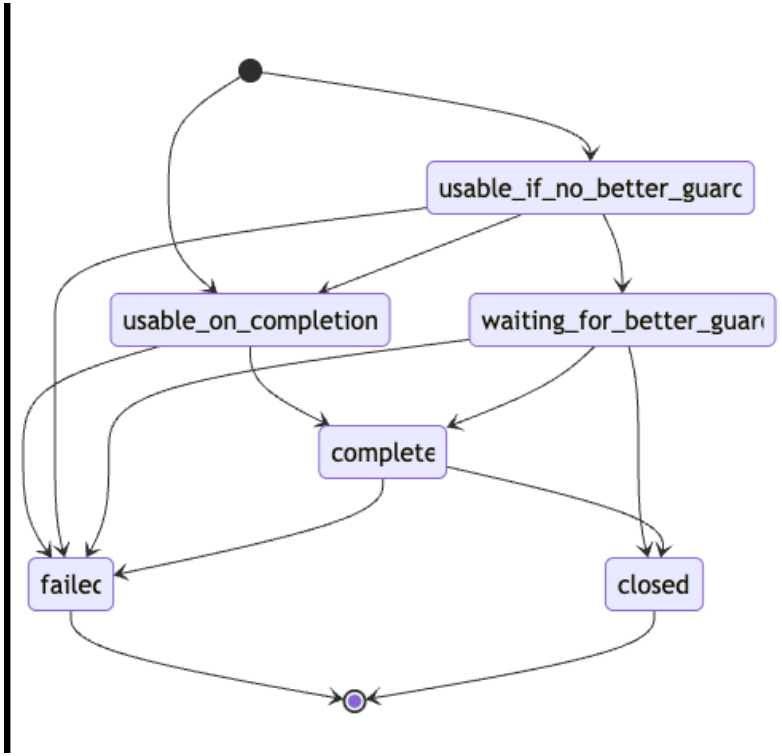
  [Note: We do not use {is_pending} on primary guards, since we are willing to try to build multiple circuits through them before we know for sure whether they work, and since we will not use any non-primary guards until we are sure that the primary guards are all down. (XX is this good?)]

- Otherwise, if the ordered intersection of {CONFIRMED_GUARDS} and {USABLE_FILTERED_GUARDS} is nonempty, return the first entry in that intersection that has {is_pending} set to false. Set its value of {is_pending} to true, and set its {pending_since} to the current time. The circuit is now `<usable_if_no_better_guard>`. (If all entries have {is_pending} true, pick the first one.)

- Otherwise, if there is no such entry, select a member from {USABLE_FILTERED_GUARDS} in sample order. Set its {is_pending} field to true, and set its {pending_since} to the current time. The circuit is `<usable_if_no_better_guard>`.

- Otherwise, in the worst case, if USABLE_FILTERED_GUARDS is empty, we have exhausted all the sampled guards. In this case we proceed by marking all guards as `<maybe>` reachable so that we can keep on trying circuits.

Whenever we select a guard for a new circuit attempt, we update the {last_tried_connect} time for the guard to 'now.'



Each of these transitions is described in sections below.

Selecting guards for circuits. [Section:SELECTING]

Now that we have described the various lists of guards, we can explain how guards are chosen for each circuit.

We keep, as global transient state:

- {tvar:last_time_on_internet} – the last time at which we successfully used a circuit or connected to a guard. At startup we set this to “infinitely far in the past.”

As an input to the algorithm, we take a list of *restrictions*. These may include specific relays or families that we need to avoid.

The algorithm

The guards listed in the current consensus.

By {set:GUARDS} we mean the set of all guards in the current consensus that are usable for all circuits and directory requests. (They must have the flags: Stable, Fast, V2Dir, Guard.)

Rationale

We require all guards to have the flags that we potentially need from any guard, so that all guards are usable for all circuits.

The Sampled Guard Set.

We maintain a set, {set:SAMPLED_GUARDS}, that persists across invocations of Tor. It is a subset of the nodes ordered by a sample idx that we have seen listed as a guard in the consensus at some point. For each such guard, we record persistently:

```
    - {pvar:ADDED_ON_DATE}: The date on which it was added to
      sampled_guards.

      We set this value to a point in the past, using
      RAND(now, {GUARD_LIFETIME}/10). See
      Appendix [RANDOM] below.

    - {pvar:ADDED_BY_VERSION}: The version of Tor that added it to
      sampled_guards.

    - {pvar:IS_LISTED}: Whether it was listed as a usable Guard in
      the _most recent_ consensus we have seen.

    - {pvar:FIRST_UNLISTED_AT}: If IS_LISTED is false, the
publication date
      of the earliest consensus in which this guard was listed such
that we
      have not seen it listed in any later consensus.  Otherwise
"None."
      We randomize this to a point in the past, based on
        RAND(added_at_time, {REMOVE_UNLISTED_GUARDS_AFTER} / 5)
```

For each guard in {SAMPLED_GUARDS}, we also record this data, non-persistently:

```
- {tvar:last_tried_connect}: A 'last tried to connect at'
  time.  Default 'never'.

- {tvar:is_reachable}: an "is reachable" tristate, with
  possible values { <state:yes>, <state:no>, <state:maybe> }.
  Default '<maybe>.'

        [Note: "yes" is not strictly necessary, but I'm
         making it distinct from "maybe" anyway, to make our
         logic clearer.  A guard is "maybe" reachable if it's
         worth trying. A guard is "yes" reachable if we tried
         it and succeeded.]

        [Note 2: This variable is, in fact, a combination
        of different context-sensitive variables depending
        on the _purpose_ for which we are selecting a guard.
        When we are selecing a guard for an ordinary
        circuit, we look at the regular {is_reachable}.
        But when we are selecting the guard for a one-hop
        directory circuit, we also look at an instance
        of {is_reachable} that tracks whether
        downloads of the types we are making have failed
        recently.]

- {tvar:failing_since}: The first time when we failed to
  connect to this guard. Defaults to "never".  Reset to
  "never" when we successfully connect to this guard.

- {tvar:is_pending} A "pending" flag.  This indicates that we
  are trying to build an exploratory circuit through the
  guard, and we don't know whether it will succeed.

- {tvar:pending_since}: A timestamp.  Set whenever we set
  {tvar:is_pending} to true; cleared whenever we set
  {tvar:is_pending} to false. NOTE
```

We require that {SAMPLED_GUARDS} contain at least {MIN_FILTERED_SAMPLE} guards from the consensus (if possible), but not more than {MAX_SAMPLE_THRESHOLD} of the number of guards in the consensus, and not more than {MAX_SAMPLE_SIZE} in total. (But if the maximum would be smaller than {MIN_FILTERED_SAMPLE}, we set the maximum at {MIN_FILTERED_SAMPLE}.)

To add a new guard to {SAMPLED_GUARDS}, pick an entry at random from ({GUARDS} - {SAMPLED_GUARDS}), according to the path selection rules.

We remove an entry from {SAMPLED_GUARDS} if:

Circuit status

Sometimes the guard selection algorithm will return a guard that we would always be happy to use; but in other cases, the guard selection algorithm can return a guard that we shouldn't use without gathering additional information.

From the point of view of guard selection, every circuit we build is considered to be in one of these states:

- `<state:usable_on_completion>`
- `<state:usable_if_no_better_guard>`
- `<state:waiting_for_better_guard>`
- `<state:complete>`

You may only attach streams to `<complete>` circuits. (Additionally, you may only send RENDEZVOUS messages, ESTABLISH_INTRO messages, and INTRODUCE messages on `<complete>` circuits.)

The per-circuit state machine is:

- New circuits are `<usable_on_completion>` or `<usable_if_no_better_guard>`.
- A `<usable_on_completion>` circuit may become `<complete>`, or may fail.
- A `<usable_if_no_better_guard>` circuit may become `<usable_on_completion>`; may become `<waiting_for_better_guard>`; or may fail.
- A `<waiting_for_better_guard>` circuit will become `<complete>`, or will be closed, or will fail.
- A `<complete>` circuit remains `<complete>` until it fails or is closed.

To compute primary guards, take the ordered intersection of {CONFIRMED_GUARDS} and {FILTERED_GUARDS}, and take the first {N_PRIMARY_GUARDS} elements. If there are fewer than {N_PRIMARY_GUARDS} elements, append additional elements to PRIMARY_GUARDS chosen from ({FILTERED_GUARDS} - {CONFIRMED_GUARDS}), ordered in "sample order" (that is, by {ADDED_ON_DATE}).

Once an element has been added to {PRIMARY_GUARDS}, we do not remove it until it is replaced by some element from {CONFIRMED_GUARDS}. That is: if a non-primary guard becomes confirmed and not every primary guard is confirmed, then the list of primary guards list is regenerated, first from the confirmed guards (as before), and then from any non-confirmed primary guards.

Note that {PRIMARY_GUARDS} do not have to be in {USABLE_FILTERED_GUARDS}: they might be unreachable.

Rationale

These guards are treated differently from other guards. If one of them is usable, then we use it right away. For other guards {FILTERED_GUARDS}, if it's usable, then before using it we might first double-check whether perhaps one of the primary guards is usable after all.

Retrying guards.

(We run this process as frequently as needed. It can be done once a second, or just-in-time.)

If a primary sampled guard's {is_reachable} status is `<no>`, then we decide whether to update its {is_reachable} status to `<maybe>` based on its {last_tried_connect} time, its {failing_since} time, and the {PRIMARY_GUARDS_RETRY_SCHED} schedule.

If a non-primary sampled guard's {is_reachable} status is `<no>`, then we decide whether to update its {is_reachable} status to `<maybe>` based on its {last_tried_connect} time, its {failing_since} time, and the {GUARDS_RETRY_SCHED} schedule.

Rationale

An observation that a guard has been 'unreachable' only lasts for a given amount of time, since we can't infer that it's unreachable now from the fact that it was unreachable a few minutes ago.

```
  * We have a live consensus, and {IS_LISTED} is false, and
    {FIRST_UNLISTED_AT} is over {REMOVE_UNLISTED_GUARDS_AFTER}
    days in the past.

OR

  * We have a live consensus, and {ADDED_ON_DATE} is over
    {GUARD_LIFETIME} ago, *and* {CONFIRMED_ON_DATE} is either
    "never", or over {GUARD_CONFIRMED_MIN_LIFETIME} ago.
```

Note that {SAMPLED_GUARDS} does not depend on our configuration. It is possible that we can't actually connect to any of these guards.

Rationale

The {SAMPLED_GUARDS} set is meant to limit the total number of guards that a client will connect to in a given period. The upper limit on its size prevents us from considering too many guards.

The first expiration mechanism is there so that our {SAMPLED_GUARDS} list does not accumulate so many dead guards that we cannot add new ones.

The second expiration mechanism makes us rotate our guards slowly over time.

Ordering the {SAMPLED_GUARDS} set in the order in which we sampled those guards and picking guards from that set according to this ordering improves load-balancing. It is closer to offer the expected usage of the guard nodes as per the path selection rules.

The ordering also improves on another objective of this proposal: trying to resist an adversary pushing clients over compromised guards, since the adversary would need the clients to exhaust all their initial {SAMPLED_GUARDS} set before having a chance to use a newly deployed adversary node.

The Usable Sample

We maintain another set, {set:FILTERED_GUARDS}, that does not persist. It is derived from:

```
  - {SAMPLED_GUARDS}
  - our current configuration,
  - the path bias information.
```

A guard is a member of {set:FILTERED_GUARDS} if and only if all of the following are true:

```
    - It is a member of {SAMPLED_GUARDS}, with {IS_LISTED} set to
      true.
    - It is not disabled because of path bias issues.
    - It is not disabled because of ReachableAddresses policy,
      the ClientUseIPv4 setting, the ClientUseIPv6 setting,
      the FascistFirewall setting, or some other
      option that prevents using some addresses.
    - It is not disabled because of ExcludeNodes.
    - It is a bridge if UseBridges is true; or it is not a
      bridge if UseBridges is false.
    - Is included in EntryNodes if EntryNodes is set and
      UseBridges is not. (But see 2.B above).
```

We have an additional subset, {set:USABLE_FILTERED_GUARDS}, which is defined to be the subset of {FILTERED_GUARDS} where {is_reachable} is <yes> or <maybe>.

We try to maintain a requirement that {USABLE_FILTERED_GUARDS} contain at least {MIN_FILTERED_SAMPLE} elements:

```
Whenever we are going to sample from {USABLE_FILTERED_GUARDS},
and it contains fewer than {MIN_FILTERED_SAMPLE} elements, we
add new elements to {SAMPLED_GUARDS} until one of the following
is true:

  * {USABLE_FILTERED_GUARDS} is large enough,
OR
  * {SAMPLED_GUARDS} is at its maximum size.

** Rationale **
```

These filters are applied *after* sampling: if we applied them before the sampling, then our sample would reflect the set of filtering restrictions that we had in the past.

The confirmed-guard list.

[formerly USED_GUARDS]

We maintain a persistent ordered list, {list:CONFIRMED_GUARDS}. It contains guards that we have used before, in our preference order of using them. It is a subset of {SAMPLED_GUARDS}. For each guard in this list, we store persistently:

- {pvar:IDENTITY} Its fingerprint.

```
    - {pvar:CONFIRMED_ON_DATE} When we added this guard to
      {CONFIRMED_GUARDS}.

      Randomized to a point in the past as RAND(now,
{GUARD_LIFETIME}/10).
```

We append new members to {CONFIRMED_GUARDS} when we mark a circuit built through a guard as "for user traffic."

Whenever we remove a member from {SAMPLED_GUARDS}, we also remove it from {CONFIRMED_GUARDS}.

```
    [Note: You can also regard the {CONFIRMED_GUARDS} list as a
    total ordering defined over a subset of {SAMPLED_GUARDS}.]
```

Definition: we call Guard A "higher priority" than another Guard B if, when A and B are both reachable, we would rather use A. We define priority as follows:

```
    * Every guard in {CONFIRMED_GUARDS} has a higher priority
      than every guard not in {CONFIRMED_GUARDS}.

    * Among guards in {CONFIRMED_GUARDS}, the one appearing earlier
      on the {CONFIRMED_GUARDS} list has a higher priority.

    * Among guards that do not appear in {CONFIRMED_GUARDS},
      {is_pending}==true guards have higher priority.

    * Among those, the guard with earlier {last_tried_connect} time
      has higher priority.

    * Finally, among guards that do not appear in
      {CONFIRMED_GUARDS} with {is_pending==false}, all have equal
      priority.

  ** Rationale **
```

We add elements to this ordering when we have actually used them for building a usable circuit. We could mark them at some other time (such as when we attempt to connect to them, or when we actually connect to them), but this approach keeps us from committing to a guard before we actually use it for sensitive traffic.

The Primary guards

We keep a run-time non-persistent ordered list of {list:PRIMARY_GUARDS}. It is a subset of {FILTERED_GUARDS}. It contains {N_PRIMARY_GUARDS} elements.