Tor: The Second-Generation Onion Router (2012 DRAFT)

Roger Dingledine
The Free Haven Project
arma@freehaven.net

Nick Mathewson
The Free Haven Project
nickm@freehaven.net

Steven Murdoch
affiliation
email

Paul Syverson
Naval Research Lab
syverson@itd.nrl.navy.mil

Abstract

THIS IS A DRAFT. IT IS NOT FINISHED. We present Tor, a circuit-based low-latency anonymous communication service. This Onion Routing system addresses limitations in the earlier design by adding perfect forward secrecy, congestion control, directory servers, integrity checking, configurable exit policies, anticensorship features, guard nodes, application- and user-selectable stream isolation, and a practical design for location-hidden services via rendezvous points. Tor is deployed on the real-world Internet, requires no special privileges or kernel modifications, requires little synchronization or coordination between nodes, and provides a reasonable tradeoff between anonymity, usability, and efficiency. An earlier paper in 2004 described Tor's original design; here we explain Tor's current design as of late 2012, and describe our experiences with an international network of approximately 3000 nodes and 500000 users. We close with a list of open problems in anonymous communication.

1 Overview

THIS IS A DRAFT. IT IS NOT FINISHED. IT IS BASED ON THE 2004 PAPER, WITH PARTIAL UPDATES.

Onion Routing is a distributed overlay network designed to anonymize TCP-based applications like web browsing, secure shell, and instant messaging. Clients choose a path through the network and build a *circuit*, in which each node (or "onion router" or "OR") in the path knows its predecessor and successor, but no other nodes in the circuit. Traffic flows down the circuit in fixed-size *cells*, which are unwrapped by a symmetric key at each node (like the layers of an onion) and relayed downstream. The Onion Routing project published several design and analysis papers [25, 38, 45, 46]. While a wide area Onion Routing network was deployed briefly, the only long-running public implementation was a fragile proof-of-concept that ran on a single machine. Even this simple deployment processed connections from over sixty thousand distinct IP addresses from all over the world at a rate of about fifty thousand per day. Here we describe Tor, a protocol for asynchronous, loosely federated onion routers. Tor provides provides the following improvements over the old Onion Routing design:

Perfect forward secrecy: In the original Onion Routing design, a single hostile node could record traffic and later compromise successive nodes in the circuit and force them to decrypt it. Rather than using a single multiply encrypted data structure (an *onion*) to lay each circuit, Tor now uses an incremental or *telescoping* path-building design, where the initiator negotiates session keys with each successive hop in the circuit. Once these keys are deleted, subsequently compromised nodes cannot decrypt old traffic. As a side benefit, onion replay detection is no longer necessary, and the process of building circuits is more reliable, since the initiator knows when a hop fails and can then try extending to a new node.

Separation of "protocol cleaning" from anonymity: Onion Routing originally required a separate "application proxy" for each supported application protocol—most of which were never written, so many applications were never supported. Tor uses the standard and near-ubiquitous SOCKS [30] proxy interface, allowing us to support most TCP-based programs without modification. For the protocol cleaning of HTTP and HTTPS, Tor relies on Torbutton (a Firefox add-on) and modifications made to the version of Firefox delivered to users as part of the Tor Browser Bundle.

No mixing, padding, or traffic shaping (yet): Onion Routing originally called for batching and reordering cells as they arrived, assumed padding between ORs, and in later designs added padding between onion proxies (users) and ORs [25, 38]. Tradeoffs between padding protection and cost were discussed, and *traffic shaping* algorithms were theorized [46] to provide good security without expensive padding, but no concrete padding scheme was suggested. As of our first writing, research [1] and deployment experience [4] suggest that this level of resource use is not practical or economical; and even full link padding is still vulnerable [31]. This has not changed since our first publication, though we remain hopeful.

Many TCP streams can share one circuit: Onion Routing originally built a separate circuit for each application-level request, but this required multiple public key operations for every request, and also presented a threat to anonymity from building so many circuits; see Section 9. Tor multiplexes multiple TCP streams along each circuit to improve efficiency and anonymity, but allows the user to control which streams may share a circuit with which other streams to prevent unintended linking of pseudonyms.

Leaky-pipe circuit topology: Through in-band signaling within the circuit, Tor initiators can direct traffic to nodes partway down the circuit. This novel approach allows traffic to exit the circuit from the middle—possibly frustrating traffic shape and volume attacks based on observing the end of the circuit. (It also allows for long-range padding if future research shows this to be worthwhile.)

Congestion control: Earlier anonymity designs do not address traffic bottlenecks. Unfortunately, typical approaches to load balancing and flow control in overlay networks involve inter-node control communication and global views of traffic. Tor's decentralized congestion control uses end-to-end acks to maintain anonymity while allowing nodes at the edges of the network to detect congestion or flooding and send less data until the congestion subsides. This is an area of active experimentation and research.

Directory authorities: The earlier Onion Routing design planned to flood state information through the network—an approach that can be unreliable and complex. Tor takes a simplified view toward distributing this information. Certain more trusted nodes act as *directory authorities*: they collaborate to generate signed directory documents describing known routers and their current state. Users periodically download these documents directly from the authorities or a mirror, via HTTP tunelled over a Tor circuit.

Variable exit policies: Tor provides a consistent mechanism for each node to advertise a policy describing the hosts and ports to which it will connect. These exit policies are critical in a volunteer-based distributed infrastructure, because each operator is comfortable with allowing different types of traffic to exit from his node.

End-to-end integrity checking: The original Onion Routing design did no integrity checking on data. Any node on the circuit could change the contents of data cells as they passed by—for example, to alter a connection request so it would connect to a different webserver, or to 'tag' encrypted traffic and look for corresponding corrupted traffic at the network edges [13]. Tor hampers these attacks by verifying data integrity before it leaves the network.

Rendezvous points and hidden services: Tor provides an integrated mechanism for responder anonymity via location-protected servers. Previous Onion Routing designs included long-lived "reply onions" that could be used to build circuits to a hidden server, but these reply onions did not provide forward security, and became useless if any node in the path went down or rotated its keys. In Tor, clients negotiate *rendezvous points* to connect with hidden servers; reply onions are no longer required.

Censorship resistance: A growing number of Tor users require not only anonymous communications but also censorship resistance. Tor circumvents attempts to block access to the network by allowing users to access the network via a "bridge" – a special purpose Tor node which does not appear in the directory and whose IP address is only known by a small number of individuals to reduce the chance of it being blocked by IP address. The Tor protocol has also been designed to be similar to HTTPS such that blocking Tor, without blocking HTTPS, is made more difficult.

Modular architecture: The Tor program is only one part of an effective anonymous communication system and Tor provides functionality to integrate with other components to fulfil a wide variety of user requirements. The graphical user interface is a separate program (Vidalia, in the Tor Browser Bundle, but alternatives exist) which communicates with Tor via a local socket – the "control port". Special-purpose controllers have also been developed by researchers to analyse Tor and prototype modifications. Additional resistance against protocol fingerprinting, for the purposes of censorship resistance, may be provided by an external "pluggable transport" obfuscator.

Unlike Freedom [7], Tor does not require OS kernel patches or network stack support. This prevents us from anonymizing non-TCP protocols, but has greatly helped our portability and deployability.

Tor has been operational since 2003. Our source code is available under a free license, and Tor is not covered by the patent that affected distribution and use of earlier versions of Onion Routing. As of this writing, the Tor network stands at about 3000 nodes.

We review previous work in Section 2, describe our goals and assumptions in Section 3, and then address the above list of improvements in Sections 4, 5, and 6. We summarize in Section 7 how our design stands up to known attacks, and talk about our early deployment experiences in Section 8. We conclude with a list of open problems in Section 9 and future work for the Onion Routing project in Section 10.

2 Related work

Modern anonymity systems date to Chaum's **Mix-Net** design [9]. Chaum proposed hiding the correspondence between sender and recipient by wrapping messages in layers of public-key cryptography, and relaying them through a path composed of "mixes." Each mix in turn decrypts, delays, and re-orders messages before relaying them onward.

Subsequent relay-based anonymity designs have diverged in two main directions. Systems like **Babel** [26], **Mixmaster** [34], and **Mixminion** [13] have tried to maximize anonymity at the cost of introducing comparatively large

and variable latencies. Because of this decision, these *high-latency* networks resist strong global adversaries, but introduce too much lag for interactive tasks like web browsing, Internet chat, or SSH connections.

Tor belongs to the second category: *low-latency* designs that try to anonymize interactive network traffic. These systems handle a variety of bidirectional protocols. They also provide more convenient mail delivery than the high-latency anonymous email networks, because the remote mail server provides explicit and timely delivery confirmation. But because these designs typically involve many packets that must be delivered quickly, it is difficult for them to prevent an attacker who can eavesdrop both ends of the communication from correlating the timing and volume of traffic entering the anonymity network with traffic leaving it [42]. These protocols are similarly vulnerable to an active adversary who introduces timing patterns into traffic entering the network and looks for correlated patterns among exiting traffic. Although some work has been done to frustrate these attacks, most designs protect primarily against traffic analysis rather than traffic confirmation (see Section 3.1).

The simplest low-latency designs are single-hop proxies such as the **Anonymizer** [3]: a single trusted server strips the data's origin before relaying it. These designs are easy to analyze, but users must trust the anonymizing proxy. Concentrating the traffic to this single point increases the anonymity set (the people a given user is hiding among), but it is vulnerable if the adversary can observe all traffic entering and leaving the proxy.

More complex are distributed-trust, circuit-based anonymizing systems. In these designs, a user establishes one or more medium-term bidirectional end-to-end circuits, and tunnels data in fixed-size cells. Establishing circuits is computationally expensive and typically requires public-key cryptography, whereas relaying cells is comparatively inexpensive and typically requires only symmetric encryption. Because a circuit crosses several servers, and each server only knows the adjacent servers in the circuit, no single server can link a user to her communication partners.

The **Java Anon Proxy** (also known as JAP or Web MIXes) uses fixed shared routes known as *cascades*. As with a single-hop proxy, this approach aggregates users into larger anonymity sets, but again an attacker only needs to observe both ends of the cascade to bridge all the system's traffic. The Java Anon Proxy's design calls for padding between end users and the head of the cascade [6]. However, it is not demonstrated whether the current implementation's padding policy improves anonymity.

PipeNet [5, 10], another low-latency design proposed around the same time as Onion Routing, gave stronger anonymity but allowed a single user to shut down the network by not sending. Systems like **ISDN mixes** [36] were designed for other environments with different assumptions.

In P2P designs like **Tarzan** [22] and **MorphMix** [40], all participants both generate traffic and relay traffic for others. These systems aim to conceal whether a given peer originated a request or just relayed it from another peer. While Tarzan and MorphMix use layered encryption as above, **Crowds** [39] simply assumes an adversary who cannot observe the initiator: it uses no public-key encryption, so any node on a circuit can read users' traffic.

Hordes [32] is based on Crowds but also uses multicast responses to hide the initiator. **Herbivore** [23] and **P^5** [43] go even further, requiring broadcast. These systems are designed primarily for communication among peers, although Herbivore users can make external connections by requesting a peer to serve as a proxy.

Systems like **Freedom** and the original Onion Routing build circuits all at once, using a layered "onion" of public-key encrypted messages, each layer of which provides session keys and the address of the next server in the circuit. Tor as described herein, Tarzan, MorphMix, **Cebolla** [8], and Rennhard's **Anonymity Network** [41] build circuits in stages, extending them one hop at a time. Section 4.3 describes how this approach enables perfect forward secrecy.

Circuit-based designs must choose which protocol layer to anonymize. They may intercept IP packets directly, and relay them whole (stripping the source address) along the circuit [7, 22]. Like Tor, they may accept TCP streams and relay the data in those streams, ignoring the breakdown of that data into TCP segments [40, 41]. Finally, like Crowds, they may accept application-level protocols such as HTTP and relay the application requests themselves. Making this protocol-layer decision requires a compromise between flexibility and anonymity. For example, a system that understands HTTP can strip identifying information from requests, can take advantage of caching to limit the number of requests that leave the network, and can batch or encode requests to minimize the number of connections. On the other hand, an IP-level anonymizer can handle nearly any protocol, even ones unforeseen by its designers (though these systems require kernel-level modifications to some operating systems, and so are more complex and less portable). TCP-level anonymity networks like Tor present a middle approach: they are application neutral (so long as the application supports, or can be tunneled across, TCP), but by treating application connections as data streams rather than raw TCP packets, they avoid the inefficiencies of tunneling TCP over TCP.

Distributed-trust anonymizing systems need to prevent attackers from adding too many servers and thus compromising user paths. Tor relies on a small set of well-known directory servers, run by independent parties, to decide which nodes can join. Tarzan and MorphMix allow unknown users to run servers, and use a limited resource (like IP addresses) to prevent an attacker from controlling too much of the network. Crowds suggests requiring written, notarized requests from potential crowd members.

Anonymous communication is essential for censorship-

resistant systems like Eternity [2], Free Haven [17], Publius [50], and Tangler [49]. Tor's rendezvous points enable connections between mutually anonymous entities; they are a building block for location-hidden servers, which are needed by Eternity and Free Haven.

3 Design goals and assumptions

Goals

Like other low-latency anonymity designs, Tor seeks to frustrate attackers from linking communication partners, or from linking multiple communications to or from a single user. Within this main goal, however, several considerations have directed Tor's evolution.

Deployability: The design must be deployed and used in the real world. Thus it must not be expensive to run (for example, by requiring more bandwidth than volunteers are willing to provide); must not place a heavy liability burden on operators (for example, by allowing attackers to implicate onion routers in illegal activities); and must not be difficult or expensive to implement (for example, by requiring kernel patches, or separate proxies for every protocol). We also cannot require non-anonymous parties (such as websites) to run our software. (Our rendezvous point design does not meet this goal for non-anonymous users talking to hidden servers, however; see Section 5.)

Usability: A hard-to-use system has fewer users—and because anonymity systems hide users among users, a system with fewer users provides less anonymity. Usability is thus not only a convenience: it is a security requirement [1, 5]. Tor should therefore not require modifying familiar applications; should not introduce prohibitive delays; and should require as few configuration decisions as possible. Finally, Tor should be easily implementable on all common platforms; we cannot require users to change their operating system to be anonymous. (Tor currently runs on Win32, Linux, Solaris, BSD-style Unix, MacOS X, and probably others.)

Flexibility: The protocol must be flexible and well-specified, so Tor can serve as a test-bed for future research. Many of the open problems in low-latency anonymity networks, such as generating dummy traffic or preventing Sybil attacks [20], may be solvable independently from the issues solved by Tor. Hopefully future systems will not need to reinvent Tor's design.

Simple design: The protocol's design and security parameters must be well-understood. Additional features impose implementation and complexity costs; adding unproven techniques to the design threatens deployability, readability, and ease of security analysis. Tor aims to deploy a simple and stable system that integrates the best accepted approaches to protecting anonymity.

Resistant to censorship: Many users will be using an anonymous communication system because they are accessing websites which may be frowned upon by their country, and so are blocked. Consequently countries may choose to block access to the Tor network in order to meet their goal of blocking access to the sites in question. Tor should therefore be resistant to censorship, both blocking by IP address and blocking as a result of protocol fingerprinting.

Non-goals

In favoring simple, deployable designs, we have explicitly deferred several possible goals, either because they are solved elsewhere, or because they are not yet solved.

Not peer-to-peer: Tarzan and MorphMix aim to scale to completely decentralized peer-to-peer environments with thousands of short-lived servers, many of which may be controlled by an adversary. This approach is appealing, but still has many open problems [22, 40].

Not secure against end-to-end attacks: Tor does not claim to completely solve end-to-end timing or intersection attacks. Some approaches, such as having users run their own onion routers, may help; see Section 9 for more discussion.

No protocol normalization: Tor does not provide *protocol normalization* like Privoxy or the Anonymizer. If senders want anonymity from responders while using complex and variable protocols like HTTP or HTTPS, Tor must be layered with a specialized web browser to hide differences between clients, and expunge protocol features that leak identity. Note that by this separation Tor can also provide services that are anonymous to the network yet authenticated to the responder, like SSH. Similarly, Tor does not integrate tunneling for non-stream-based protocols like UDP; this must be provided by an external service if appropriate.

3.1 Threat Model

A global passive adversary is the most commonly assumed threat when analyzing theoretical anonymity designs. But like all practical low-latency systems, Tor does not protect against such a strong adversary. Instead, we assume an adversary who can observe some fraction of network traffic; who can generate, modify, delete, or delay traffic; who can operate onion routers of his own; and who can compromise some fraction of the onion routers.

In low-latency anonymity systems that use layered encryption, the adversary's typical goal is to observe both the initiator and the responder. By observing both ends, passive attackers can confirm a suspicion that Alice is talking to Bob if the timing and volume patterns of the traffic on the connection are distinct enough; active attackers can induce timing signatures on the traffic to force distinct patterns. Rather than focusing on these *traffic confirmation* attacks, we aim to prevent *traffic analysis* attacks, where the adversary uses traffic patterns to learn which points in the network he should attack.

Our adversary might try to link an initiator Alice with her communication partners, or try to build a profile of Alice's behavior. He might mount passive attacks by observing the

network edges and correlating traffic entering and leaving the network—by relationships in packet timing, volume, or externally visible user-selected options. The adversary can also mount active attacks by compromising routers or keys; by replaying traffic; by selectively denying service to trustworthy routers to move users to compromised routers, or denying service to users to see if traffic elsewhere in the network stops; or by introducing patterns into traffic that can later be detected. The adversary might subvert the directory servers to give users differing views of network state. Additionally, he can try to decrease the network's reliability by attacking nodes or by performing antisocial activities from reliable nodes and trying to get them taken down—making the network unreliable flushes users to other less anonymous systems, where they may be easier to attack. We summarize in Section 7 how well the Tor design defends against each of these attacks.

4 The Tor Design

The Tor network is an overlay network; each onion router (OR) runs as a normal user-level process without any special privileges. Each onion router maintains TLS [15] connections to other onion routers it has been recently communicating with. Each user runs local software called an onion proxy (OP) to fetch directories, establish circuits across the network, and handle connections from user applications. These onion proxies accept TCP streams and multiplex them across the circuits. The onion router on the other side of the circuit connects to the requested destinations and relays data.

Each onion router maintains a long-term identity key and a short-term onion key. The identity key is used to sign TLS certificates, to sign the OR's *router descriptor* (a summary of its keys, address, bandwidth, exit policy, and so on). The onion key is used to decrypt requests from users to set up a circuit and negotiate ephemeral keys. The TLS protocol also establishes a short-term link key when communicating between ORs. Short-term keys are rotated periodically and independently, to limit the impact of key compromise.

Section 4.1 presents the fixed-size *cells* that are the unit of most communication in Tor. We describe in Section 4.3 how circuits are built, extended, truncated, and destroyed. Section 4.5 describes how TCP streams are routed through the network. We address integrity checking in Section 4.6, and resource limiting in Section 4.7. Finally, Section 4.8 talks about congestion control and fairness issues.

4.1 Cells

Onion routers communicate with one another, and with users' OPs, via TLS connections with ephemeral keys. Using TLS conceals the data on the connection with perfect forward secrecy, and prevents an attacker from modifying data on the wire or impersonating an OR.

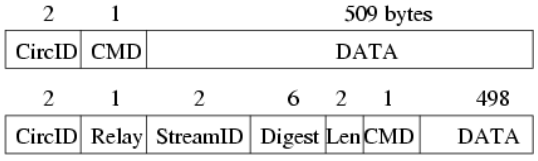
Most traffic passes along these connections in fixed-size cells.[1] Each fixed-size cell is 512 bytes, and consists of a header and a payload. The header includes a circuit identifier (circID) that specifies which circuit the cell refers to (many circuits can be multiplexed over the single TLS connection), and a command to describe what to do with the cell's payload. (Circuit identifiers are connection-specific: each circuit has a different circID on each OP/OR or OR/OR connection it traverses.) Fixed-size cells provide some resistance to traffic analysis but are inefficient, so some control cells are variable length, where the ability of an attacker to detect their presence doesn't affect security. Fixed-size cells also make the packet-size distribution of Tor distinctive, contrary to the goal of protocol-fingerprinting resistance. Therefore a variable-length padding cell was introduced (but is currently unused) to allow the implementation of schemes to disguise packet length.

Based on their command, cells are either *control* cells, which are always interpreted by the node that receives them, *relay* cells, which carry end-to-end stream data, or *relay_early* cells, which work similarly to *relay* cells but are distinguished to enforce the maximum path length (see 6.1). The fixed-size control cell commands are: *padding* (currently used for keepalive, but also usable for link padding); *create* or *created* (used to set up a new circuit); *create_fast* or *created_fast* (used to set up a new circuit to the first hop, without public key computation); *netinfo* (used to help nodes discover the time and their own address); and *destroy* (to tear down a circuit). The variable-length control cell commands are: *versions* (used for link-protocol negotiation); *vpadding* (variable length padding); and *certs*, *auth_challenge*, *authenticate*, and *authorize* (used for OR-OR and OP-OR authentication).

Relay cells have an additional header (the relay header) at the front of the payload, containing a streamID (stream identifier: many streams can be multiplexed over a circuit); an end-to-end truncated digest for integrity checking; the length of the relay payload; and a relay command. The entire contents of the relay header and the relay cell payload are encrypted or decrypted together as the relay cell moves along the circuit, using the 128-bit AES cipher in counter mode to generate a cipher stream. The relay commands are: *relay data* (for data flowing down the stream), *relay begin* (to open a stream), *relay begin dir* (to open a local stream for directory information), *relay end* (to close a stream cleanly), *relay teardown* (to close a broken stream), *relay connected* (to notify the OP that a relay begin has succeeded), *relay extend* and *relay extended* (to extend the circuit by a hop, and to acknowledge), *relay truncate* and *relay truncated* (to tear down only part of the circuit, and to acknowledge), *relay sendme* (used for congestion control), *relay resolve* and *relay resolved* (used for anonymous DNS), and *relay drop* (used to implement long-range dummies). We give a visual overview of cell structure

[1] A few cell types, notably those used for connection establishment, are variable-sized.

plus the details of relay cell structure, and then describe each of these cell types and commands in more detail below.



4.2 TLS details

Tor's original (version 1) TLS handshake was fairly straightforward. The initiator said that it supported a sensible set of cryptographic algorithms and parameters (ciphersuites, in TLS terminology) and the responder selected one. If one side wanted to prove to the other that it was a Tor node, it would send a two-element certificate chain signed by the key published in the Tor directory.

This approach met all the security properties envisaged at the time the 2004 design paper was written, but Tor's increasing use in censorship resistance changed the requirements Tor's protocol signature also had to look (to the extent possible) like that of HTTPS web traffic, to prevent censors using deep-packet-inspection to detect and block Tor. Tor's use of fixed two-certificate chains was a giveaway.

After an intermediary design that relied (fragilely and observably) on TLS renegotiation, Tor shifted to a mixed authentication model, where the TLS handshake can complete with any (secure) credentials and ciphersuites desired, and an inner handshake done within the TLS protocol provides the authentication that Tor actually wants.[2]

To perform the inner handshake once the TLS handshake is done, the parties negotiate a Tor link protocol version by exchanging *versions* cells containing the list of link protocol versions each supports, then choosing the highest version supported by both. Next, the responder sends a *certs* cell containing the actual certificate chain authenticating the public key it used for the TLS handshake with its identity key. The responder also sends a random nonce as a challenge. If the initiator also wishes to authenticate herself as an OR, she sends a *certs* cell of her own, followed by an *authenenticate* cell signed by her link key, containing: a digest of both identity keys, a digest of all messages she has sent and received so far, a digest of the responder's TLS link certificate, the current time, a random nonce, and a MAC using the TLS master secret as its key, of the TLS handshake's client_random and server_random parameters.

[2]To determine that this newer version of the link protocol handshake is to be used, the initiator avoids using the exact set of ciphersuites used by early Tor versions, and the Tor responder uses an X.509 certificate unlike those generated by earlier versions of Tor. This may be too clever for Tor's own good; we mean to eliminate it once every supported version of Tor supports this version of Tor's link protocol.

4.3 Circuits and streams

Onion Routing originally built one circuit for each TCP stream. Because building a circuit can take several tenths of a second (due to public-key cryptography and network latency), this design imposed high costs on applications like web browsing that open many TCP streams.

In Tor, each circuit can be shared by many TCP streams. To avoid delays, OPs construct circuits preemptively. To limit linkability among their streams, the user's OP will not assign a new stream to a circuit if the circuit[3] has previously carried a stream which the user has indicated should be isolated from the new one. By default, a user signals that two streams should not be linkable by making SOCKS connections to different ports, from a different IP address, or with different SOCKS authentication credentials. Tor's SOCKS ports can additionally be configured to isolate streams based on destination port[4] or address. Even when a stream would otherwise be permitted to be carried by a circuit, if the circuit's first stream was created more than 10 minutes (by default) ago, that circuit will not be considered for re-use and closed once there are no remaining streams, then the OP will build a new circuit preemptively.

With careful configuration, this system can be used to avoid numerous linking attacks. For example, a user accessing multiple pseudonymous chat accounts could configure her chat application to use a separate SOCKS username for each one, thus telling Tor not to place any of their streams on the same circuit (which would reveal to the exit node and suggest to the exit that the accounts were shared by the same user). Or for applications that don't support SOCKS authentication, the user might configure multiple SOCKS ports, and tell each application a different port, so that for example her anonymous web browsing never shares a circuit with her pseudonymous IM usage.

OPs consider rotating to a new circuit once a minute: thus even heavy users spend negligible time building circuits, but a limited number of requests can be linked to each other through a given exit node. Also, because circuits are built in the background, OPs can recover from failed circuit creation without harming user experience.

Constructing a circuit

A user's OP constructs circuits incrementally, negotiating a symmetric key with each OR on the circuit, one hop at a time,

[3]Occasionally people suggest that isolating *exits* would be better than isolating circuits, so that two isolated streams would never appear to come from the same IP as one another. A little analysis shows that this approach would hurt anonymity, however: a destination service could observe that two accounts both used Tor, but never arrived from the same exit node IP at the same time, and thereby conclude that those accounts were probably run by the same user.

[4]Some designs have suggested port-based isolation as a means for keeping use of separate protocols from becoming linked to each other. This is non-workable, though, if one of the protocols is one such as HTTP or HTTPS where applications can typically be made to use any attacker-selected port.

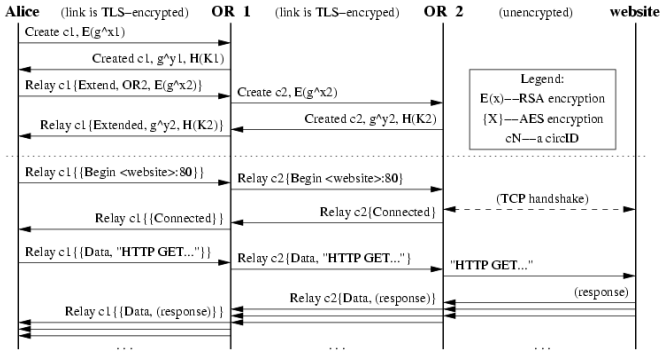


Figure 1: Alice builds a two-hop circuit and begins fetching a web page.

and using each partially created circuit to communicate with the next hop. To begin creating a new circuit, the OP (call her Alice) sends a *create* cell to the first node in her chosen path (call him Bob). (She chooses a new circID C_{AB} not currently used on the connection from her to Bob.) The *create* cell's payload contains the first half of the Diffie-Hellman handshake (g^x), encrypted to the onion key of Bob. Bob responds with a *created* cell containing g^y along with a hash of the negotiated key $K = g^{xy}$.

Once the circuit has been established, Alice and Bob can send one another relay cells encrypted with the negotiated key.[5] More detail is given in the next section.

To extend the circuit further, Alice sends a *relay extend* cell to Bob, specifying the address of the next OR (call her Carol), and an encrypted g^{x_2} for her. Bob copies the half-handshake into a *create* cell, and passes it to Carol to extend the circuit. (Bob chooses a new circID C_{BC} not currently used on the connection between him and Carol. Alice never needs to know this circID; only Bob associates C_{AB} on his connection with Alice to C_{BC} on his connection with Carol.) When Carol responds with a *created* cell, Bob wraps the payload into a *relay extended* cell and passes it back to Alice. Now the circuit is extended to Carol, and Alice and Carol share a common key $K_2 = g^{x_2 y_2}$.

To extend the circuit to a third node or beyond, Alice proceeds as above, always telling the last node in the circuit to extend one hop further.

This circuit-level handshake protocol achieves unilateral entity authentication (Alice knows she's handshaking with the OR, but the OR doesn't care who is opening the circuit—Alice uses no public key and remains anonymous) and unilateral key authentication (Alice and the OR agree on a key, and Alice knows only the OR learns it). It also achieves forward secrecy and key freshness. More formally, the protocol is as follows (where $E_{PK_{Bob}}(\cdot)$ is encryption with Bob's public key, H is a secure hash function, and $|$ is concatenation):

$$\text{Alice} \rightarrow \text{Bob} : E_{PK_{Bob}}(g^x)$$
$$\text{Bob} \rightarrow \text{Alice} : g^y, H(K|\text{“handshake”})$$

In the second step, Bob proves that it was he who received g^x, and who chose y. We use PK encryption in the first step (rather than, say, using the first two steps of STS, which has a signature in the second step) because a single cell is too small to hold both a public key and a signature. Preliminary analysis with the NRL protocol analyzer [33] shows this protocol to be secure (including perfect forward secrecy) under the traditional Dolev-Yao model. In practice, since the most data that can be encrypted with padded RSA-1024 is less than the size needed to hold an DH-1024 value, we need to use hybrid encryption. Tor's original hybrid encryption approach here was somewhat poorly designed, but turns out to be secure anyway.

As an optimization, Alice client may sent a *create_fast* cell in place of her first *create* cell: instead of sending an encrypted g^x value, she simply sends a random value x, Bob replies with a *created_fast* cell containing a random value y, and they base their shared keys on $H(x|y)$. This handshake saves the expense of RSA and Diffie-Hellman, but provides no authentication, integrity, confidentiality or forward secrecy on its own: it relies on the TLS protocol that Alice and Bob are already using for their link in order to achieve these properties.

Relay cells

Once Alice has established the circuit (so she shares keys with each OR on the circuit), she can send relay cells. Upon receiving a relay cell, an OR looks up the corresponding circuit, and decrypts the relay header and payload with the session key for that circuit. If the cell is headed away from Alice the OR then checks whether the decrypted cell has a valid digest (as an optimization, the first two bytes of the integrity check are zero, so in most cases we can avoid computing the hash). If the digest is valid (See 4.6, it accepts the relay cell and processes it as described below. Otherwise, the OR looks up the circID and OR for the next step in the circuit, replaces the circID as appropriate, and sends the decrypted relay cell to the next OR. (If the OR at the end of the circuit receives an unrecognized relay cell, an error has occurred, and the circuit is torn down.)

OPs treat incoming relay cells similarly: they iteratively unwrap the relay header and payload with the session keys shared with each OR on the circuit, from the closest to farthest. If at any stage the digest is valid, the cell must have originated at the OR whose encryption has just been removed.

To construct a relay cell addressed to a given OR, Alice assigns the digest, and then iteratively encrypts the cell payload (that is, the relay header and payload) with the symmetric key of each hop up to that OR. Because the digest is encrypted to a different value at each step, only at the targeted OR will

[5] Actually, the negotiated key is used to derive four symmetric keys: one for each direction for AES, and one in each direction for integrity. To generate enough key material, Tor uses an ad hoc key derivation function where K is expanded to $H(K|[00])|H(K|[01])|...$

it have a meaningful value.[6] This *leaky pipe* circuit topology allows Alice's streams to exit at different ORs on a single circuit. Alice may choose different exit points because of their exit policies, or to keep the ORs from knowing that two streams originate from the same person.

When an OR later replies to Alice with a relay cell, it encrypts the cell's relay header and payload with the single key it shares with Alice, and sends the cell back toward Alice along the circuit. Subsequent ORs add further layers of encryption as they relay the cell back to Alice.

To tear down a circuit, Alice sends a *destroy* control cell. Each OR in the circuit receives the *destroy* cell, closes all streams on that circuit, and passes a new *destroy* cell forward. But just as circuits are built incrementally, they can also be torn down incrementally: Alice can send a *relay truncate* cell to a single OR on a circuit. That OR then sends a *destroy* cell forward, and acknowledges with a *relay truncated* cell. Alice can then extend the circuit to different nodes, without signaling to the intermediate nodes (or a limited observer) that she has changed her circuit. Similarly, if a node on the circuit goes down, the adjacent node can send a *relay truncated* cell back to Alice. Thus the "break a node and see which circuits go down" attack [4] is weakened.

4.4 Choosing nodes for circuits

Early onion routing designs assumed a set of uniform nodes, from which clients would therefore choose uniformly at random. But this approach creates terrible bandwidth bottlenecks: a server that would allow 10x as many bytes per second as another would still get the same number of circuits constructed through it, leading to overloaded low-capacity relays and underutilized high-capacity bandwidth.

Therefore, Tor now weights[7] its choice of nodes by servers' bandwidths, so that a server with more bandwidth gets more circuits, and therefore (probabilistically) more of the traffic.

Later, it proved that weighting by bandwidth was also suboptimal, because of nonuniformity in path selection rules. Consider that if node A is suitable for use at any point in a circuit, but node B is suitable only as the middle node, then node A will be considered for use three times as often as B. If the two nodes have equal bandwidth, node A will be chosen three times as often, leading to it being overloaded in comparison with B. As of 0.2.2.10-alpha, we moved to a more sophisticated approach, where nodes are chosen proportionally to their bandwidth, as weighted by an algorithm to optimize load-balancing between nodes of different capabilities.

[6] With 48 bits of digest per cell, the probability of an accidental collision is far lower than the chance of hardware failure.

[7] In the original paper, we imagined that we might take Morphmix's approach, and divide nodes into "bandwidth classes", such that clients would choose only from among nodes having at least the same approximate bandwidth as the clients. This may be a good design for peer-to-peer anonymity networks, but it doesn't seem to work for the Tor network: the most useful high-capacity nodes have more capacity than nearly any typical client.

Weighting by *advertised* bandwidth would open the possibility of an attacker trying to see a disproportionate number of circuits—not by running an extra-high number of nodes—but by claiming to have a very large bandwidth.

For a while, Tor tried to limit the impact of this attack by limiting the maximum bandwidth that a client would believe, so that a single rogue node couldn't just claim to have infinite bandwidth. This approach proved unuseful, since some real nodes did in fact have very high bandwidth.

But now, clients use *measured* bandwidth values published in the network status consensus document (see section 6.3). A subset of the authorities measure and vote on nodes' observed bandwidth, to prevent misbehaving nodes from claiming (intentionally or accidentally) to have too much capacity.

4.4.1 Guard nodes

We assume that if an attacker controls or monitors the first hop and last hop of a circuit, then the attacker can deanonymize the user by correlating timing and volume information. Many of the security improvements to path selection discussed in this post concentrate on reducing the probability that an attacker can be in this position, but no reasonably efficient proposal can eliminate the possibility.

Therefore, each time a user creates a circuit, there is a small chance that the circuit will be compromised. However, most users create a large number of Tor circuits, so with the original path selection algorithm, these small chances would build up into a large chance that at least one of their circuits would be compromised. For users whose goal is to avoid having any of their communications partners learned, having a fraction of their circuits compromised is as unacceptable as having all of them compromised.

To help improve this situation Tor uses Guard Nodes (initially called "helper nodes", invented by Wright, Adler, Levine, and Shields and proposed for use in Tor by verlier and Syverson). Each Tor client picks a few Tor nodes to use persistently as its "guards", and uses one of them as the first hop for all circuits (as long as those nodes remain operational).

This doesn't affect the probability that the first circuit is compromised, but it does mean that if the guard nodes chosen by a user are not attacker-controlled all their future circuits will be safe. On the other hand, users who choose attacker-controlled guards will have about c/N of their circuits compromised, where c is the amount of attacker-controlled network resource and N is the total network resource. Without guard nodes every circuit has a $(c/N)^2$ probability of being compromised.

Essentially, the guard node approach recognises that some circuits are going to be compromised, but it's better to increase your probability of having no compromised circuits at the expense of also increasing the proportion of your circuits that will be compromised if any of them are. This is because compromising a fraction of a user's circuits—sometimes even

just one—can be enough to compromise a user's anonymity. For users who have good guard nodes, the situation is much better, and for users with bad guard nodes the situation is not much worse than before.

4.4.2 Avoiding duplicate node families in the same circuit

As mentioned above, if the first and last node in a circuit are controlled by an adversary, they can use traffic correlation attacks to notice that the traffic entering the network at the first hop matches traffic leaving the circuit at the last hop, and thereby trace a client's activity with high probability. Research on preventing this attack has not yet come up with any affordable, effective defense suitable for use in a low-latency anonymity network. Therefore, the most promising mitigation strategies seem to involve lowering the attacker's chances of controlling both ends of a circuit.

To this end, clients do not use any two nodes in a circuit whose IP addresses are in the same /16. (When we designed the network, it was marginally more difficult to acquire a large number of disparate addresses than it was to get a large number of concentrated addresses. This approach is imperfect, but possibly better than nothing.

To allow honest node operators to run more than one server without inadvertently giving themselves the chance to see more traffic than they should, we also allow nodes to declare themselves to be members of the same "family," such that a client won't use two nodes in the same family in the same circuit. (Clients only believe mutual family declarations, so that an adversary can't capture routes by having his nodes claim unilaterally to be in a family with every node the adversary doesn't control.)

4.5 Opening and closing streams

When Alice's application wants a TCP connection to a given address and port, it asks the OP (via SOCKS) to make the connection. The OP chooses the newest open circuit (or creates one if needed), and chooses a suitable OR on that circuit to be the exit node (usually the last node, but maybe others due to exit policy conflicts; see Section 6.2.) The OP then opens the stream by sending a *relay begin* cell to the exit node, using a new random streamID. Once the exit node connects to the remote host, it responds with a *relay connected* cell. Upon receipt, the OP sends a SOCKS reply to notify the application of its success. The OP now accepts data from the application's TCP stream, packaging it into *relay data* cells and sending those cells along the circuit to the chosen OR.

(As an optimization, to avoid a round-trip while waiting for a connected reply, clients may send data immediately after the connected cell. They need to be ready to send the same data to another stream, though, if no connected cell arrives.)

There's a catch to using SOCKS, however—some applications pass the alphanumeric hostname to the Tor client, while others resolve it into an IP address first and then pass the IP address to the Tor client. If the application does DNS resolution first, Alice thereby reveals her destination to the remote DNS server, rather than sending the hostname through the Tor network to be resolved at the far end. Common applications like Firefox and SSH need to be configured to use SOCKS4A or SOCKS5 (with the option to send hostnames rather than IP address) to avoid this vulnerability.

With Firefox, the Torbutton add-on ensures that the browser sends requests via Tor by configuring Firefox to correctly use a SOCKS proxy. However, this is not in itself sufficient to provide private web browsing, because the browser provides many ways for a malicious site to link separate accesses to being from the from the same user. Therefore Torbutton and patches applied to the version of Firefox delivered with Tor also restrict tracking capabilities – both intended ones (such as cookies, and more modern variants like DOM storage) and unintended channels (such as TLS session IDs), normalizes browser characteristics accessible from Javascript (such as screen size and system colors), and blocks plugins which may leak identifying information. Previously the Privoxy filtering proxy was used for this purpose, but its major weakness is that it is unable to protect the user from being tracked over HTTPS because a proxy sees only encrypted content.

Other applications, which can be configured to use SOCKS (and send the proxy a hostname rather than IP address), may be connected directly to Tor. Other options available are to intercept calls to the resolver and sockets libraries with *torsocks* or to use an operating system firewall's transparent port-forwarding mechanism (Tor supports BSD and Linux's in-built port-forwarding) to intercept outgoing DNS requests and TCP connections, sending them via Tor.

Closing a Tor stream is analogous to closing a TCP stream: it uses a two-step handshake for normal operation, or a one-step handshake for errors. If the stream closes abnormally, the adjacent node simply sends a *relay teardown* cell. If the stream closes normally, the node sends a *relay end* cell down the circuit, and the other side responds with its own *relay end* cell. Because all relay cells use layered encryption, only the destination OR knows that a given relay cell is a request to close a stream. This two-step handshake allows Tor to support TCP-based applications that use half-closed connections.

4.6 Integrity checking on streams

Because the old Onion Routing design used a stream cipher without integrity checking, traffic was vulnerable to a malleability attack: though the attacker could not decrypt cells, any changes to encrypted data would create corresponding changes to the data leaving the network. This weakness allowed an adversary who could guess the encrypted content to

change a padding cell to a destroy cell; change the destination address in a *relay begin* cell to the adversary's webserver; or change an FTP command from `dir` to `rm *`. (Even an external adversary could do this, because the link encryption similarly used a stream cipher.)

Because Tor uses TLS on its links, external adversaries cannot modify data. Addressing the insider malleability attack, however, is more complex.

We could do integrity checking of the relay cells at each hop, either by including MACs or by using an authenticating cipher mode like GCM, but there are some problems. First, these approaches impose a message-expansion overhead at each hop, and so we would have to either leak the path length or waste bytes by padding to a maximum path length. Second, these solutions can only verify traffic coming from Alice: ORs would not be able to produce suitable hashes for the intermediate hops, since the ORs on a circuit do not know the other ORs' session keys. Third, we have already accepted that our design is vulnerable to end-to-end timing attacks; so tagging attacks performed within the circuit provide no additional information to the attacker.

Thus, we check integrity only at the edges of each stream. (Remember that in our leaky-pipe circuit topology, a stream's edge could be any hop in the circuit.) When Alice negotiates a key with a new hop, they each initialize a SHA-1 digest with a derivative of that key, thus beginning with randomness that only the two of them know. Then they each incrementally add to the SHA-1 digest the contents of all relay cells they create, and include with each relay cell the first four bytes of the current digest. Each also keeps a SHA-1 digest of data received, to verify that the received hashes are correct.

To be sure of removing or modifying a cell, the attacker must be able to deduce the current digest state (which depends on all traffic between Alice and Bob, starting with their negotiated key). Attacks on SHA-1 where the adversary can incrementally add to a hash to produce a new valid hash don't work, because all hashes are end-to-end encrypted across the circuit. The computational overhead of computing the digests is minimal compared to doing the AES encryption performed at each hop of the circuit. We use only four bytes per cell to minimize overhead; the chance that an adversary will correctly guess a valid hash is acceptably low, given that the OP or OR tear down the circuit if they receive a bad hash.

This approach has, however, appeared less robust than we hoped; while tagging attacks don't provide more information than an end-to-end attacker could get through passive correlation attacks, they succeed more quickly. Even that isn't such a big deal, were it not for a class of attacks that become possible if an attacker can detect non-corelatable circuits early and kill them. We are therefore looking into improved constructions for integrity, especially ones based on wide-block ciphers. We hope to also take the opportunity to move the authentication mechanism away from the moribund SHA-1.

4.7 Rate limiting and fairness

Volunteers are more willing to run services that can limit their bandwidth usage. To accommodate them, Tor servers use a token bucket approach [47] to enforce a long-term average rate of incoming bytes, while still permitting short-term bursts above the allowed bandwidth. To accommodate volunteers who are charged when their daily, weekly, or monthly bandwidth usage exceeds a limit, Tor servers can be configured to "hibernate", closing existing connections and refusing new ones, once the limit has been reached. The server will become usable again at the start of the next period.

Because the Tor protocol outputs about the same number of bytes as it takes in, it is sufficient in practice to limit only incoming bytes. With TCP streams, however, the correspondence is not one-to-one: relaying a single incoming byte can require an entire 512-byte cell. (We can't just wait for more bytes, because the local application may be awaiting a reply.) Therefore, we treat this case as if the entire cell size had been read, regardless of the cell's fullness.

To provide good latency for interactive service, Tor chooses which cells to deliver favouring circuits that had been quiet recently. Specifically, when Tor is about to put a cell on an outgoing connection it chooses the circuit which has sent the lowest total exponentially-decaying number of cells so far. Currently each cell has a 30-second half-life. Such preferential treatment presents a possible end-to-end attack, but an adversary observing both ends of the stream can already learn this information through timing attacks.

4.8 Congestion control

Even with bandwidth rate limiting, we still need to worry about congestion, either accidental or intentional. If enough users choose the same OR-to-OR connection for their circuits, that connection can become saturated. For example, an attacker could send a large file through the Tor network to a webserver he runs, and then refuse to read any of the bytes at the webserver end of the circuit. Without some congestion control mechanism, these bottlenecks can propagate back through the entire network. We don't need to reimplement full TCP windows (with sequence numbers, the ability to drop cells when we're full and retransmit later, and so on), because TCP already guarantees in-order delivery of each cell. We describe our response below.

Circuit-level throttling: To control a circuit's bandwidth usage, each OR keeps track of two windows. The *packaging window* tracks how many relay data cells the OR is allowed to package (from incoming TCP streams) for transmission back to the OP, and the *delivery window* tracks how many relay data cells it is willing to deliver to TCP streams outside the network. Each window is initialized (say, to 1000 data cells). When a data cell is packaged or delivered, the appropriate window is decremented. When an OR has received enough

data cells (currently 100), it sends a *relay sendme* cell towards the OP, with streamID zero. When an OR receives a *relay sendme* cell with streamID zero, it increments its packaging window. Either of these cells increments the corresponding window by 100. If the packaging window reaches 0, the OR stops reading from TCP connections for all streams on the corresponding circuit, and sends no more relay data cells until receiving a *relay sendme* cell.

The OP behaves identically, except that it must track a packaging window and a delivery window for every OR in the circuit. If a packaging window reaches 0, it stops reading from streams destined for that OR.

Stream-level throttling: The stream-level congestion control mechanism is similar to the circuit-level mechanism. ORs and OPs use *relay sendme* cells to implement end-to-end flow control for individual streams across circuits. Each stream begins with a packaging window (currently 500 cells), and increments the window by a fixed value (50) upon receiving a *relay sendme* cell. Rather than always returning a *relay sendme* cell as soon as enough cells have arrived, the stream-level congestion control also has to check whether data has been successfully flushed onto the TCP stream; it sends the *relay sendme* cell only when the number of bytes pending to be flushed is under some threshold (currently 10 cells worth).

These arbitrarily chosen parameters give tolerable but not great throughput and delay; see Section 8. See also Section **??** for discussion of future work directions on the topic.

5 Rendezvous Points and hidden services

Rendezvous points are a building block for *location-hidden services* (also known as *responder anonymity*) in the Tor network. Location-hidden services allow Bob to offer a TCP service, such as a webserver, without revealing his IP address. This type of anonymity protects against distributed DoS attacks: attackers are forced to attack the onion routing network because they do not know Bob's IP address.

Our design for location-hidden servers has the following goals. **Access-control:** Bob needs a way to filter incoming requests, so an attacker cannot flood Bob simply by making many connections to him. **Robustness:** Bob should be able to maintain a long-term pseudonymous identity even in the presence of router failure. Bob's service must not be tied to a single OR, and Bob must be able to migrate his service across ORs. **Smear-resistance:** A social attacker should not be able to "frame" a rendezvous router by offering an illegal or disreputable location-hidden service and making observers believe the router created that service. **Application-transparency:** Although we require users to run special software to access location-hidden servers, we must not require them to modify their applications.

We provide location-hiding for Bob by allowing him to advertise several onion routers (his *introduction points*) as contact points, in a distributed hash table (DHT). He does this by publishing the hidden service descriptor (containing introduction point's addresses) to the ORs whose identity keys are closest to a hash of the location-hidden service's identity key, the current date, and a replica number. Optionally, the portion of the hidden service descriptor describing the introduction points can be encrypted under a key shared with authorized users of the hidden service. Therefore not only can unauthorized users not connect to the hidden service or its introduction points (the descriptor contains an authentication credential), they also cannot discover whether the hidden service is online.

Alice, the client, chooses an OR as her *rendezvous point*. She connects to one of Bob's introduction points, informs him of her rendezvous point, and then waits for him to connect to the rendezvous point. This extra level of indirection helps Bob's introduction points avoid problems associated with serving unpopular files directly (for example, if Bob serves material that the introduction point's community finds objectionable, or if Bob's service tends to get attacked by network vandals). The extra level of indirection also allows Bob to respond to some requests and ignore others.

5.1 Rendezvous points in Tor

The following steps are performed on behalf of Alice and Bob by their local OPs; application integration is described more fully below.

- Bob generates a long-term public key pair to identify his service.
- Bob chooses some introduction points, and advertises them on the lookup service, signing the advertisement with his public key. He can add more later.
- Bob builds a circuit to each of his introduction points, and tells them to wait for requests.
- Alice learns about Bob's service out of band (perhaps Bob told her, or she found it on a website). She retrieves the details of Bob's service from the lookup service. If Alice wants to access Bob's service anonymously, she must connect to the lookup service via Tor.
- Alice chooses an OR as the rendezvous point (RP) for her connection to Bob's service. She builds a circuit to the RP, and gives it a randomly chosen "rendezvous cookie" to recognize Bob.
- Alice opens an anonymous stream to one of Bob's introduction points, and gives it a message (encrypted with Bob's public key) telling it about herself, her RP and rendezvous cookie, and the start of a DH handshake. The introduction point sends the message to Bob.
- If Bob wants to talk to Alice, he builds a circuit to Alice's RP and sends the rendezvous cookie, the second half of the DH handshake, and a hash of the session key they now share. By the same argument as in Section 4.3, Alice knows she shares the key only with Bob.

- The RP connects Alice's circuit to Bob's. Note that RP can't recognize Alice, Bob, or the data they transmit.
- Alice sends a *relay begin* cell along the circuit. It arrives at Bob's OP, which connects to Bob's webserver.
- An anonymous stream has been established, and Alice and Bob communicate as normal.

When establishing an introduction point, Bob provides the onion router with the public key identifying his service. Bob signs his messages, so others cannot usurp his introduction point in the future. He uses the same public key to establish the other introduction points for his service, and periodically refreshes his entry in the lookup service.

The message that Alice gives the introduction point includes a hash of Bob's public key and an optional initial authorization token (the introduction point can do prescreening, for example to block replays). Her message to Bob may include an end-to-end authorization token so Bob can choose whether to respond. The authorization tokens can be used to provide selective access: important users can get uninterrupted access. During normal situations, Bob's service might simply be offered directly from mirrors, while Bob gives out tokens to high-priority users. If the mirrors are knocked down, those users can switch to accessing Bob's service via the Tor rendezvous system.

Bob's introduction points are themselves subject to DoS—he must open many introduction points or risk such an attack. He can provide selected users with a current list or future schedule of unadvertised introduction points; this is most practical if there is a stable and large group of introduction points available. Bob could also give secret public keys for consulting the lookup service. All of these approaches limit exposure even when some selected users collude in the DoS.

5.2 Integration with user applications

Bob configures his onion proxy to know the local IP address and port of his service, a strategy for authorizing clients, and his public key. The onion proxy anonymously publishes a signed statement of Bob's public key, an expiration time, and the current introduction points for his service onto the lookup service, indexed by the hash of his public key. Bob's webserver is unmodified, and doesn't even know that it's hidden behind the Tor network.

Alice's applications also work unchanged—her client interface remains a SOCKS proxy. We encode all of the necessary information into the fully qualified domain name (FQDN) Alice uses when establishing her connection. Location-hidden services use a virtual top level domain called `.onion`: thus hostnames take the form `y.onion` where `y` encodes the hash of the public key. Alice's onion proxy examines addresses; if they're destined for a hidden server, it decodes the key and starts the rendezvous as described above.

5.3 Previous rendezvous work

Rendezvous points in low-latency anonymity systems were first described for use in ISDN telephony [28, 36]. Later low-latency designs used rendezvous points for hiding location of mobile phones and low-power location trackers [21, 37]. Rendezvous for anonymizing low-latency Internet connections was suggested in early Onion Routing work [25], but the first published design was by Ian Goldberg [24]. His design differs from ours in three ways. First, Goldberg suggests that Alice should manually hunt down a current location of the service via Gnutella; our approach makes lookup transparent to the user, as well as faster and more robust. Second, in Tor the client and server negotiate session keys with Diffie-Hellman, so plaintext is not exposed even at the rendezvous point. Third, our design minimizes the exposure from running the service, to encourage volunteers to offer introduction and rendezvous services. Tor's introduction points do not output any bytes to the clients; the rendezvous points don't know the client or the server, and can't read the data being transmitted. The indirection scheme is also designed to include authentication/authorization—if Alice doesn't include the right cookie with her request for service, Bob need not even acknowledge his existence.

6 Other design decisions

6.1 Denial of service

Providing Tor as a public service creates many opportunities for denial-of-service attacks against the network. While flow control and rate limiting (discussed in Section 4.8) prevent users from consuming more bandwidth than routers are willing to provide, opportunities remain for users to consume more network resources than their fair share, or to render the network unusable for others.

First of all, there are several CPU-consuming denial-of-service attacks wherein an attacker can force an OR to perform expensive cryptographic operations at little cost to the attacker. For example, an attacker can fake the start of a TLS handshake, forcing the OR to carry out its (comparatively expensive) half of the handshake at no real computational cost to the attacker.

We have not yet implemented any defenses for these attacks, but several approaches are possible. First, ORs can require clients to solve a puzzle [14] while beginning new TLS handshakes or accepting *create* cells. So long as these tokens are easy to verify and computationally expensive to produce, this approach limits the attack multiplier. Additionally, ORs can limit the rate at which they accept *create* cells and TLS connections, so that the computational work of processing them does not drown out the symmetric cryptography operations that keep cells flowing. This rate limiting could, however, allow an attacker to slow down other users when

they build new circuits.

The Tor network itself can be exploited as a DoS amplifier, because for every relay cell sent into an OP, a cell is generated at each hop on the circuit. An adversary could create a long path, potentially going through the same node many times, and overload CPU or network resources with only a small investment of both. To resist this attack, the length of a path is limited to 8, enforced by the distinction between *relay* and *relay_early* cells. Incoming *relay_early* cells may contain any type of relay cell but if they are not destined for the OR which receives them, result in a further *relay_early* cell being generated. Only 8 *relay_early* cells are permitted to be sent on a circuit. Similarly *relay* cells result in a *relay* cell being created, and may be sent without limit, but *relay* cells cannot contain an extend request. In this way, intermediate ORs cannot know how long the path length is (they always see up to 8 *relay_early* cells, and don't know what they contain) but an OP cannot send more than 8 extend requests and so cannot generate a path of longer than 8 hops. This does not however prevent an adversary tunneling Tor over Tor, and connecting from an exit node back to the Tor network.

Adversaries can also attack the Tor network's hosts and network links. Disrupting a single circuit or link breaks all streams passing along that part of the circuit. Users similarly lose service when a router crashes or its operator restarts it. The current Tor design treats such attacks as intermittent network failures, and depends on users and applications to respond or recover as appropriate. A future design could use an end-to-end TCP-like acknowledgment protocol, so no streams are lost unless the entry or exit point is disrupted. This solution would require more buffering at the network edges, however, and the performance and anonymity implications from this extra complexity still require investigation.

In general, deliberate denial of service attacks have not yet had a noticeable impact on the Tor network in the wild. While we are aware to the possibility, and looking for more ways to mitigate possible DoS attack vectors, we are currently more concerned with circumstances under which misbehaving nodes accidentally "attack" part of the network. For example, our decision to have the directory authorities act both as the initial contact point for clients to get directory information has led to clients placing excessive load on those authorities during times when they couldn't find pieces of directory information.

6.2 Exit policies, node history, and abuse

The prospect of exit abuse limits the number of users willing to run Tor nodes. Exit abuse is a serious barrier to wide-scale Tor deployment. Anonymity presents would-be vandals and abusers with an opportunity to hide the origins of their activities. Attackers can harm the Tor network by implicating exit servers for their abuse. Also, applications that commonly use IP-based authentication (such as institutional mail or webservers) can be fooled by the fact that anonymous connections appear to originate at the exit OR.

To enable volunteers to run nodes without having to necessarily worry about these abuse issues, each onion router's *exit policy* describes to which external addresses and ports the router will connect. On one end of the spectrum are *open exit* nodes that will connect anywhere. On the other end are *middleman* nodes that only relay traffic to other Tor nodes, and *private exit* nodes that only connect to a local host or network. A private exit can allow a client to connect to a given host or network more securely—an external adversary cannot eavesdrop traffic between the private exit and the final destination, and so is less sure of Alice's destination and activities. Many onion routers in the current network function as *restricted exits* that permit connections to the world at large, but prevent access to certain abuse-prone addresses and services such as SMTP. The OR might also be able to authenticate clients to prevent exit abuse without harming anonymity [45].

Many administrators use port restrictions to support only a limited set of services, such as HTTP, SSH, or AIM. This is not a complete solution, of course, since abuse opportunities for these protocols are still well known.

To manage the abuse potential, The Tor Project operates a DNS blacklist system, allowing service operators to easily identify whether a particular incoming connection may have arrived over the Tor network. The service may then choose to block the connection, subject it to extra scrutiny, or restrict the rights of the user who is connecting via Tor (such as giving read-only, rather than read-write, access to a wiki). A similar service also allows retrospective queries over the list of exit nodes to allow exit-node operators to show that their computer was an exit node at the time an abusive connection was made, and therefore that they should not be liable for any harm cause.

A mixture of open and restricted exit nodes allows the most flexibility for volunteers running servers. But while having many middleman nodes provides a large and robust network, having only a few exit nodes reduces the number of points an adversary needs to monitor for traffic analysis, and places a greater burden on the exit nodes. This tension can be seen in the Java Anon Proxy cascade model, wherein only one node in each cascade needs to handle abuse complaints—but an adversary only needs to observe the entry and exit of a cascade to perform traffic analysis on all that cascade's users. The hydra model (many entries, few exits) presents a different compromise: only a few exit nodes are needed, but an adversary needs to work harder to watch all the clients; see Section 10.

Finally, we note that exit abuse must not be dismissed as a peripheral issue: when a system's public image suffers, it can reduce the number and diversity of that system's users, and thereby reduce the anonymity of the system itself. Like usability, public perception is a security parameter. Sadly, preventing abuse of open exit nodes is an unsolved problem, and will probably remain an arms race for the foreseeable

future. The abuse problems faced by Princeton's CoDeeN project [35] give us a glimpse of likely issues.

6.3 Directory Servers

First-generation Onion Routing designs [7, 38] used in-band network status updates: each router flooded a signed statement to its neighbors, which propagated it onward. But anonymizing networks have different security goals than typical link-state routing protocols. For example, delays (accidental or intentional) that can cause different parts of the network to have different views of link-state and topology are not only inconvenient: they give attackers an opportunity to exploit differences in client knowledge. We also worry about attacks to deceive a client about the router membership list, topology, or current network state. Such *partitioning attacks* on client knowledge help an adversary to efficiently deploy resources against a target [13].

Tor uses a small group of redundant, well-known onion routers to track changes in network topology and node state, including keys and exit policies. Each such *directory server* acts as an HTTP server, so clients can fetch current network state and router lists, and so other ORs can upload state information.

A small number of partially trusted directory servers (nine as of late 2012) are "directory authorities." Onion routers periodically publish signed statements of their state to each directory authority. The directory servers combine this information with their own views of network liveness, and periodically collaborate to vote on a description (a consensus *directory*) of the entire network state, signed by as many of the authorities as possible. Client software is pre-loaded with a list of the directory authorities and their public keys, to bootstrap each client's view of the network.

When a directory authority receives a signed statement for an OR, it does not advertise the node as running until it tested that it correctly responds to direct and anonymous circuit creation attempts. The number of nodes that can run with a single IP address is limited, and authority administrators try to keep a lookout for nodes that appear to be configured too similarly or running all on the same subnet. Other than that, the authority subsystem takes no action to prevent Sybil attacks [20]. Previous designs had declared that authority operators should hand-approve each new node, but this system proved ineffective in practice.

To avoid centralizing trust in any single authority, clients will not use a consensus document unless it has been signed by a threshold (half, rounded up) of the authorities that the client recognizes. To prevent rollback attacks, each consensus document has a range of times in which it's valid, and clients don't use a consensus which have been invalid for too long.

Requiring a consensus view of the network prevents individual directory authorities from mounting a variety of attacks: if clients trusted a single directory authority, then an attacker who controlled that server can track clients by providing each client different information—perhaps by listing only nodes under its control, or by informing only certain clients about a given node. Even an external adversary could exploit differences in client knowledge: clients who use a node listed by one authority server but not another are distinguishable, and hence vulnerable.

The directory authorities use a voting algorithm chosen more for simplicity of implementation than for byzantine fault tolerance. At an interval before a vote is to be taken, every authority floods the others with a signed vote document containing its view of the composition of the network and the status of all routers in it. In the next interval, each authority asks all the other authorities for votes from any authority it didn't receive a vote from. Then, each authorities follows a well-specified voting algorithm such that, if each has the same set of votes, each will produce the same consensus as an output. Finally, they sign this consensus document, and collect signatures from every authority that signed the same consensus.

This voting system is not robust to ill-timed authority failures, ill-behaved authorities giving their peers different votes, authorities who disagree about the composition of the set of authorities, and similar issues. In practice, we handle accidental failures in directory authority operation by setting consensus validity intervals so that an occasional day or two of missing consensus votes doesn't hurt the network, and by keeping in touch with the authority operators, who try to keep the number of running authorities well above the threshold. We have not yet needed to deal with a hostile or compromised authority: our design restricts the damage that such an authority could do to casting a maliciously designed vote, or preventing the vote from occurring. In the event of such a denial of service from a hostile authority, it would be sufficient to detect the authority's malfeasance, and remove it from the authority set.

Authorities' long-term private keys are kept offline. Rather than signing documents with them directly, authorities use them to sign certificates containing shorter-term 'signing keys' that they keep online and use for signing documents.

To avoid excessive load on the directory authorities, clients do not contact them directly except when bootstrapping. Instead, most Tor servers act as "directory caches," and periodically fetch network consensus documents; clients can contact a cache instead, once they know who the caches are.

6.4 The Tor controller protocol

Tor has always had a minimalist user interface—it can be configured on the command line or a configuration file and sends output to a log file. This was fine for advanced users, but most users will prefer a GUI. Building a GUI into Tor would be difficult, and would force certain choices (e.g. GUI toolkit) to be made which might not suit all users and all platforms. There-

fore Tor includes an interface for other programs to communicate with the Tor daemon, extracting information to display on the GUI and changing the Tor configuration based on user actions. This interface is an ASCII-based protocol, implemented over a local socket, to allow another program to control Tor.

The control protocol has also proven useful to researchers experimenting with Tor. Initially the functionality exposed in the control protocol was simply that exposed by the configuration file and log files. Providing status information in a specified and machine-readable format made the task of monitoring and controlling Tor easier. Later, functionality was added to the control protocol which should not be exposed to ordinary Tor users but is useful to researchers, such as allowing controllers to arbitrarily control the path selection process.

To prevent arbitrary local processes from changing Tor's configuration to make it less secure, the control protocol provides authentication mechanisms so that only authorized local processes (ones that can read an appropriate file on the filesystem, or that know an appropriate passord) can connect to the controller port.

7 Attacks and Defenses

Below we summarize a variety of attacks, and discuss how well our design withstands them.

Passive attacks

Observing user traffic patterns. Observing a user's connection will not reveal her destination or data, but it will reveal traffic patterns (both sent and received). Profiling via user connection patterns requires further processing, because multiple application streams may be operating simultaneously or in series over a single circuit.

Observing user content. While content at the user end is encrypted, connections to responders may not be (indeed, the responding website itself may be hostile). While filtering content is not a primary goal of Onion Routing, Tor can directly use Privoxy and related filtering services to anonymize application data streams.

Option distinguishability. We allow clients to choose configuration options. For example, clients concerned about request linkability should rotate circuits more often than those concerned about traceability. Allowing choice may attract users with different needs; but clients who are in the minority may lose more anonymity by appearing distinct than they gain by optimizing their behavior [1].

End-to-end timing correlation. Tor only minimally hides such correlations. An attacker watching patterns of traffic at the initiator and the responder will be able to confirm the correspondence with high probability. The greatest protection currently available against such confirmation is to hide the connection between the onion proxy and the first Tor node, by running the OP on the Tor node or behind a firewall. This approach requires an observer to separate traffic originating at the onion router from traffic passing through it: a global observer can do this, but it might be beyond a limited observer's capabilities.

End-to-end size correlation. Simple packet counting will also be effective in confirming endpoints of a stream. However, even without padding, we may have some limited protection: the leaky pipe topology means different numbers of packets may enter one end of a circuit than exit at the other.

Website fingerprinting. All the effective passive attacks above are traffic confirmation attacks, which puts them outside our design goals. There is also a passive traffic analysis attack that is potentially effective. Rather than searching exit connections for timing and volume correlations, the adversary may build up a database of "fingerprints" containing file sizes and access patterns for targeted websites. He can later confirm a user's connection to a given site simply by consulting the database. This attack has been shown to be effective against SafeWeb [27]. It may be less effective against Tor, since streams are multiplexed within the same circuit, and fingerprinting will be limited to the granularity of cells (currently 512 bytes). Additional defenses could include larger cell sizes, padding schemes to group websites into large sets, and link padding or long-range dummies.[8]

Active attacks

Compromise keys. An attacker who learns the TLS session key can see control cells and encrypted relay cells on every circuit on that connection; learning a circuit session key lets him unwrap one layer of the encryption. An attacker who learns an OR's TLS private key can impersonate that OR for the TLS key's lifetime, but he must also learn the onion key to decrypt *create* cells (and because of perfect forward secrecy, he cannot hijack already established circuits without also compromising their session keys). Periodic key rotation limits the window of opportunity for these attacks. On the other hand, an attacker who learns a node's identity key can replace that node indefinitely by sending new forged descriptors to the directory servers.

Iterated compromise. A roving adversary who can compromise ORs (by system intrusion, legal coercion, or extralegal coercion) could march down the circuit compromising the nodes until he reaches the end. Unless the adversary can complete this attack within the lifetime of the circuit, however, the ORs will have discarded the necessary information before the attack can be completed. (Thanks to the perfect forward secrecy of session keys, the attacker cannot force nodes to decrypt recorded traffic once the circuits have been closed.) Additionally, building circuits that cross jurisdictions can make

[8] Note that this fingerprinting attack should not be confused with the much more complicated latency attacks of [5], which require a fingerprint of the latencies of all circuits through the network, combined with those from the network edges to the target user and the responder website.

legal coercion harder—this phenomenon is commonly called "jurisdictional arbitrage." The Java Anon Proxy project recently experienced the need for this approach, when a German court forced them to add a backdoor to their nodes [48].

Run a recipient. An adversary running a webserver trivially learns the timing patterns of users connecting to it, and can introduce arbitrary patterns in its responses. End-to-end attacks become easier: if the adversary can induce users to connect to his webserver (perhaps by advertising content targeted to those users), he now holds one end of their connection. There is also a danger that application protocols and associated programs can be induced to reveal information about the initiator. Tor depends on Privoxy and similar protocol cleaners to solve this latter problem.

Run an onion proxy. It is expected that end users will nearly always run their own local onion proxy. However, in some settings, it may be necessary for the proxy to run remotely—typically, in institutions that want to monitor the activity of those connecting to the proxy. Compromising an onion proxy compromises all future connections through it.

DoS non-observed nodes. An observer who can only watch some of the Tor network can increase the value of this traffic by attacking non-observed nodes to shut them down, reduce their reliability, or persuade users that they are not trustworthy. The best defense here is robustness.

Run a hostile OR. In addition to being a local observer, an isolated hostile node can create circuits through itself, or alter traffic patterns to affect traffic at other nodes. Nonetheless, a hostile node must be immediately adjacent to both endpoints to compromise the anonymity of a circuit. If an adversary can run multiple ORs, and can persuade the directory servers that those ORs are trustworthy and independent, then occasionally some user will choose one of those ORs for the start and another as the end of a circuit. If an adversary controls m bandwidth out of the total network bandwidth N, he can correlate approximately $\left(\frac{m}{N}\right)^2$ of the circuits—although an adversary could still attract a disproportionately large amount of traffic by running an OR with a permissive exit policy, or by degrading the reliability of other routers.

If the path chosen for each circuit was chosen independently of the paths chosen for previous circuits, the probability that each circuit will be compromised would be equal. Therefore, even if there is only a small probability that any individual circuit is compromised, if a user creates many circuits, over time the probability that at least one circuit will be compromised could be quite large. Since compromising only a fraction of a user's circuits (perhaps even just one) will likely be enough to compromise a user's anonymity, the security offered by such a system could be quite poor.

Therefore Tor adopts "guard nodes" (sometimes called "helper nodes") where each user choses a few nodes to act as that user's entry point to the network, and keeps this selection the same for as long as enough of the chosen guards remain operational. This approach doesn't affect the probability that a user's first circuit will be compromised, but if a user has chosen honest guards their future circuits will remain safe. A user which has chosen a dishonest guard however will have a large proportion of their circuits compromised. An adversary still is able to compromise $\left(\frac{m}{N}\right)^2$ of the circuits but the use of guard nodes concentrates these compromised circuits over a small group of users. Since we assume that compromising a few circuits is only marginally better for user security than compromising them all, guard nodes improves the average security of the network.

Introduce timing into messages. This is simply a stronger version of passive timing attacks already discussed earlier.

Tagging attacks. A hostile node could "tag" a cell by altering it. If the stream were, for example, an unencrypted request to a Web site, the garbled content coming out at the appropriate time would confirm the association. However, integrity checks on cells prevent this attack.

Replace contents of unauthenticated protocols. When relaying an unauthenticated protocol like HTTP, a hostile exit node can impersonate the target server. Clients should prefer protocols with end-to-end authentication.

Replay attacks. Some anonymity protocols are vulnerable to replay attacks. Tor is not; replaying one side of a handshake will result in a different negotiated session key, and so the rest of the recorded session can't be used.

Smear attacks. An attacker could use the Tor network for socially disapproved acts, to bring the network into disrepute and get its operators to shut it down. Exit policies reduce the possibilities for abuse, but ultimately the network requires volunteers who can tolerate some political heat.

Distribute hostile code. An attacker could trick users into running subverted Tor software that did not, in fact, anonymize their connections—or worse, could trick ORs into running weakened software that provided users with less anonymity. We address this problem (but do not solve it completely) by signing all Tor releases with an official public key, and including an entry in the directory that lists which versions are currently believed to be secure. To prevent an attacker from subverting the official release itself (through threats, bribery, or insider attacks), we provide all releases in source code form, encourage source audits, and frequently warn our users never to trust any software (even from us) that comes without source.

Block access to the network. An attacker who controls a user's Internet connection can block access to the Tor network by blocking connections to the directory authorities and/or Tor nodes. The IP addresses of the former are embedded in every copy of Tor and the IP addresses of the latter can be easily found by asking the directory authorities. Tor resists this attack by having an additional type of OR – the "bridge node" which is distinguished from other ORs by not having its IP address included in the directory. Operators of bridge nodes publish their IP address to a single bridge authority which

distributes IP addresses to users in a way to resist an attacker being able to enumerate (and thus block) them all. Currently bridge IP addresses are made available on a website (where requests from the same source IP address always get the same answer) and via email (where requests from the same email address always get the same answer). Bridge IP addresses are also distributed by personal contacts.

Bridges resist blocking access to the Tor network by IP address, but do not prevent an attacker blocking by protocol fingerprint. Tor's use of TLS is designed to provide some resistance against this attack, through impersonating HTTPS, but due to efficiency and simplicity considerations, it does not give perfect protection. Steganographic transports (e.g. embedding data in images) would improve resistance to fingerprinting but at a high cost to effeciency so would not be appropriate for all users. Also, users in some countries may need to disguise their traffic as different protocols due to particular policies in place. Therefore the protocol-fingerprinting-resistance part of Tor has been left the responsibility of an external "pluggable transport" program, which is responsible for obfuscating Tor's TLS traffic at the OP end, and converting it back to TLS at the bridge. Since the pluggable transport operates on TLS ciphertext, which would otherwise be sent directly over the network, it can't harm the security properties Tor provides, and so Tor users can accept pluggable transports written by third-parties, as long as they are confident the software is not malicious.

Directory attacks

Destroy directory servers. If a few directory servers disappear, the others still decide on a valid directory. So long as any directory servers remain in operation, they will still broadcast their views of the network and generate a consensus directory. (If more than half are destroyed, this directory will not, however, have enough signatures for clients to use it automatically; human intervention will be necessary for clients to decide whether to trust the resulting directory.)

Subvert a directory server. By taking over a directory server, an attacker can partially influence the final directory. Since ORs are included or excluded by majority vote, the corrupt directory can at worst cast a tie-breaking vote to decide whether to include marginal ORs. It remains to be seen how often such marginal cases occur in practice.

Subvert a majority of directory servers. An adversary who controls more than half the directory servers can include as many compromised ORs in the final directory as he wishes. We must ensure that directory server operators are independent and attack-resistant.

Encourage directory server dissent. The directory agreement protocol assumes that directory server operators agree on the set of directory servers. An adversary who can persuade some of the directory server operators to distrust one another could split the quorum into mutually hostile camps, thus partitioning users based on which directory they use. Tor does not address this attack.

Trick the directory servers into listing a hostile OR. Our threat model explicitly assumes directory server operators will be able to filter out most hostile ORs.

Convince the directories that a malfunctioning OR is working. In the current Tor implementation, directory servers assume that an OR is running correctly if they can start a TLS connection to it. A hostile OR could easily subvert this test by accepting TLS connections from ORs but ignoring all cells. Directory servers must actively test ORs by building circuits and streams as appropriate. The tradeoffs of a similar approach are discussed in [16].

Attacks against rendezvous points

Make many introduction requests. An attacker could try to deny Bob service by flooding his introduction points with requests. Because the introduction points can block requests that lack authorization tokens, however, Bob can restrict the volume of requests he receives, or require a certain amount of computation for every request he receives.

Attack an introduction point. An attacker could disrupt a location-hidden service by disabling its introduction points. But because a service's identity is attached to its public key, the service can simply re-advertise itself at a different introduction point. Advertisements can also be done secretly so that only high-priority clients know the address of Bob's introduction points or so that different clients know of different introduction points. This forces the attacker to disable all possible introduction points.

Compromise an introduction point. An attacker who controls Bob's introduction point can flood Bob with introduction requests, or prevent valid introduction requests from reaching him. Bob can notice a flood, and close the circuit. To notice blocking of valid requests, however, he should periodically test the introduction point by sending rendezvous requests and making sure he receives them.

Compromise a rendezvous point. A rendezvous point is no more sensitive than any other OR on a circuit, since all data passing through the rendezvous is encrypted with a session key shared by Alice and Bob.

8 Early experiences: Tor in the Wild

As of mid-May 2004, the Tor network consists of 32 nodes (24 in the US, 8 in Europe), and more are joining each week as the code matures. (For comparison, the current remailer network has about 40 nodes.) Each node has at least a 768Kb/768Kb connection, and many have 10Mb. The number of users varies (and of course, it's hard to tell for sure), but we sometimes have several hundred users—administrators at several companies have begun sending their entire departments' web traffic through Tor, to block other divisions of their company from reading their traffic. Tor users have reported using the network for web browsing, FTP, IRC, AIM, Kazaa, SSH, and recipient-anonymous email via rendezvous

points. One user has anonymously set up a Wiki as a hidden service, where other users anonymously publish the addresses of their hidden services.

Each Tor node currently processes roughly 800,000 relay cells (a bit under half a gigabyte) per week. On average, about 80% of each 498-byte payload is full for cells going back to the client, whereas about 40% is full for cells coming from the client. (The difference arises because most of the network's traffic is web browsing.) Interactive traffic like SSH brings down the average a lot—once we have more experience, and assuming we can resolve the anonymity issues, we may partition traffic into two relay cell sizes: one to handle bulk traffic and one for interactive traffic.

Based in part on our restrictive default exit policy (we reject SMTP requests) and our low profile, we have had no abuse issues since the network was deployed in October 2003. Our slow growth rate gives us time to add features, resolve bugs, and get a feel for what users actually want from an anonymity system. Even though having more users would bolster our anonymity sets, we are not eager to attract the Kazaa or warez communities—we feel that we must build a reputation for privacy, human rights, research, and other socially laudable activities.

As for performance, profiling shows that Tor spends almost all its CPU time in AES, which is fast. Current latency is attributable to two factors. First, network latency is critical: we are intentionally bouncing traffic around the world several times. Second, our end-to-end congestion control algorithm focuses on protecting volunteer servers from accidental DoS rather than on optimizing performance. To quantify these effects, we did some informal tests using a network of 4 nodes on the same machine (a heavily loaded 1GHz Athlon). We downloaded a 60 megabyte file from `debian.org` every 30 minutes for 54 hours (108 sample points). It arrived in about 300 seconds on average, compared to 210s for a direct download. We ran a similar test on the production Tor network, fetching the front page of `cnn.com` (55 kilobytes): while a direct download consistently took about 0.3s, the performance through Tor varied. Some downloads were as fast as 0.4s, with a median at 2.8s, and 90% finishing within 5.3s. It seems that as the network expands, the chance of building a slow circuit (one that includes a slow or heavily loaded node or link) is increasing. On the other hand, as our users remain satisfied with this increased latency, we can address our performance incrementally as we proceed with development.

Although Tor's clique topology and full-visibility directories present scaling problems, we still expect the network to support a few hundred nodes and maybe 10,000 users before we're forced to become more distributed. With luck, the experience we gain running the current topology will help us choose among alternatives when the time comes.

9 Open Questions in Low-latency Anonymity

In addition to the non-goals in Section 3, many questions must be solved before we can be confident of Tor's security.

Many of these open issues are questions of balance. For example, how often should users rotate to fresh circuits? Frequent rotation is inefficient, expensive, and may lead to intersection attacks and predecessor attacks [51], but infrequent rotation makes the user's traffic linkable. Besides opening fresh circuits, clients can also exit from the middle of the circuit, or truncate and re-extend the circuit. More analysis is needed to determine the proper tradeoff.

How should we choose path lengths? If Alice always uses two hops, then both ORs can be certain that by colluding they will learn about Alice and Bob. In our current approach, Alice always chooses at least three nodes unrelated to herself and her destination. Should Alice choose a random path length (e.g. from a geometric distribution) to foil an attacker who uses timing to learn that he is the fifth hop and thus concludes that both Alice and the responder are running ORs?

Throughout this paper, we have assumed that end-to-end traffic confirmation will immediately and automatically defeat a low-latency anonymity system. Even high-latency anonymity systems can be vulnerable to end-to-end traffic confirmation, if the traffic volumes are high enough, and if users' habits are sufficiently distinct [12, 29]. Can anything be done to make low-latency systems resist these attacks as well as high-latency systems? Tor already makes some effort to conceal the starts and ends of streams by wrapping long-range control commands in identical-looking relay cells. Link padding could frustrate passive observers who count packets; long-range padding could work against observers who own the first hop in a circuit. But more research remains to find an efficient and practical approach. Volunteers prefer not to run constant-bandwidth padding; but no convincing traffic shaping approach has been specified. Recent work on long-range padding [31] shows promise. One could also try to reduce correlation in packet timing by batching and reordering packets, but it is unclear whether this could improve anonymity without introducing so much latency as to render the network unusable.

A cascade topology may better defend against traffic confirmation by aggregating users, and making padding and mixing more affordable. Does the hydra topology (many input nodes, few output nodes) work better against some adversaries? Are we going to get a hydra anyway because most nodes will be middleman nodes?

Common wisdom suggests that Alice should run her own OR for best anonymity, because traffic coming from her node could plausibly have come from elsewhere. How much mixing does this approach need? Is it immediately beneficial because of real-world adversaries that can't observe Alice's router, but can run routers of their own?

To scale to many users, and to prevent an attacker from

observing the whole network, it may be necessary to support far more servers than Tor currently anticipates. This introduces several issues. First, if approval by a central set of directory servers is no longer feasible, what mechanism should be used to prevent adversaries from signing up many colluding servers? Second, if clients can no longer have a complete picture of the network, how can they perform discovery while preventing attackers from manipulating or exploiting gaps in their knowledge? Third, if there are too many servers for every server to constantly communicate with every other, which non-clique topology should the network use? (Restricted-route topologies promise comparable anonymity with better scalability [11], but whatever topology we choose, we need some way to keep attackers from manipulating their position within it [19].) Fourth, if no central authority is tracking server reliability, how do we stop unreliable servers from making the network unusable? Fifth, do clients receive so much anonymity from running their own ORs that we should expect them all to do so [1], or do we need another incentive structure to motivate them? Tarzan and MorphMix present possible solutions.

When a Tor node goes down, all its circuits (and thus streams) must break. Will users abandon the system because of this brittleness? How well does the method in Section 6.1 allow streams to survive node failure? If affected users rebuild circuits immediately, how much anonymity is lost? It seems the problem is even worse in a peer-to-peer environment—such systems don't yet provide an incentive for peers to stay connected when they're done retrieving content, so we would expect a higher churn rate.

10 Future Directions

Tor brings together many innovations into a unified deployable system. The next immediate steps include:

Scalability: Tor's emphasis on deployability and design simplicity has led us to adopt a clique topology, semi-centralized directories, and a full-network-visibility model for client knowledge. These properties will not scale past a few hundred servers. Section 9 describes some promising approaches, but more deployment experience will be helpful in learning the relative importance of these bottlenecks.

Bandwidth classes: This paper assumes that all ORs have good bandwidth and latency. We should instead adopt the MorphMix model, where nodes advertise their bandwidth level (DSL, T1, T3), and Alice avoids bottlenecks by choosing nodes that match or exceed her bandwidth. In this way DSL users can usefully join the Tor network.

Incentives: Volunteers who run nodes are rewarded with publicity and possibly better anonymity [1]. More nodes means increased scalability, and more users can mean more anonymity. We need to continue examining the incentive structures for participating in Tor. Further, we need to explore more approaches to limiting abuse, and understand why most people don't bother using privacy systems.

Cover traffic: Currently Tor omits cover traffic—its costs in performance and bandwidth are clear but its security benefits are not well understood. We must pursue more research on link-level cover traffic and long-range cover traffic to determine whether some simple padding method offers provable protection against our chosen adversary.

Caching at exit nodes: Perhaps each exit node should run a caching web proxy [44], to improve anonymity for cached pages (Alice's request never leaves the Tor network), to improve speed, and to reduce bandwidth cost. On the other hand, forward security is weakened because caches constitute a record of retrieved files. We must find the right balance between usability and security.

Better directory distribution: Clients currently download a description of the entire network every 15 minutes. As the state grows larger and clients more numerous, we may need a solution in which clients receive incremental updates to directory state. More generally, we must find more scalable yet practical ways to distribute up-to-date snapshots of network status without introducing new attacks.

Further specification review: Our public byte-level specification [18] needs external review. We hope that as Tor is deployed, more people will examine its specification.

Multisystem interoperability: We are currently working with the designer of MorphMix to unify the specification and implementation of the common elements of our two systems. So far, this seems to be relatively straightforward. Interoperability will allow testing and direct comparison of the two designs for trust and scalability.

Wider-scale deployment: The original goal of Tor was to gain experience in deploying an anonymizing overlay network, and learn from having actual users. We are now at a point in design and development where we can start deploying a wider network. Once we have many actual users, we will doubtlessly be better able to evaluate some of our design decisions, including our robustness/latency tradeoffs, our performance tradeoffs (including cell size), our abuse-prevention mechanisms, and our overall usability.

Acknowledgments

We thank Peter Palfrader, Geoff Goodell, Adam Shostack, Joseph Sokol-Margolis, John Bashinski, and Zack Brown for editing and comments; Matej Pfajfar, Andrei Serjantov, Marc Rennhard for design discussions; Bram Cohen for congestion control discussions; Adam Back for suggesting telescoping circuits; and Cathy Meadows for formal analysis of the *extend* protocol. This work has been supported by ONR and DARPA.

References

[1] A. Acquisti, R. Dingledine, and P. Syverson. On the economics of anonymity. In R. N. Wright, editor, *Financial Cryptography*. Springer-Verlag, LNCS 2742, 2003.

[2] R. Anderson. The eternity service. In *Pragocrypt '96*, 1996.

[3] The Anonymizer. <http://anonymizer.com/>.

[4] A. Back, I. Goldberg, and A. Shostack. Freedom systems 2.1 security issues and analysis. White paper, Zero Knowledge Systems, Inc., May 2001.

[5] A. Back, U. Möller, and A. Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. In I. S. Moskowitz, editor, *Information Hiding (IH 2001)*, pages 245–257. Springer-Verlag, LNCS 2137, 2001.

[6] O. Berthold, H. Federrath, and S. Köpsell. Web MIXes: A system for anonymous and unobservable Internet access. In H. Federrath, editor, *Designing Privacy Enhancing Technologies: Workshop on Design Issue in Anonymity and Unobservability*. Springer-Verlag, LNCS 2009, 2000.

[7] P. Boucher, A. Shostack, and I. Goldberg. Freedom systems 2.0 architecture. White paper, Zero Knowledge Systems, Inc., December 2000.

[8] Z. Brown. Cebolla: Pragmatic IP Anonymity. In *Ottawa Linux Symposium*, June 2002.

[9] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudo-nyms. *Communications of the ACM*, 4(2), February 1981.

[10] W. Dai. Pipenet 1.1. Usenet post, August 1996. <http://www.eskimo.com/~weidai/pipenet.txt> First mentioned in a post to the cypherpunks list, Feb. 1995.

[11] G. Danezis. Mix-networks with restricted routes. In R. Dingledine, editor, *Privacy Enhancing Technologies (PET 2003)*. Springer-Verlag LNCS 2760, 2003.

[12] G. Danezis. Statistical disclosure attacks. In *Security and Privacy in the Age of Uncertainty (SEC2003)*, pages 421–426, Athens, May 2003. IFIP TC11, Kluwer.

[13] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *2003 IEEE Symposium on Security and Privacy*, pages 2–15. IEEE CS, May 2003.

[14] D. Dean and A. Stubblefield. Using Client Puzzles to Protect TLS. In *Proceedings of the 10th USENIX Security Symposium*. USENIX, Aug. 2001.

[15] T. Dierks and C. Allen. The TLS Protocol — Version 1.0. IETF RFC 2246, January 1999.

[16] R. Dingledine, M. J. Freedman, D. Hopwood, and D. Molnar. A Reputation System to Increase MIX-net Reliability. In I. S. Moskowitz, editor, *Information Hiding (IH 2001)*, pages 126–141. Springer-Verlag, LNCS 2137, 2001.

[17] R. Dingledine, M. J. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. In H. Federrath, editor, *Designing Privacy Enhancing Technologies: Workshop on Design Issue in Anonymity and Unobservability*. Springer-Verlag, LNCS 2009, July 2000.

[18] R. Dingledine and N. Mathewson. Tor protocol specifications. <https://www.torproject.org/svn/trunk/doc/tor-spec.txt>.

[19] R. Dingledine and P. Syverson. Reliable MIX Cascade Networks through Reputation. In M. Blaze, editor, *Financial Cryptography*. Springer-Verlag, LNCS 2357, 2002.

[20] J. Douceur. The Sybil Attack. In *Proceedings of the 1st International Peer To Peer Systems Workshop (IPTPS)*, Mar. 2002.

[21] H. Federrath, A. Jerichow, and A. Pfitzmann. MIXes in mobile communication systems: Location management with privacy. In R. Anderson, editor, *Information Hiding, First International Workshop*, pages 121–135. Springer-Verlag, LNCS 1174, May 1996.

[22] M. J. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *9th ACM Conference on Computer and Communications Security (CCS 2002)*, Washington, DC, November 2002.

[23] S. Goel, M. Robson, M. Polte, and E. G. Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Technical Report TR2003-1890, Cornell University Computing and Information Science, February 2003.

[24] I. Goldberg. *A Pseudonymous Communications Infrastructure for the Internet*. PhD thesis, UC Berkeley, Dec 2000.

[25] D. M. Goldschlag, M. G. Reed, and P. F. Syverson. Hiding routing information. In R. Anderson, editor, *Information Hiding, First International Workshop*, pages 137–150. Springer-Verlag, LNCS 1174, May 1996.

[26] C. Gülcü and G. Tsudik. Mixing E-mail with Babel. In *Network and Distributed Security Symposium (NDSS 96)*, pages 2–16. IEEE, February 1996.

[27] A. Hintz. Fingerprinting websites using traffic analysis. In R. Dingledine and P. Syverson, editors, *Privacy Enhancing Technologies (PET 2002)*, pages 171–178. Springer-Verlag, LNCS 2482, 2002.

[28] A. Jerichow, J. Müller, A. Pfitzmann, B. Pfitzmann, and M. Waidner. Real-time mixes: A bandwidth-efficient anonymity protocol. *IEEE Journal on Selected Areas in Communications*, 16(4):495–509, May 1998.

[29] D. Kesdogan, D. Agrawal, and S. Penz. Limits of anonymity in open environments. In F. Petitcolas, editor, *Information Hiding Workshop (IH 2002)*. Springer-Verlag, LNCS 2578, October 2002.

[30] D. Koblas and M. R. Koblas. SOCKS. In *UNIX Security III Symposium (1992 USENIX Security Symposium)*, pages 77–83. USENIX, 1992.

[31] B. N. Levine, M. K. Reiter, C. Wang, and M. Wright. Timing analysis in low-latency mix-based systems. In A. Juels, editor, *Financial Cryptography*. Springer-Verlag, LNCS (forthcoming), 2004.

[32] B. N. Levine and C. Shields. Hordes: A multicast-based protocol for anonymity. *Journal of Computer Security*, 10(3):213–240, 2002.

[33] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[34] U. Möller, L. Cottrell, P. Palfrader, and L. Sassaman. Mixmaster Protocol — Version 2. Draft, July 2003. <http://www.abditum.com/mixmaster-spec.txt>.

[35] V. S. Pai, L. Wang, K. Park, R. Pang, and L. Peterson. The Dark Side of the Web: An Open Proxy's View. <http://codeen.cs.princeton.edu/>.

[36] A. Pfitzmann, B. Pfitzmann, and M. Waidner. ISDN-mixes: Untraceable communication with very small bandwidth overhead. In *GI/ITG Conference on Communication in Distributed Systems*, pages 451–463, February 1991.

[37] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Protocols using anonymous connections: Mobile applications. In B. Christianson, B. Crispo, M. Lomas, and M. Roe, editors, *Security Protocols: 5th International Workshop*, pages 13–23. Springer-Verlag, LNCS 1361, April 1997.

[38] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, May 1998.

[39] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for web transactions. *ACM TISSEC*, 1(1):66–92, June 1998.

[40] M. Rennhard and B. Plattner. Practical anonymity for the masses with morphmix. In A. Juels, editor, *Financial Cryptography*. Springer-Verlag, LNCS (forthcoming), 2004.

[41] M. Rennhard, S. Rafaeli, L. Mathy, B. Plattner, and D. Hutchison. Analysis of an Anonymity Network for Web Browsing. In *IEEE 7th Intl. Workshop on Enterprise Security (WET ICE 2002)*, Pittsburgh, USA, June 2002.

[42] A. Serjantov and P. Sewell. Passive attack analysis for connection-based anonymity systems. In *Computer Security – ESORICS 2003*. Springer-Verlag, LNCS 2808, October 2003.

[43] R. Sherwood, B. Bhattacharjee, and A. Srinivasan. p^5: A protocol for scalable anonymous communication. In *IEEE Symposium on Security and Privacy*, pages 58–70. IEEE CS, 2002.

[44] A. Shubina and S. Smith. Using caching for browsing anonymity. *ACM SIGEcom Exchanges*, 4(2), Sept 2003. <http://www.acm.org/sigs/sigecom/exchanges/volume_4_(03)/4.2-Shubina.pdf>.

[45] P. Syverson, M. Reed, and D. Goldschlag. Onion Routing access configurations. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, volume 1, pages 34–40. IEEE CS Press, 2000.

[46] P. Syverson, G. Tsudik, M. Reed, and C. Landwehr. Towards an Analysis of Onion Routing Security. In H. Federrath, editor, *Designing Privacy Enhancing Technologies: Workshop on Design Issue in Anonymity and Unobservability*, pages 96–114. Springer-Verlag, LNCS 2009, July 2000.

[47] A. Tannenbaum. Computer networks, 1996.

[48] The AN.ON Project. German police proceeds against anonymity service. Press release, September 2003. <http://www.datenschutzzentrum.de/material/themen/presse/anon-bka_e.htm>.

[49] M. Waldman and D. Mazières. Tangler: A censorship-resistant publishing system based on document entanglements. In 8^{th} *ACM Conference on Computer and Communications Security (CCS-8)*, pages 86–135. ACM Press, 2001.

[50] M. Waldman, A. Rubin, and L. Cranor. Publius: A robust, tamper-evident, censorship-resistant and source-anonymous web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.

[51] M. Wright, M. Adler, B. N. Levine, and C. Shields. Defending anonymous communication against passive logging attacks. In *IEEE Symposium on Security and Privacy*, pages 28–41. IEEE CS, May 2003.