Acknowledgements

First and foremost, I'd like to thank my thesis advisors, Daniel Krizanc and Norman Danner, who were invaluable in all phases of this project. Roger Dingledine was also consistently friendly and helpful in answering my questions about Tor's protocol on IRC. I would also like to extend my thanks to the following projects for creating the tools that made this research possible: User-mode Linux, for making our simulation method possible, the GNU Project, for providing so many essential standard tools, and the Debian project, for creating a lightweight and usable operating system that made an excellent base for all our virtual machines.

Finally, I want to thank caffeine, for getting me through so many sleepless nights, and my friends and family, for making the days *after* the sleepless nights bearable.

Contents

CHAPTER 1

Introduction

1. Anonymity-providing Systems

In general, when a client accesses any server over the Internet, a great deal of information is trivially obtainable. In particular, the sender's IP address is included in every packet of data they transmit, and this information can be used to determine their Internet Service Provider and in many cases their approximate geographic location. In addition, each packet contains the IP address of its destination. Anonymous networks provide a way of preventing outside observers from determining that an initiator and a responder are, in fact, communicating.

The most common way of achieving this anonymity is by routing the connection through a series of intermediate proxy servers. The goal is that the recipient will only be able to uncover the address of the proxy server that directly contacted him, allowing the identity of the original sender to remain hidden. Along the way, the intermediate nodes in the route may *mix* (send packets from many users out in an different order than they were received) or delay packets in order to frustrate attempts at uncovering the sender through traffic analysis. Such networks are known as *mix-nets*, and were first described by Chaum [11].

In many cases, however, it is not feasible to employ mixing or delaying of packets; for most interactive applications, such as web browsing, SSH sessions, or live chat, too much mixing or delaying will result in a poor or unusable experience for the user. A web browsing session, would quickly become frustrating if it took five minutes for each page to load. It seems, then, that there is a trade-off between anonymity and speed.[1] This thesis examines what threats are actually posed by traffic analysis against low-latency mix-nets and how practical it is to implement them on a real network, using Tor as our case study.

2. Anonymity Networks

Many different designs have been proposed to allow users to achieve anonymity. Although we will primarily treat Tor, a brief overview of the systems available may be useful.

- **The Anonymizer** [1] and other anonymous proxy services offer anonymity by allowing a user to connect to remote web sites through a single proxy server, which strips out identifying information and then forwards the request on to the destination. While this approach is quite fast and easy to understand, users must trust the operator of the proxy server not to record information about the identities of clients. It also creates a single point of failure—an attacker can focus his efforts on compromising the proxy and, if successful, establish the identities of all clients using the proxy.
- **Java Anon Proxy (JAP)** [5] uses static routes called *cascades*, made up of several mixes with encryption between each mix. Many users share the same route, which makes it harder to distinguish any one user by observing network traffic along the cascade. Currently, there are four cascades, but only one (Dresden-Dresden) receives a significant amount of traffic.[2]

[1]A much more in-depth analysis of this notion of trading strong anonymity for efficiency is given in Back et al. [10].
[2]Source: `http://anon.inf.tu-dresden.de/status.php`, accessed 4/4/2006.

- **Mixminion** [12] is an anonymous remailing system that allows users to send messages anonymously. Because e-mail does not require a high level of interactivity, Mixminion can afford to add large amounts of delay as well as reorder messages. This allows it to protect anonymity against extremely powerful adversaries. These strategies, however, are unsuitable for applications which require low latency, such as web browsing and Internet Relay Chat.
- Peer-to-peer systems such as **Tarzan** [14] and **MorphMix** [19] have all participants generate cover traffic (fake traffic intended to prevent an outside observer from compromising anonymity by the numbers and sizes of packets at a source and destination) as well as relay traffic for other users.
- **Tor** uses layered encryption to ensure that each intermediate proxy only knows its predecessor and successor in a path. It does not use any mixing or delaying, which makes it fast and responsive for interactive traffic.

3. Why Tor?

Tor was chosen over the other networks mentioned for several reasons. First, it is one of the most popular anonymizing networks currently in use: there are over 450 Tor nodes currently running[3] and it was estimated to be serving over 125,000 clients as of January 2006.[4] Tor is also free and open-source, and can be set up on a private network, which made simulations using the real Tor software possible. It is well-documented and continues to receive active development.

[3]According to `http://www.noreply.org/tor-running-routers/`, retrieved 4/4/2006.
[4]Roger Dingledine, personal communication. March 8, 2006.

4. Organization

Chapter 2 describes in detail the network protocol Tor uses to establish connections and send data. Chapter 3 lists several known attacks on the anonymity Tor provides and includes for each attack a description of the adversary's goals and the capabilities he needs to achieve those goals. This chapter also discusses the efficiency and feasibility of each attack. In chapter 4, we present an attack of our own based on a method of timing analysis used in Levine et al. [16] (described in detail in Chapter 3, and describe how we tested its effectiveness on a small network of 20 Tor nodes and 60 clients. Finally, chapter 5 describes and analyzes results of the simulations.

CHAPTER 2

The Design of Tor

1. Introduction

Tor, the “second-generation onion router,” is an overlay network that attempts to provide anonymity for its users.[1] Like most other anonymity-providing systems currently in use, it is a mix network [11]: it routes each connection through a series of nodes, each of which should know only the previous and next node in the circuit.

Currently, Tor routes connections through three intermediate nodes before forwarding them on to the destination. There are more than 450 nodes running, and the Tor network is used by at least 125,000 people worldwide.

2. Cells

To make use of Tor, a client connects using an application called the *Onion Proxy*, which selects a set of *Onion Routers*, establishes a path through Tor (called a circuit; the protocol for creating circuits is described below), and sends data. All data sent through Tor is encapsulated into *cells*, which are fixed-length (512 bytes) packets. Cells come in two forms: *control* cells, which are intended to be interpreted directly by the node that receives them, and *relay* cells, which are forwarded from one end of the circuit to the other.

Each cell begins with a two byte *Circuit ID*, which is used by the Onion Router to keep track of the multiple circuits that pass through it. In a control cell, the

[1]The description of Tor here is based almost exclusively on the Tor design document [13], though some clarification on the congestion control mechanisms comes from Murdoch and Danezis [17] and personal communication with Roger Dingledine on IRC.

Circuit ID is followed by a one byte specifying the type of control command, and then 509 bytes of data. Relay cells have an additional header that includes a *Stream ID* (to allow a single circuit to carry multiple TCP streams), a digest (which provides integrity checking), and a relay command. There are 498 bytes available for data in each relay cell.

3. Circuit Establishment

Circuit establishment is performed incrementally. The user's Onion Proxy randomly selects a sequence of nodes to use in the circuit,[2] and then sends a control cell with the command *create* with the first half of the Diffie-Hellman handshake g^x to the initial node in the sequence. The Onion Router will then reply with a control cell of type *created* and place the other half of the DH handshake, g^y, into the data section along with a hash of the symmetric key that has been negotiated. All communications between the Onion Proxy and the Onion Router, as well as connections between Onion Routers, are sent over a TLS-encrypted link.

A circuit can be extended by sending a relay cell with the *relay extend* command destined for the final node in the circuit (the creation and transmission of relay cells is described below) containing the address of the new node to add and the first half of the DH handshake. Upon receiving a *relay extend* message, a node will establish a connection to the specified address (using the same *create*/*created* sequence of messages described earlier), and send a *relay extended* message with the other half of the DH handshake back along the circuit. The client now shares a symmetric key with the new node, and the circuit has been extended.

[2]In fact the selection is not entirely random; the client prefers nodes with good bandwidth and uptime, and the last node in the circuit will be chosen so that its *exit policy* (the type of traffic a node will forward outside the Tor network) permits the traffic that the user wants to send.

4. Circuit Lifetime

According to the Tor technical FAQ [8], Tor will use the same circuit for new TCP streams for ten minutes. It is important to note, however, that once a circuit is built and is transporting one or more TCP streams, it will stay open until all streams it is carrying have closed. This feature is necessary to support many protocols such as SSH, which would be unable to keep track of a session if it switched circuits (and hence exit nodes) during the session.

5. Relaying Messages

Relay cells are created by choosing a recipient on the circuit (usually this is the last node, but it need not be—this property of Tor is called *leaky pipe* circuit topology, since a relay cell need not travel the entire length of the circuit), and successively encrypting the relay header and its payload using the keys negotiated with each node in the circuit, starting with the final node. For example, suppose that Alice shares symmetric keys K_1, K_2, and K_3 with routers N_1, N_2, N_3, and she wants to send a message M, using N_3 as her exit node. She encrypts the message first with K_3, then K_2, and finally with K_1, producing $E(K_1, E(K_2, E(K_3, (H(M), M))))$, which she can then send to N_1 ($H(M)$ here is digest mentioned in Section 2). When an Onion Router receives a relay cell, it looks up the key corresponding to the Circuit ID, decrypts the contents, and checks to see if the digest contained in the cell matches the computed digest of the payload. If it does match, then the Onion Router can accept it and process the message. Otherwise, it simply looks up the next node in the circuit and sends the (decrypted) message on. Only when the message has reached its intended recipient and all layers of encryption have been removed will the digest be correct (since the hash is 48 bits, the chance of an accidental collision is very low).

Sending data destined for the outside world (for example, a web site) is simply a matter of establishing a connection using a relay cell with the command *relay begin* (which will be acknowledged by the exit node with a *relay connected* cell) and then sending data through *relay data* cells. Any TCP stream can be transported in this way; user applications communicate with the Onion Proxy using the SOCKS protocol [15].

6. Tearing Down Circuits

Circuits, once created, can be destroyed in two ways: all at once, or incrementally. To tear down a circuit entirely, the Onion Proxy sends a control cell with the command *destroy* to the first node. Each node closes all connections associated with that circuit and then passes the destroy message on to the next node, until it reaches the final node in the circuit. Alternatively, a user can tear down a connection by sending a *relay truncate* cell to any node on the chain. When a *relay truncate* command is received, the node sends a *destroy* control cell forward, removing all the nodes after it from the circuit, and sends back a *relay truncated* acknowledgement to the client. The circuit can then be extended to new nodes if desired. Tearing down connections incrementally allows a user to change the path through Tor that a connection takes without alerting any of the intermediate nodes that anything has changed. It also provides a way for intermediate nodes to let the user know if the next node in a circuit goes down—the node simply sends a *relay truncated* message back to the initiator.

7. Quality of Service Measures

Several different mechanisms are used by Tor to provide congestion control and the ability for Onion Routers to limit the amount of bandwidth used. Allowing rate limiting is intended to make running an Onion Router more attractive (ORs are run entirely by volunteers). For this type of bandwidth limiting, Tor uses a token bucket algorithm: tokens are added to the bucket at a fixed rate, and when n bytes need to be transmitted, n tokens are removed from the bucket. If the bucket is empty, no data is transmitted. This allows the node operator to specify an average amount of bandwidth usage that will be maintained, while permitting occasional bursts above the specified average.

Links between Onion Routers may also become congested if many circuits share the same nodes. To prevent this, each Onion Router keeps track of two windows for every circuit it is a part of: the *packaging window* and the *delivery window*. The *packaging window* represents the number of cells the node is willing to package and send back to the client while the *delivery window* counts the number of cells the node is willing to pass towards the final destination. Each window is initialized to 1000 cells. When an Onion Router sends a cell towards the destination, it decrements the *delivery window*; likewise, when it sends a cell back towards the client, it decrements the *packaging window*. If the count on a window reaches zero, the node stops relaying data on that circuit in the corresponding direction.

An Onion Router increments its windows when it receives a *relay sendme* cell from an adjacent node; such a cell indicates that the node that sent it is capable of receiving more data. If a link becomes congested, no *relay sendme* messages will be sent, the window will eventually reach zero and the node will stop sending data in that direction, easing the congestion. If a node receives a *relay sendme* from the node ahead of it, it increments its *delivery window* by 100; conversely, if it receives a *relay sendme* from the node that precedes it, the *packaging window* is incremented by 100.

Finally, Tor also distinguishes bulk transfers from interactive traffic by comparing the frequency at which different streams send cells, and gives interactive traffic higher priority, which allows decent throughput for bulk traffic while improving performance for the most common application of Tor: web browsing.

8. Hidden Services

Tor also allows for the possibility of creating a *hidden service*, which is a resource that can be accessed while allowing the person providing the service to remain anonymous. The creator of a resource chooses several nodes within Tor to act as *introduction points*, builds a circuit to each of them, and then advertises the introduction points through Tor's lookup service. He also creates a long-term public/private keypair, which he uses to sign the advertisement.

When a user wants to connect to the service (users must learn about the service out-of-band), he looks up the introduction points and then randomly chooses an Onion Router to act as a *rendezvous point*. The user then creates a circuit to the rendezvous point and gives it a randomly generated cookie that will be used to identify the service. Finally, the user builds a circuit to an introduction point and sends a message (encrypted with the service's public key) that contains the address of the rendezvous point, the cookie, and the first half of a DH handshake. If the service is willing to respond, it can build a circuit to the rendezvous point, provide the cookie, and send the second half of the handshake, establishing a shared key between user and service. Communication can now proceed as usual.

By adding a layer of indirection through the use of the rendezvous point, Tor also allows the hidden service to decide whether or not to accept any given connection.

9. Directory Servers

In order for clients to connect through Tor, they must have information about what Onion Routers are available. To achieve this, Tor uses several nodes that are trusted more than others, called *directory servers*. The IP address, public key, and fingerprint of the directory servers (there are currently three) is built into every Onion Proxy, but the defaults can also be changed through a configuration option.

When an Onion Router wishes to make itself available for circuits, it publishes its server description to the directory servers. If the OR's fingerprint is not already known to the directory server, its information will not be made available on the directory server—at present, the operators of the directory servers must manually approve all new Onion Routers. If the OR is recognized, however, it will be added to the list of running Onion Routers and information about it will be sent whenever a client requests a list of all running routers, including its IP address and exit policy.

Clients, meanwhile, obtain a list of running routers by simply requesting one from the directory servers through HTTP. These lists are signed using the directory server's private key, to prevent an attack where an adversary tricks a client into believing that some machine the adversary controls is a directory server. There is no mechanism for ensuring that all directory servers agree on the state of the network; each directory server just publishes the information it knows about. Clients then decide on the state of the network by going with the majority opinion from the directories they have received.

It should be noted that the use of centralized directory servers does leave Tor vulnerable to attacks on those servers. If an adversary can compromise more than half of the directories, he can then choose to only publish information about onion routers under his control, and easily trace any connections.

CHAPTER 3

Attacks on Tor

One of the more controversial features of Tor is its adversary model: in contrast to most anonymous systems, Tor does not attempt to protect users from powerful adversaries capable of observing all traffic on the network (known as a *global adversary* in anonymity literature). Instead, it attempts to protect against attacks where the adversary is capable of observing or altering only some fraction of network traffic, or compromising some fraction of the Onion Routers on the network.

Despite these limitations on the types of attackers, there are still a number of attacks that have been presented against Tor. These are detailed below.[1]

1. Basic Traffic Analysis

One capability that nearly all of the following attacks require is the ability, given some guess that two parties are communicating, to confirm or reject that guess. To do this, one most often makes use of some form of traffic analysis to correlate the traffic observed on one node with that observed at another. This task is made much easier if the anonymity-providing system does not attempt to delay, reorder, or drop packets as they go between nodes. Tor does not do any explicit mixing on any packets which pass through the network and the only latency introduced is that which naturally arises as a packet travels across the Internet. To help clarify the discussion of specific attacks, we will describe here

[1]Many of these attacks do not refer specifically to Tor, but rather to low-latency mix systems in general. I have only included such attacks when they are directly applicable to Tor.

one method (given by Levine et al. [16]) for establishing such a correlation; we will later use this technique in our implementation of a "backtracking" attack.

Suppose that we have two Onion Routers O_1 and O_2, and we wish to know if they are on the same path. For each Onion Router N_1^i connected to O_1 and each Onion Router N_2^j connected to O_2, we observe network traffic between from O_1 to N_1^i and from O_2 to N_2^j. We then take fixed-size, non-overlapping *windows* of time and count the number of packets received in that time period for each of the pairs (O_1, N_1^i) and (O_2, N_2^j), giving us sequences X_1^i and X_2^j. We then compare each X_1^i to each X_2^j and attempt to determine their *cross correlation*. The cross correlation r of two sequences y and z is defined as:

$$r = \frac{\sum_i((y_i - \mu_y)(z_i - \mu_z))}{\sqrt{\sum_i(y_i - \mu_y)^2}\sqrt{\sum_i(z_i - \mu_z)^2}}$$

where μ_y and μ_z are the mean of the sequences y and z, respectively. If the correlation between two sequences is sufficiently high, and the correlation between the other pairs of sequences sufficiently low, the two nodes are likely to be on the same path.

2. Path Confirmation Attack

In *Timing Analysis in Low-Latency Mix-Based Systems*, Levine et al. [16] present an attack on mix systems that uses the traffic analysis technique given in Section 1. For this attack, the adversary observes two nodes, an entry node on path P_I and an exit node on path P_J and wishes to determine whether $I = J$ (that is, whether the two nodes are part of the same path), thus linking source and destination. To accomplish this, he records packets arriving at and leaving the entry and exit nodes, splits them into fixed size windows, and correlates the resulting sequences.

3. The Intersection Attack

If some information about a user's traffic patterns is known in advance, an *intersection attack* can be mounted. It may be known, for example, that a user whose identity the adversary wishes to discover posts messages to a web forum every day at a particular time. If the attacker can then get any information about what users are active in the Tor network at that time, he can, over a long period of time, intersect these sets of users and extract probabilities that each of the users observed is, in fact, the target. In the case of Tor, such information can be gathered by, for example, running an Onion Router and logging the IP address of every user that connects to it at the specified time of day.

Intersection attacks are quite general and apply to many anonymous systems, and the threat posed by them "seems extremely difficult to solve in an efficient manner." [18] In the specific case of Tor, the attack can be mounted with very few resources—joining the network by running an Onion Router is fairly easy. The low resource cost, however, is offset by the fact that an attacker must gather information for an extremely long time before he can establish the identity of the victim with high probability.

4. The Predecessor Attack

The *predecessor attack* targets long-lived connections through Tor that persist through multiple path reformations; that is, any connection that uses a stream of traffic that is identifiable as the client sends it over different circuits. This situation is actually quite uncommon in Tor: most connections are short-lived (for example, web browsing), and long-lived connections (FTP, SSH, IRC) stay on the same circuit throughout their entire lifetime. However, if there were some piece of information in the stream that could be used to infer that several streams seen over time actually belonged to the same user (perhaps the re-use of a fairly unique pseudonym when posting to an online forum multiple times), then the attack could be carried out. Identifying streams of traffic in Tor is further made difficult by the use of encryption between each Onion Router, and between the Onion Proxy and the first node in the circuit. It is only when the connection exits Tor that any data is sent over the network unencrypted.

The attacker is also assumed to have some way to tell when the user has switched to a new circuit. This might be achieved by causing one of the routers in the current circuit to drop offline, which would necessitate rebuilding the circuit—although Tor has the capability to incrementally rebuild connections after one node goes offline, this capability is not currently implemented in the client. Alternatively, the attacker could simply wait for a long periods of time during the attack, and assume that since most connections are short-lived, the victim will have changed circuits by the time the attack begins again.

If these preconditions are met, the attack proceeds as follows. At each stage, the attacker compromises some constant number of Onion Routers (the minimum is two for a successful attack). If the attacker determines that the routers compromised are on the circuit that carries the stream the attacker is interested in (as noted above, determining this may require that the attacker get lucky and compromise the last node in the victim's circuit), the attacker logs the IP address of all users that connect to the earliest Onion Router on the path that has been compromised. As the attacker collects this data over a long period of time, across many changes in the circuit used, the IP of the initiator will be logged more frequently than that of any other user, since he necessarily participates in all connections that travel along the path he has chosen. For Onion Routing, the expected number of rounds before the attack is successful is $O((\frac{n}{c})^2)$, where

n is the total number of routers and c is the number of nodes the attacker can compromise per round.

5. Work Out from the Middle

In the course of general analysis of Onion Routing security, Syverson et al. [20] present an attack that allows a moderately powerful adversary to discover the initiator and receiver of a connection, provided the initial route setup has been observed. Unlike most of the attacks described in this section, the goal is not to target any particular user; rather, the adversary wishes to establish *any* sender-receiver pair. For the attack to succeed, the adversary must have the capability to compromise some fixed number k ($k \geq 2$) of Onion Routers within a given interval of time (referred to as a *round*). It is further assumed that the circuit will be in use for as long as it takes for the attack to succeed.

When the attack begins, the adversary is assumed to control at least one node in the Tor network, and has just seen a route being created that passes through the node he controls. During each round, the attacker compromises the node directly before the one he controls that is closest to the sender, and directly after the one he controls that is closest to the destination (in the first round, this corresponds to the nodes before and after the original corrupt Onion Router). He also compromises $k - 2$ Onion Routers randomly. Using timing analysis, he can tell if any of the random nodes he has compromised are on the same path as the connection he wishes to track. Out of this set of compromised nodes that lie on the desired path, he selects two nodes: the one closest to the sender and the one closest to the receiver (as determined, again, by the timing of packets passing through those nodes). He then releases control of all other compromised nodes and begins the cycle anew.

In each round of the attack, he gets one step closer to the initiator and the sender in the worst case. This means that at worst it will take n rounds (where n is the length of the path) to determine the endpoints of the connection. Once the endpoints are established, the adversary will be able to establish identity of the sender and receiver using that path by correlating the traffic entering at one end with that exiting at the other end.

6. Low-cost Traffic Analysis of Tor

Murdoch and Danezis [17] have presented what is, to date, the most practical and effective attack on the anonymity Tor provides. In this attack, the adversary is assumed to control a resource that a client accesses through Tor, such as a web page, and he wishes to determine the identity of the initiator. This assumption is quite reasonable; web site operators wish to known the identities of those accessing their sites, so that they can deliver targeted advertising or provide geographically specific services.

To begin the attack, the adversary chooses a connection coming into the destination server that he wishes to trace. He then acts as a client to the Tor network, creates a circuit of length one to each of the Onion Routers in the network (although the circuit length is currently hard-coded into the Tor client software, it is not difficult to modify it to use a different path length). A connection is then made through each of these circuits back to the adversary's client. This allows the adversary to easily measure the time it takes for a packet to go to the Onion Router and back. Nodes that have more traffic flowing through them will be more heavily loaded, and will therefore take more time to respond.

At this point, the attacker can begin to inject timing patterns into the data he is sending back to the victim. For example, he may send back data in pulses at regular intervals. During each of these pulses, the load on the Onion Routers that are on the path the victim is using will increase, and the latencies observed by the attacker using his connections to those nodes will be greater. In a relatively short amount of time, the adversary can establish a strong correlation between the timing of the pulses he sends to the victim and the load on the nodes the victim is using, thus revealing the victim's path through Tor.

This attack does not actually reveal the identity of the initiator; however, once the adversary knows the first node in the circuit, he needs only find a way to compromise that node or observe traffic entering it to completely succeed in identifying the initiator. The attack degrades the level of anonymity that Tor provides to that of a single proxy server.

There are several notable features of this attack. First, it falls completely within the restricted threat model that Tor uses—the attacker need not have a global view of traffic in the Tor network, and he need not have control over any of the hosts involved in the victim's connection aside from the final destination.[2] Second, since the attacker needs so few resources (the cost of the attack is given as $O(n)$, where n is the number of Onion Routers, since the attacker must create one connection to each router), it represents the first attack on Tor that is practical to implement for anyone who wishes to compromise the anonymity of a user.

[2]In fact, the attack works even if the adversary does not have full control over the destination—if he can observe network traffic at the endpoint, he can correlate timing patterns already present in the stream with his latency observations. This form of the attack will take longer, however, since the attacker has no control over the timing patterns in the data transmitted.

CHAPTER 4

Methodology

1. The Backtrack Attack

In order to test the effectiveness of basic timing analysis, we have created a simple attack against Tor's anonymity. In our attack, the adversary is able to observe packets entering or leaving some destination, such as a web server. He is also able, at each round, to observe packets on any single Tor node he chooses. This models an attacker that is moderately powerful; in the real world, being able to capture packets on any Tor node at will would require compromising another computer on the same local network, or having access to a switch or router on the same network.

The attacker's goal is to uncover the identity of a user he sees connecting to the destination server, as well as the path through Tor that user takes. To accomplish this, he first identifies the exit node the initiator is using; this is a trivial task, since it is directly connected to the destination he his monitoring. He then performs two simultaneous packet captures: on the destination, he captures all packets coming from the exit node (stream w in Figure 1); meanwhile, on the exit node, he captures any packets coming from other Onion Routers (streams x, x', and x'' in the diagram). Then for each candidate node that communicated with the exit node during the time the packets were captured, he performs the cross-correlation described in Section 1 with fixed-size windows (10 seconds each in our implementation). That is, for each candidate node O_i, he correlates the sequence $x = \{O_i^1, \dots O_i^k\}$ with the destination sequence $w = \{D^1, \dots, D^k\}$ (where

$k \approx \mathit{capturetime}/\mathit{windowsize}$). The sequence with the highest correlation over time should be correspond to the second node used in the initiator's path through Tor.

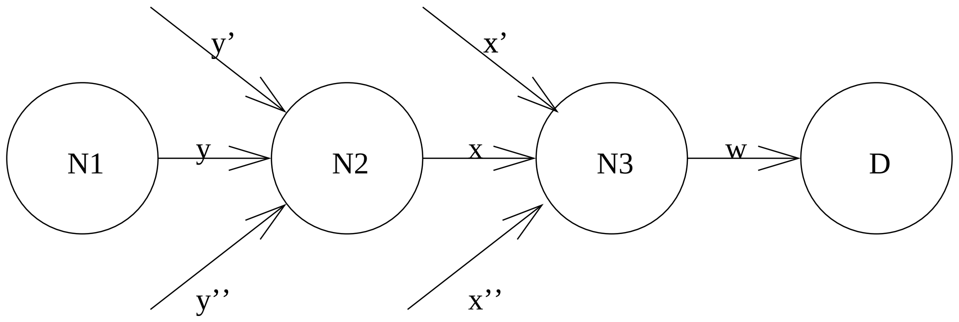

FIGURE 1. A circuit on which the backtrack attack will be run

The attacker then repeats this process and correlates packets from Onion Routers (y, y', and y'') connected to the second node with packets arriving at the destination and in this way discovers the address of the entry node. At this point the victim's entire path through Tor is known. To finally establish the identity of the initiator, the adversary correlates packets from all non-Tor nodes connected to the entry node with the packets captured at the destination (this is called *end-to-end correlation*). One possible alternative version of this attack would be to correlate each of the streams between Onion Routers with the stream going to the next node in the path (e.g., correlating y, y', and y'' with x) rather than with the stream going to the destination. However, all links between Onion Routers go over a single SSL-encrypted connection, and so there is no way of reliably separating out the streams belonging to different circuits.

Two features of this attack should be noted: first, the attack is entirely passive; the attacker need not modify or create packets, nor can he actually take over any machine. Second, the attacker never needs to have access to the machine or network the initiator is using; all captures are done at the destination or on one of the intermediate nodes. This is to the attacker's benefit, since he would naturally wish to remain undetected by the initiator.

2. Platform Setup

To test the attack, we created a network of machines running the actual Tor software, since this is more realistic than a traditional network simulation. Setting up a large number of actual machines on a network, however, is prohibitively expensive and labor-intensive. However, it is possible to create many virtual machines running on a single host through the use of User-mode Linux (UML) [9]. User-mode Linux simulates a full Linux kernel running in user space from a hard drive image on the host; any flavor of Linux may be installed onto this image, and any software that can run on the Linux platform can run inside of UML.

Our test network of UML instances consists of 91 machines spread across three hosts: 12 Onion Routers, 3 directory servers, 15 destination servers, 45 clients, and one network file server. To create the virtual machines, we installed Debian/GNU Linux 3.1 [3] on a hard drive image to create a base system, and then used UML's copy-on-write feature to allow many machines to share the same drive image. With copy-on-write, each write to the drive a virtual machine makes does not modify the original image, but rather a file specific to that instance. This allows for considerably greater space efficiency.

The first host, `kurtz`, was dedicated to running the Tor network. Each virtual machine had a copy of the Tor software (version 0.1.1.14-alpha) loaded, as well as a simple server we created that allowed remote hosts to request packet captures. `Kurtz` is a dual processor machine (each processor is an Intel® Xeon^{TM} CPU running at 2.80GHz) with 4 gigabytes of RAM and two 250 gigabyte hard drives in a RAID1 (mirroring) configuration.

The second host, `mimesis`, ran the 15 of the simulated clients and the 15 destination servers. The final host, `taki`, ran the remaining 45 clients. `Mimesis` is a Pentium® 3.20GHz machine with 512 megabytes of RAM, and `taki` is a dual-core Pentium® D 2.80GHz with 2 gigabytes of RAM.

Each client connects to a destination server and generates traffic through the Tor network according to several predefined usage patterns (uniform traffic, random message size, and random delay). The traffic generator also queries the local Onion Proxy to find out what path the connection is taking through Tor, and then writes this information out to a global "answers" file so that successful attacks can be confirmed. The destination servers use the standard UNIX discard daemon to listen for incoming connections, and also run the packet capture server mentioned above; in our current setup they do not return any data to the client aside from the standard TCP acknowledgement packets. All UML instances on `kurtz`, `mimesis`, and `taki` have access to a shared, writable network drive hosted by a UML instance running on `kurtz`. This shared space is used only to make software distribution easier, and is not necessary for the simulation in general.

3. Network Setup

To allow the virtual hosts to communicate, `kurtz`, `mimesis`, and `taki` each have a number of *tap* devices, one for each hosted UML instance. The tap devices appear as ordinary ethernet interfaces from inside User-mode Linux, and can be configured in the usual way.[1] The tap interfaces were then linked together on each

[1]In this case, we assigned each UML a static IP address from the 192.168.0.0/24 subnet, which is reserved for private networks.

host machine using the Linux virtual ethernet bridge [4], which acts in much the same way an ethernet switch does, forwarding packets between the tap interfaces.

The networks of virtual machines on the three hosts were then joined together using OpenVPN [6], which allows two machines to access each other's networks over a secure TCP connection. This allows each UML instance to contact the others as if it were on the same local subnet, regardless of its physical location.

4. Attack Software

The attack itself was implemented in the Python programming language [7], and utilizes the pcapy and Impacket libraries (versions 0.10.4 and 0.9.5.1, respectively) from CoreSecurity Technologies [2] to read and analyze packet captures. The traffic generator described in Section 2 was written in C. The attack software, when executed on a destination server,[2] randomly picks one of the incoming connections to the machine from Tor and attempts to trace its initiator using the method described above. Specifically, for each stage of the attack it requests a sixty second packet capture from each of the two machines involved in the correlation and then correlates the streams by examining the packets captured. If on all of the streams the correlation is not strong enough (we describe in greater precision what is meant by this below), more packet captures may be requested and the results combined. At the end of the attack, the results are confirmed by checking the computed path and initiator against the answers file described in Section 2.

The full source code of all software can be found in Appendix A.

[2]There is no special reason why the attack software need be executed directly on the destination server; it is merely convenient since it allows us to directly see what connections the machine has open without capturing any packets.

5. Experimental Setup

Our goal in implementing this attack was to examine its efficiency; specifically, how much data (as measured by the number of windows, assuming a fixed size window of 10 seconds) is required before the correlation is sufficiently strong to be confident that we have found the previous node in the path. We are also interested in understanding the effect of various network parameters, such as the ratio of clients to Tor nodes and the pattern of traffic sent by clients.

There are a number of criteria we might use to decide when a correlation is "strong enough." The most intuitive is a simple correlation threshold: the correlation between two streams is strong enough when it rises above, say, 0.9. The previous node in the path is then chosen as the one with the highest correlation among those whose correlation is higher than 0.9. Another useful metric might be the difference in correlation between the best candidate and the second-best candidate, which should grow as the number of windows increases. In our initial runs, including those involving the uniform traffic model (described below), we used correlation difference to determine the previous node in the circuit; in later runs, however, we found a simple threshold to give quicker results.

Our experimental setup modelled traffic according to several different patterns:

- **Uniform traffic:** Each client sends a steady stream of packets with no delay between packets. Note that this results in streams from clients which are essentially identical to one another.
- **Random message length:** Each client flips a weighted coin, and sends another packet until the coin comes up tails. This leads to bursts of packets that have an expected length that depends on the weights assigned to the coin; a coin that comes up heads 99/100 times will produce messages consisting of 100 packets using this method.

- **Random delay:** Each client waits for a random amount of time between sending messages according to a uniform distribution between two real numbers.

The first run of the attack used uniform traffic; however, because of the similarity in the traffic patterns of the streams, our attack was unable to distinguish between them, and consistently found more than one stream that was highly correlated with the destination stream. This suggests that cover traffic, the practice of enforcing some level of constant traffic between all Onion Routers (inserting dummy messages when there is not enough traffic, and refusing to relay messages when there is too much), would be an effective defense against our attack. It is possible, however, to imagine more sophisticated versions of the attack that use, for example, Wei Dai's method [10] of estimating actual traffic usage in the presence of cover traffic.[3]

We then ran tests with a combination of random message length and random delay. Each test run consisted of 10 attempts to trace a random connection to one of the destination servers through Tor; 15 such runs were performed and the results were analyzed to determine the number of 10-second windows needed to find the preceding node at each stage, as well as to determine the false positive rate. Finally, we quadrupled the number of clients and repeated the simulation. Our results are given in the next chapter.

[3]This method works by using up the node's bandwidth with the attacker's traffic and then deducing the amount of other traffic flowing through the link by subtracting the attacker's traffic from the total amount of traffic.

CHAPTER 5

Results

1. Uniform Traffic

We attempted to discover three paths using a uniform traffic model, using the difference between the two highest candidates as our criteria for selecting the next host. Because all of the client streams were essentially identical, in each case there were several candidates that were all highly correlated with stream going to the destination. In fact, because there was no termination condition imposed, the attack software was never even able to determine the middle node in the circuit successfully. Each attempt was allowed to continue until the virtual machine it was running on ran out of memory; a graph showing how the correlation difference varied with the number of windows is given in figure 1.

2. Simulation 1: 15 Clients, 15 servers, 15 Onion Routers

In total, 147 paths were discovered in this simulation, broken up into runs of 10 path discoveries. After each run the data generators on the clients were restarted in order to force them to choose a new server to connect to and a new path through Tor. Of those 147, 36 of the paths were incorrect, a false positive rate of 24.5%. We have also calculated the number of windows required to perform the correlation at each stage, and the results are summarized in table 1 (each of the “stages” listed in the table refers to one portion of the backtrack attack—i.e., Stage 1 attempts to discover the middle node, Stage 2 attempts to discover the entry node, and Stage 3 attempts to find the initiator).

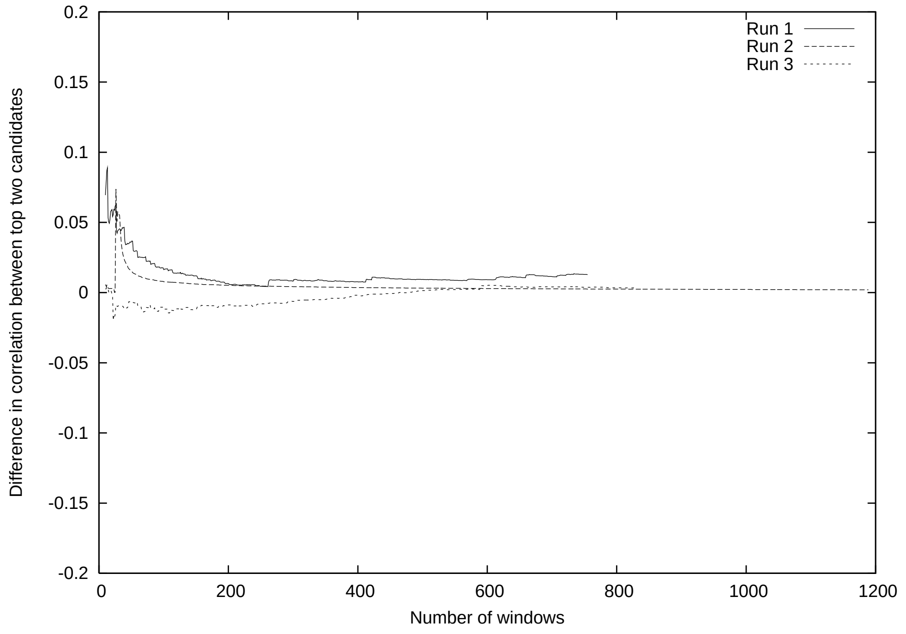

Figure 1. Correlation difference versus windows with uniform traffic

	Stage 1	**Stage 2**	**Stage 3**
Min	6	6	6
Max	67	68	1251
Mean	19.28	20.24	40.64
Median	13.00	7.00	7.00
Mode	7	7	7
Standard Deviation	19.28119	21.31357	147.35053

Table 1. Simulation 1 Results (15 clients, 15 servers, 15 Tor nodes)

3. False Positives

The rate of false positives was much higher than we anticipated. After examining the data, we noticed that on runs that produced false positives there was usually a “close call” at some stage; although the highest correlation was over the threshold, the next highest candidate was very highly correlated as well. To make this notion more precise, we separated the runs into false positives and correct runs, and measured the difference in correlation between the top two candidates at each stage. We found that in all three stages, the average difference between the

	Correct		False Positives	
	Mean	**Median**	**Mean**	**Median**
Stage 1	0.25	0.10	0.17	0.05
Stage 2	0.23	0.08	0.12	0.04
Stage 3	0.16	0.09	0.04	0.01

TABLE 2. Correlation difference comparison for false positives and correct runs

highest candidate and the second-highest was consistently greater for the correct runs than for the false positives. These results are given in Table 2.

These results clearly indicate that the choice of criteria for determining when a correlation is strong enough is critical. A strategy for choosing the next hop that incorporates both the absolute strength of the correlation and its strength relative to the other streams observed would very likely have a much better success rate.

4. The Effect of Window Size

Up to this point, all discussion has assumed that the windows were fixed at 10 seconds. Given that this window size is more than 13 times larger than the expected delay between client messages, one might expect that a smaller window size could allow for more accurate correlations for the same amount of packets captured, perhaps improving the false positive rate. Using the saved packet captures from Simulation 1, we re-did the analysis using windows of size 0.5, 2, and 5 seconds. For each false positive, we checked whether, given the same amount of packet data, the new window size would have arrived at a different choice for the next hop, and if so, whether this new choice was actually correct (according to the stored list of paths for that run). Likewise, for each correct run, we determined whether the new window size would have resulted in a new false positive.

With half-second windows, the results were generally much less accurate than the original 10-second window. Of the 36 false positives from the first run, there were only three cases where the new window size resulted in a correct choice of the next hop. At the same time, it generated 99 new false positives when the test was performed on successful runs. Even if we assume that every run where the new window size improved upon the results of the original would have continued to choose the correct nodes from that point onward (unlikely, given the rate of new false positives), this gives a new false positive rate of 89.8%. Additionally, the correlations themselves were extremely low, with a mean of 0.012.

Two factors may help explain the surprisingly poor performance of the attack with a half-second window size. First of all, the system clocks on `kurtz` and `mimesis` were not perfectly synchronized, despite attempts to keep them on the same time using the Network Time Protocol (NTP). Since the Onion Routers were located on `kurtz` and the destination servers on `mimesis`, the timestamps on packets captured at the same time on both hosts were not identical. This means that the sequence of packets observed on one host might be shifted by several windows with respect to a sequence observed on the other, which would result in the low correlations observed. In addition, the network itself adds a certain amount of latency, which could also create a similar sort of shift in the sequence of packets. This latter explanation is less convincing, however, since the delay between links is actually quite small (around 12ms, or 62 times smaller than the expected delay between messages).

With larger window sizes, the effect of various time shifts becomes smaller. With a window size of 2 seconds, we obtain a false positive rate of 51.0%, and with a window size of 5 seconds we see a false positive rate of 27%. Neither of these, however, actually improves on the original false positive rate. Going the

	Stage 1	**Stage 2**	**Stage 3**
Min	6	6	6
Max	67	66	68
Mean	18.66	20.24	23.51
Median	7.00	7.00	7.00
Mode	7	7	6
Standard Deviation	21.19650	22.37061	26.02917

Table 3. Simulation 2 Results (60 clients, 15 servers, 15 Tor nodes)

other direction, we find that with 15 and 20 second windows, we have false positive rates of 55.8% and 66.0%, respectively.

5. Simulation 2: 60 Clients, 15 servers, 15 Onion Routers

For our second simulation, we wanted to determine how effective the attack would be with a higher ratio of clients to Tor nodes. On the real Tor network, the ratio of clients to Onion Routers is around 550:1; with our current simulation setup, we have only been able to achieve a ratio of 4:1. As the amount of traffic going through Tor increases, it is expected that the difficulty of correlating the streams would also increase: the number of windows required to find a correlation high enough should be higher, as should the false positive rate.

As before, in each run we attempted to discover 10 paths and then reset the data generators. 14 runs were performed, for a total of 140 paths discovered. Table 3 summarizes the results of the simulation. Out of the 140 paths found, 55 were false positives, giving a false positive rate of 39.3%; as expected, increasing the amount of traffic on the network does create more false positives, but surprisingly, the number of windows needed to obtain a sufficiently high correlation was about the same as in the first simulation.

CHAPTER 6

Conclusion

1. Effectiveness of the Attack

In its current form, the backtrack attack is unlikely to be very effective against the real Tor network, as the rate of false positives is likely to increase further as more clients are added. However, since the attack can be re-run for as long as the connection remains alive, an attacker with the capabilities we described could reasonably find the initiator by making repeated runs, or by simply requiring a much larger amount of packet data before deciding on the next stage.

2. Possible Improvements

The simplest improvement to the attack would be to fine-tune the criteria for selecting the next node in the circuit. For example, we might require that the same node remain the highest candidate over some specified interval of time, or that it be a certain amount more highly correlated than the next-best candidate, or some combination of both. Imposing stricter conditions on the quality of the correlation would most likely increase the number of windows required and hence the time required for the attack, but it should also reduce the rate of false positives. Another simple improvement would be to incorporate some method of detecting when the algorithm has chosen incorrectly (perhaps by noting when the correlations are much lower than one would expect for a node on the path) and allow it to go back and try the next-highest candidate at the previous hop. This, again, would cause the attack to take longer, but it would improve the accuracy significantly.

One might also consider various techniques to eliminate the effect of network delay on the correlation. If we allow the attacker some active abilities, such as the ability to measure timings along routes, it should be possible to re-align the sequences before correlating them, since the attacker would know approximately what delay the network has introduced into the timings. It may also be possible to use statistical techniques to calculate the appropriate delay without adding any abilities to the attacker. If the effect of delay can be effectively nullified, the attack could operate reliably with much smaller window sizes than we currently use, which in turn would allow for far faster runs–possibly even bringing the expected time for the attack down to something that would be able to track ordinary users browsing web sites through Tor.

Finally, rather than using cross correlation, some other means of determining how similar two streams are could be used. Zhu et al. [21], for example, have used mutual information as their measure of similarity. It is unknown, however, what the effect of a different measure of similarity would be.

3. Conclusion

Current low-latency mix networks are, in fact, quite vulnerable against moderately powerful attackers using the techniques we have described. An attacker who has the ability to capture packets on some large percentage of the Tor network can do much better than random chance in determining who the initiator of any connection is, especially against long-lived connections such as large file downloads, SSH connections, or IRC sessions. Unfortunately, most mechanisms for preventing timing analysis also increase the latency of traffic going through

Tor, which makes the network less attractive to end users; with fewer users, attacks such as the intersection attack (described in Section 3) become much more effective, and such attacks can be mounted by weaker adversaries (recall that the intersection attack requires only that the adversary be able to control a single Tor node).

For the moment, then, there is no effective defense against the types of attackers we have described. Users must decide how much convenience they are willing to sacrifice for anonymity.

APPENDIX A

Source Code Listings

1. auto_attack.py

```
#!/usr/bin/env python

import threading
import sys, os
import socket
import random
import time
from bdg_config import *
from data_sources import nicknames,hostnames
from correlations import correlate_or_multi,correlate_endtoend_multi
from netstat import get_open_connections

dest_pcap_filenames = []
or_pcap_filenames = []

class AutoFlush:
    def __init__(self, path, mode):
        self.fp = open(path, mode)
    def write(self, data):
        self.fp.write(data)
        self.fp.flush()
    def close(self):
        self.fp.close()

def setup_output_dir(prefix="/mnt/results"):
    dirname = os.path.join(prefix,time.strftime("run-%Y-%m-%d-%H:%M:%S"))
    stage1 = os.path.join(dirname,"stage1")
    stage2 = os.path.join(dirname,"stage2")
    stage3 = os.path.join(dirname,"stage3")
    os.mkdir(dirname)
    os.mkdir(stage1)
    os.mkdir(stage2)
    os.mkdir(stage3)
    for stage in (stage1, stage2, stage3):
        os.mkdir(os.path.join(stage, "dest"))
        os.mkdir(os.path.join(stage, "or"))
    log = AutoFlush(os.path.join(dirname, "log.txt"), 'w')
    return (stage1, stage2, stage3, log)

def move_pcaps(dest_pcaps, or_pcaps, dir):
    for name in dest_pcaps:
        filename = os.path.basename(name)
        os.rename(name, os.path.join(dir, "dest", filename))
    for name in or_pcaps:
        filename = os.path.basename(name)
        os.rename(name, os.path.join(dir, "or", filename))

class CaptureThread ( threading.Thread ):
    def __init__(self, capserver, cap_port=CAP_PORT, cap_time=300,
                 filterstr="", dest=True, logfile=sys.stdout):
        self.capserver = capserver
        self.cap_port = cap_port
        self.cap_time = cap_time
        self.filterstr = filterstr
        self.dest = dest
        self.log = logfile
        threading.Thread.__init__ ( self )
    def run(self):
        global dest_pcap_filenames, or_pcap_filenames
        cap_id = int(time.time())
        s = socket.socket()
        self.log.write("[%s] capturing on host %s\n" %
                       (self.getName(), self.capserver))

        # We're getting some odd timeouts--try connecting 3 times
        for i in range(3):
            try:
                s.connect( (self.capserver, self.cap_port) )
                break
            except: pass

        f = s.makefile()
        s.send("CAPTURE|%s|%d|%d\n" % (self.filterstr, self.cap_time, cap_id))
        fname = f.readline().strip()
        self.log.write("[%s] capture finished, output filename is %s\n" %
                      (self.getName(), fname))
        s.send("QUIT\n")
        s.close()

        if self.dest:
            dest_pcap_filenames.append(fname)
        else:
            or_pcap_filenames.append(fname)

        return

def get_ip_address(ifname):
    import fcntl
    import struct
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    return socket.inet_ntoa(fcntl.ioctl(
        s.fileno(),
        0x8915,  # SIOCGIFADDR
        struct.pack('256s', ifname[:15])
    )[20:24])

START_TIME = time.time()

if len(sys.argv) != 2:
    print "usage: %s <output directory>" % sys,argv[0]
    sys.exit(1)

stage1, stage2, stage3, log = setup_output_dir(sys.argv[1])
#log = sys.stdout

# Pick a random stream to trace
dest_ip = get_ip_address("eth0")
conns = [ dst for src, dst in get_open_connections() if src == (dest_ip, 9) ]
if len(conns) == 0:
    print "No connections from Tor on this machine"
    sys.exit(0)
exit_ip, exit_port = random.choice(conns)
log.write("Tracing incoming connection from %s, port %d\n" %
          (exit_ip, exit_port))

correlation = 0.0
corrdiff = 0.0
node2 = ""
iter = 1
empty_tries = 1
#while corrdiff < CORR_DIFF_THRESHOLD:
while correlation < CORR_THRESHOLD:
    # Connect to exit_ip and dest_ip, cap on both
    # correlate_or two cap files, determine node2
    dest_capthread = CaptureThread(dest_ip, CAP_PORT, 60,
        "ip src %s and tcp dst port %d and tcp src port %d" %
            (exit_ip, PORT, exit_port),
        dest=True, logfile=log)
```

```
or_capthread = CaptureThread(exit_ip, CAP_PORT, 60,
    "ip dst %s and tcp port 9001" % exit_ip,
    dest=False, logfile=log)

# Start the threads and then wait for them to finish
# when they both return, dest_pcap_filename and
# or_pcap_filename should be filled in with the values
# of the packet captures
dest_capthread.start()
or_capthread.start()
dest_capthread.join()
or_capthread.join()

dest_wins, results = correlate_or_multi(dest_pcap_filenames,
                        or_pcap_filenames, exit_ip, exit_ip, WINDOW_SIZE)
if len(results) < 2:
    if empty_tries < UTTER_FAILURE:
        empty_tries = empty_tries + 1
        log.write("Not enough data to correlate; %d tries left\n" %
                  (UTTER_FAILURE - empty_tries))
        continue
    else:
        log.write("Didn't receive enough packets to correlate"
                  "after %d tries, aborting.\n" % empty_tries)
        # One last hope--maybe there was only one host talking
        # the entire time
        if len(results) == 1:
            node2, correlation, _ = results[-1]
            log.write("Correlation after %d windows is %f,"
                      " candidate %s.\n" % (len(dest_wins),
                          correlation, hostnames[node2]))
            log.write("No other candidate.\n")
            break
        END_TIME = time.time()
```

```
            log.write("Time taken: %f seconds\n" % (END_TIME - START_TIME))
            move_pcaps(dest_pcap_filenames, or_pcap_filenames, stage1)
            sys.exit(1)
    node2, correlation, _ = results[-1]
    corrdiff = correlation - results[-2][1]
    log.write("Correlation after %d windows is %f, candidate %s.\n" %
            (len(dest_wins), correlation, hostnames[node2]))
    log.write("Next highest candidate: %s %f\n" %
            (hostnames[results[-2][0]], results[-2][1]))
    if iter >= TRIES:
        # Abort and go with what we have
        log.write("Reached cutoff point, going with current best candidate\n")
        break
    else:
        iter = iter + 1

log.write("STAGE 1: found hop: %s, nickname %s, hostname %s, "
        "correlation %f\n" % (node2, nicknames[node2],
            hostnames[node2], correlation))

move_pcaps(dest_pcap_filenames, or_pcap_filenames, stage1)

dest_pcap_filenames = []
or_pcap_filenames = []
correlation = 0.0
corrdiff = 0.0
node1 = ""
iter = 1
empty_tries = 1
#while corrdiff < CORR_DIFF_THRESHOLD:
while correlation < CORR_THRESHOLD:
    # connect to dest_ip, node2, cap on both
    # correlate_or two cap files, determine node1
```

```
dest_capthread = CaptureThread(dest_ip, CAP_PORT, 60,
    "ip src %s and tcp dst port %d and tcp src port %d" %
        (exit_ip, PORT, exit_port),
    dest=True, logfile=log)
or_capthread = CaptureThread(node2, CAP_PORT, 60,
    "ip dst %s and tcp port 9001" % node2,
    dest=False, logfile=log)

dest_capthread.start()
or_capthread.start()
dest_capthread.join()
or_capthread.join()

dest_wins, results = correlate_or_multi(dest_pcap_filenames,
                        or_pcap_filenames, exit_ip, node2, WINDOW_SIZE)
if len(results) < 2:
    if empty_tries < UTTER_FAILURE:
        empty_tries = empty_tries + 1
        log.write("Not enough data to correlate; %d tries left\n" %
                  (UTTER_FAILURE - empty_tries))
        continue
    else:
        log.write("Didn't receive enough packets to correlate "
                  "after %d tries, aborting.\n" % empty_tries)
        # One last hope--maybe there was only one host
        # talking the entire time
        if len(results) == 1:
            node1, correlation, _ = results[-1]
            log.write("Correlation after %d windows is %f, "
                      "candidate %s.\n" % (len(dest_wins),
                            correlation, hostnames[node1]))
            log.write("No other candidate.\n")
            break
        END_TIME = time.time()
```

```
            log.write("Time taken: %f seconds\n" % (END_TIME - START_TIME))
            move_pcaps(dest_pcap_filenames, or_pcap_filenames, stage2)
            sys.exit(1)
    node1, correlation, _ = results[-1]
    corrdiff = correlation - results[-2][1]
    log.write("Correlation after %d windows is %f, candidate %s.\n" %
            (len(dest_wins), correlation, hostnames[node1]))
    log.write("Next highest candidate: %s %f\n" %
            (hostnames[results[-2][0]], results[-2][1]))
    if iter >= TRIES:
        # Abort and go with what we have
        log.write("Reached cutoff point, going with current best candidate\n")
        break
    else:
        iter = iter + 1

log.write("STAGE 2: found hop: %s, nickname %s, hostname %s, "
          "correlation %f\n" % (node1, nicknames[node1],
                hostnames[node1], correlation))

move_pcaps(dest_pcap_filenames, or_pcap_filenames, stage2)

# connect to dest_ip, node1, cap on both
# correlate_endtoend two cap files, find initiator
dest_pcap_filenames = []
or_pcap_filenames = []
correlation = 0.0
corrdiff = 0.0
init_ip = ""
iter = 1
empty_tries = 1
#while corrdiff < CORR_DIFF_THRESHOLD:
while correlation < CORR_THRESHOLD:
    dest_capthread = CaptureThread(dest_ip, CAP_PORT, 60,
    "ip src %s and tcp dst port %d and tcp src port %d" %
        (exit_ip, PORT, exit_port),
    dest=True, logfile=log)
or_capthread = CaptureThread(node1, CAP_PORT, 60,
    "ip dst %s and tcp port 9001" % node1,
    dest=False, logfile=log)

dest_capthread.start()
or_capthread.start()
dest_capthread.join()
or_capthread.join()

dest_wins, results = correlate_endtoend_multi(dest_pcap_filenames,
                        or_pcap_filenames, exit_ip, node1, WINDOW_SIZE)
if len(results) < 2:
    if empty_tries < UTTER_FAILURE:
        empty_tries = empty_tries + 1
        log.write("Not enough data to correlate; %d tries left\n" %
                  (UTTER_FAILURE - empty_tries))
        continue
    else:
        log.write("Didn't receive enough packets to correlate "
                  "after %d tries, aborting.\n" % empty_tries)
        # One last hope--maybe there was only one host
        # talking the entire time
        if len(results) == 1:
            init_ip, correlation, _ = results[-1]
            log.write("Correlation after %d windows is %f, "
                      "candidate %s.\n" % (len(dest_wins),
                            correlation, hostnames[init_ip]))
            log.write("No other candidate.\n")
            break
        END_TIME = time.time()
        log.write("Time taken: %f seconds\n" % (END_TIME - START_TIME))
```

```
            move_pcaps(dest_pcap_filenames, or_pcap_filenames, stage3)
            sys.exit(1)
    init_ip, correlation, _ = results[-1]
    corrdiff = correlation - results[-2][1]
    log.write("Correlation after %d windows is %f, candidate %s.\n" %
              (len(dest_wins), correlation, hostnames[init_ip]))
    log.write("Next highest candidate: %s %f\n" %
              (hostnames[results[-2][0]], results[-2][1]))
    if iter >= TRIES:
        # Abort and go with what we have
        log.write("Reached cutoff point, going with current "
                  "best candidate\n")
        break
    else:
        iter = iter + 1

log.write("STAGE 3: found initiator: %s, hostname %s, correlation %f\n" %
          (init_ip, hostnames[init_ip], correlation))

log.write("Full path: %s -> %s -> %s -> %s -> %s\n" %
          (hostnames[init_ip], hostnames[node1], hostnames[node2],
              hostnames[exit_ip], hostnames[dest_ip]))

move_pcaps(dest_pcap_filenames, or_pcap_filenames, stage3)

END_TIME = time.time()

path = [hostnames[init_ip], nicknames[node1], nicknames[node2],
        nicknames[exit_ip], hostnames[dest_ip]]

# Check our answer
try:
    f = open("/mnt/answers/" + ",".join(path))
    log.write("Initiator and path match!\n")
    f.close()
except IOError:
    log.write("Error! Path was incorrect.\n")

log.write("Time taken: %f seconds\n" % (END_TIME - START_TIME))
log.close()
```

2. bdg_config.py

```
from timestamp import TimeStamp
PORT=9
WINDOW_SIZE=TimeStamp((10, 0))
CAP_PORT=10000
CORR_THRESHOLD = .9
CORR_DIFF_THRESHOLD = .2
TRIES = 10
UTTER_FAILURE = 10
```

3. correlations.py

```
#!/usr/bin/env python

import sys
import pcap_utils
from timestamp import TimeStamp
from cross_corr import cross_corr
from makewin import make_windows
from data_sources import nicknames,hostnames
from pprint import pprint

from bdg_config import *

def usage():
    sys.stderr.write("usage: %s <\"end\"|\"or\"> <dest pcap> <or pcap> "
                     "<exit node IP> <or node IP>\n" % sys.argv[0])
```

```
def is_empty_all(pcaps, ip):
    for pcap in pcaps:
        if not pcap_utils.is_empty(pcap, "ip src %s and tcp src "
                                          "or dst port 9001" % ip):
            return False
    return True


def remove_inactive(pcaps, ips):
    # Remove any IPs that didn't talk at all during our packet captures
    active_ips = ips[:] # make a copy of the original

    for o in active_ips:
        if is_empty_all(pcaps, o):
            active_ips.remove(o)
    return active_ips


def make_dest_wins(dest_pcap, exit_ip, winsize=WINDOW_SIZE):
    dest_filter = "tcp src or dst port %d and ip src %s" % (PORT, exit_ip)
    dest_start = pcap_utils.get_start_ts(dest_pcap, dest_filter)
    dest_end = pcap_utils.get_end_ts(dest_pcap, dest_filter)
    dest_wins = make_windows(dest_pcap, winsize, dest_filter,
                             dest_start, dest_end)
    return (dest_wins, dest_start, dest_end)


def make_other_wins(or_pcap, onion, current_or_ip,
                    start, end, winsize=WINDOW_SIZE):
    or_filter = ("ip src %s and ip dst %s and tcp src or dst port 9001" %
                 (onion, current_or_ip))
    or_wins = make_windows(or_pcap, winsize, or_filter, start, end)
    return or_wins


def get_non_or_ips(pcaps, or_ips):
    # Get a list of all non-OR nodes in the OR packet captures
    or_filter_single = [ "not ip src %s" % o for o in or_ips ]
    or_filter = " and ".join(or_filter_single)
    non_or_ips = pcap_utils.get_all_ips(pcaps, or_filter)
    return non_or_ips


def correlate_endtoend_multi(dest_pcaps, or_pcaps, exit_ip,
                             current_or_ip, winsize=WINDOW_SIZE):
    from data_sources import or_ips

    non_or_ips = get_non_or_ips(or_pcaps, or_ips)
    active_ips = remove_inactive(or_pcaps, non_or_ips)

    cl_wins = [[] for i in range(len(active_ips))]
    dest_wins = []
    for i in range(len(dest_pcaps)):
        dest_win, start, end = make_dest_wins(dest_pcaps[i], exit_ip,
                                              winsize)
        dest_wins.extend(dest_win)
        for j in range(len(cl_wins)):
            client = active_ips[j]
            cl_win = make_other_wins(or_pcaps[i], client, current_or_ip,
                                     start, end, winsize)
            cl_wins[j].extend(cl_win)

    # Cross correlate dest_wins with each cl_win
    results = []
    for client, cl_win in zip(active_ips, cl_wins):
        results.append((client, cross_corr(dest_wins, cl_win), cl_win))

    results.sort(lambda x, y: cmp(x[1], y[1]))

    return (dest_wins, results)

def correlate_or_multi(dest_pcaps, or_pcaps, exit_ip, current_or_ip,
                       winsize=WINDOW_SIZE):
```

```
from data_sources import or_ips

active_ors = remove_inactive(or_pcaps, or_ips)
try: active_ors.remove(current_or_ip)
except: pass
try: active_ors.remove(exit_ip)
except: pass


# list of lists of windows, one for each active_or
or_wins = [[] for i in range(len(active_ors))]

dest_wins = []
for i in range(len(dest_pcaps)):
    dest_win, start, end = make_dest_wins(dest_pcaps[i], exit_ip,
                                          winsize)
    dest_wins.extend(dest_win)
    # go through each active_or, and extend the corresponding
    # entry in or_wins using the current or_pcap file
    for j in range(len(or_wins)):
        onion = active_ors[j]
        or_win = make_other_wins(or_pcaps[i], onion, current_or_ip,
                                 start, end, winsize)
        or_wins[j].extend(or_win)

# Cross correlate dest_wins with each or_win
results = []
for onion, or_win in zip(active_ors, or_wins):
    results.append((onion, cross_corr(dest_wins, or_win), or_win))
results.sort(lambda x, y: cmp(x[1], y[1]))

#pprint(dest_wins)
#pprint(results)
return (dest_wins, results)
```

```
# Convenience functions that deal only with single pcaps
def correlate_or(dest_pcap, or_pcap, exit_ip, current_or_ip,
                 winsize=WINDOW_SIZE):
    return correlate_or_multi([dest_pcap], [or_pcap], exit_ip,
                              current_or_ip, winsize)

def correlate_endtoend(dest_pcap, or_pcap, exit_ip, current_or_ip,
                       winsize=WINDOW_SIZE):
    return correlate_endtoend_multi([dest_pcap], [or_pcap],
                                    exit_ip, current_or_ip, winsize)

if __name__ == "__main__":
    if len(sys.argv) != 6:
        usage()
        sys.exit(1)

    type = sys.argv[1]
    dest_pcap = sys.argv[2]
    or_pcap = sys.argv[3]
    exit_ip = sys.argv[4]
    current_or_ip = sys.argv[5]

    if type == "end":
        dest_wins, results = correlate_endtoend(dest_pcap, or_pcap,
                                                exit_ip, current_or_ip)
        for r in results: print r[0:2], hostnames[r[0]]
    elif type == "or":
        dest_wins, results = correlate_or(dest_pcap, or_pcap,
                                          exit_ip, current_or_ip)
        for r in results: print r[0:2], nicknames[r[0]], hostnames[r[0]]
```

4. cross_corr.py

```
#!/usr/bin/env python
```

```
import math


def mean(seq):
    return float(sum(seq)) / float(len(seq))


# Note: this implementation assumes that values out of range are not ignored,
# but rather wrap around to the beginning of the sequence. This is probably
# not what we actually want, but it doesn't matter for delay = 0
def cross_corr(seq1, seq2, delay=0):
    m1 = mean(seq1)
    m2 = mean(seq2)

    wins = len(seq1) # assume number of windows is same for both seqs

    top = sum( [ (seq1[i] - m1)*(seq2[(i+delay) % wins] - m2)
                 for i in range(wins) ] )
    bot1 = sum ( [ (seq1[i] - m1)*(seq1[i] - m1) for i in range(wins) ] )
    bot1 = math.sqrt(bot1)
    bot2 = sum ( [ (seq2[i] - m2)*(seq2[(i+delay) % wins] - m2)
                   for i in range(wins) ] )
    bot2 = math.sqrt(bot2)

    try:
        result = top / (bot1*bot2)
    except ZeroDivisionError:
        result = 0.0
    return result
```

5. data_sources.py

```
#!/usr/bin/env python

# Things that this file offers:
# or_ips: a list of all onion router IPs (including dirservers)
# client_ips: a list of all client IPs
# nicknames: a dictionary of IP->nickname and nickname->IP mappings
# hostnames: a dictionary of IP->hostname and hostname->IP mappings

import os

nick_list = os.path.join("..", "data", "nicknames.txt")
ornode_list = os.path.join("..", "data", "ornodes.txt")
client_list = os.path.join("..", "data", "clients.txt")
hostname_list = os.path.join("..", "data", "hostnames.txt")

f = open(ornode_list)
or_ips = [ line.strip() for line in f.readlines() ]
f.close()

f = open(client_list)
client_ips = [ line.strip() for line in f.readlines() ]
f.close()

f = open(nick_list)
nicknames = {}
for line in f.readlines():
    fields = line.split()
    nicknames[fields[1]] = fields[2]
    nicknames[fields[2]] = fields[1]
f.close()

f = open(hostname_list)
hostnames = {}
for line in f.readlines():
    fields = line.split()
    hostnames[fields[0]] = fields[1]
    hostnames[fields[1]] = fields[0]
f.close()
```

6. make_answers.py

```
#!/usr/bin/env python

import TorCtl
import socket
import time

PORT = 9

def get_all_tor_paths():
    # Find out what path the connection is taking through Tor
    t = socket.socket()
    t.connect( ('127.0.0.1', 9100) )
    conn = TorCtl.get_connection(t)
    conn.authenticate("")

    # Get the appropriate circuit ID for our stream
    inf = conn.get_info("stream-status")
    circids = {} # Keys: targets Vals: circIDs
    for line in inf['stream-status'].splitlines():
        line = line.strip()
        _, _, id, target = line.split()
        if target.endswith(":" + str(PORT)):
            circids[target.split(':')[0]] = id

    paths = []
    initiator = socket.gethostname()
    circinf = conn.get_info("circuit-status")
    for line in circinf['circuit-status'].splitlines():
        line = line.strip()
        id, _, path = line.split()
        path = path.split(',')
        for (target, circid) in circids.items():
            if id == circid:
                target_hostname = socket.gethostbyaddr(target)[0]
                paths.append([initiator] + path + [target_hostname])
    return paths


paths = []
while len(paths) != 1:
    time.sleep(10)
    paths = get_all_tor_paths()


for path in paths:
    fname = "/mnt/answers/" + ",".join(path)
    f = open(fname, 'w')
    f.close()


print ",".join(paths[0])
```

7. makewin.py

```
#!/usr/bin/env python


import sys
import pcapy
import getopt
import math
from impacket.ImpactDecoder import EthDecoder
from timestamp import TimeStamp, posinf, neginf


def new_make_windows(pcap_filename, winsize, filterstr, start, end):
    rdr = pcapy.open_offline(pcap_filename)
    rdr.setfilter(filterstr)

    # We assume that any packets being read are from Ethernet.
    # Change this line to use a different decoder.
    decoder = EthDecoder()
    num_buckets = int(math.ceil((end - start).to_float() /
                      winsize.to_float()))
    buckets = [0]*num_buckets

    while True:
        try:
            packet_header, packet_data = rdr.next()
            packet_ts = TimeStamp(packet_header.getts())
            if packet_ts >= start and packet_ts <= end:
                i = (packet_ts - start).idiv(winsize)
                buckets[i] = buckets[i] + 1
            else:
                continue
        except:
            break

    return buckets

def old_make_windows(pcap_filename, winsize, filterstr, start, end):
    rdr = pcapy.open_offline(pcap_filename)
    rdr.setfilter(filterstr)

    # We assume that any packets being read are from Ethernet.
    # Change this line to use a different decoder.
    decoder = EthDecoder()

    # Seek to the period of time we're interested in
    packet_header, packet_data = rdr.next()
    start_ts = TimeStamp(packet_header.getts())
    while start_ts < start:
        packet_header, packet_data = rdr.next()
        start_ts = TimeStamp(packet_header.getts())
    # Get our baseline timestamp
    start_ts = TimeStamp(packet_header.getts())
    end_ts = start_ts + winsize
    packet_list = [1] # put the first packet in the first window

    while True:
        try:
            packet_header, packet_data = rdr.next()
            packet_ts = TimeStamp(packet_header.getts())
            if packet_ts >= start and packet_ts <= end:
                if packet_ts >= start_ts and packet_ts < end_ts:
                    packet_list[-1] += 1
                elif packet_ts < start_ts:
                    sys.stderr.write("ERROR! Packet read out of order: %f < %f"
                                     % (packet_ts, start_ts))
                    sys.exit(1)
                elif packet_ts >= end_ts:
                    # Increment the window and add this packet to it
                    packet_list.append(1)
                    start_ts = end_ts
                    end_ts = start_ts + winsize
            else:
                # Skip this packet
                continue
        except: # Reader throws an exception at EOF
            break

    return packet_list



def make_windows(pcap_filename, winsize=TimeStamp((10,0)), filterstr="",
                 start=neginf, end=posinf):
    if start == neginf or end == posinf:
        return old_make_windows(pcap_filename, winsize, filterstr, start, end)
    else:
        return new_make_windows(pcap_filename, winsize, filterstr, start, end)


def usage(errstr = ""):
    if errstr != "": sys.stderr.write("ERROR: " + errstr + "\n")
    sys.stderr.write("usage: %s [OPTION] <pcap file>" % sys.argv[0])
    sys.stderr.write("""
OPTION is one of:
    -f, --filter: a BPF-style filter string, used to select packets.
    -s, --window-secs: what size window to use, in seconds (default is 10).
    -u, --window-usecs: what size window to use, in microseconds
    -o, --output: file to write the data to. If not specified, stdout will
                  be used.\n""")


if __name__ == "__main__":
    # Parse options
    try:
        opts, args = getopt.getopt(sys.argv[1:], "s:u:f:o:",
            ["window-secs=" "window-usecs=", "filter=", "output="])
    except getopt.GetoptError:
        usage("Unable to parse command line options.")
        sys.exit(2)

    sopts = [ o[0] for o in opts ]
    if "-s" in sopts or "--window-secs" in sopts:
        if "-u" in sopts or "--window-usecs" in sopts:
            usage("Only one of seconds or microseconds may be specified")
            sys.exit(2)

    winsize = 10.0
    filter = ""
    output_file = ""
    for (opt, arg) in opts:
        if opt in ("-s", "--window-secs"):
            winsize = float(arg)
        elif opt in ("-u", "--window-usecs"):
            winsize = (0, int(arg))
        elif opt in ("-f", "--filter"):
            filter = arg
        elif opt in ("-o", "--output"):
            output_file = arg
        else:
            usage("Unrecognized command line option.")
            sys.exit(2)

    # Make sure we have a pcap file
    if len(args) != 1:
        usage("Must specify a PCAP file.")
        sys.exit(2)

    if output_file != "":
        outfp = open(output_file, "w")
    else:
        outfp = sys.stdout

    # Convert the window size to a TimeStamp
    winsize = TimeStamp(winsize)

    wins = make_windows(args[0], winsize, filter)
    for i in range(len(wins)):
        outfp.write("%d,%d\n" % (i, wins[i]))

    outfp.close()
```

8. netstat.py

```
#!/usr/bin/env python

import socket
```

```
def int_to_bytes(itg):
    s = ""
    s += (chr(itg & 0x000000FF) + chr((itg >> 8) & 0x0000FF) +
          chr((itg >> 16) & 0x00FF) + chr(itg >> 24))
    return s

def packed_to_addr(packed):
    ip, port = packed.split(':')
    ip = socket.inet_ntoa(int_to_bytes(int(ip, 16)))
    port = int(port, 16)
    return (ip, port)

def get_open_connections():
    open_connections = []
    f = open("/proc/net/tcp")
    f.readline()
    for line in f.readlines():
        fields = line.split()
        src = fields[1]
        dst = fields[2]
        open_connections.append((packed_to_addr(src), packed_to_addr(dst)))
    f.close()
    return open_connections
```

9. pcap_utils.py

```
#!/usr/bin/env python

import pcapy
from timestamp import TimeStamp
from impacket.ImpactDecoder import EthDecoder

def get_start_ts(pcap_file, filterstr=""):
    rdr = pcapy.open_offline(pcap_file)
    rdr.setfilter(filterstr)
    pkh, pkd = rdr.next()
    return TimeStamp(pkh.getts())

def get_end_ts(pcap_file, filterstr=""):
    rdr = pcapy.open_offline(pcap_file)
    rdr.setfilter(filterstr)
    end_ts = None
    while 1:
        try:
            pkh, pkd = rdr.next()
            end_ts = TimeStamp(pkh.getts())
        except:
            break
    return end_ts

def get_intersection(pcaps):
    """finds a window of time shared by all pcap files in the list
    pcaps: list of (pcapfile, filterstring) pairs
    returns: a pair of TimeStamps"""
    starts = [ get_start_ts(pcapfile, filterstr)
               for (pcapfile, filterstr) in pcaps ]
    ends = [ get_end_ts(pcapfile, filterstr)
             for (pcapfile, filterstr) in pcaps ]
    maxstart = max(starts)
    minend = min(ends)
    assert maxstart < minend
    return (maxstart, minend)

def packet_count(pcap_file, filterstr=""):
    count = 0
    rdr = pcapy.open_offline(pcap_file)
    rdr.setfilter(filterstr)
    while 1:
        try:
            rdr.next()
            count += 1
        except:
            break
    return count


def is_empty(pcap_file, filterstr=""):
    return packet_count(pcap_file, filterstr) == 0


def get_all_ips(pcap_files, filterstr=""):
    ips = []
    for pcap_file in pcap_files:
        dec = EthDecoder()
        rdr = pcapy.open_offline(pcap_file)
        rdr.setfilter(filterstr)
        while 1:
            try:
                pkh, pkd = rdr.next()
                ethdecoded = dec.decode(pkd)
                if ethdecoded.child().ethertype == 2048:
                    ethdecoded = ethdecoded.child()
                    ipsrc = ethdecoded.get_ip_src()
                    ipdst = ethdecoded.get_ip_dst()
                    if ipsrc not in ips: ips.append(ipsrc)
                    if ipdst not in ips: ips.append(ipdst)
            except:
                break
    return ips
```

10. tcpdump_server.py

```
#!/usr/bin/env python


import SocketServer
import pcapy
import socket
import time

PORT = 10000

class TCPDumpRequestHandler(SocketServer.StreamRequestHandler):
    def handle(self):
        print "Connection from", self.client_address
        while True:
            command = self.rfile.readline().strip()
            command = command.split('|')
            if command[0].upper() == 'CAPTURE':
                filterstr = command[1]
                capturetime = int(command[2])
                captureid = command[3]
                filename = self.capture(captureid, capturetime, filterstr)
                self.wfile.write(filename +"\n")
            elif command[0] == 'QUIT':
                return
            else:
                self.wfile.write("500 Unrecognized.\n")
    def capture(self, captureid, capturetime=300, filterstr=""):
        rdr = pcapy.open_live("eth0", 96, 0, 0)
        rdr.setfilter(filterstr)
        start = time.time()
        ofname = socket.gethostname() + "-" + str(captureid) + ".pcap"
        ofname = "/mnt/tcpdumps/" + ofname
        dumper = rdr.dump_open(ofname)

        while time.time() < start + capturetime:
            pkh, pkd = rdr.next()
            dumper.dump(pkh, pkd)
        return ofname


server = SocketServer.ThreadingTCPServer(("", PORT), TCPDumpRequestHandler)
server.serve_forever()
```

11. timestamp.py

```
#!/usr/bin/env python


class TimeStamp:
    def __init__(self, ts, infinity=0):
        self.infinity = infinity
        if type(ts) == type(()):
            self.seconds = ts[0]
            self.useconds = ts[1]
            if self.useconds > 1000000:
                self.seconds = self.seconds + (self.useconds / 1000000)
                self.useconds = self.useconds % 1000000
            elif self.useconds < 0:
                while self.useconds < 0:
                    self.seconds -= 1
                    self.useconds = 1000000 + self.useconds
        elif type(ts) == type(.1):
            self.seconds = int(ts)
            if ts < 1:
                self.useconds = int(ts * 1000000)
            else:
                self.useconds = int( (ts % int(ts)) * 1000000)
    def to_float(self):
        secs = float(self.seconds)
        usecs = float(self.useconds)
        return float(secs + (usecs / float(1000000)))
    def __cmp__(self, other):
        if self.infinity > 0:
            if other.infinity > 0:
                return 0
            else:
                return 1
        elif self.infinity < 0:
            if other.infinity < 0:
                return 0
            else:
                return -1
        elif other.infinity > 0:
            if self.infinity > 0:
                return 0
            else:
                return -1
        elif other.infinity < 0:
            if self.infinity < 0:
                return 0
            else:
                return 1
        else: # Both timestamps are finite
            secdiff = self.seconds - other.seconds
            if secdiff != 0: return secdiff
            else: return self.useconds - other.useconds
    def __add__(self, other):
        secs = self.seconds + other.seconds
        usecs = self.useconds + other.useconds
        return TimeStamp((secs, usecs))
    def __sub__(self, other):
        secs = self.seconds - other.seconds
        usecs = self.useconds - other.useconds
        return TimeStamp((secs, usecs))
    def __str__(self):
        return str(self.to_float())
    def __repr__(self):
        return str( (self.seconds, self.useconds) )
    def idiv(self, other):
        c = TimeStamp( (0,0) )
        n = 0
        c = c + other
        while c < self:
            n = n + 1
            c = c + other
        return n

posinf = TimeStamp((0,0), infinity=1)
neginf = TimeStamp((0,0), infinity=-1)
```

12. datagen.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#include <unistd.h>

#define SLEEP_MIN 500000
#define SLEEP_MAX 1000000

int socks_connect(char *host, int port) {
    int buflen = 10 + strlen(host);
    int sock = socket(AF_INET, SOCK_STREAM, 0);

    if (sock < 0) {
        perror("socket");
        exit(1);
    }
```

```
struct sockaddr_in socks_server;
struct in_addr socks_addr;
char buf[128];
char rbuf[16];

memset(&buf, 0, sizeof(buf));
memset(&rbuf, 0, sizeof(rbuf));
memset(&socks_server, 0, sizeof(socks_server));
socks_server.sin_family = AF_INET;
socks_server.sin_port = htons(9050);
inet_aton("127.0.0.1", &(socks_server.sin_addr));

int ret;
ret = connect(sock, (struct sockaddr *) &socks_server,
              sizeof(socks_server));
if (ret < 0) {
    perror("connect");
    exit(1);
}

buf[0] = 0x04;
buf[1] = 0x01;
buf[2] = (port & 0xFF00) >> 8;
buf[3] = (port & 0x00FF);
buf[4] = 0x00;
buf[5] = 0x00;
buf[6] = 0x00;
buf[7] = 0x01;
buf[8] = 0x00;
sprintf(&buf[9], "%s", host);

send(sock, buf, buflen, 0);
recv(sock, rbuf, 8, 0);
```

```
    if (rbuf[1] != 0x5a) {
        printf("Socks error: code 0x%02x (%d)", rbuf[1], rbuf[1]);
        exit(1);
    }
    return sock;
}

int main(int argc, char **argv) {
    int s;
    int flag = 1;
    int sleep_delay;
    int i;

    // Seed the PRNG
    srand( (unsigned) time(NULL) );

    s = socks_connect(argv[1], 9);

    // Disable Nagle's algorithm:
    // We want packets to be sent immediately
    // rather then buffering.
    setsockopt(s, IPPROTO_TCP, TCP_NODELAY, (char *) &flag, sizeof(int));



    // Daemonize
    pid_t pid, sid;

    /* Fork off the parent process */
    pid = fork();
    if (pid < 0) {
        exit(EXIT_FAILURE);
    }
    /* If we got a good PID, then
```

```
 * we can exit the parent process. */
if (pid > 0) {
    exit(EXIT_SUCCESS);
}

/* Change the file mode mask */
umask(0);

/* Create a new SID for the child process */
sid = setsid();
if (sid < 0) {
    /* Log any failure here */
    exit(EXIT_FAILURE);
}

/* Change the current working directory */
if ((chdir("/")) < 0) {
        /* Log any failure here */
        exit(EXIT_FAILURE);
}

/* Close out the standard file descriptors */
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);

while(1) {
    // Decide on the number of packets
    // Expected value: 10 packets
    int num_packets = 1;
    while( (rand() % 10) > 0 ) num_packets++;
    //printf("Sending %d packets\n", num_packets);
    for(i = 0; i < num_packets; i++)
        send(s, "BEEF", strlen("BEEF"), 0);
        sleep_delay = (rand() % SLEEP_MAX - SLEEP_MIN) + SLEEP_MIN;
        //printf("Sleeping for %d microseconds\n", sleep_delay);
        usleep(sleep_delay);
    }
    return 0;
}
```

Bibliography

[1] The Anonymizer. http://anonymizer.com/.

[2] Core Security Technologies. http://oss.coresecurity.com/index.html.

[3] Debian GNU/Linux. http://www.debian.org/.

[4] Bridge - LinuxNet. http://linux-net.osdl.org/index.php/Bridge.

[5] Java Anon Proxy. http://anon.inf.tu-dresden.de/index_en.html.

[6] OpenVPN - an Open Source SSL VPN solution by James Yonan. http://openvpn.net/.

[7] Python programming language — official website. http://python.org/.

[8] Tor technical FAQ. http://wiki.noreply.org/noreply/TheOnionRouter/TorFAQ.

[9] User-mode Linux. http://user-mode-linux.sourceforge.net/.

[10] 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.

[11] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, 1981. ISSN 0001-0782.

[12] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a type iii anonymous remailer protocol. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 2, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1940-7.

[13] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, August 2004. http://tor.eff.org/tor-design.pdf.

[14] M. J. Freedman and R. Morris. Tarzan: a peer-to-peer anonymizing network layer. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 193–206, 2002. ISBN 1-58113-612-9.

[15] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. RFC 1928 (Proposed Standard), Mar. 1996. http://www.ietf.org/rfc/rfc1928.txt.

[16] 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, 2004.

[17] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *IEEE Symposium on Security and Privacy*. IEEE CS, May 2005.

[18] J. F. Raymond. Traffic Analysis: Protocols, Attacks, Design Issues, and Open Problems. In H. Federrath, editor, *Designing Privacy Enhancing Technologies: Workshop on Design Issue in Anonymity and Unobservability*, pages 10–29. Springer-Verlag, LNCS 2009, July 2000.

[19] M. Rennhard and B. Plattner. Introducing morphmix: peer-to-peer based anonymous internet usage with collusion detection. In *WPES '02: Proceedings of the 2002 ACM workshop on Privacy in the Electronic Society*, pages 91–102, 2002. ISBN 1-58113-633-1.

[20] 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.

[21] Y. Zhu, X. Fu, B. Graham, R. Bettati, and W. Zhao. On flow correlation attacks and countermeasures in mix networks. In *Proceedings of Privacy Enhancing Technologies workshop (PET 2004)*, volume 3424 of *LNCS*, May 2004.