David M. Goldschlag, Michael G. Reed, and Paul F. Syverson. "Hiding Routing Information," Workshop on Information Hiding, Cambridge, UK, May, 1996.

Hiding Routing Information

David M. Goldschlag, Michael G. Reed, and Paul F. Syverson

Naval Research Laboratory, Center For High Assurance Computer Systems, Washington, D.C. 20375-5337, USA, phone: +1 202.404.2389, fax: +1 202.404.7942, e-mail: {*last name*}@itd.nrl.navy.mil.

Abstract. This paper describes an architecture, *Onion Routing*, that limits a network's vulnerability to traffic analysis. The architecture provides anonymous socket connections by means of proxy servers. It provides real-time, bi-directional, anonymous communication for any protocol that can be adapted to use a proxy service. Specifically, the architecture provides for bi-directional communication even though no-one but the initiator's proxy server knows anything but previous and next hops in the communication chain. This implies that neither the respondent nor his proxy server nor any external observer need know the identity of the initiator or his proxy server. A prototype of *Onion Routing* has been implemented. This prototype works with HTTP (World Wide Web) proxies. In addition, an analogous proxy for TELNET has been implemented. Proxies for FTP and SMTP are under development.

1 Introduction

This paper presents an architecture that limits a network's vulnerability to traffic analysis. We call this approach *Onion Routing*, because it relies upon a layered object to direct the construction of an anonymous, bi-directional, real-time virtual circuit between two communicating parties, an *initiator* and *responder*. Because individual *routing nodes* in each circuit only know the identities of adjacent nodes (as in [1]), and because the nodes further encrypt multiplexed virtual circuits, studying traffic patterns does not yield much information about the paths of messages. This makes it difficult to use traffic analysis to determine who is communicating with whom.

Onion Routing provides an anonymous socket connection through a proxy server. Since proxies are a well defined interface at the application layer [12, 11], and many protocols have been adapted to work with proxy servers in order to accommodate firewalls, Onion Routing can be easily used by many applications. Our prototype works with HTTP (World Wide Web) proxies. In addition, a proxy for TELNET has been implemented.

Traffic analysis can be used to help deduce who is communicating with whom by analyzing traffic patterns instead of the data that is sent. For example, in most networks, it is relatively easy to determine which pairs of machines are communicating by watching the routing information that is part of each packet. Even if data is encrypted, routing information is still sent in the clear because routers need to know packets' destinations, in order to route them in the right direction. Traffic analysis can also be done by watching particular data move through a network, by matching amounts of data, or by examining coincidences, such as connections opening and closing at about the same time.

Onion Routing hides routing information by making a data stream follow a path through several nodes en route to its destination. The path is defined by the first node, which is also a proxy for the service being requested (e.g., HTTP requests). Therefore, this Proxy/Routing Node is the most sensitive one, so sites that are concerned about traffic analysis should also manage a Proxy/Routing Node. We will see later that it is important that this Proxy/Routing Node also be used as an intermediate routing node in other virtual circuits. Although the compromise of all routing nodes compromises the hiding, one uncompromised routing node is sufficient to complicate traffic analysis. Figure 1 illustrates the topology of an Onion Routing network with five nodes, one of which (W) is the Proxy/Routing node for the initiator's site.

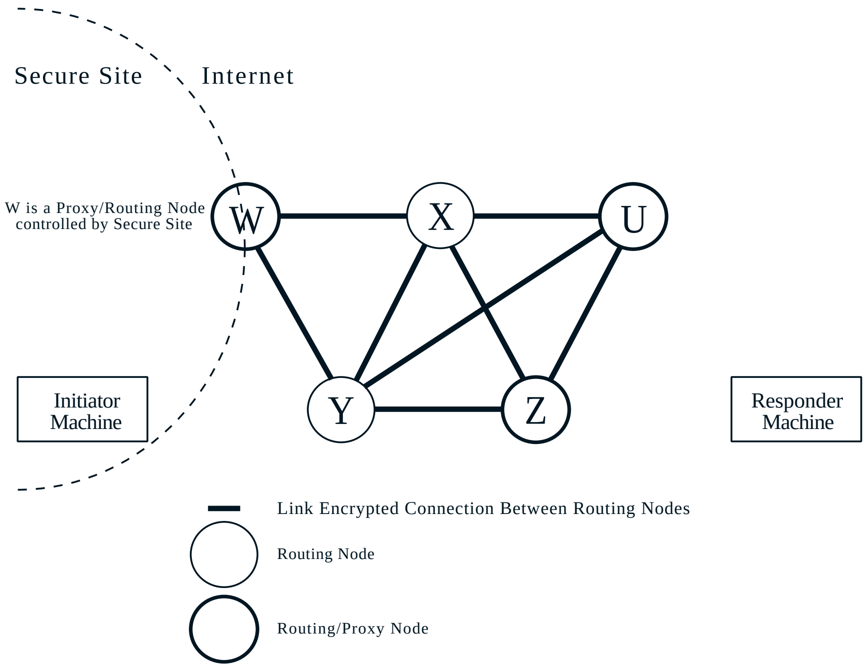

Fig. 1. Routing Topology.

The goal of Onion Routing is not to provide anonymous communication. Parties are free to (and usually should) identify themselves within a message. But the use of a public network should not automatically give away the identities and locations of the communicating parties. For example, imagine a researcher who uses the World Wide Web to collect data from a variety of sources. Although each piece of information that he retrieves is publicly known, it may be possible for an outside observer to determine his sensitive interests by studying the patterns in his requests. Onion Routing makes it very difficult to match his HTTP requests to his site.

Anonymous re-mailers [5, 6] attempt to limit the feasibility of traffic analysis by providing an anonymous store and forward architecture. To prevent replay attacks, re-mailers keep a log of sent messages. These two characteristics make the anonymous re-mailer approach unsuitable for HTTP applications, as HTTP requests would both generate an enormous log and require bi-directional communication. Anonymous ISDN [8] has even more severe real-time and bi-directional requirements than HTTP, but, the architecture of an ISDN network is considerably different from the architecture of the Internet [4].

Onion Routing provides bi-directional communication, without requiring that the responder know the initiator's identity or location. Individual messages are not logged. In addition, Onion Routing is easily adapted to electronic mail. Messages can include *Reply Onions* that permit a later reply to the sender without knowing his address and without keeping the original virtual circuit open.

The rest of the paper is organized in the following way: Section 2 presents background information. Section 3 describes the *Onion*, the object that directs the construction of the virtual circuit. Section 4 describes the construction and use of these virtual circuits. Section 5 describes the vulnerabilities in the Onion Routing architecture. Section 6 presents some concluding remarks.

2 Background

Chaum [1] defines a layered object that routes data through intermediate nodes, called *mixes*. These intermediate nodes may reorder, delay, and pad traffic to complicate traffic analysis. Some work has been done using mixes in ATM networks [3].

Anonymous Remailers like [5, 6] use mixes to provide anonymous e-mail services and also to invent an address through which mail can be forwarded back to the original sender. Remailers work in a store and forward manner at the mail application layer, by stripping off headers at each mix, and forwarding the mail message to the next mix. These remailers provide confirmation of delivery.

In [8], mixes are used to provide untraceable communication in an ISDN network. In a phone system, each telephone line is assigned to a particular local switch (i.e., local exchange), and switches are interconnected by a (long distance) network. Anonymous calls in ISDN rely upon an anonymous connection within each switch between the caller and the long distance network, which is obtained by routing calls through a predefined series of mixes. The long distance endpoints of the connection are then mated to complete the call. (Notice that observers can tell which local switches are connected.) This approach relies upon two unique features of ISDN switches. Since each phone line has a subset of the switch's total capacity pre-allocated to it, there is no (real) cost associated with keeping a phone line active all the time, either by making calls to itself, to other phone lines on the same switch, or to the long distance network. Keeping phone lines active complicates traffic analysis because an observer cannot track coincidences.

Also, since each phone line has a control circuit connection to the switch, the switch can broadcast messages to each line using these control circuits. So, within a switch a truly anonymous connection can be established: A phone line makes an anonymous connection to some mix. That mix broadcasts a token identifying itself and the connection. A recipient of that token can make another anonymous connection to the specified mix, which mates the two connections to complete the call.

Our goal of anonymous socket connections over the Internet differs from anonymous remailers and anonymous ISDN. The data is different, with real-time constraints more severe than mail, but somewhat looser than voice. Both HTTP and ISDN connections are bidirectional, but, unlike ISDN, HTTP connections are likely to be small requests followed by short bursts of returned data. In a local switch capacity is pre-allocated to each phone line, and broadcasting is efficient. But broadcasting over the Internet is not free, and defining broadcasts domains is not trivial. Most importantly, the network topology of the Internet is more akin to the network topology of the long distance network between switches, where capacity is a shared resource. In anonymous ISDN, the mixes hide communication within the local switch, but connections between switches are not hidden. This implies that all calls between two businesses, each large enough to use an entire switch, reveal which businesses are communicating. In Onion Routing, mixing is dispersed throughout the Internet, which improves hiding.

3 Onions

To begin a session between an initiator and a responder, the initiator's proxy identifies a series of routing nodes forming a route through the network and constructs an *onion* which encapsulates that route. Figure 2 illustrates an onion constructed by the initiator's Proxy/Routing Node W for an anonymous route to the responder's Proxy/Routing Node Z through intermediate routing nodes X and Y. The initiator's proxy then sends the onion along that route to establish a virtual circuit between himself and the responder's proxy.

The onion data structure is composed of layer upon layer of encryption wrapped around a payload. Leaving aside the shape of the payload at the very center, the basic structure of the onion is based on the route to the responder that is chosen by the initiator's proxy. Based on this route, the initiator's proxy encrypts first for the responder's proxy, then for the preceding node on the route, and so on back to the first routing node to whom he will send the onion. When the onion is received, each node knows who sent him the onion and to whom he should pass the onion. But, he knows nothing about the other nodes, nor about how many there are in the chain or his place in it (unless he is last). What a

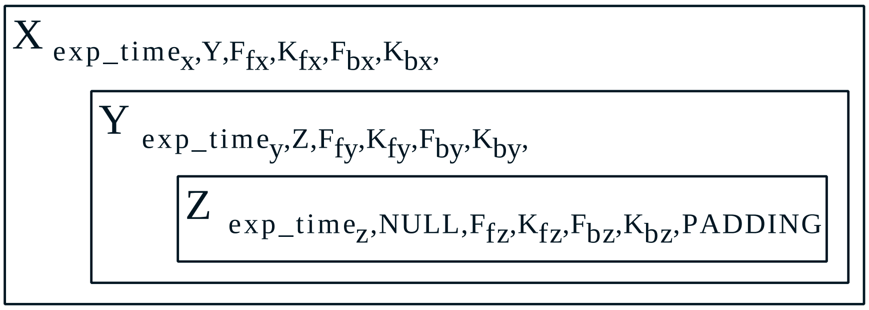

Fig. 2. A Forward Onion.

node P_x receives looks like this

$$\{exp\_time, next\_hop, F_f, K_f, F_b, K_b, payload\}_{PK_x}$$

Here PK_x is a public encryption key for routing node P_x, who is assumed to have the corresponding decryption key.[1] The decrypted message contains an expiration time for the onion, the next routing node to which the payload is to be sent, the payload, and two function/key pairs specifying the cryptographic operations and keys to be applied to data that will be sent along the virtual circuit. The forward pair (F_f, K_f) is applied to data moving in the forward direction (along the route that the onion is traveling) the backward pair (F_b, K_b) is applied to data moving in the opposite direction (along the onion's reverse route).[2] (If the receiving node is the responder's proxy, then the *next_hop* field is *null*.) For any intermediate routing node the payload will be another onion. The expiration time is used to detect replays, which pairs of compromised nodes could use to try to correlate messages. Each node holds a copy of the onion until *exp_time*. If he receives another copy of the same onion within that time he simply ignores it. And, if he receives an onion that has expired, he ignores that as well.

Notice that at each hop the onion shrinks as a layer is peeled off. To avoid compromised nodes inferring route information from this monotonically diminishing size, a random bit string the size of the peeled off layer is appended to the end of the *payload* before forwarding. No proxy except the last will know how much of the *payload* he receives is such padding because he won't know where

[1] Depending on certain assumptions about the fields in each onion layer, a naive RSA implementation of the simple public key encryption implied by our notation could be vulnerable to an attack as described in [7]. In our implementation, this potential vulnerability is illusory since the public key is only used to encrypt a secret key, and that secret key is used to encrypt the remainder of the message using an efficient symmetric algorithm. This also makes for a more efficient implementation than the simple, straightforward implementation using only public keys.

[2] Specifying two pairs of functions unifies the virtual circuits that are constructed by forward and reply onions. See section 3.3.

he is in the chain. He simply 'decrypts' the padding along with the rest of the onion. Even a constant size onion might be traced unless all onions are the same size, so we fix the size of the onion. To maintain this constant size to hide the length of the chain from the responder's proxy, the initiator's proxy will pad the central *payload* according to the size of the onion, i.e., the number of hops. So, when any onion arrives at the responder's proxy it will always have the same amount of padding, either added initially or en route.

3.1 Creating the circuit

The goal in sending the onion is to produce virtual circuits within link encrypted connections already running between routing nodes.[3] More details will be given in section 4. An onion occurs as the data field in one of the presently described 'messages'. Such messages contain a circuit identifier, a command (*create*, *destroy*, and *data*), and data. Any other command is considered an error, and the node who receives such a message ignores that message except to return a *destroy* command back through that virtual circuit. The *create* command accompanies an onion. When a node receives a create command along with an onion, he chooses a virtual circuit identifier and sends another *create* message containing this identifier to the next node and the onion (padded with his layer peeled off). He also stores the virtual circuit identifier he received and virtual circuit identifier he sent as a pair. Until the circuit is destroyed, whenever he receives data on the one connection he sends it off on the other. He applies the forward cryptographic function and key (obtained from the onion) to data moving in the forward direction (along the route the onion traveled) and the backward cryptographic function and key to data moving in the opposite direction (along the onion's reverse route). The virtual circuit established by the onion in figure 2 is illustrated in figure 3:

Data sent by the initiator over a virtual circuit is "pre-crypted"[4] repeatedly by his proxy by applying the inverse of all the forward cryptographic operations specified in the onion, innermost first. Therefore, these layers of cryptography will be peeled off as the data travels forward through the virtual circuit. Data sent by the responder is "crypted" once by his proxy and again by each previous node in the virtual circuit using the backward cryptographic operation specified at the corresponding layer of the onion. The initiator's proxy applies the inverse of the backward cryptographic operations specified in the onion, outermost first, to this stream, to obtain the plaintext.

3.2 Loose Routing

It is not necessary that the entire route be prespecified by the initiator's proxy. He can instruct various nodes along the route to choose their own route to the

[3] Onions could be used to carry data also, but since onions have to be tracked to prevent replay, this would introduce a large cost.

[4] We define the verb *crypt* to mean the application of a cryptographic operation, be it encryption or decryption, where the two are logically interchangeable.

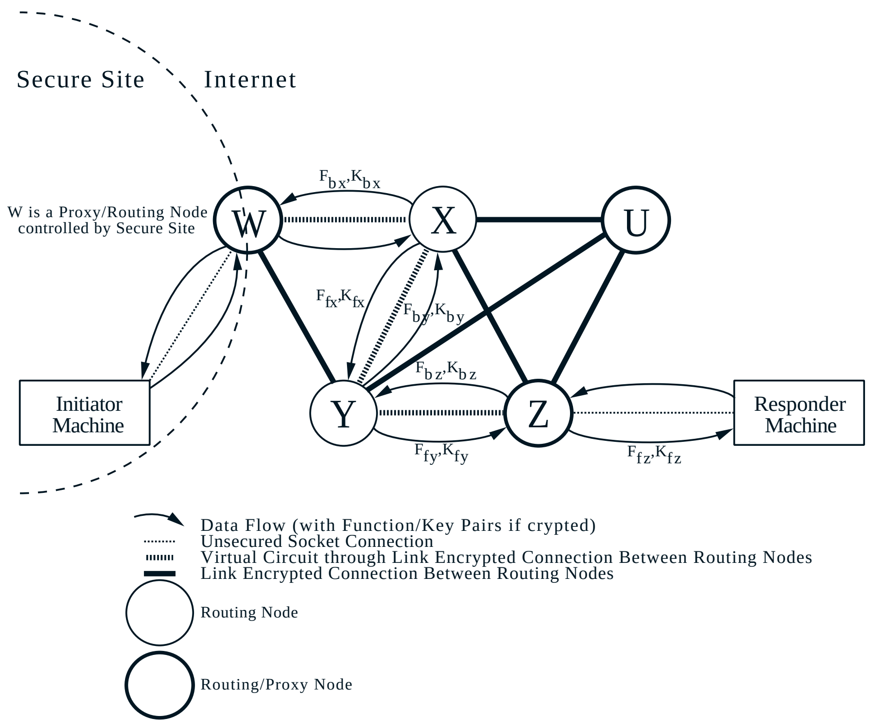

Fig. 3. A Virtual Circuit.

next prespecified node. This can be useful for security, adding more hops to the chain. It could also be used if the initiating proxy does not know a complete, connected route to the responder but believes that the node where any break occurs can construct a route to the next node. Or, loose routing can be used to handle connection changes that occur of which the initiator was unaware. Also, since onions are all of fixed size, there is a fixed maximum length to the route from the initiator's proxy to the responder's proxy. Loose routing allows us to increase the size of that maximum for the same fixed onion size. Why this is so should become clear presently.

It is also possible to iterate the loose routing process, allowing nodes on the added route to themselves add to the chain. Obviously, we need a mechanism to prevent the chain from lengthening indefinitely. This can be incorporated into the onion structure. An onion for a system that allows for loose routing is as follows:

$$\{exp\_time, next\_hop, max\_loosecount, F_f, K_f, F_b, K_b, payload\}_{PK_x}$$

If the node receiving this onion decides to loose-route the onion, he prepares a new onion with up to *max_loosecount* layers. The payload of this onion is simply the onion he received with PK_x changed for the last (innermost) node he added to the chain. In other words, he behaves as an initiator's proxy except that his payload is itself already an onion. (This node behaves like an initiator's proxy with respect to data also, since he must repeatedly pre- and post- crypt data that moves along the diverted route.) To keep the onion a constant length he must truncate the payload by an amount commensurate with the layers he has added to the onion. The initiating proxy must anticipate the amount of padding (both present initially and any added and/or truncated en route) that will be on the central payload at the time loose routing occurs to allow for this truncation. Failure to pre-pad correctly or ignoring an onion's fixed size will result in a malformed onion later in the route. The total of the *max_loosecount* values occurring in the added layers plus the number of added layers must be less than or equal to the *max_loosecount* value that the adding node received.

3.3 Reply Onions

There are applications in which it would be useful for a responder to send back a reply after the original circuit is broken. This would allow answers (like e-mail replies) to be sent to queries that were not available at the time of the original connection. As we shall see presently, this also allows the responder as well as the initiator to remain hidden. The way we allow for these delayed replies is by sending a reply onion to accompany the reply. Like the forward onion, it reveals to each node en route only the next step to be taken. It has the same structure as the forward onion and is treated the same way by nodes en route. Intermediate nodes processing an onion cannot differentiate between forward and reply onions. Furthermore, the behavior of the original initiator and responder proxies are the same, once the circuit is formed.

The primary difference between a forward and a reply onion is the innermost payload. The payload of the forward onion can be effectively empty (containing only padding). The reply onion payload contains enough information to enable the initiator's proxy to reach the initiator and all the cryptographic function and key pairs that are to crypt data along the virtual circuit. The initiator's proxy retrieves the keys from the onion. Figure 4 illustrates a reply onion constructed by the initiator's Proxy/Routing Node W for an anonymous route back to him starting at the responder's Proxy/Routing Node Z through intermediate routing nodes Y and X:

There is no difference between virtual circuits established by reply onions and forward onions, except that in circuits established by reply onions intermediate routing nodes appear to think that forward points toward the initiator's proxy. But since the behavior of intermediate routing nodes is symmetric, this difference is irrelevant. The terminal Proxy/Routing nodes, however, have the same behavior in circuits established by forward and reply onions. Therefore, a figure of the virtual circuit formed by the reply onion illustrated in figure 4 would be identical to the virtual circuit illustrated in figure 3 even though the circuit was formed by the reply onion moving from the responder's proxy node to the

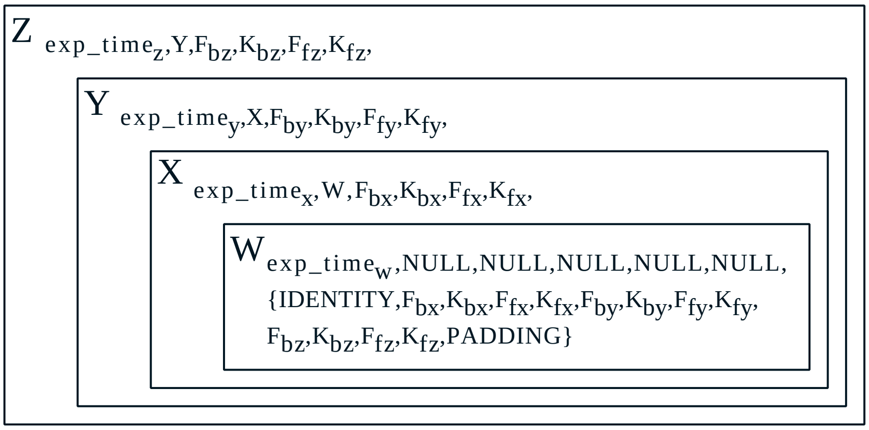

Fig. 4. A Reply Onion.

initiator's proxy node. Internally to the intermediate nodes, the forward cryptographic functions are applied to data moving in the direction that the circuit was established, and the backward cryptographic functions are applied to data moving in the opposite direction. The location of the terminal Proxy/Routing Nodes are in this sense reversed, with the initiator's proxy at the end of the circuit and the responder's proxy at the beginning of the circuit. However, the behavior of the initiator and responder proxies is identical to their behavior in the virtual circuit formed by a forward onion. This is the reason for having forward and backward function/key pairs at each layer of the onion.

Like a forward onion, a reply onion can only be used once. When a node receives an onion it is kept until it expires, and any onion received is compared to detect replay. If a replay is detected, it is treated as an error and ignored. Since reply onions can only be used once, if multiple replies are desired, multiple reply onions must be sent. Of course, they need not all follow the same return route; although they may. If replies are only likely to be forthcoming if they are anonymous, one or more reply onions can be broadcast. Anyone can then reply with an unused onion. If he can maintain anonymity from or in cooperation with the responder's proxy for that reply onion, then he can do so anonymously.

4 Implementation

The easiest way to build our system without requiring the complete redesign and deployment of new client and server software is to make use of existing proxy technologies. Historically, proxy technologies have been used to create tunnels through a firewall. The use of proxy technologies requires that the client applications be 'proxy aware'. The widespread deployment of firewalls on the Internet has created the demand for such proxy aware applications, which software manufacturers are rushing to meet.

In the firewall setting, a system administrator will set up a proxy server on the firewall machine which will be responsible for forwarding requests from the protected domain out onto the open Internet, and maintain a return path for the response to the request. A proxy server can be divided into two parts: the front end that receives and parses the request, and the back end that processes the request and returns the results back to the requester. Classically, the front and back ends are the same process running on one machine.

Under our system we will use a traditional proxy front end and back end, but, they will be separate processes on separate machines with a tunnel connecting them. In this manner, our Proxy/Routing Nodes will look no different to the client and server software than any other proxy server. A couple of assumptions will hold for the remainder of this paper: 1) Proxy/Routing Nodes and intermediate routing nodes know about each other in advance of their operation, and 2) public key certificates for each node have been securely distributed to all others prior to operation.

All nodes are connected by link encrypted connections which multiplex many virtual circuits between initiator and responder proxy nodes. These connections are link encrypted in an odd way (for efficiency). All messages moving through these connections are of fixed size and have two components, header and payload fields. Header fields contain the virtual circuit identifier and the command and are link encrypted using a stream cipher [10]. Since all payload fields will be encrypted via other mechanisms (public keys or onion keys), they need not be link encrypted.

There are three commands that nodes understand. The first is to *create* a virtual circuit. At each node, a virtual circuit has two connections. Data arriving on one is passed along on the other. The circuit is defined by the labels for these two connections. Creating a virtual circuit is the process of defining these labels for each node along the route. For the first Proxy/Routing Node, one connection is a link to the initiator, and the other is a link to the next routing node. The Proxy/Routing Node creates an onion defining the sequence of intermediate routing nodes to the responder's Proxy/Routing Node. It breaks the onion up into payload sized chunks and transmits these chunks in order to the next node with a control field containing both the label of the connection and a *create* command. Each subsequent node reassembles the onion and peels off a layer from the onion which reveals the next node in the route and two cryptographic function/key pairs. Before acting on the *create* command, the node checks whether the onion has expired or is a replay. To check for replay, the node consults a table of unexpired onions. If the onion is valid, it is inserted into the table, and the node then labels a new connection to the next node and passes the peeled and padded onion in a similar sequence of messages to the next node. It also updates a table containing the labels and cryptographic function/key pairs associated with the new virtual circuit. The appropriate (forward or backward) function/key pair should be used to crypt data moving along that circuit. The responder's Proxy/Routing Node, recognizing that the onion is empty, will partially update its tables. As with standard proxies the next *data* message along this circuit will identify the responder.

The second command is *data*. The second role of the initiator's Proxy/Routing Node is to pass a stream of data from the initiator along the virtual circuit together with other control information for the responder's Proxy/Routing Node. To do this, he breaks the incoming stream into (at most) payload sized chunks, and repeatedly pre-crypts each chunk using the inverse of the cryptographic operations specified in the onion, innermost first. The function/key pairs that are applied, and the virtual circuit identifier of the connection to the next node are obtained from a table. The header field for each payload is the label of the connection and a *data* command. Each subsequent node looks at its table, obtaining the cryptographic function/key pair associated with the circuit (for the appropriate direction) and the virtual circuit identifier of the connection to the next node. It then peels off a layer of cryptography and forwards the peeled payload to the next node. Once the data reaches the responder's proxy, its final cryption will produce the plaintext that is to be processed or forwarded to the responder.

The *data* command can also be used to move data from the responder's Proxy/Routing Node to the initiator's Proxy/Routing Node. The responder's Proxy/Routing Node obtains the cryptographic function/key pair and the virtual circuit identifier for the next node from its tables, and crypts the stream. It breaks the crypted stream into payload sized chunks and forwards them to the next node with the appropriate control field. Each subsequent node further stream crypts each payload using the appropriate function/key associated with that virtual circuit. Once a messages arrives at the initiator's Proxy/Routing Node he looks at his table and applies the inverse of the backward cryptographic operations specified in the onion, outermost first, to this stream to obtain the plaintext. The plaintext is forwarded to the initiator.

The third command is *destroy* which is used to tear down a virtual circuit when it is no longer needed or in response to certain error conditions. Notice that *destroy* messages can be initiated by any node along a virtual circuit, and it is a node's obligation to forward the *destroy* messages in the appropriate directions. (A node initiating a *destroy* message in an active virtual circuit forwards it in both directions. A node that receives a *destroy* message passes it along in the same direction.) The payload of a *destroy* command is empty padding. Nonetheless, this payload is still crypted with the appropriate function/key pair. In addition to the *destroy* command, the control field contains the virtual circuit identifier of the recipient of the *destroy* command. Upon receipt of a *destroy* command a node deletes the table entries associated with that virtual circuit.

5 Vulnerabilities

Onion Routing is not invulnerable to traffic analysis attacks. With enough data, it is still possible to analyze usage patterns and make educated guesses about the routing of messages. Also, since our application requires real time communication, it may be possible to detect the near simultaneous opening of socket connections on the first and last proxy servers revealing who is requesting what information. However, these sorts of attacks require the collection and analysis of huge amounts of data by external observers.

Other attacks depend upon compromised Proxy Servers and Routing Nodes. If the initiator's proxy is compromised then all information is revealed. In general it is sufficient for a single routing node to be uncompromised to complicate traffic analysis. However, a single compromised routing node can destroy connections or stop forwarding messages, resulting in denial of service attacks.

Onion Routing uses expiration times to prevent replay attacks. It is curious that, unlike timestamps, the vulnerability due to poorly synchronized clocks here is a denial of service attack, instead of a replay attack. If a node's clock is too fast, otherwise timely onions will appear to have already expired. Also, since expiration times define the window during which nodes must store used onions, a node with a slow clock will end up storing more information.

If the responder's proxy is compromised, and can determine when the unencrypted data stream has been corrupted, it is possible for compromised nodes earlier in the virtual circuit to corrupt the stream and ask which responder's proxy received uncorrupted data. By working with compromised nodes around a suspected initiator's proxy, one can identify the beginning of the virtual circuit. The difficulty with this attack is that once the data stream has been corrupted, it will remain corrupted (because we use a stream cipher), limiting further analysis.

In order for Onion Routing to be effective, there must be significant use of all the nodes, and Proxy Nodes must also be intermediate routing nodes. Choosing the appropriate balance between efficient use of network capacity and security is a hard problem both from a theoretical and practical standpoint. Theoretically, it is difficult to calculate the value of the tradeoff. For more security, network traffic must be relatively constant. This requires sending dummy traffic over a connection when traffic is light and buffering data when traffic is heavy. If traffic is very bursty and response time is important, smoothing out network traffic requires wasting capacity. If however, traffic is relatively constant, additional smoothing may not be necessary. From a practical point of view, the Internet may not provide the control necessary to smooth out traffic: unlike ATM, users do not own capacity on shared connections. The important observation, however, is that Onion Routing forms an architecture within which these tradeoffs can be made and explored.

6 Conclusion

Onion Routing is an architecture that hides routing information while providing real-time, bi-directional communication. Since it provides a virtual circuit that can replace a socket connection, Onion Routing can be used in any protocol that can be adapted to use a proxy service. Although our first use is in HTTP and TELNET, it is easy to imagine other applications. In e-mail, for example, Onion Routing would create an anonymous socket connection between two sendmail daemons. This contrasts with Anonymous Remailers, where each remailer provides a single hop in a chain of mail forwarding. In this sense, in Onion Routing, the rerouting of messages is independent of the type of message.

Other extensions are also possible and integrate nicely with the proxy approach to anonymity. For example, to create a completely anonymous conversation between two parties, each party would make an anonymous connection to some anonymity server, which mates connections sharing some token. This approach, similar to IRC servers, can also be used if the responder does not trust the initiator, especially with (broadcast) reply onions. The responder builds his own (trusted) connection to some anonymity server, and asks that anonymity server to build another connection to the initiator using a reply onion and to mate the two connections. Each party is therefore protected by a route that he determined.

In Onion Routing the encryption burden on connected intermediate nodes is less than the burden of link encryption on routers. In link encryption, each packet is encrypted by the sender and decrypted by the recipient. In Onion Routing the header and payload of each message are crypted separately: the header is encrypted and decrypted using the connection's key, and the payload is crypted (only by the recipient) using the appropriate function/key pair associated with the virtual circuit.

Our goal here is not to provide anonymous communication, but, to place identification where it belongs. The use of a public network should not automatically reveal the identities of communicating parties. If anonymous communication is undesirable, it is easy to imagine filters on the endpoint machines that restrict communication to signed messages.

Onion Routing will only be effective in complicating traffic analysis if its Proxy and Routing Nodes become widespread and widely used. There is an obvious tension between anonymity and law enforcement. If this tension is resolved in favor of law enforcement, it would be straightforward to integrate a key escrow system within the onion, which would make routing information available to the lawful authorities.

7 Acknowledgements

Discussions with many people helped develop the ideas in this paper. We would like to thank Ran Atkinson, Markus Jakobbsen, John McLean, Cathy Meadows, Andy Moore, Moni Naor, Holger Peterson, Birgit Pfitzmann, Michael Steiner, and the anonymous referees for their helpful suggestions.

References

1. D. Chaum. *Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms,* Communications of the ACM, v. 24, n. 2, Feb. 1981, pages 84-88.
2. D. Chaum, *The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability,* Journal of Cryptology, 1/1, 1988, pages 65-75.

3. S. Chuang. *Security Management of ATM Networks,* Ph.D. thesis, in progress, Cambridge University.
4. D. E. Comer. *Internetworking with TCP/IP, Volume 1: Principles, Protocols, and Architecture,* Prentice–Hall, Engelwood Cliffs, New Jersey, 1995.
5. L. Cottrell. *Mixmaster and Remailer Attacks,* http://obscura.obscura.com/~loki/remailer/remailer-essay.html
6. C. Gulcu and G. Tsudik. *Mixing Email with* Babel, 1996 Symposium on Network and Distributed System Security, San Diego, February 1996.
7. A. Pfitzmann and B. Pfitzmann. *How to Break the Direct RSA-implementation of MIXes,* Advances in Cryptology–EUROCRYPT '89 Proceedings, Springer-Verlag, Berlin, 1990, pages 373-381.
8. A. Pfitzmann, B. Pfitzmann, and M. Waidner. *ISDN-Mixes: Untraceable Communication with Very Small Bandwidth Overhead,* GI/ITG Conference: Communication in Distributed Systems, Mannheim Feb, 1991, Informatik-Fachberichte 267, Springer-Verlag, Heildelberg 1991, pages 451-463.
9. A. Pfitzmann and M. Waidner. *Networks Without User Observability,* Computers & Security, 6/2 1987, pages 158-166.
10. B. Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C,* John Wiley and Sons, 1994.
11. W. R. Stevens. *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols,* Addison–Wesley, Reading, Mass., 1996.
12. L. D. Stein. *How to Set up and Maintain a World Wide Web Site: The Guide for Information Providers,* Addison–Wesley, Reading, Mass., 1995.

This article was processed using the LaTeX macro package with LLNCS style