Rebound: Decoy Routing on Asymmetric Routes Via Error Messages

Daniel Ellard, Christine Jones, Victoria Manfredi,
W. Timothy Strayer, Bishal Thapa, and Megan Van Welie
Raytheon BBN Technologies
10 Moulton Street
Cambridge, MA 02138
Email: {dellard,cej,vmanfred,strayer,bthapa,mvanweli}@bbn.com

Alden Jackson
Brookline, MA 02445
Email: awjacks@gmail.com

***Abstract*—Decoy routing is a powerful circumvention mechanism intended to provide secure communications that cannot be monitored, detected, or disrupted by a third party who controls the user's network infrastructure. Current decoy routing protocols have weaknesses, however: they either make the unrealistic assumption that routes through the network are symmetric (i.e., the router implementing the decoy routing protocol must see all of the traffic, in both directions, from each connection it modifies), or their protocol requires modifying the route taken by packets in connections that use the protocol, and these route changes are detectable by a third party. We present *Rebound*, a decoy routing protocol that tolerates asymmetric routes without modifying the route taken by any packet that passes through the decoy router, making it more difficult to detect or disrupt than previous decoy routing protocols.**

***Index Terms*—Decoy Routing, Circumvention, Privacy**

I. Introduction

Decoy routing [15], [18], [20], [21] is motivated by the desire to mitigate developments in network infrastructure that enable network operators or other third parties to filter or monitor access to portions of the Internet.

As shown in Figure 1, users of decoy routing establish ordinary connections to *decoy hosts* that the third party does not block or tamper with. These decoy hosts are ordinary destinations on the Internet, such as popular web sites, that are oblivious to decoy routing. The decoy routing client encodes a signal within each connection to the decoy hosts. If a connection transits a *decoy router*, the router decodes the signal and then uses a handshake protocol embedded in the ordinary traffic of the connection to validate that the connection was initiated by an authorized user. Once validated, the connection is redirected to a proxy service used by the decoy router. This proxy service can be used to reach any service, using any protocol desired, regardless of the protocol used for the original connection. Meanwhile, a third party anywhere on the network between the user and the decoy router only sees what appears to be ordinary traffic to and from the decoy host.

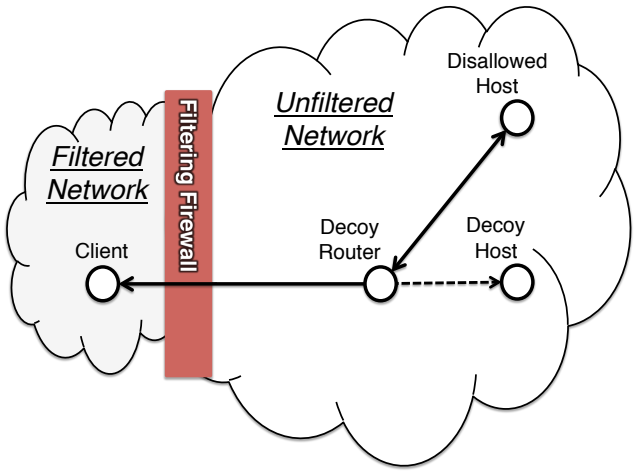

Fig. 1. A high-level illustration of decoy routing. The client accesses the Internet via a *filtered network*, which is controlled by a third party who may monitor, eavesdrop, or block the client communication with specific web sites. That third party can only control or monitor a section of the network; the rest of the Internet is *unfiltered*. Decoy routing provides secure communications that appear to be ordinary Internet connections to ordinary, public web sites (decoy hosts) in the unfiltered Internet. These connections contain hidden signals that are detected and processed by Curveball routers within the network, enabling communications with other hosts.

The essential difference between decoy routing and conventional proxy services [2], [3], [4], [5], [6], [10], covert channels [1], [7], [8], [12], or anonymization tools such as Tor [11], is that decoy routing uses real connections to *allowed* (or *decoy*) hosts and services outside of the filtered area as a conduit for clients within the filtered area to exchange information with *disallowed* hosts and services outside of it. A third party who wishes to block use of a conventional proxy service only needs to block the addresses of each proxy, but a third party who wishes to block use of decoy routing must block the addresses of *all* possible decoy hosts – every host that might be reached

This material is based upon work supported by the Defense Advanced Research Project Agency (DARPA) and Space and Naval Warfare Systems Center Pacific under Contract No. N66001-11-C-4017. The views, opinions, and/or findings contained in this article/presentation are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)

by the client via a route that includes a decoy router.

Some of the early decoy routing protocols made the simplifying assumption that routes between two hosts on the Internet are *symmetric*—that is, traffic takes the same route in both directions. There is significant evidence, however, that the converse is usually true for routes across the Internet. He *et al.* observed that 65% of sampled routes between public traceroute servers have some degree of asymmetry at the AS level [13]. John *et al.* found that asymmetry increases dramatically as the routes use networks closer to the core of the Internet: on two Tier1 ISP backbone links, as many as 96% of the routes were asymmetric because of hot-potato routing [17].

Other earlier decoy routing protocols required that the decoy router forge messages, addressed to the client, that appear to be from the decoy host [21], [18], [20]. This requirement introduces critical vulnerabilities, as we discuss in Section VIII-A.

We present Rebound, a novel decoy routing protocol that only needs to observe packets from the client to the decoy host, and does not forge packets from the decoy host. Rebound conveys *all* messages to the client, securely and privately, via the decoy host; the decoy router never sends messages directly to the client.

The rest of this paper is organized as follows. In Section II, we describe our threat model for decoy routing. In Section III, we give an overview of Rebound, and in Section IV, we give an overview of related work on decoy routing. In Section V, we introduce the fully asymmetric decoy routing protocol that we propose in this paper. In Section VI, we describe how we implement our proposed solution. In Section VII, we present results. In Section VIII, we assess the vulnerabilities to our protocol. Section IX concludes this paper.

II. Decoy Routing Threat Model

Our threat model for decoy routing is that there is a third party that can monitor and control the network to which the client is attached. The goals of this third party may include blocking access to specific web sites, actively modifying the contents returned by these sites, or passively monitoring whether users access these sites. The third party can observe all the packets to and from the client, and can delay, reorder, modify, drop, or inject packets within this network. There are parts of the network (shown as the *unfiltered network* in Figure 1) that this third party cannot monitor and does not control: the decoy routers, decoy proxies, decoy hosts, and disallowed hosts are within the *unfiltered network*. For Rebound, we assume that the third party permits the use of encrypted connections (i.e., TLS), between the client and sites in the unfiltered network, and cannot break common ciphersuites or undetectably forge host certificates for the decoy hosts. We assume that the third party cannot access the logs of the decoy hosts or disallowed hosts, and that the internal state of the client computer cannot be controlled or monitored by the third party. We also assume that there is a way for the Rebound software and keys to be installed on the client computer without the knowledge of the third party. Finally, we assume that the user can trust the Rebound software and the decoy router; compromise of the decoy router would compromise its users as well.

III. An overview of Rebound

In Rebound, the client in the filtered network first connects to a *decoy host*, which may be any allowed host located within the unfiltered Internet. This is shown in Figure 1 as the solid and then dotted line between the client and the decoy host. During the connection between the client and the decoy host, a *decoy router* along the path observes a *sentinel* hidden within the payload of the packets. The sentinel is a specific string of bits that the client puts into the packet stream. The decoy router looks for these bits in the packet stream and, upon finding them, recognizes that the connection might be from a client wishing to use Rebound.

Note that exactly how the sentinel is encoded within the packets is protocol-dependent. In this paper, we focus on the HTTPS protocol (HTTP over TLS), although Rebound has been implemented over HTTP as well. In the TLS implementations of the Rebound protocol, the sentinel is inserted into the *Random* field of the TLS *ClientHello* message packet.

When the decoy router detects the sentinel in the client connection, it begins the process of determining whether the connection represents a valid decoy routing connection (rather than a spurious match or an attempt by a malicious third party to replay an old connection). This process is called the *handshake*. The handshake establishes that the client is communicating with a valid decoy router and that the decoy router is communicating with a valid client.

Rebound is based on the Curveball protocol [18]. In the original Curveball protocol, once the handshake was complete, the decoy router terminated the client's connection with the decoy host with a TCP RST, and established a bidirectional tunnel between the client and the decoy router. The client still appears to be communicating with the decoy host, but the Curveball router is actually tunneling the conversation to the *disallowed host* in the unfiltered Internet—shown as the solid line in Figure 1. The Curveball router acts on behalf of the client to extract information ostensibly headed to the decoy host (but in reality destined to disallowed hosts). The Curveball router also acts on behalf of the disallowed host for information headed back to the client. The decoy host is never aware of this tunnel to the disallowed host, and the packets that may be observed by a third party appear to be part of a normal connection to the decoy host. In this way, the true destination and content of the client communication is hidden from anyone filtering or monitoring the client's network.

While the Curveball approach works on asymmetric routes, the Curveball protocol is not truly asymmetric because once the handshake is complete, the connection to the decoy host is closed, and the decoy router handles both the traffic to and from the client. The fact that the decoy router forges packets addressed from the decoy host creates opportunities for a third party to detect that the connection has been altered,

if the resulting packets from the decoy host to the client take a different route than expected, or are created with different characteristics than packets created by the decoy.

To address this weakness, we present Rebound, a novel way to handle asymmetric routes where the packets from the client to the decoy host go through the decoy router, but the packets from the decoy host to the client are not required to. While there are other decoy routing protocols that also accommodate asymmetric routing, they do so by either terminating the client to decoy host connection, or primarily using only the forward direction of the connection and having the decoy router communicate directly back to the client by spoofing packets from the decoy host. In the Rebound approach, *all* information to be conveyed to the client is securely and privately conveyed by the decoy host to the client; the decoy router never sends information directly to the client.

IV. Related Work

Telex [21] is a decoy routing approach that requires symmetric routes, works over TLS, and uses a public key system. Public-key systems do not require key distribution and management (although some care must be taken to ensure that the public key is real, and not a false key planted by a malicious third party). Telex appears to the user as an HTTP proxy, so it can only be used for HTTP/HTTPS traffic.

The Curveball protocol, described in Karlin *et al.* [18], is an approach that tolerates asymmetric routes, works over TLS, and uses a private key system. Curveball provides a general SOCKS proxy, which enables it to handle other traffic types, as well as a VPN, which can handle virtually any traffic type from the client. It is an example of a system that terminates the connection between the client and the decoy host at the decoy router and has the decoy router spoof packets that appear to be from the decoy host. Private-key systems such as Curveball require key distribution and management, but have the benefit that keys can be revoked. Rebound is based on Curveball and inherits its SOCKS and VPN proxies.

Cirripede [15] requires symmetric routes, works over TLS, and uses a public key system. Cirripede provides a general SOCKS proxy, which enables it to handle other traffic types. The Cirripede handshake uses a fundamentally different approach than Telex or Curveball: the client "registers" with the Cirripede service by encoding a message in a connection. This connection is not redirected or terminated, but instead serves to tell the Cirripede servers that a later connection will be running the Cirripede protocol. The later connection is then redirected entirely, so that it is never connected to the decoy. The Telex and Curveball protocols, on the other hand, encode the signaling within the same connection that is later redirected or terminated.

TapDance [20] is a more recent approach. TapDance tolerates asymmetric routes, works over TLS, and uses a public key system. In order to cope with asymmetric routes, TapDance takes advantage of the fact that an incomplete HTTP request will cause the decoy host to wait some period of time (without responding) for the request to be completed. The TapDance client encodes its hidden messages within incomplete HTTP requests, and the TapDance router spoofs responses from the decoy host to send information to the client.

While the TapDance protocol is able to effectively handle asymmetric routes, it does so by forging packets from the decoy host. Once the TapDance connection is established, most of the traffic flows between the client and the TapDance router, and only occasionally through the decoy host.

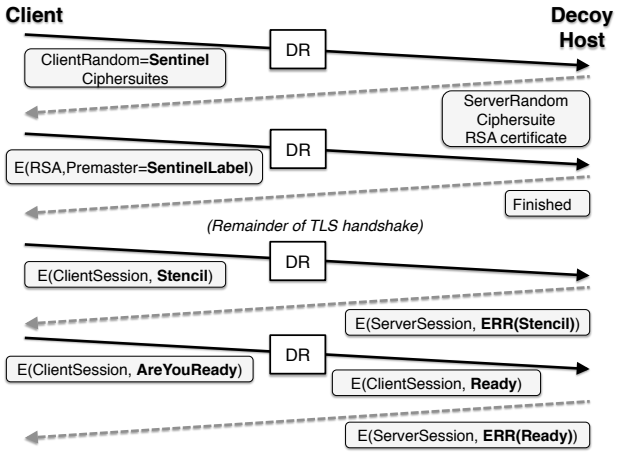

Fig. 2. An overview of Rebound's handshake and tunnel protocols. The decoy router observes all of the messages from the client to the decoy host, but does not observe messages from the decoy host to the client.

V. Fully Asymmetric Decoy Routing

We next describe Rebound's approach to perform fully asymmetric decoy routing without forging packets, using the TLS protocol.

Any decoy routing protocol requires both a *handshake protocol*, to securely determine whether a user wishes to use decoy routing, and a *tunnel protocol*, to securely tunnel information between the user and disallowed hosts. We first describe the Rebound handshake protocol that operates over an asymmetrically-routed HTTPS connection, and then describe the Rebound tunnel protocol, which operates over the same connection.

A. The Rebound Handshake

The Rebound handshake, illustrated in Figure 2, is the protocol that a Rebound router uses to verify that a connection flagged by the presence of a sentinel is a legitimate Rebound connection. The Rebound handshake must ensure not only that the connection is a well-formed Rebound connection, but that it also originates from a machine with a legitimate Rebound sentinel-key. This is complicated by the fact that since the decoy router cannot observe the reverse traffic from the decoy host, it does not have an obvious way to send information back to the client.

To authenticate the client and decoy router (and provide confidentiality and data integrity), we use TLS, with some slight modifications. We do not describe the details of the TLS protocol here, but only how Rebound uses the protocol. The modifications to the initial messages in the TLS handshake are illustrated in Figure 3. The underlined fields are TLS fields whose values are determined by the Rebound protocol. In general, the TLS handshake is unmodified, except that the `Random` field of the `ClientHello` message and the `PremasterSecret` field of the `ClientKeyExchange` message are generated from the client's Rebound key, rather than chosen at random. The sentinel is indistinguishable from an ordinary ClientRandom, unless the key is known. The client can generate many sentinels from the same key, and the client never uses the same sentinel twice.

Our Rebound implementation uses a private-key cipher to generate each sentinel, which requires that the user possess a secret key that is known to the decoy router. Rebound could be extended to permit use of a public-key mechanism (similar to how Telex [21] and TapDance [20] generate their sentinels) if desired. In different contexts, both private- and public-key systems have advantages: public-key systems simplify key distribution, but are susceptible to flooding attacks (a malicious third party can use the public key to create huge numbers of connections and overwhelm the decoy router). Private-key systems require key distribution and management mechanisms, but if a key is compromised and used to flood the system it can be revoked without impacting the other users of the system.

To describe the handshake and tunnel protocols, we use the following notation:

- *C* refers to the Rebound client, *DH* to the decoy host, and *DR* to the decoy router.
- $S(m) \rightarrow R$ denotes that message *m* is sent by the sender *S* to receiver *R*.
- $S(m) \xrightarrow{observer} R$ denotes that message *m* is sent by the sender *S* to receiver *R*, and the message is observed by the specified *observer*.
- $S(m) \xrightarrow{MITM} R(\bar{m})$ denotes that message *m* is sent by the sender *S* to receiver *R*, is observed by the specified man-in-the-middle *MITM*, and changed by the man-in-the-middle, so that *R* receives $\bar{m}$ instead of *m*.
- $E(k, m)$ denotes message *m* encrypted with key *k*.
- $Err(m)$ denotes the error response that would be returned in response to message *m*. In Rebound, *m* is typically an HTTP GET request whose URL encodes a message to the decoy router or the client. In order for the Rebound protocol to work, $Err(m)$ must contain *m* as a substring.

1) Sentinel Location: We modify the TLS protocol to hide the sentinel by replacing the TLS *Random* with the Rebound *Sentinel* in the first step of the TLS protocol, and then allow the next few steps of the TLS protocol to proceed unmodified.

The sentinel must be hidden in a location in a message within the TLS handshake where it can be easily detected by the decoy router, yet cannot be modified by an adversary without being detected and causing the TLS protocol to fail.

1) TLS ClientHello:
   $C(\underline{\mathbf{Sentinel}}) \xrightarrow{DR} DH$
2) TLS ServerHello:
   $DH(ServerRandom) \rightarrow C$
3) TLS ServerCertificate:
   $DH(PublicKey_{DH}) \rightarrow C$
4) TLS ServerHelloDone:
   $DH(ServerHelloDone) \rightarrow C$

Fig. 3. The initial messages of the TLS and Rebound protocol. The TLS protocol is unmodified, except that part of the ClientRandom field is replaced with the Rebound sentinel.

1) TLS ClientKeyExchange:
   $C(E(PublicKey_{DH}, \underline{\mathbf{SentinelLabel}})) \xrightarrow{DR} DH$
2) TLS ClientChangeCipherSpec:
   $C(E(ClientSession, Finished)) \xrightarrow{DR} DH$
3) TLS ServerChangeCipherSpec:
   $DH(E(DecoySession, Finished) \rightarrow C$

Fig. 4. The Premaster secret is replaced with a value computed based on the sentinel and the client's secret key. The decoy router, which already knows the value of the sentinel and the client's secret key, can compute the same Premaster secret.

Rebound hides the sentinel in the *Random* field of the TLS *ClientHello* message. This field is always at a fixed offset and so can be easily located by the decoy router. Because this field is protected by integrity checks in the TLS protocol, and is used to generate the session keys, this field cannot be modified without making it impossible for the client to establish a TLS connection with a decoy host.

2) Obtaining the TLS Premaster Secret: The TLS protocol generates its session keys from the *ClientRandom*, the *ServerRandom*, the *Premaster* secret generated by the client, and the selected *Ciphersuite*. In order for Rebound to infer the session keys, Rebound must be able to determine the values of all of these fields.

Even if routes are symmetric, the decoy router is unable to observe the *Premaster* secret, because it is encrypted (typically by the public key of the decoy). To address this, the Rebound protocol specifies which *Premaster* the client should choose, based on the *Sentinel* and the key used to create that *Sentinel*. Thus, when the decoy router sees the sentinel for a flow, the decoy router can compute the value of the *Premaster* (if it knows the client's key) for the flow if the flow is following the Rebound protocol and the client has a valid key. When the decoy router receives the message containing the encrypted form of the *Premaster*, it confirms that the *Premaster* is consistent with the protocol. The modification to the ClientKeyExchange message is described in Figure 4.

When routes are asymmetric, the decoy router is unable to observe any of the messages from the decoy host to the client. The client conveys the *ServerRandom* and *Ciphersuite* to the decoy router using *stencil coding*, which is described next.

3) Stencil Coding: Stencil coding permits a sender to secretly communicate a value to an observer via an encrypted channel to a third party, without requiring that the observer be able to decrypt the channel. Stencil coding does require that the sender and the observer either share a private key, or that the sender possess the public key of the observer, in order to ensure the privacy and integrity of the communicated value.

For Rebound, the client uses stencil coding to share, with the decoy router, the values of *ServerRandom* and *Ciphersuite* it received from the decoy host during the TLS handshake. Once the decoy router has these values, it can reconstruct the TLS session keys, and then read and write messages in the TLS connection.

The basic idea of stencil coding is straightforward: the client chooses a plaintext message to send to the decoy host such that the ciphertext for the message (as created by the TLS session between the client and the decoy host) encodes the encrypted value of the secret that the client wishes to share with the decoy router. The message must be chosen with some care because it will also be received and interpreted by the decoy host: it must, in our case, decrypt to one or more properly formed HTTP requests.

The *stencil* is a specification of which bits of the encrypted TLS message correspond to the encrypted message from the sender to the observer. The optimal stencil depends on the ciphersuite chosen, but since neither the client nor the decoy router know the ciphersuite ahead of time (and, in fact, the client uses stencil coding to tell the decoy router which ciphersuite was chosen) we must use a stencil that can work regardless of the ciphersuite.

For Rebound, we use a stencil which uses the low-order bit of the last eight bytes of each 16-byte block in a TLS data record, omitting the final checksum and padding. This stencil has the advantage of working with all popular ciphersuites.

Stencil coding is similar to a more efficient scheme that was developed for a similar purpose as part of the TapDance protocol [20]. The advantage of stencil coding is that it works with TLS 1.0, which is still used by a large percentage of web servers, while the advantages of the encoding used by TapDance are higher efficiency and simplicity of implementation for TLS 1.1 and 1.2. The essential difference (with respect to stencil coding) between TLS 1.0 and later versions of TLS is that later versions permit the creator of each TLS data record to choose the initialization vector associated with that record, and the TapDance protocol chooses the initialization vector in a way that makes finding a plaintext with the necessary properties more efficient. In TLS 1.0, the initialization vector of each TLS record is defined implicitly, and therefore Rebound cannot use the initialization vector as a free variable in its search.

4) Obtaining the TLS ServerRandom and Ciphersuite: In our case, once the TLS handshake has finished, the client uses stencil coding to communicate to the decoy router the *Random* and *Ciphersuite* fields of the *ServerHello* of the TLS handshake. The *Random* field is 32 bytes long, and the *Ciphersuite* field is two bytes long. Rebound uses a stencil that encodes eight bits in 16 bytes of plaintext, so this requires finding a 544-byte plaintext whose ciphertext (when encrypted by the TLS session key) encodes these 34 bytes. Once the client chooses the plaintext, it is sent to the decoy host over the TLS connection, and the decoy router observes the ciphertext. This process is illustrated in Figure 5.

1) TLS ApplicationRecord
   $C(E(ClientSession, \mathbf{StencilMessage})) \xrightarrow{DR} DH$
2) TLS ApplicationRecord
   $DH(E(ServerSession, \mathbf{ERR}(\mathbf{StencilMessage})))$
   $\rightarrow C$

Fig. 5. Send *StencilMessage*, an HTTP GET request constructed from the Stencil encoding of ServerRandom and the Ciphersuite identifier, to the decoy host via TLS. The Rebound router can extract the value of the ServerRandom and Ciphersuite identifier, even though the message is encrypted by TLS.

1) TLS ApplicationRecord
   $C(E(ClientSession, \mathbf{AreYouReady}))$
   $\xrightarrow{DR} DH(E(ClientSession, \mathbf{Ready}))$
2) TLS ApplicationRecord
   $DH(E(DecoySession, \mathbf{ERR}(\mathbf{Ready}))$
   $\rightarrow C$

Fig. 6. To finish the handshake, the client sends an *AreYouReady* GET request to the decoy host. If there is a Rebound router along the path from the client to the decoy host, the Rebound router detects the *AreYouReady* message and replaces it with a *Ready* GET request. If the client receives a response matching the *Ready* request, then it knows that there is a Rebound router along the path.

At this point the decoy router extracts the *ServerRandom* and *CipherSuite* from the stencil-encoded message. The decoy router is now able to compute the session keys.

5) Completing the Handshake: Once the decoy router has the keys for the TLS session, it can begin to monitor the data sent from the client to the decoy host, and even rewrite this data. To announce its presence to the client, the Rebound router waits until it sees a GET request containing an URL encoding the *AreYouReady* message from the client. The Rebound router then rewrites the URL so that encodes the *Ready* message, and sends the resulting request to the decoy host. The decoy router replaces the *AreYouReady* message with a *Ready* message by rewriting the packets as it routes them to the decoy host, as shown in Figure 6.

When the client decrypts the response from the decoy host, it will either receive an error message containing the *AreYouReady* message, if there is no decoy router along the path, or the *Ready* message, if it has successfully completed the handshake with a Rebound router.

This handshake relies on several features to ensure that the decoy router is talking with a valid Rebound client and vice versa. First, the *PremasterSecret* is set by the client in such a way that the decoy router can infer it from previously shared information. Second, the client conveys the *ServerRandom* and *Ciphersuite*, encrypted using previously shared information known only to the client and decoy router. Third, using this information in a normal TLS handshake gives the connection

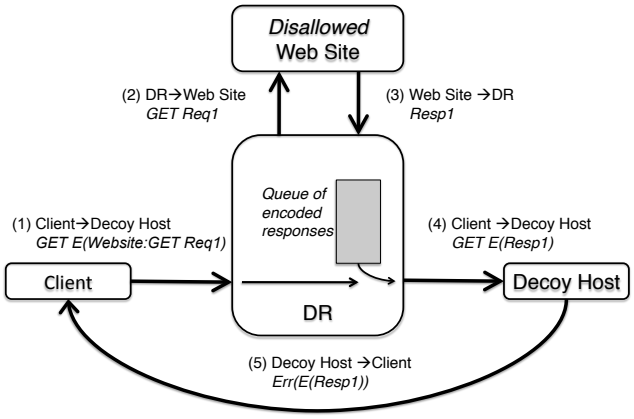

Fig. 7. The Mole Protocol

the authentication and security attributes of TLS.

B. The Rebound Tunnel

For the Rebound tunnel to function effectively, the client and decoy router must be able to exchange non-trivial amounts of information bidirectionally over what is effectively a unidirectional connection. Since there is no path from the decoy host back through the decoy router, the information from the decoy router must be conveyed to the decoy host in such a way that this information eventually reaches the client. This is the purpose of the Rebound *mole* protocol, which is illustrated in Figure 7.

The key rules for the mole protocol are that packets must always be forwarded immediately from the client to the decoy host (in order to avoid susceptibility to timing analysis) and that the data that is delivered to the decoy host must be a valid TLS stream containing well-formed HTTP requests.

As with the handshake, the mole protocol addresses this problem by taking advantage of the fact that the error message many web servers return in response to a failed GET request contains the URL that triggered the error. By rewriting client requests to contain HTTP GET requests that contain URLs that are extremely likely to cause errors and that encode the desired messages, the decoy router is able to send information back to the client via the decoy host.

When the mole protocol was conceived, we planned to use TRACE requests instead of GET requests. Using TRACE requests is more efficient than using intentionally erroneous GET requests, but TRACE requests are a vector for several attacks against the privacy and credentials of web browsers, and therefore most web servers no longer accept TRACE requests. We find that many sites still support TRACE requests, however, and many sites close connections that have multiple erroneous GET requests, so an ideal implementation of the mole protocol would be able to use both types of requests, depending on what the decoy host supports.

Communication between the client and the decoy router is done via a protocol for multiplexing SOCKS or VPN connections over a single virtual connection. We do not describe the multiplexing protocol in this paper, but only describe how messages are exchanged over this virtual connection.

1) The client creates an HTTP GET request whose URL encodes a message to the Rebound decoy router. In our example, this is a request to open a connection to a given web site and then send message *Req1* to that site. The GET request with the URL encoding this message is sent to the decoy host via the TLS connection.
   Note that this encoding includes encryption; a third party who observes the URL cannot decipher the message and discover either the name of the web site or *Req1*. This encryption is in addition to the ordinary encryption done as part of the TLS protocol, so that knowledge of the TLS session keys is not sufficient to decrypt the message and discover *Req1*.
2) The decoy router intercepts and decrypts the request, opens a connection to the web site, and sends the *Req1* message to the web site.
   At the same time, the decoy router immediately forwards the packets containing the request to the decoy host. If the decoy router has pending data for the client, it will replace the URL of the original request with a new URL that encodes as much data as will fit. If there is no pending data, then the decoy router may rewrite the packets with chaff (as discussed in Section V-B1) or forward them as-is, depending on its heuristics for maintaining an active connection to the decoy host. In any case, the packets that are received by the decoy router are immediately forwarded–possibly with altered contents. The need to forward these packets immediately is discussed in Section VIII-A3.
3) At some later point in time, the web site responds to *Req1* by sending *Resp1* back to the decoy router. The decoy router adds *Resp1* to the queue of information to send to the client in the future.
4) When *Resp1* reaches the head of the queue, and another request is received from the client on the Rebound connection, the decoy router rewrites the URL of the request with a new URL that encodes *Resp1* (and any other pending data for the client that will fit in the request), and then forwards the packets immediately to the decoy host. This URL is constructed in a way that makes it extremely unlikely to match any resource on the decoy host.
   This encoding also includes encryption (using the same mechanism as the client used to encrypt *Req1*), so that a third party (or the decoy host) cannot decipher the message and learn *Resp1*.
   If the decoy router has no information to send, it simply rewrites the request using a chaff URL, as described in Section V-B1.
5) Because the URL created by the decoy router in the

previous step does *not* request a real web page that the decoy host can serve, it will cause an error, such as a 404 "page not found" error. Such error responses often contain the offending URL, which encodes the desired message *Resp1*. The client then extracts *Resp1* from the error response and delivers it.

1) Chaff Generation: Observe that the responses from the disallowed host in *Step 2* of the mole protocol are queued up at the decoy router (in the *mole queue*), awaiting subsequent requests from the client into which they can be packed and conveyed back to the client, via error responses from the decoy host. This means that the client must generate a steady stream of GET requests that the Rebound router can fill in with the contents of this queue, because the only way that the decoy router can send information to the client is by encoding it in messages from the client to the decoy host. We call such requests in this stream *chaff* requests, because they encode no application data, but exist only to be filled in by the decoy router.

The client must generate chaff with sufficient frequency to ensure that the mole queue is emptied in a timely manner, yet not so frequently as to overwhelm the network, the decoy router, or the decoy host. To rate-limit the chaff, we use a multi-level throttling approach with several layers of chaff. The first layer comprises background chaff that is sent at a regular fixed rate of 1 chaff message every 0.5 seconds. The second layer comprises additional chaff that is sent depending on whether the responses received by the client contain data, or are simply responses to the chaff. The intuition here is that if the client is seeing data in responses, then there is probably more data pending in the mole queue at the decoy router, and the client should increase its rate of chaff until it begins to receive chaff responses. Once the client begins to receive chaff responses, it should then reduce the rate at which it sends chaff until either it reaches a minimum rate or the percentage of chaff responses falls below a threshold.

To use the mole protocol within TLS, we must ensure that the rewritten requests can be decrypted by the decoy host. The decoy router cannot rewrite client requests with unencrypted requests taken from the mole queue (containing URLs that encode either data or chaff). Instead, the decoy router must rewrite properly encrypted TLS requests. Thus, the mole queue must store requests encrypted using the TLS keys; the requests themselves, however, are as described earlier. Thus, the flow of information from the client to the decoy host via the Rebound tunnel is as described in Figure 8.

VI. Implementation

Rebound is written in C, C++, and Python. The Rebound decoy router runs on Linux, and the client software runs on Windows, MacOS X, Linux, and Android. Our implementation of Rebound is available as open source, as part of the Curveball software release, at https://curveball.nct.bbn.com.

1) TLS ApplicationRecord
$C(E(ClientSession, MsgToDR))$
$\xrightarrow{DR} DH(E(ClientSession, MsgToClient)$
2) TLS ApplicationRecord
$DH(E(DecoySession, Err(MsgToClient))) \rightarrow C$

Fig. 8. The mole tunnel. The client sends messages to Rebound by encoding them in the URLs of GET requests that it sends to the decoy host. The Rebound router replaces those URLs with URLs that encode the messages it wishes to send to the client. The decoy host responds to the GET requests with error messages containing those URLs, and the client observes these error messages and decodes them to receive the messages from the Rebound router.

TABLE I
Bandwidth of HTTP, Curveball, and Rebound

Protocol	Bytes/s	stdev
HTTP	1,174,240	83,812
Curveball	354,676	24,238
Rebound	129,398	9,655

VII. Performance Results

We measure our implementation of Rebound across the Internet in order to characterize its performance in a realistic setting. Our client is a laptop running MacOSX 10.10, connected to the Internet via wifi through a residential Verizon FiOS connection. The client is 12 hops away from the Rebound router, with a typical round-trip latency between the client and Rebound router of 26 milliseconds. The decoy host is adjacent to the Rebound router.

Recall from Section V-B1 that the rate at which Rebound downloads data to the client is a function of the rate at which the client sends messages to the decoy, because the only way that the Rebound router can send information to the client is by encoding it in a packet sent from the client to the decoy. For the benchmarks we describe in this paper, the client is configured to send messages (either containing real data or chaff) at a rate that fits within the characteristics of the route. We can send data more quickly in short bursts, and we can also send data more slowly (or at random intervals) to make it more difficult to detect the Rebound traffic.

Table I shows typical transfer rates for 1 megabyte transfers from the disallowed host to the client. Over this route, ordinary HTTP achieves 1147 KB/s; Curveball achieves 346 KB/s. Rebound achieves 126 KB/s for 1 megabyte transfers. which is fast enough to stream 360p video or a standard-quality two-

TABLE II
Latency to load web pages using HTTP/HTTPS or Rebound

	Page load time in seconds (stdev)	
Site	**Rebound**	**Ordinary Web Connections**
www.cnn.com	38.7 (4.24)	7.68 (6.28)
www.nytimes.com	17.5 (8.55)	3.31 (0.85)
en.wikipedia.org	2.1 (0.89)	0.38 (0.05)
slashdot.org	23.9 (6.04)	4.32 (0.66)
twitter search	9.44 (1.39)	0.91 (0.09)
google search	4.96 (1.30)	0.27 (0.09)

way video conference.

In a test environment, where the client, decoy host, and disallowed hosts are all adjacent to the Rebound router and there is no other traffic on the network, HTTP and Curveball are at least an order of magnitude faster. Rebound is not fast, even in ideal circumstances, but its performance degrades substantially less in a shared, busy network.

For sustained transfers at 126 KB/s, our unoptimized Python implementation of the Rebound router uses less than half of the processing bandwidth of a single core of an Intel Xeon E5620 2.4 GHz processor. For ordinary web browsing (rather than sustained transfers at full speed) our implementation can support multiple concurrent users, and we believe that an optimized, multicore implementation will be able to support many concurrent users.

Table II illustrates the impact of using Rebound on the user experience for browsing the web. We use the "Page load time" plugin to the Google Chrome web browser to measure the time to load web pages from several popular web sites. Although browsing the web via Rebound is substantially slower than browsing the web in the normal manner, it is still acceptable. Most of the page load time for graphics- or ad-intensive sites (such as www.cnn.com, www.nytimes.com, or slashdot.org) is used to load advertisement graphics and trackers, and this does not impact the user experience; the text the users are waiting to read is rendered before the ads.

VIII. Vulnerability Assessment

Rebound's approach to asymmetric decoy routing over TLS eliminates a number of vulnerabilities present in other decoy routing protocols, but also introduces some vulnerabilities.

We note that route flapping (or intentional manipulation of routes by a third party [19]) is a concern for every decoy routing protocol, including Rebound. Recent analyses have shown that proper placement of decoy routers can reduce this problem [9] and that decoy routing is reasonably robust against route-based attacks [16].

In the rest of this section we describe vulnerabilities that Rebound mitigates, and then the new vulnerabilities to address in our future work.

A. Vulnerabilities Mitigated

1) Stack Fingerprinting: IP and TCP implementations (often referred to as "network stacks") differ in the way that they implement aspects of protocols that are not completely specified, and in practice each implementation has a distinct fingerprint. For example, different network stacks use different TCP options, order the TCP options differently, use different TCP clock rates, or use different TCP window size initialization and update mechanisms, etc. If the decoy router is running a different network stack than the decoy host, then the difference between the stacks can be detected by a third party unless the decoy router implementation actively imitates the stack of the decoy host.

If the decoy router creates packets addressed to the client, spoofing the address of the decoy host, then it must do so in such a way that the spoofed packets are identical to the packets that would have been created by the decoy in the same circumstances. The Cirripede, Curveball, Telex, and TapDance protocols are all susceptible to detection via stack fingerprinting, to some extent, because they all forge packets from the decoy host. Rebound does not need to mimic the network stack of the decoy host, and therefore is immune to stack fingerprint analysis. All information sent from Rebound to the client goes through the decoy host. The Rebound protocol never creates packets addressed to the client; only the decoy host creates such packets.

2) Connection State Probes: An active third party can probe the decoy host to see whether it is still connected to the client, and whether that connection reflects the same state that is observed at the client. The simplest probe is sending TCP in-window data, and seeing how the decoy host responds. If the connection has been terminated, the decoy host will respond with an error.

There are several variations on this probe. If the decoy router observes all of the traffic from the decoy host to the client, then it can detect simple probes and either drop the probes, or forge a response from the decoy host.

A more difficult type of probe to defeat uses the same mechanism as 0trace [22], but for a different purpose. To create this probe, the third party clones data packets from the client to the decoy host and sends them via a different route (so that the decoy router cannot observe them) to the decoy host. Before sending the packets, however, the third party replaces the packet TTL with a value calculated to expire at a router very close to the decoy host. If the connection has been closed, and if the decoy host is protected by a stateful firewall that drops all packets for dead or malformed flows, then the packets will never reach the router in front of the decoy host because they will be discarded by the firewall. If the connection is open, however, the firewall will admit the packets, and then the packets will expire at the router and the third party will observe the resulting ICMP packet generated by the router. This probe is impossible to detect if it is successful.

There are also several ICMP message types and IP options (such as IP option 7, "Record Route") that can be used to probe connection state. Although in some cases these messages are archaic or deprecated, they are still supported by many routers, and therefore must be handled correctly by a decoy router.

Protocols such as Telex and Curveball, which terminate the original connection between the client and the decoy host and replace it with a spoofed connection that appears to be between the client and the decoy host but is actually between the client and the decoy router, are susceptible to connection probes. The Rebound solution to connection probes is to not terminate and spoof, but instead use the original connection to the decoy host as the conduit for its communication with the client. This means that the connection to the decoy host is, by definition, always in exactly the same state as observed by the client or a third party. Connection probes will not reveal any discrepancy between the client and decoy host states, because there is no discrepancy to reveal.

3) Timing Analysis: Earlier decoy routing protocols were susceptible to detection via timing and/or analysis; if the latency between requests and responses from the decoy does not match the typical latency of other connections to the decoy, a third party may infer that the responses are coming from another host. Some analyses are even able to deduce what hosts generated the responses, by looking at a combination of the timing and the length of the responses [14].

Rebound defeats timing analysis. Packets from the client to the decoy host are always forwarded immediately to the decoy, so the latency of the responses from the decoy is decoupled from the latency of the responses from the disallowed host. The same requests sent to the decoy host by an ordinary client will result in responses with the same latency.

Recall from Section V-B that if a packet arrives at the Rebound router, but it doesn't yet have any "real" data from the disallowed host to send to the client, it must still forward the packet immediately, so it forwards chaff. It cannot simply drop or delay the packet, because then there would be a detectable difference in timing, or between the acknowledged and expected sequence numbers.

Traffic analysis is also complicated by Rebound, since the length of each message is obscured. The messages from the client to the decoy host (or the disallowed host) are encoded at a constant effective rate. The same number of messages, of the same length, will be sent regardless of whether the client has any data to send to the disallowed host, or whether the disallowed host has any data to send to the client.

B. Vulnerabilities Introduced

Although Rebound masks the identity of the disallowed host and the data it sends and receives, Rebound may be detectable via traffic analysis because the traffic it generates (a sequence of long GET requests and error responses) has a characteristic pattern that may be identified even when the channel is encrypted. Although these error responses are encrypted by TLS, they will often differ, to an extent that is observable, from ordinary traffic. To reduce the observability of this signature, it is possible to intermingle the Rebound requests with ordinary requests, but this further diminishes the effective throughput of the hidden connection.

IX. Conclusions

In this paper, we presented Rebound, a novel decoy routing handshake and tunnel protocol that handles asymmetric routes and continues to actively use both directions of the connection between the client and the decoy. In Rebound, *all* information to be conveyed to the client is securely and privately conveyed to the client via the decoy host; the decoy router never sends information directly to the client. This approach eliminates several vulnerabilities present in previous approaches to decoy routing over asymmetric routes.

Acknowledgment

David Mankins and Josh Karlin made substantial contributions to the design and implementation of Rebound.

We thank the reviewers for their comments and suggestions for improving this paper.

References

[1] "Analogbit: Tcp-over-dns tunnel software howto," http://analogbit.com/tcp-over-dns_howto.

[2] "Freegate," http://www.dit-inc.us/freegate.

[3] "Global pass," http://gpass1.com/gpass/.

[4] "Guardster," http://www.guardster.com.

[5] "Proxify web proxy," https://proxify.com.

[6] "Ultrasurf," http://www.ultrareach.com.

[7] A. Baliga, J. Kilian, and L. Iftode, "A web based covert file system," *Proceedings of the 11th USENIX workshop on Hot topics in operating systems HOTOS*, 2007.

[8] S. Burnett, N. Feamster, and S. Vempala, "Chipping away at censorship with user-generated content," *USENIX Security Symposium*, 2010.

[9] J. Cesareo, J. Karlin, M. Schapira, and J. Rexford, "Optimizing the placement of implicit proxies," http://www.cs.princeton.edu/~jrex/papers/decoy-routing.pdf, June 2012. [Online]. Available: http://www.cs.princeton.edu/~jrex/papers/decoy-routing.pdf

[10] R. Deibert, "Psiphon," http://psiphon.civisec.org/.

[11] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," *13th USENIX Security Symposium*, 2004.

[12] N. Feamster, M. Balazinska, G. Harfst, H. Balakrishnan, and D. Karger, "Infranet: Circumventing Web Censorship and Surveillance," in *11th USENIX Security Symposium*, San Francisco, CA, August 2002. [Online]. Available: http://wind.lcs.mit.edu/papers/

[13] Y. He, M. Faloutsos, S. Krishnamurthy, and B. Huffaker, "On Routing Asymmetry in the Internet," in *IEEE GLOBECOM 2005*, November 28–December 2, 2005.

[14] A. Hintz, "Fingerprinting websites using traffic analysis," in *Privacy Enhancing Technologies*, ser. Lecture Notes in Computer Science, R. Dingledine and P. Syverson, Eds. Springer Berlin Heidelberg, 2003, vol. 2482, pp. 171–178. [Online]. Available: http://dx.doi.org/10.1007/3-540-36467-6_13

[15] A. Houmansadr, G. T. K. Nguyen, M. Caesar, and N. Borisov, "Cirripede: circumvention infrastructure using router redirection with plausible deniability," in *Proceedings of the 2011 ACM Conference on Computer and Communications Security (CCS)*, October 2011, pp. 187–200.

[16] A. Houmansadr, E. L. Wong, and V. Shmatikov, "No direction home: The true cost of routing around decoys," in *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS)*, ser. NDSS '14, February 2014.

[17] W. John, M. Dusi, and k. claffy, "Estimating Routing Symmetry on Single Links by Passive Flow Measurements," in *The 6th International Wireless Communications and Mobile Computing Conference (IWCMC 2010)*, June 28–July 2, 2010.

[18] J. Karlin, D. Ellard, A. W. Jackson, C. E. Jones, G. Lauer, D. P. Mankins, and W. T. Strayer, "Decoy routing: Toward unblockable internet communication," in *Proceedings of the 2011 USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, August 2011.

[19] M. Schuchard, J. Geddes, C. Thompson, and N. Hopper, "Routing around decoys," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 85–96. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382209

[20] E. Wustrow, C. M. Swanson, and J. A. Halderman, "Tapdance: End-to-middle anticensorship without flow blocking," in *23rd USENIX Security Symposium*, 2014.

[21] E. Wustrow, S. Wolchok, I. Goldberg, and J. A. Halderman, "Telex: Anticensorship in the network infrastructure," in *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[22] M. Zalewski, "0trace - traceroute on established connections," http://lwn.net/Articles/217023, January 2007.