MultiFlow: Cross-Connection Decoy Routing using TLS 1.3 Session Resumption

Victoria Manfredi
Wesleyan University

Pi Songkuntham
Wesleyan University

Abstract

Most approaches to circumventing Internet censorship and monitoring use conventional proxies which are accessed directly by their IP addresses and so are easily blocked. Decoy routing is an alternative approach that deploys a proxy in association with a router, called a decoy router, that is only accessible indirectly when traffic traverses the router. In this work, we design MultiFlow, a new decoy routing protocol that re-uses the TLS protocol's session resumption mechanism to enable the decoy router itself to resume a client's session. As a consequence, MultiFlow is able to (1) authenticate a client without blocking traffic inline on the decoy router, and (2) use information provided by the client to bootstrap the establishment of additional secure connections for covert communication. The client and decoy router then use a message board-like tunnel to communicate across multiple connections in a way that mitigates probing and traffic analysis attacks.

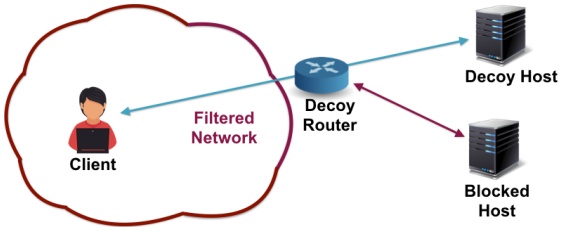

Figure 1: Decoy routing overview. The client is located in a network controlled by an adversary. The adversary can monitor, filter, modify, or block any traffic entering or leaving its network. While the adversary can see client to decoy host traffic that is sourced inside its network, and can probe the decoy host, the adversary cannot observe the actual packets that arrive at the decoy host, nor monitor the internal activity of the client. The adversary also cannot see all traffic sent to or from the decoy router, decoy host, or blocked host. The decoy host and blocked host are oblivious to the use of decoy routing.

1 Introduction

Nation-state censorship and monitoring of user traffic on the Internet is widespread, affecting the majority of people in the world [8]. In some countries, a user may be prevented from accessing certain websites, or redirected unwittingly to an alternate version of a website. Most approaches to circumventing such Internet censorship and monitoring, like Tor [12], use conventional proxies which are accessed directly by their IP addresses and so are easily blocked. Decoy routing [16, 19, 24] is an alternative approach in which a proxy or monitoring point is deployed in association with a router, called a *decoy router*. The proxy is then only accessible indirectly when traffic is routed though the decoy router.

As in Figure 1, in decoy routing, a client typically opens a TLS [5] connection to any unblocked website, called the *decoy host*. If that connection traverses a decoy router on its way to the decoy host, the decoy router can connect to a blocked host on the client's behalf. The decoy router then tunnels traffic from the blocked host back to the client via the client-decoy host traffic. To an adversary, the client appears to be communicating only with the decoy host. To block the operation of decoy routing, an adversary must block most or all websites on the Internet: for instance, in 2012 it was shown that 30 autonomous systems with decoy routers will be on the paths of about 90% of client-decoy host paths [11].

The deployment of decoy routers, however, requires the cooperation of ISPs, who are unwilling to add any in-path or blocking elements to their routers. Thus, *tap-based* implementations of decoy routers are desired, in which the proxy associated with the router observes the packets passing through the router but does not delay or modify them. Instead, the proxy can additionally cre-

ate and inject new packets as needed. Indeed, the first decoy routing protocol deployed in an ISP [14] was TapDance [23], the first tap-based protocol. However, because Internet routes are typically asymmetric, most decoy routing protocols assume that only the forward direction of traffic can be observed by the decoy router. TapDance also makes this assumption and as a consequence is unable to verify client liveness [23].

In this work, we design a tap-based decoy routing protocol, MultiFlow, which uses the TLS session resumption mechanism to enable the following key features.

1. *Client authentication with a tap-based decoy router.* The decoy router opens its own connection to the decoy host, by resuming the client's session. If the decoy host is willing to resume the session, the client is considered a valid decoy routing client.

2. *Asynchronous and out-of-band communication.* The tunnel operates as a virtual message board. The decoy router connects to a host accessible to the client and uploads covert data. The client then downloads the covert data using its own connection to the same host. Thus, the client and decoy never communicate within the same connection, mitigating probing and traffic analysis attacks.

3. *Cross-connection and cross-server decoy routing.* A decoy routing session can be created across multiple connections and decoy hosts. To do so, the client exfiltrates to the decoy router the TLS session resumption information for a second, different connection. This (1) amortizes the cost of setting up a new decoy routing connection and (2) protects against an adversary that terminates a TLS connection after one exchange of application data.

2 Related Work

Fig. 2 summarizes the characteristics of the different decoy routing protocols, which we describe in detail next.

1st generation decoy routing protocols. These protocols [24, 19, 16] primarily differ with respect to (1) whether they support the much more likely case of asymmetric Internet routes [15, 22, 18], and (2) how the client signals its wish to use decoy routing and the subsequent handshaking between the client and decoy router.

Telex [24] operates over symmetric routes while Cirripede [16] and Curveball [19] operate over asymmetric routes. But in all three protocols, the decoy router uses a TCP RST to reset the client to decoy host connection once the handshake completes. The decoy router then forges responses from the decoy host. Consequently all three protocols are vulnerable to connection probing attacks when probes bypass the decoy router on their way to the decoy host. To securely reset the client connection with the decoy host, the decoy router must block traffic inline. Otherwise client traffic risks reaching the decoy host after the connection has been reset, with the decoy host subsequently sending its own TCP RST back to the client, visible to an adversary. Cirripede and Curveball, by operating over asymmetric routes, must also ensure the decoy router correctly fingerprints the network stack of the decoy host when forging traffic.

Decoy routing is typically implemented within the TLS protocol [5], leveraging the security properties of TLS. In both Telex and Curveball, the client signals its wish to use decoy routing via the TLS ClientRandom, which cannot be modified by an adversary without breaking the TLS protocol itself. Cirripede instead combines the TCP initial sequence numbers (ISNs) of multiple connections to obtain the client signal. ISNs are more easily modified than the ClientRandom but also let Cirripede use a later different TLS connection to tunnel covert data back to the client. Attacks replaying client handshake traffic can be mitigated by limiting how long a signal is valid, but only Telex can definitively check client liveness since it operates over symmetric routes.

2nd generation decoy routing protocols. These protocols [23, 13] improve the security and deployability of decoy routing with respect to two issues: (1) termination of the client-decoy host connection which introduces connection probing attacks and (2) inline blocking of traffic at the decoy router which hinders deployability.

Like Telex [24] and Curveball [19], TapDance [23] and Rebound [13] operate over asymmetric routes, and use the ClientRandom field to signal the client's wish to use decoy routing. Unlike 1st generation protocols, TapDance and Rebound do not terminate the client-decoy host connection. Instead, the decoy router rewrites or injects traffic on this connection, which requires knowledge of the TLS keys. Some of the information needed to compute these keys is generated by the decoy host but is never seen by the decoy router since routes are assumed asymmetric. Thus, TapDance and Rebound each design techniques by which the client exfiltrates the needed information to the decoy router, respectively called chosen ciphertext steganography and stencil coding. In both techniques the client chooses plaintext messages in such a way that when encrypted the resulting ciphertext steganographically hides, or stencils, the information to be exfiltrated in bits of ciphertext. By working through the encryption backwards, TapDance is more efficient than Rebound in finding the right plaintext.

TapDance also implements the first non-blocking (tap-based) decoy router. In TapDance, the client hides covert data in incomplete HTTP requests: while the decoy host waits for the HTTP request to complete, the decoy router forges decoy host responses. While TapDance does not

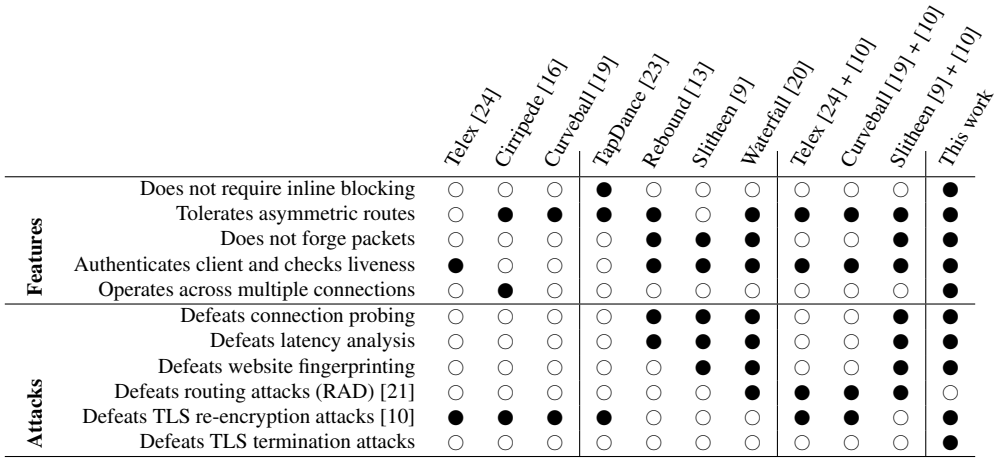

Figure 2: Comparison of decoy routing protocols (extension of the table found in [10]). Filled black circles indicate features that a protocol supports or attacks a protocol defeats. More filled circles is considered more positive.

terminate the client-decoy host connection, the forging of packets leaves TapDance vulnerable to connection probing since the decoy host's TCP state will not match the client's view. TapDance is also unable to validate client liveness, a consequence in part of the tap-based design.

While not tap-based, Rebound also does not forge responses. Instead, Rebound takes advantage of malformed URLs in GET requests triggering error messages from the decoy host which can include the offending URL. The decoy router then blocks client traffic inline: the URL in the client's GET request is immediately rewritten, either with random data that the client detects as chaff, or covert data to send to the client. Rebound is vulnerable to traffic analysis attacks, in part because of the large amount of chaff traffic that is generated. Rebound relies on the decoy host to check client liveness on its behalf: if the client is live the decoy host sends the error message back to the client, otherwise the decoy host sends a TCP RST. The decoy router itself never knows whether the client is live, but still behaves appropriately.

3rd generation decoy routing protocols. These protocols [9, 20, 10] focus on (1) resisting attacks that route traffic around decoy routers [21] or (2) addressing vulnerabilities from operating over asymmetric routes. None of these protocols is tap-based. The routing around decoys attack (RAD) [21] shows that a powerful adversary can make routing decisions that bypass decoy routers. Further analysis shows that, in addition to the high costs of launching such an attack, more strategic placement of decoy routers can mitigate the attack [17].

Slitheen [9] operates over symmetric routes, so the decoy router can easily obtain the TLS keys and check client liveness. Slitheen mitigates traffic analysis by having the decoy router only replace leaf content from the decoy host, like images, with covert data. Waterfall [20] operates over asymmetric routes and protects against RAD attacks by having the decoy router rewrite traffic on the reverse rather than forward path. The client leverages Rebound's error channel [13] to communicate covert data to the decoy router. By seeing traffic from the decoy host, Waterfall is able to check client liveness.

Recent work [10] proposes collaboration among decoy routers, enabling asymmetric protocols to gain the security properties of operating over symmetric routes, as well as mitigating RAD attacks. This work also introduces a TLS re-encryption attack that occurs when an adversary sees re-encrypted traffic both entering and leaving a decoy router as in the Rebound [13] and Slitheen [9] protocols (a capability that decoy routing protocols typically assume the adversary does not have).

This work. MultiFlow makes three contributions: (1) checking client liveness with a tap-based decoy router by having the decoy router resume the client's session, (2) mitigating traffic analysis and connection probing attacks by never having client-decoy router communication in the same connection, and (3) amortizing setup cost and mitigating TLS termination attacks via cross-connection and cross-server decoy routing. By TLS termination attacks we mean attacks that terminate a connection after one client-decoy host data exchange, blocking decoy routing while still enabling normal users to use TLS. The current MultiFlow tunnel, however, does require the client to share private information, like a username/password or email address, with the decoy router.

3 MultiFlow Protocol

We overview the MultiFlow protocol in Fig. 3 and Fig. 4. In the MultiFlow *handshake* in Fig. 3, the decoy router

passively observes TLS traffic for a colluding client on a possibly asymmetric route. Once the client exfiltrates the necessary information in TLS traffic, the decoy router opens its own TLS connection to the decoy host and resumes the client's session. The goal of this resumption is to check client liveness. In the MultiFlow *tunnel* in Fig. 4, the client and decoy router communicate via a virtual *message board*. The client uses its original connection to the decoy host, which traverses the decoy router, to communicate requests to the decoy router. The decoy router uses its own connection to the decoy host to *post* data for the client to later download. The decoy router does this by uploading data to a location specified by the client such as by replaying a client's HTTP POST observed on the client-decoy host connection, or alternatively, by sending a message, with an appropriately tagged subject line, to an email address specified by the client.

Client-decoy router communication is thus asynchronous and does not occur within a single connection. Instead, the client and decoy router post and read information independently from each other on separate connections to the same or different decoy hosts. The different connections are made part of a *decoy routing session* by having the client share not just the TLS session keys with the decoy router but also the associated TLS session resumption information (e.g., pre-shared key or ticket). In a decoy routing session, once a client has authenticated with a decoy router on one connection, that authentication can be bootstrapped to establish another connection to a possibly different decoy host. This differs from the use of "quilting" in Curveball [19] and Rebound [13] which completes handshakes on multiple connections and then uses those connections in parallel to allow the client to more quickly download data.

MultiFlow is designed to work within the TLS 1.3 protocol [6], specifically Internet-Draft draft-ietf-tls-tls13-28. Much of the design extends to session tickets and session IDs as used in TLS 1.2 [5]. There are, however, a number of differences between TLS 1.3 and TLS 1.2 that impact decoy routing. First, TLS 1.3 incorporates forward secrecy via a Diffie-Hellman key exchange in which the ClientHello and SeverHello use the *key_share* field to derive a shared secret that is used as the Handshake Secret. Second, all handshake traffic after the ClientHello and ServerHello is encrypted using handshake traffic keys. Third, rather than using primarily the ClientRandom, ServerRandom, and Premaster Secret to derive keys, as in TLS 1.1 and 1.2, in TLS 1.3, hashes of all messages exchanged are incorporated.

We next overview the MultiFlow protocol. Our notation is based on that of Rebound [13]. We use *C* to refer to the MultiFlow client, *DH* to refer to the decoy host, and *DR* to refer to the decoy router.

- $S(m) \rightarrow R$ denotes that message m is sent by sender S to receiver R.
- $S(m) \xrightarrow{observer} R$ denotes that message m is sent by sender S to receiver R, and observed by *observer*.
- $E_k^{i,j}(m)$ denotes message m is encrypted with key k established between endpoints i and j.

3.1 MultiFlow Handshake

Fig. 3 overviews the MultiFlow handshake and details the changes to the TLS protocol. In Step 1, a ClientHello is sent. Like in Rebound [13], the client and decoy router share a secret key from which sentinel values are generated periodically. Each sentinel must be at least 112 bytes and is split into a number of sub-strings. The first 32 bytes, i.e., $\underline{\textbf{Sentinel}[\textbf{0}:\textbf{31}]}$, are used as the ClientRandom to signal the client's presence to the decoy router and authenticate the client. When the decoy router observes an appropriate ClientRandom, the decoy router starts passively recording traffic on that connection. $\underline{\textbf{Sentinel}[\textbf{32}:\textbf{63}]}$ is used as the public part of the client's *key_share*, corresponding to the client's Diffie-Hellman public key value while *Sentinel[64:95]* corresponds to the client's Diffie-Hellman private key value. Thus, by seeing the ClientRandom, the decoy router immediately obtains the client's Diffie-Hellman public and private keys. *Sentinel[96:111]* is a shared symmetric key, $K_{Sym}^{C,DR}$ between the client and decoy router.

In Step 2 of Fig. 3, the decoy host responds to the ClientHello with its own ServerHello. As the decoy router does not see traffic from the decoy host, the client uses stencil coding [13] or chosen-ciphertext steganography [23] to covertly exfiltrate the 98 bytes of information the client receives in Steps 2-7 and needed by the decoy router for key computation and session resumption. In Step 8, from $\underline{\textbf{StencilMessage}(\textbf{98 bytes})}$, the decoy router obtains the 2 byte ciphersuite and 32 byte *key_share* selected by the decoy host, the 32 byte hash, $Hash(ClientHello...ServerHello)$, and the 32 byte hash, $Hash(ClientHello...ServerFinish)$. From this information and the already known Client Diffie-Hellman keys, the decoy router computes the HandshakeSecret and the *handshake traffic keys*, and then the MasterSecret and *application traffic keys*.

In the TLS 1.3 draft [6] the record size limit is 2^{14} bytes. Using stencil coding [13] we expect to be able to hide 1 byte in every 16-byte block of the TLS record, thus hiding 98 bytes requires a 1568 byte record. We observe that for many websites, the initial GET request is small, unless large cookies or data are uploaded, with initial TLS 1.3 application record sizes often less than 100 bytes. To thwart traffic analysis, it may thus be necessary

Client opens new connection to DH

1. ClientHello: $C(\underline{\textbf{Sentinel}[\textbf{0}:\textbf{31}]}, \underline{\textbf{Sentinel}[\textbf{32}:\textbf{63}]}, ciphersuites, groups, signature\_algorithms, psk\_modes) \xrightarrow{DR} DH$
2. ServerHello: $DH(random, key\_share, ciphersuite, group, signature\_algorithm) \rightarrow C$
3. Certificate: $DH(E_{HS}^{C,DH}(Certificate)) \rightarrow C$
4. CertificateVerify: $DH(E_{HS}^{C,DH}(Signature(Hash(ClientHello...Certificate)))) \rightarrow C$
5. ServerFinish: $DH(E_{HS}^{C,DH}(MAC(Hash(ClientHello...CertificateVerify)))) \rightarrow C$
6. ClientFinish: $C(E_{HS}^{C,DH}(MAC(Hash(ClientHello...CertificateVerify)))) \xrightarrow{DR} DH$
7. NewSessionTicket: $DH(E_{App}^{C,DH}(ticket\_lifetime, ticket\_nonce, ticket, psk\_identity, extensions)) \rightarrow C$
8. ApplicationRecord: $C(E_{App}^{C,DH}(\underline{\textbf{StencilMessage}(\textbf{98 bytes})})) \xrightarrow{DR} DH$
9. ApplicationRecord: $DH(E_{App}^{C,DH}(\underline{\textbf{HTTP}_{\textbf{Req}}(\textbf{E}_{\textbf{Sym}}^{\textbf{C,DR}}(\textbf{NewSessionTicket}))})) \rightarrow C$

Authentication: DR resumes client session

10. ClientHello: $DR(random, key\_share, \underline{\textbf{psk\_identity}}, \underline{\textbf{psk\_binder}}, ciphersuites) \rightarrow DH$
11. ServerHello: $DH(random, key\_share, selected\_psk\_identity, ciphersuite) \rightarrow DR$
12. ServerFinish: $DH(E_{HS}^{DR,DH}(MAC(Hash(ClientHello...ServerHello)))) \rightarrow DR$
13. ClientFinish: $DR(E_{HS}^{DR,DH}(MAC(Hash(ClientHello...ServerHello)))) \rightarrow DH$

Figure 3: MultiFlow handshake using TLS 1.3 Internet-Draft draft-ietf-tls-tls13-28 [6]. $E_{HS}^{i,j}$ and $E_{App}^{i,j}$refer to encryption using the handshake and application keys respectively, with superscripts indicating the two endpoints that established the keys. $E_{Sym}^{C,DR}$ refers to encryption using the symmetric key shared between the client and decoy router. $Hash(Msg_1...Msg_N)$ indicates a hash taken over all messages exchanged starting with Msg_1 and ending with Msg_N. Bold-face underlined fields correspond to covert data exfiltrated by the client or decoy router. C denotes the client, DH denotes the decoy host, and DR denotes the decoy router.

to stencil across multiple application records (or attempt more efficient stenciling, as in TapDance [23]).

To resume a session, the decoy router requires the 219 byte *NewSessionTicket*, sent by the decoy host in Step 7. In Step 9, the client exfiltrates these 219 bytes as application data encrypted with $K_{Sym}^{C,DR}$ and hidden in an HTTP request. Using the *application traffic keys*, the decoy router decrypts the records in Steps 8-9 to get the **NewSessionTicket**, which comprises 219 bytes in total: a 4 byte *ticket_lifetime*, a 4 byte *ticket_age_add*, a 1 byte *ticket_nonce*, a 208 byte *ticket* (aka **psk_identity**), and a 2 byte *extensions*. In Step 10, the decoy router resumes the session using the **psk_identity** and the **psk_binder** (which corresponds to $Hash(ClientHello')$ where $ClientHello'$ is the $ClientHello$ excluding the list of binders). The *pre_shared_key* used in cryptographic operations for the resumed session is computed from the ResumptionMasterSecret, which is computed from the previous connection's MasterSecret, and from the *ticket_nonce* in the NewSessionTicket.

Decoy router authentication of client. The $Sentinel[0:31]$ sent in Step 1 of Fig. 3 is used to authenticate the client, while the session resumption in Steps 10-13 is used to check client liveness. If the decoy host accepts the resumed session as valid rather than falling back to a new handshake, then the decoy router considers the client to be a live client. Now instead of an adversary probing a connection to determine whether decoy routing is being used, instead, the decoy router is probing the connection to determine client liveness.

Suppose an adversary replays client traffic to the decoy host and the decoy router is still able to resume the session. Then the decoy router will still consider the client to be valid. However, the adversary will not be able to detect the presence of a decoy router on the path to the decoy host, because the decoy router does not modify, delay, or inject traffic on the client-decoy host connection. Suppose instead an adversary replays client traffic to a different host under adversary control. If the adversary observes the decoy router attempting to resume the client's connection to this new host then the adversary can infer that decoy routing is in use. We can mitigate this attack by having the client exfiltrate the destination server to which to resume the connection somewhere within the encrypted data.

Client authentication of decoy router. Since the decoy router never directly communicates with the client, the client only becomes aware of the presence of a decoy router when the client is able to later download covert data that the decoy router has posted using the tunnel.

Tunnel 1: On its own connection to DH, DR uploads covert data to URI specified by client

14. ApplicationRecord: $C(E_{App}^{C,DH}(\underline{\mathbf{HTTP_{POST}}(\mathbf{URI}=\mathbf{X},\mathbf{E}_{\mathbf{Sym}}^{\mathbf{C,DR}}(\mathbf{CovertRequest}))})) \xrightarrow{DR} DH$
15. ApplicationRecord: $DR(E_{App}^{DR,DH}(\underline{\mathbf{HTTP_{POST}}(\mathbf{URI}=\mathbf{X},\mathbf{E}_{\mathbf{Sym}}^{\mathbf{C,DR}}(\mathbf{CovertResponse}))})) \rightarrow DH$
16. ApplicationRecord: $C(E_{App}^{C,DH}(\underline{\mathbf{HTTP_{GET}}(\mathbf{URI}=\mathbf{X})})) \xrightarrow{DR} DH$

Tunnel 2: DR sends covert data as body of email to email address specified by client

14. ApplicationRecord: $C(E_{App}^{C,DH}(\underline{\mathbf{HTTP_{Req}}(\mathbf{E}_{\mathbf{Sym}}^{\mathbf{C,DR}}(\mathbf{EmailAddress}=\mathbf{X},\mathbf{CovertRequest}))})) \xrightarrow{DR} DH$
15. ApplicationRecord: $DR(E_{Email}^{DR,ES}(\underline{\mathbf{EmailAddress}=\mathbf{X},\mathbf{EmailBody}=\mathbf{E}_{\mathbf{Sym}}^{\mathbf{C,DR}}(\mathbf{CovertResponse})})) \rightarrow ES$
16. ApplicationRecord: $C(E_{Email}^{C,ES}\underline{(\mathbf{Download\ emails\ for\ EmailAddress}=\mathbf{X})}) \rightarrow ES$

Figure 4: Possible MultiFlow tunnels. We use the same notation as Fig. 3. ES denotes an email server.

3.2 MultiFlow Tunnel

Decoy routing protocols typically require the decoy router to interfere with the client-decoy host connection in some way: either by blocking connection traffic and re-writing packets, or by passively watching traffic and forging and injecting new packets impersonating the decoy host. This can introduce traffic analysis attacks (or probing attacks on the decoy host when forging is used). Instead, the MultiFlow tunnel uses a message-board based approach. This enables the client and decoy router to communicate with each other on different connections to the same or different decoy hosts, and removes the need for tight time coupling when tunneling data back to the client. Fig. 4 overviews two possible MultiFlow tunnels.

Client to decoy router communication. In both tunnels in Fig. 4, the client sends $\underline{\mathbf{E}_{\mathbf{Sym}}^{\mathbf{C,DR}}(\mathbf{CovertRequest})}$ to the decoy router via the client's connection with the decoy host. This covert data can be hidden inefficiently in traffic via the use of stencil coding [13] or chosen-ciphertext steganography [23]. Covert data can be more efficiently exfiltrated by hiding the data in application data encrypted within a TLS record such as via a malformed GET request URI [13, 20] or a cookie field.

Decoy router to client communication. The decoy router sends data to the client by uploading the requested covert data to a location from which the client can later access and download. Because the adversary does not have the TLS keys or the client's username and password, the adversary cannot decrypt, identify, or access the resource being modified on the decoy host. Because the decoy router never rewrites or modifies traffic on the connection between the client and decoy host, traffic analysis attacks are mitigated. There are several options for how covert data can be uploaded, described next.

HTTP POST based tunnel. In Step 14 of Tunnel 1 in Fig. 4, the client sends an HTTP POST to the decoy host. In Step 15, the decoy router replays this HTTP POST on its own connection to the decoy router, but now modified to upload different data, $\underline{\mathbf{E}_{\mathbf{Sym}}^{\mathbf{C,DR}}(\mathbf{CovertResponse})}$. In Step 16, the client re-downloads the URI associated with the HTTP POST in Step 14 and obtains the covert data that the decoy router posted. For some websites, the client may also need to exfiltrate its login credentials and specific URIs to which the decoy router should post.

Email-based tunnel. In Step 14 of Tunnel 2 in Fig. 4, the client exfiltrates to the decoy router an email address to which covert data should be sent. In Step 15, the decoy router sends an email to the email address specified by the client, with $\underline{\mathbf{E}_{\mathbf{Sym}}^{\mathbf{C,DR}}(\mathbf{CovertResponse})}$ as the email body while the email subject line is appropriately tagged with a sequence number and other identifying information. Even though tunneling data via email gives higher delay, it allows for much higher throughput than, for instance, an error-based channel.

Cloud-based tunnels. The decoy router uploads covert data to Dropbox or another cloud service via a command line interface. Cloud service command line interfaces are implemented as an application that makes API calls to the cloud server. The client thus needs a cloud service account, permission for the cloud service command line interface to access its account, and an authentication token which the client must then exfiltrate to the decoy router. The decoy router must then navigate the interface, which may be hard to automate using a tool such as Selenium [4] because the decoy router may potentially need to visit a URL on a web browser to obtain a token.

3.3 Extensions

Virtual symmetric routes. Since MultiFlow communication between the client and decoy router is asynchronous, there is no strict timing requirement. Thus, it might be acceptable for the decoy router to incur the delays necessary to coordinate with other decoy routers and de-

tect the reverse traffic going from the decoy host back to the client. The idea of virtual symmetric routes is similar to the collaboration among decoy routers done in recent work [10] which enables symmetric operation over asymmetric routes. The ability to see both directions of traffic simplifies the work that the decoy router must do to authenticate the client and tunnel covert data back to the client, and lets the decoy router more easily obtain the information needed to resume a client's session and compute the handshake and application traffic keys.

Cross-server decoy routing. The first decoy host to which a client connects must be on a path that traverses a decoy router, so that the decoy router can obtain the information needed to authenticate the client as well as resume the client's session. The decoy host used for tunneling covert data from the decoy router to the client, however, need not go through a decoy router, depending on the kind of tunnel used. As long as the decoy router knows the necessary session information, client login credentials, and posting location (e.g., email address or URI), the decoy router to client tunnel tunnel can use different decoys hosts than does the client to decoy router tunnel. In fact, once the handshake is completed, the client and decoy router can choose to discard their TLS connections to the original decoy host, and instead obtain covert data from each other via a virtual message board on yet another decoy host which need not be on the path of a decoy router. This mitigates the TLS termination attack described at the end of Section 2.

Use with other decoy routing protocols. The MultiFlow handshake could be used with other decoy routing protocols, particularly those that operate as a tap but do not authenticate their client such as TapDance [23].

3.4 Implementation Issues

We do not have an implementation of MultiFlow at this time. Instead, our preliminary experiments have focused on the feasiblity of MultiFlow using the s_client from OpenSSL's 1.1.1-pre2 release [1, 7] which works with TLS 1.3, and Scapy's TLS library [2, 3]. We have verified session resumption using a different source IP address is feasible. We next describe some expected implementation issues.

Information leakage from session resumption. One potential issue is the possibility of single-use tickets as described for 0-RTT data [6]. The adversary will not, however, itself be able to resume the session as it would need to know the NewSessionTicket, which is encrypted by the decoy host using the handshake traffic keys. Single-use tickets may actually benefit MultiFlow as they would alert the decoy router to an adverary's attempts to resume the session. If the decoy router is able to resume the session, the decoy router should then get an additional single-use session ticket to use.

Replaying HTTP POSTs. The decoy router can identify which HTTP POSTs to replay by tags inserted by the client. A harder problem to solve is how the client identifies or generates appropriate HTTP POSTs to replay. The client could require the user to web browse and perform actions that are likely to generate an HTTP POSTs, such as posting comments on a web page. But whether the HTTP POST is accepted by the decoy host may also depend on whether it contains the appropriate cookies and other fields set in the header, requiring the decoy router's replayed HTTP POST to contain the same fields.

Privacy of HTTP-posted data. An adversary must be prevented from accessing covert data uploaded by the decoy router. Thus, HTTP POSTs must be done using a private account that requires the client (and consequently also the decoy router) to login. This will prevent the adversary from simply crawling the entire decoy host site to identify the posted data or trying to confuse the client by posting nonsense. The client must thus exfiltrate its login credentials to the decoy router: for this reason, the email based tunnel may be preferable for the client, as the decoy router would not need to login as the client.

Leakage of private information. The proposed MultiFlow tunnels require some leakage of private information to the decoy router, such as a username and password or an email address. Alternative approaches that do not require leakage of private information are thus of interest. For instance, while most message boards are implemented using identities linked to passwords, this is not a requirement of such a system. It is possible to have a capability-based rather than identity-based system as well as quasi-anonymous boards, such as where the board provides anonymity but the messages themselves are keyed with identities.

4 Conclusions

In this work, we design MultiFlow, a new decoy routing protocol that (1) checks client liveness with a tap-based decoy router, (2) mitigates traffic analysis and connection probing attacks, and (3) amortizes decoy routing setup cost and mitigates TLS termination attacks. MultiFlow achieves this by leveraging the TLS session resumption mechanism to enable cross-connection and cross-server decoy routing.

5 Acknowledgments

The authors are grateful to Daniel Ellard for his insightful comments on this work, and to our shepherd, Eric Wustrow, and the anonymous reviewers for their feedback that has much improved this paper.

References

[1] OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org. Accessed: 2018-05-15.

[2] Scapy: interactive packet manipulation program. https://scapy.net/. Accessed: 2018-05-15.

[3] Scapy with SSL/TLS. https://github.com/tintinweb/scapy-ssl_tls. Accessed: 2018-05-15.

[4] SeleniumHQ Browser Automation. https://www.seleniumhq.org/. Accessed: 2018-05-15.

[5] The Transport Layer Security (TLS) Protocol Version 1.2. https://www.rfc-editor.org/rfc/rfc5246.txt. Accessed: 2018-05-15.

[6] The Transport Layer Security (TLS) Protocol Version 1.3. https://tools.ietf.org/html/draft-ietf-tls-tls13-28. Accessed: 2018-05-15.

[7] Using TLS 1.3 with OpenSSL. https://www.openssl.org/blog/blog/2018/02/08/tlsv1.3/. Accessed: 2018-05-15.

[8] ABRAMOWITZ, M. J. Freedom in the wworld 2018: The annual survey of political rights and civil liberties. https://freedomhouse.org/report/freedom-world/freedom-world-2018, 2018.

[9] BOCOVICH, C., AND GOLDBERG, I. Slitheen: Perfectly imitated decoy routing through traffic replacement. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1702–1714.

[10] BOCOVICH, C., AND GOLDBERG, I. Secure asymmetry and deployability for decoy routing systems. *Proceedings on Privacy Enhancing Technologies 3* (2018), 1–20.

[11] CESAREO, J., KARLIN, J., REXFORD, J., AND SCHAPIRA, M. Optimizing the placement of implicit proxies. Tech. rep., Princeton University, 2012.

[12] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. Tech. rep., Naval Research Lab Washington DC, 2004.

[13] ELLARD, D., JONES, C., MANFREDI, V., STRAYER, W. T., THAPA, B., VAN WELIE, M., AND JACKSON, A. Rebound: Decoy routing on asymmetric routes via error messages. In *Local Computer Networks (LCN), 2015 IEEE 40th Conference on* (2015), IEEE, pp. 91–99.

[14] FROLOV, S., DOUGLAS, F., SCOTT, W., MCDONALD, A., VANDERSLOOT, B., HYNES, R., KRUGER, A., KALLITSIS, M., ROBINSON, D. G., SCHULTZE, S., ET AL. An isp-scale deployment of tapdance. In *7th USENIX Workshop on Free and Open Communications on the Internet (FOCI 17)* (2017).

[15] HE, Y., FALOUTSOS, M., KRISHNAMURTHY, S., AND HUFFAKER, B. On routing asymmetry in the internet. In *Global Telecommunications Conference, 2005. GLOBECOM'05. IEEE* (2005), vol. 2, IEEE, pp. 6–pp.

[16] HOUMANSADR, A., NGUYEN, G. T., CAESAR, M., AND BORISOV, N. Cirripede: circumvention infrastructure using router redirection with plausible deniability. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 187–200.

[17] HOUMANSADR, A., WONG, E. L., AND SHMATIKOV, V. No direction home: The true cost of routing around decoys. In *NDSS* (2014).

[18] JOHN, W., DUSI, M., AND CLAFFY, K. C. Estimating routing symmetry on single links by passive flow measurements. In *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference* (2010), ACM, pp. 473–478.

[19] KARLIN, J., ELLARD, D., JACKSON, A. W., JONES, C. E., LAUER, G., MANKINS, D., AND STRAYER, W. T. Decoy routing: Toward unblockable internet communication. In *FOCI* (2011).

[20] NASR, M., ZOLFAGHARI, H., AND HOUMANSADR, A. The waterfall of liberty: Decoy routing circumvention that resists routing attacks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 2037–2052.

[21] SCHUCHARD, M., GEDDES, J., THOMPSON, C., AND HOPPER, N. Routing around decoys. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 85–96.

[22] SCHWARTZ, Y., SHAVITT, Y., AND WEINSBERG, U. On the diversity, stability and symmetry of end-to-end internet routes. In *INFOCOM IEEE Conference on Computer Communications Workshops, 2010* (2010), IEEE, pp. 1–6.

[23] WUSTROW, E., SWANSON, C., AND HALDERMAN, J. A. Tapdance: End-to-middle anticensorship without flow blocking. In *USENIX Security Symposium* (2014), pp. 159–174.

[24] WUSTROW, E., WOLCHOK, S., GOLDBERG, I., AND HALDERMAN, J. A. Telex: Anticensorship in the network infrastructure. In *USENIX Security Symposium* (2011).