Slitheen: Perfectly imitated decoy routing through traffic replacement

Cecylia Bocovich
University of Waterloo
cbocovic@uwateroo.ca

Ian Goldberg
University of Waterloo
iang@cs.uwateroo.ca

ABSTRACT

As the capabilities of censors increase and their ability to perform more powerful deep-packet inspection techniques grows, more powerful systems are needed in turn to disguise user traffic and allow users under a censor's influence to access blocked content on the Internet. Decoy routing is a censorship resistance technique that hides traffic under the guise of a HTTPS connection to a benign, uncensored "overt" site. However, existing techniques far from perfectly mimic a typical access of content on the overt server. Artificial latency introduced by the system, as well as differences in packet sizes and timings betray their use to a censor capable of performing basic packet and latency analysis. While many of the more recent decoy routing systems focus on deployability concerns, they do so at the cost of security, adding vulnerabilities to both passive and active attacks. We propose Slitheen, a decoy routing system capable of perfectly mimicking the traffic patterns of overt sites. Our system is secure against previously undefended passive attacks, as well as known active attacks. Further, we show how recent innovations in traffic-shaping technology for ISPs mitigate previous deployability challenges.

CCS Concepts

•**Security and privacy → Privacy protections;** *Network security;*
•**Networks → Network protocols;**

Keywords

censorship resistance, decoy routing, network latency, TLS, HTTP state

1. INTRODUCTION

Historically, censorship efforts paralleled advances in society and technology that promoted the production and dissemination of information. From literature [6], to music and art [2], almost all forms of human expression have undergone scrutiny and censorship by powerful authorities, including governments, religious organizations, and individuals. Resistance to censorship comes in the form of faster and more reliable methods of distributing information, prompting in turn an increase in the efforts of the censor to combat these innovations.

The cat-and-mouse relationship between censors in power and their resisting constituents continues to the present day. The Internet, although originally a tool developed by the United States government and research laboratories as a backup communication network, has blossomed into a diverse and powerful means of communication, social organization, and distribution of information. Unfortunately, governments and other political authorities have recognized the potential of uninhibited international communication to change the balance of power. Many countries as a result have adopted a centralized network infrastructure [1,16] that makes traffic easy to surveil and filter.

The capabilities of censors to monitor and control traffic inside their sphere of influence has expanded in the last few years, necessitating increasingly rigorous techniques to counteract state-level censorship of the web. In response to increasingly popular censorship-resistance tools such as Tor [4], censoring authorities such as the Great Firewall of China (GFC) have used sophisticated deep-packet inspection (DPI) techniques [23] to analyze application-level data in TLS connections to detect the usage of censorship circumvention techniques.

Decoy routing [7,12,14], or end-to-middle (E2M) proxying [25, 26], is a proposed solution to censorship that combats state-level adversaries with state-level defenses. Decoy routing relies on the deployment of relay stations to routers belonging to friendly participating ISPs outside of the censor's sphere of influence. Users access blocked content by tagging traffic to an uncensored, *overt* website, indicating to a deployed relay station—on the network path between the user and the benign site—that they wish to access blocked content *covertly*. The tag provides the means for the relay station to open a proxy connection to the censored website. By moving the client side of the proxy to the middle of the network, the client's use of decoy routing systems remains undetectable by the censor.

Existing decoy routing systems are vulnerable to latency analysis and website fingerprinting attacks. In 2012, Schuchard et al. [18] showed that simple latency analysis allows a censor to not only distinguish decoy routing traffic from regular traffic, but also determine which censored sites were accessed by the client. Although there is no evidence of censors using more traditional website fingerprinting techniques [11,21] to take advantage of the differences in packet sizes, timings, and directionality to distinguish between a regular visit to the overt site and a disguised visit to a different censored page, the capabilities of modern censors are sufficient to support these more sophisticated techniques.

There is a trade-off in existing systems between addressing challenges to deployment and resisting active attacks. The success of decoy routing schemes relies on adoption by a large and diverse group of ISPs. As such, challenges to deployment such as requiring in-line blocking and symmetric flows have proven to be prohibitive enough to inhibit the use of decoy routing schemes. Recent attempts at reducing these requirements have exposed the system to active attacks [25] that definitively identify decoy traffic, or simple passive attacks due to their highly unusual traffic patterns [7]. Whether through an active attack or by identifying unusual browsing patterns, a censor that is able to identify the use of decoy routing may block the client's use of the system or seek out the client for recrimination, rendering the service in the best case useless and dangerous at worst. We argue that as the capabilities of both censors and friendly ISPs continue to grow, the challenges to deployability will lessen and the need for highly secure systems will grow.

Our contribution. In this paper we propose Slitheen,[1] a novel decoy routing system that defends perfectly against latency analysis and fingerprinting attacks by perfectly mimicking an access to an allowed, uncensored site. We use careful knowledge of the HTTP protocol to deliver censored content to users in the place of image or video resources from the overt, decoy site. By only replacing unnecessary "leaf" resources, we ensure that the client will fully load the overt site in the same manner and timing as a typical access. A key feature of our replacement protocol is to forward packets as soon as possible after they arrive at the station to prevent adding latency in the replacement process. This requires keeping track of the TCP, TLS, and HTTP states of each flow in order to correctly handle delayed or missing packets. Our technique forces censored content to hold the same shape as benign traffic to the overt destination, eliminating the censor's ability to use latency or packet sizes and timings to identify Slitheen traffic.

We stand firmly on the side of security in the above-mentioned security-deployability trade-off, but argue that advances in traffic shaping technology mitigate previously prohibitive barriers faced by potential participants. Traffic shapers support the practice of in-line blocking and allow ISPs to force traffic that crosses into their domain into symmetric flows. Still, to reduce the amount of overhead our system adds to regular traffic flow, we only institute flow blocking of downstream data.

In the next section, we give a general overview of decoy routing and discuss existing techniques. We then introduce Slitheen, our proposed decoy routing scheme in Section 3. In Section 4, we discuss the security of our system and our defenses to both passive and active attacks, including latency analysis and website fingerprinting techniques. We end the discussion with a comparison to existing decoy routing schemes. In Section 5, we discuss our proof-of-concept implementation and follow with an evaluation of its performance in Section 6. Finally, we conclude in Section 7.

2. RELATED WORK

Early censorship circumvention systems consisted of a simple proxy. A user could hide their destination site from a censoring ISP by instead making an encrypted connection to a proxy server outside of the censor's area of influence. A censor would see that the client had made a connection to the proxy, but would be unable to determine which webpage the client visited. Tor [4] is a much more robust system, with stronger anonymity properties, that is widely used for censorship circumvention in a similar manner. Tor extends the simple proxy model by routing the user's traffic through a circuit of three proxies, or relays. The additional hops guarantee web-browsing anonymity for the client, even in the event that the censor has compromised one of the relays. However, Tor itself does not mask a client's participation in the system. Clients select relays from a publicly available list, one that is also available to a censoring authority. Censors have been known to block access to Tor by simply blacklisting connections to known Tor entry, or guard, relays [20].

In response to the blocking of Tor guards, the Tor Project has begun to gradually and selectively release the location of secret or hidden entry relays, called bridges [19]. A client may use these relays to continue circuit construction with publicly listed relays. As Tor bridges are not included in public Tor directories, they are much more difficult for a censor to track down and block. However, censors such as the Great Firewall of China (GFC) have employed other techniques to identify Tor traffic in the event that the client is using a hidden entry to Tor.

Tor traffic has several distinguishing characteristics that are necessary for providing anonymity, but allow a censor to distinguish Tor traffic from regular traffic to an unknown IP address. In 2012, Winter and Lindskog experimentally confirmed that the GFC could identify the use of Tor bridges with deep packet inspection (DPI) boxes due to the unique ciphersuite list sent by Tor clients in the TLS ClientHello message [23]. Furthermore, Tor traffic is distinguishable in the fact that all packets entering and leaving the Tor network are padded to 512-byte cells.

2.1 Pluggable Transports

In an effort to disguise connections to Tor, various *pluggable transports* have been proposed to change the shape and protect the contents of connections to Tor bridges and guard relays. To disguise easily identifiable traffic patterns, transports take two main approaches: obscuring the traffic patterns randomly, or mimicking existing protocols. Transports such as Obfsproxy [3] and ScrambleSuit [24] aim to make the traffic look as random as possible, relying on the censor's use of blacklisting to ignore a connection that does not match typical Tor traffic. Others, such as Marionnete [5], SkypeMorph [15], and StegoTorus [22] aim to mimic allowed protocols as closely as possible to avoid raising the suspicion of the censor.

Meek [10] is a recently proposed system that relies on an innovative technique called domain fronting to hide the true destination of a client's traffic. Unlike existing pluggable transports, Meek traffic appears to the censor to be heading to a legitimate, allowed website. The censored, covert destination (typically a proxy running on the same cloud service as the allowed website) is instead hidden in the `Host:` header field of the HTTP header to the allowed site. Destination information appears in three different places in an HTTPS request: the IP address, the TLS Server Name Indication (SNI) extension, and the Host header of the HTTP request. While the first two are viewable by a censor or any router between the client and the destination, the HTTP request is encrypted with all other application data after the completion of the TLS handshake. Domain fronting is the practice of specifying one domain, usually an edge server for a cloud service, in the IP address and SNI fields, while setting the encrypted HTTP host header to a different domain.

To make a connection to Tor using Meek, the client establishes a TLS connection with an edge server of an overt destination that allows domain fronting. Many Content Distribution Networks (CDNs) and large websites, such as Google App Engine and Amazon CloudFront, allow domain fronting for web applications that subscribe to their services. The proxy only has to subscribe and pay for bandwidth (shown to be between $0.10 and $0.20 USD per GB [10]) in order for their service to be accessible from an overt edge server. After establishing a connection to the overt destination, the client can issue HTTP requests to the Meek proxy. Packets are redirected to the proxy by the overt destination according to the host field of the HTTP header. The client's Tor traffic is then tunneled over HTTP to the proxy and sent to a Tor guard. In this way, the proxy to Tor "hides" behind the cloud service. To block all traffic to the Meek proxy, the censor would have to block all traffic to the front service, causing collateral damage.

Domain fronting relies on the deterrence of collateral damage. If the censor knows the existence of the domain fronted proxy, they may block it only by blocking the entire front service. This may be

[1]Slitheen is named after a Doctor Who alien capable of taking the exact form of its victims.

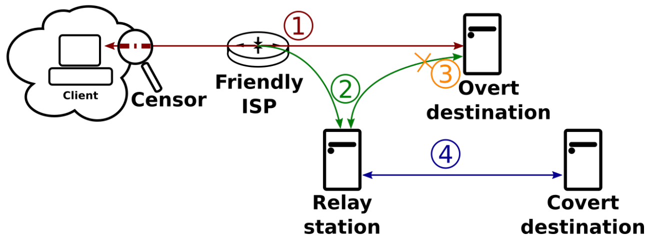


Figure 1: An overview of the Telex [26] architecture. A client first initiates a TLS handshake with the overt destination, tagging the ClientHello message (1). The relay station recognizes the tag, and then continues to passively monitor the TLS handshake (2). Upon receipt of the TLS Finished messages from both sides, the station decrypts and verifies the Finished messages with the session's TLS master secret, computed from the client's tag. Finally, the station will sever the connection to the overt site (3), and assume its role as a proxy to the censored, covert site (4). Slitheen is very similar, except that in step (3), the connection between the client and the overt destination is maintained and actively used.

prohibitively expensive in terms of collateral damage for large web services and therefore unappealing to the censor.

2.2 Decoy Routing

The first generation of decoy routers surfaced in 2011, proposed by three independent research groups. Telex [26], Cirripede [12], and Curveball [14] all use the same basic technique in which the client steganographically expresses a desire to access censored content covertly by tagging the setup messages in a seemingly benign connection to an *overt destination*, sometimes referred to as a *decoy server*. These tags are recognized by a friendly ISP with a deployed *relay station* on the path between the client and the overt destination, but are provably invisible to a censoring ISP without the relay station's private key. After the tag has been recognized, the station facilitates the flow of information between the client and a censored website via a man-in-the-middle proxy. The details of the tagging process and proxy setup vary; Figure 1 shows the general architecture of Telex, which is most similar to our system.

In Telex, the tag is placed in the random nonce of the ClientHello message that initiates a TLS handshake with the overt site. From this tag and the station's private key, the station can compute the client's Diffie-Hellman (DH) exponent, allowing it to compute the session's TLS master secret and man-in-the-middle the connection between the client and the overt destination. Upon the receipt, decryption, and verification of both TLS Finished messages, the station severs the connection to the overt destination and assumes its role as a proxy, preserving the server-side TCP and TLS state. The client may then connect to a censored covert webpage through the relay station; the traffic between the client and the covert destination appears to the censor as encrypted traffic to and from the overt site.

Cirripede takes a different approach by inserting a tag in the Initial Sequence Numbers (ISNs) of TCP SYN packets to register the client with the relay station over the course of 12 TCP connections. Once registered, the client initiates a TLS connection with an overt destination, now routed by the relay station through a *service proxy*, and sends an initial HTTP GET request. After the request goes through, the service proxy terminates the connection on behalf of the client, and begins to impersonate the overt destination. The proxy generates a new TLS session key, computed from the registration tag and station private key, and issues a ChangeCipherSpec message and Finished message to the client. Once the client responds with a valid Finished message, the service proxy begins to relay traffic between the client and the censored site until a predetermined time interval has passed.

In Curveball, the client and the relay station share a predetermined secret, obtained through the use of a covert channel. This secret is used to generate a tag recognizable by the station and is inserted in the ClientHello message of a TLS handshake to the overt destination, similar to Telex. Upon receipt of this tagged message, the relay station observes the completion of the TLS handshake with the overt destination, and assumes the role of the server, sending the client a Hello message in the form of a TLS record encrypted with the client-station shared secret. Once the client responds with a similar Hello message, the station begins to proxy traffic between the client and the censored site.

Despite the strong deniability properties of the three first-generation schemes, none achieved widespread adoption. All three systems require symmetric flows between the client and the overt destination (i.e., flows for which the upstream and downstream paths both traverse the same relay station), as well as in-line flow blocking (i.e., where the relay station must actively modify traffic by dropping, delaying, or modifying packets), speculated to be a prohibitively large barrier to participation by friendly ISPs [25]. To combat this shortcoming, Wustrow et al. proposed TapDance [25], a non-blocking, asymmetric decoy routing system. TapDance flows are tagged in an initial, incomplete HTTP GET request to the overt site, after the negotiated TLS handshake. The TapDance station recovers the tag (steganographically embedded in the ciphertext of an ignored GET request header), and uses the corresponding secret to encrypt a confirmation, mimicking an HTTP response from the overt site. The client then sends the station upstream requests for a censored site, making sure to never signify the completion of the initial GET request to the overt site. As such, the overt site will continue to receive upstream data from the client without complaint, thereby eliminating the need for flow blocking. The station fulfills the client's requests for censored content, issuing encrypted responses on behalf of the overt site. As the only downstream data from the server are the original TLS handshake messages (which are not needed by the TapDance station), and the TCP ACK messages following extra data from the incomplete header, the station does not need to witness this downstream traffic, making the scheme amenable to asymmetric flows.

Although TapDance solves two major challenges to deployment, it also exemplifies the trade-off between deployability and security. By leaving the connection between the client and overt destination open but abandoned, TapDance becomes vulnerable to active attacks by an adversarial censor. The station sends HTTP responses in place of the overt server, resulting in a discrepancy between the TCP state of the overt site and the TCP state witnessed by the censor. A passive censor would not notice this discrepancy, as the overt site will ignore client packets with an acknowledgement number higher than the overt site's TCP SND.NXT value. An active attacker, however, may determine the actual TCP state of the overt site by replaying a TCP ACK packet with a stale sequence number, prompting the server to reveal its current TCP state, and incriminating the client.

Rebound, proposed by Ellard et al. [7], is the most recent decoy routing system and aims to solve the trade-off mentioned above, providing an asymmetric solution that defends against an active adversary. Clients tag Rebound flows in a manner similar to Telex, by inserting a tag in the ClientHello message of a TLS handshake, enabling the Rebound station to compute the master secret of the TLS connection between the client and the overt site. They achieve an asymmetric version of the tagging procedure by leaking the values of the server random nonce and the ciphersuite to the Rebound station by embedding them in the ciphertext of initial HTTP GET requests from the client to the overt site. Once the Rebound station is able to man-in-the-middle the TLS connection, the client begins issuing requests for censored content embedded in invalid HTTP GET requests to the overt site. The relay station fetches the censored content and stores it in a queue. When the next invalid GET request is received by the Rebound station, the station replaces the URL field of the request with content from the censored site. This information is forwarded to the overt site, which "rebounds" the encrypted content, inside an HTTP error response.

Rebound maintains the connection between the client and the overt site, making it resistant to the TCP replay attack mentioned above. The TCP state of the overt site will report the TCP sequence and acknowledgement numbers expected by the censor. Furthermore, by rebounding content off of the overt site in the form of error messages, Rebound delivers downstream data from the proxy to the client even if the underlying network routes are asymmetric. There are two major barriers to the adoption of Rebound as a censorship resistance technique. The first barrier is that a client must send upstream data in an equal amount to the downstream data they receive to avoid mismatched TCP sequence numbers upstream, alerting the censor of a decoy session. In typical Internet usage, the ratio between upstream data sent and downstream data received by the client is very low; this trend is reflected in the bandwidth provided by most ISPs. The second barrier to adoption is the flood of bad HTTP GET requests that the client sends to the overt site. The frequency and size of these requests are reminiscent of HTTP flooding, a class of Denial of Service attacks, and will likely be blocked by the overt site.

The two most recent decoy routing systems have addressed the shortcomings of first generation systems, notably those of deployability in the case of TapDance, and security against some active (but not all passive) attacks in the case of Rebound. However, all existing systems are vulnerable to the timing analysis attacks introduced by Schuchard et al. [18]. They showed that differences in latency due to fetching content from a possibly distant censored server through the decoy routing proxy is enough to not only detect the usage of a decoy routing system, but also fingerprint the censored webpage accessed. Although Rebound's stored queue of censored content reduces the latency that stems from proxying traffic between the client the covert destination, it does not account for the latency introduced by the relay station in replacing the contents of the HTTP GET request. Furthermore, all previous systems are vulnerable to traditional website fingerprinting techniques, in which the censor can compare packet sizes, timings, and directionality to differentiate between decoy and regular traffic while fingerprinting the censored site.

We designed Slitheen to defend against latency analysis and website fingerprinting attacks. We present a system that perfectly mimics the expected packet sequences from the overt site and techniques for reducing the latency introduced by the Slitheen station. Our system also provides strong defenses against active and routing-based attacks, including the TCP replay attack mentioned above. We give the details of our system and provide a security analysis and comparison to previous systems in Section 3 and Section 4, respectively.

3. SLITHEEN SYSTEM DETAILS

Slitheen defends against passive latency analysis and website fingerprinting attacks by perfectly imitating a typical access to an allowed, overt destination. We accomplish this by maintaining the connection to the overt site after the completion of the tagging protocol in the TLS handshake. At this point, the relay station is able to man-in-the-middle the TLS connection to the overt site and monitor or modify both upstream and downstream data. The station extracts upstream data to the covert destination from specialized headers in *valid* HTTP GET requests to the overt site, and replaces image or video resources from the overt site with downstream data from the covert destination. In this section, we give a high-level overview of the Slitheen architecture and follow with a detailed discussion of the replacement protocol. We show an overview of the Slitheen architecture in Figure 2.

3.1 Architecture Overview

A typical access to content on the web consists of a collection of TCP connections, over which flow HTTP requests and responses. The first connection to the overt destination typically requests an HTML document, which in turn prompts the client's browser to issue several more requests to HTTP servers (the same or different) to collect various resources such as cascading style sheets (CSS), JavaScript, images, and videos. Certain types of resources, such as CSS or Javascript, may in turn require additional resources to be fetched by the client's browser. Others, such as images or videos, are "leaf" content types in that they never contain a request for an additional resource.

In Slitheen, we have the client access the overt site exactly as a regular user would, fetching both the original HTML page from the overt site as well as all additional resources necessary to completely load the page. When a client initiates a Slitheen session with the desire to proxy information to a censored, blocked site, they first randomly generate a 32-byte Slitheen identification number. They then proceed to access the overt site through the use of an overt user simulator (OUS) running on the client's machine (as part of the Slitheen client code). The OUS is a headless browser that requests a page from an overt destination over a tagged TLS connection. When the OUS receives a resource that contains a request for additional resources, it issues these requests over tagged TLS connections to the servers that host them.

The tagging procedure of Slitheen is identical to that of Telex [26], though in principle we could use something different, as the tagging procedure is not the contribution of Slitheen. The client's OUS initiates a tagged flow with a decoy routing station by inserting a Telex tag into the random nonce of the ClientHello message at the beginning of the TLS handshake with the overt site. If the path between the client and the overt site crosses into the territory of a friendly ISP with a deployed Slitheen station, the tag is recognized and the station continues to passively observe the TLS handshake, allowing it to compute the shared master secret for the TLS connection.

After the TLS session has been established and the tagging procedure is complete, the Slitheen protocol deviates from Telex. Rather than terminating the connection to the overt site on behalf of the client or leaving it stale, the station continues to passively monitor the session as the OUS proceeds to request content from the overt site in the usual manner. When the OUS issues a valid HTTP GET request for a resource to the overt site, the Slitheen station inspects the headers of this request for an X-Slitheen header containing the Slitheen identification number of the client, and any upstream data meant for a covert destination. The station now associates the tagged flow with the given Slitheen ID, replaces the contents of this header with garbage bytes, and allows it to continue to the overt site. If the X-Slitheen header contained upstream data to be proxied to a covert destination, the station simultaneously relays this data, and stores any responses in a queue of content for the Slitheen ID, to later replace leaf content from overt sites

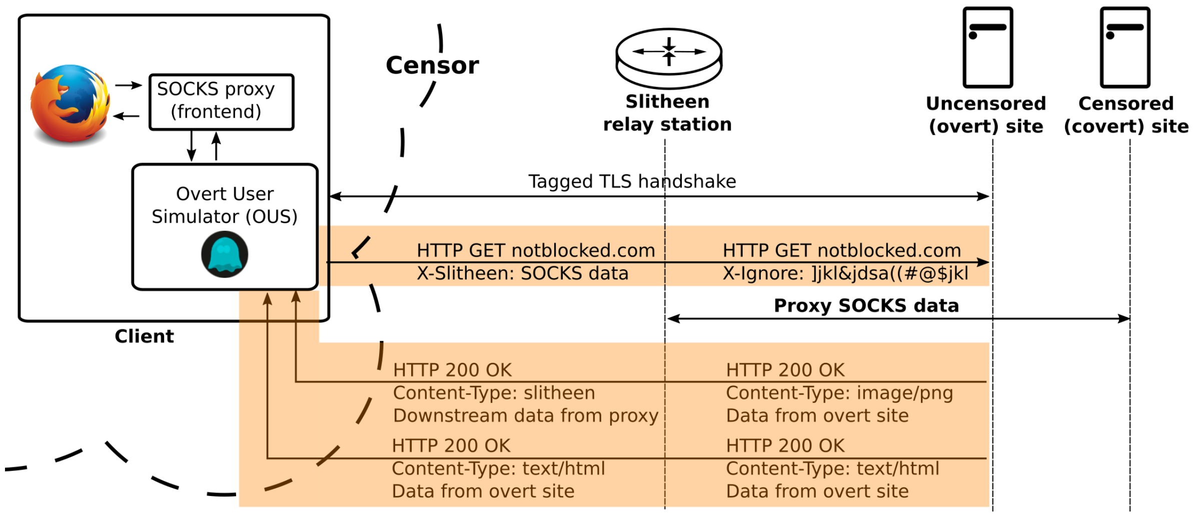


Figure 2: After the establishment of a TLS session between the client's Overt User Simulator (OUS) and the overt site, the Slitheen station may monitor the encrypted traffic (shaded) in both directions. The station receives upstream proxy data from the client in an X-Slitheen header of a valid HTTP GET request to the overt site. Once the station has relayed the upstream data to the censored site, it stores the downstream responses in a queue. When the station receives responses from the overt site, it replaces leaf content types such as images with the queued data. This data is then forwarded by the OUS to the SOCKS frontend and finally received by the client's browser. The censor sees only the TLS handshake and encrypted traffic to and from the overt site.

to the client associated with the given ID. To keep the size of HTTP requests consistent with the addition of the X-Slitheen header and data, the client may replace only non-essential headers or compress existing headers to be later decompressed by the relay station before they are forwarded to the overt site. If the existing headers are compressed, the X-Slitheen header is simply removed by the relay station.

When the Slitheen station receives downstream traffic from an overt site, it first decrypts the TLS record and inspects the HTML response for the content type of the resource. If the content type is not a leaf type, the station will re-encrypt the record and let the resource pass unaltered to the client. If the resource has a leaf content type, the station will replace the response body with data from the downstream queue pertaining to the Slitheen ID of the flow and change the content type of the resource to "slitheen". It then re-encrypts the modified record, recomputes the TCP checksum, and sends the packet on its way. If there is a shortage of downstream data, the station will replace the resource with garbage bytes, padding the response body to the expected length. When the OUS receives the resource, it sends all resources of the "slitheen" content type to be processed and sent to the client's (real, not OUS) browser. All other resources, it processes in the usual manner.

By replacing the leaf resources of valid HTTP requests, Slitheen perfectly imitates an access to an overt site. Regardless of advances in website fingerprinting techniques, a censor will be unable to distinguish between a Slitheen decoy routing session and a regular access to the overt site based on packet sequence patterns such as packet lengths, directionalities, and timings. As in Rebound, we completely eliminate one form of latency identified by Schuchard et al. by not waiting for responses from possibly distant covert destinations; unlike Rebound, however, we instead immediately replace leaf responses with content that is available in the saved downstream queue. The **key insight of Slitheen** is that whenever a packet arrives at the relay station from the overt destination, the relay station will immediately forward a packet toward the client with the same size and TCP state; only the (encrypted) contents of the packet will be possibly replaced with (again encrypted) censored content, and only when the replaced content is a leaf type. We show that this replacement process introduces a minimal amount of latency, leaving the censor unable to detect the usage of a decoy routing system, and give the results of timing analysis in Section 6.

Not only does our system defend against passive attacks by design, but also against known active attacks on decoy routing schemes. Slitheen defends against TCP replay attacks by actively maintaining the connection between the client and the overt site. Since our replacements match the sizes of requests and responses exactly, the TCP state between the client and the overt site as seen by the censor is the true TCP state. Furthermore, Slitheen eliminates the ability of the censor to identify its use through TCP/IP protocol fingerprinting. The station modifies only application-level data, which is unidentifiable by the censor as ciphertext. We do not need to mimic the server's TCP options, or IP TLS values as these are supplied by the overt site itself. We give a more complete security analysis of Slitheen and a comparison to existing systems in Section 4.

3.2 Content Replacement Details

While the replacement of the X-Slitheen headers and leaf content types is straightforward in theory, it is difficult to achieve in practice while also minimizing the latency introduced by the station. HTTP responses may be spread across multiple TLS records, and each record may contain multiple responses. Additionally, a record may be spread across multiple packets, leaving the station unable to decrypt a record to replace its contents or determine the content type of the responses it contains until the rest of the record has been received. Furthermore, packets may be delayed or dropped and arrive at the relay station out of order. Waiting for the receipt of an entire record before sending the observed packet to the client introduces an identifiable amount of latency, which may be used by the censor to detect the usage of Slitheen.

A simple solution to receiving a record fragment is to forward the record unchanged and forego any possibly replaceable responses it

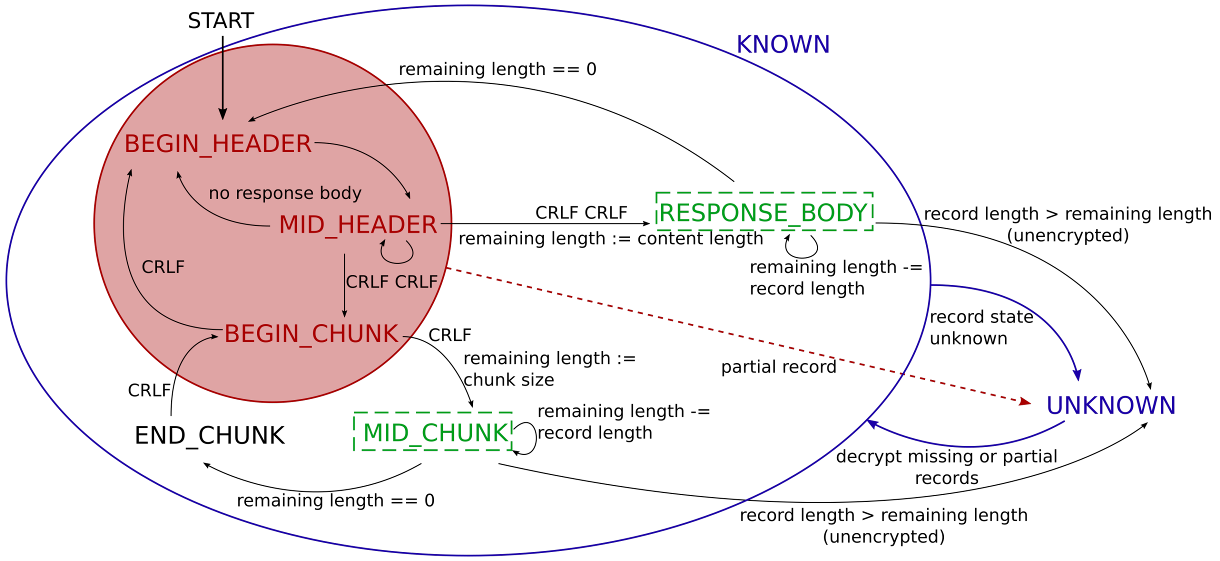


Figure 3: A flow may be in one of several TLS (blue) and HTTP (red, green, and black) states. When a new packet arrives that allows the relay station to find the beginning of a new TLS record, the station uses the record's length, its (possibly) decrypted data, and the length of the packet to determine the next HTTP state. States in the shaded red circle must be decrypted to decide the next state. If the flow is in a red, shaded state when the relay receives a partial TLS record it cannot decrypt due to missing data, the flow will enter into the UNKNOWN state until the remainder of the record is received and decrypted. This is represented by the dashed red arrow. States in green dashed boxes indicate states where data may be replaced. If the HTTP header showed a leaf content type, the relay station will construct a new record to replace the one(s) it receives. A flow with an HTTP state of UNKNOWN may recover its state by reconstructing partial or missing records and analyzing the decrypted data, along with the previous known state.

contains. However, as record sizes for large image files are frequently large themselves, this results in a significant drop in the bandwidth available for delivering censored content. To address this trade-off, we analyzed all possible states that may occur at the relay station upon the receipt of a packet from the overt site and determined a replacement procedure that maximizes the amount of downstream data that can be replaced without delaying packets that contain partial records.

Record state. When a packet is received by the relay station, the TLS record state of the flow determines whether the packet's contents begin with a new record, contain the contents of a previously processed record, or contain the remnants of a previous record and the beginning of a new record. The record's length, specified in the record header, determines how many full or partial records are contained in the current packet and how many bytes of subsequent packets contain the contents of the record. Although the relay station may not be able to decrypt a record if it is spread across multiple packets, it is still able to maintain a view of the record state. Depending on the HTTP state of the flow, these records may be safely replaced without being decrypted by the station.

A flow can have an unknown record state if packets arrive at the station out of order. If the delayed packet does not contain any new record headers, the station is able to maintain the record state and processes the received packet in the usual manner, assuming the eventual receipt of the missing packet. However, if the delayed packet contained the beginning of a new record, the station has lost the record state and can only regain it after the missing packet arrives. While the record state is unknown, the station is unable to encrypt modified records for the client, as it does not know the lengths or contents of the record(s) in a received packet.

HTTP state. The station also maintains information about the HTTP state of each flow, indicating whether the next record will contain all or part of a response header, or response body. We give the state machine for HTTP responses in Figure 3. The end of a header is determined, as specified in RFC 2616 [9], by the receipt of two consecutive carriage return and line feed characters (CRLF): one to signify the end of the last header field, and one to signify that there are no more header fields in the message. The length of the response is determined by the status code of the response, and the transfer encoding (in which case the length is updated with each subsequent "chunk") or the content length. The Content-Type header indicates to the station whether the subsequent response should be replaced.

The HTTP state of the flow is updated upon the receipt of a new record header. Depending on the HTTP state, a record does not need to be decrypted in order to be replaced. When the station receives a new record, it checks the record header to determine whether the record is contained in the TCP segment and may be decrypted, or whether the record is spread across multiple packets. It then determines, based on the HTTP state, whether the record may be replaced. If the HTTP state and the record's length indicates that it contains only a replaceable HTTP response body, the station will then construct a new record of the same length and fill it with downstream data from the client's queue. After encrypting the modified record, it sends the first part, matching the length of the record fragment in the received TCP segment, and stores the remainder of the modified record to replace the data in subsequent packets. After the entire record has been sent, the next TCP segment data will contain the header of a new TLS record.

If, however, the station is unable to decrypt a record that contains information about the response length or content type, the HTTP state of the flow will be unknown until the station receives the rest of the record and decrypts it. In this case, the contents of the record will be forwarded immediately to the client, without modification and a copy saved by the station to decrypted when the entire record arrives. Upon its receipt, the station can re-evaluate the HTTP state of the flow. Similarly, when the record state of the flow is unknown due to a delayed or dropped packet, the HTTP state of the flow will remain unknown until the station receives the missing packet. At this point, the station will determine the updated state and continue processing records.

3.3 Future Changes to HTTP

We have designed our system for use with HTTP/1.1, in which a user issues a sequence of HTTP GET requests to retrieve the resources on a page. However, the recently proposed HTTP/2.0 specification suggests several changes that increase the efficiency of page loads by reducing header sizes and allowing concurrently loaded HTTP responses. Our system requires minor changes to function with the new specification.

Header reduction in HTTP/2.0 is achieved through compression. The header fields of an HTTP request or response are compressed before transmission to their destination. We can still add our own headers to the list before compression, but special care must be taken to ensure that the total compressed size of the headers with our extra data does not vary significantly from the size of a typical HTTP header to the site. When the headers are received by the relay station, the station must uncompress the headers to analyze or modify them, and then re-compress the headers before forwarding the traffic. If modifications are made, the station must ensure that the size of the re-compressed HTTP message stays consistent.

To allow for multiple concurrently loaded resources, HTTP/2.0 encapsulates requests and responses in HTTP frames. Each request and corresponding response is identified by a unique stream ID. These stream IDs are later used by the client to demultiplex the sequence of frames into separate resources. The multiplexing of resources complicates our calculation of the HTTP state at the relay station as an incoming encrypted record may contain data from several streams. The station will need to keep track of the HTTP state of each stream, but without the ability to decrypt the record and determine which stream(s) it contains, the station will be unable to determine the next HTTP state of each stream and will not be able to replace the record contents with proxied downstream data. Without testing to determine how often the station will lose track of the HTTP state of a tagged flow, we are unable to guess at how difficult it will be to maintain a steady bandwidth for downstream proxy data. We leave this analysis for future work.

4. SECURITY ANALYSIS

We have analyzed the security of our system by examining the effectiveness of previously proposed decoy routing attacks [18,25, 26]. These attacks consider three different types of adversaries: a *passive adversary* capable of only monitoring traffic, an *active adversary* capable of both monitoring and modifying traffic by dropping, injecting, or changing packets inside their area of influence, and finally a *routing-capable adversary* who is able to not only change traffic, but also make routing decisions on traffic that leaves their network.

The goal of the adversary is ultimately to identify a decoy routing session. An adversary may also try to identify the censored content that the client is accessing through the decoy routing session. We do not consider attacks that allow the adversarial censor to perform unrealistic computations or utilize unrealistic amounts of resources; For example, we assume that the adversary is unable to distinguish a tagged ClientHello message from a truly random nonce, as doing so would violate a cryptographic assumption. Similarly, we assume that the adversary may not compromise the TLS session between the client and the overt site by brute-forcing the overt site's private key, or performing a TLS downgrade attack. Furthermore, deployed relay stations and overt sites are assumed to be geographically outside the censor's sphere of influence.

4.1 Latency and Fingerprinting Attacks

Added latency in decoy routing systems stems from two sources: (1) the additional time it takes for the relay station proxy to communicate with a possibly distant censored server, and (2) from the mechanisms of the proxy itself in processing and verifying the TLS handshake, and manipulating data that flows through it. In their experiments, Schuchard et al. found that there was a significant amount of latency from both sources independently, enough to identify the usage of previous decoy routing systems.

Slitheen defends perfectly against the first type of latency. The relay station does not wait to communicate with the covert destination, but forwards packets from the overt site immediately after possibly replacing their contents with queued downstream proxy data. Similarly, the station forwards upstream data immediately after processing the record's contents to extract the Slitheen ID of the flow and upstream data for the proxy.

The second type of latency, from the Slitheen proxy itself, is more difficult to prevent. Schuchard et al. show that Telex exhibited enough latency to detect its usage even by choosing a covert destination on the same server as the overt destination (effectively reducing the first type of latency to zero). Although Schuchard et al. were unable to determine the cause of the latency, their findings suggest some amount of overhead imposed by the relay proxy and TLS handshake protocol. While the tagging procedure of Slitheen matches that of Telex, our proxying protocol behaves very differently. If there is overhead introduced by the proxy on the relay station, it will not affect the rate at which incoming packets to the relay station are processed, replaced, and forwarded to their destination. We performed a latency analysis of our system by accessing an overt destination as both an overt site for tagged flows and as a regular, untagged access. Our results, given in Section 6.2, show that we do not introduce enough latency to identify the use of our decoy routing system by timing page loads.

In addition to identifying the use of decoy routing, Schuchard et al. show that latency can be used to fingerprint packet sequences and determine which censored webpage a client has accessed. Slitheen defends against both this and other traditional website fingerprinting attacks by eliminating not only the latency from accessing distant covert destinations, but also by forcing the packet timings, sizes, and directionality to exactly follow that of a regular access to the overt site. The latency fingerprinting method relies on differences between the latency distributions of visits to different censored sites. With Slitheen, accesses to different censored sites will all produce the same latency distribution, as the latency source is only in the decryption and re-encryption of records passing through the relay station.

This, coupled with the fact that the observed packet sequences of a regular access and a decoy access to any overt destination will be identical in terms of packet sizes, relative timing, and direction drastically reduces the censor's ability to distinguish between the two types of traffic.

4.2 Passive Attacks

In addition to timing and latency attacks, there are a number of other attacks an adversary may employ to detect the use of Slitheen.

Protocol Fingerprinting. Previous decoy routing schemes are susceptible to protocol fingerprinting attacks, in which the adversary leverages the possible differences in the TCP/IP implementations of the overt destination and the proxy. Mimicry is an inherently difficult problem, as any difference in options, parameters, or variable values can alert the censor to a suspicious change in the connection. The defense proposed by Wustrow et al. [25] requires each station to build a profile of each overt site that accounts for all possible variations in TCP options and IP header values. This solution is costly in terms of storage and also slow to update; a change in the TCP options or headers requires an immediate change at the station to evade censor scrutiny.

Slitheen eliminates risk of protocol fingerprinting by reusing the TCP and IP headers sent by the overt site. The only differences in the data sent by the overt site and the Slitheen proxy are the encrypted payload and the TCP checksum. Neither of these values provides the censor with any information that suggests the replacement of the requested resource with proxy data.

Station Malfunction. In the event that the relay station fails to recognize a tagged flow, a Slitheen client's OUS will interact with the overt site in the normal manner, avoiding suspicion. Telex and TapDance clients assume that the station is present and able to block or monitor upstream flows to the overt destination. If the station is absent in Telex or Tapdance, the connection to the overt site terminates early or results in an HTTP error message that may indicate their use to a passively monitoring censor.

4.3 Active Attacks

An active adversary is capable of modifying, injecting, or dropping traffic in addition to passive monitoring. The following attacks are known active attacks against previous decoy routing systems.

Tag Replay Attack. Our system inherits protection against a tag or handshake replay attack from the Telex handshake procedure. If an adversary attempts to replay a tag, they will not be able to successfully construct the TLS Finished message without knowledge of the shared secret, resulting in a connection terminated by the client and overt destination.

State-Controlled Root Certificates

In deployments where censors are actively performing man-in-the-middle attacks on TLS traffic by mandating the installation of a state-controlled root certificate, the resultant flows will not be tagged and will pass through the station unaltered. While this performs a denial of service attack on Slitheen, it does not reveal a client's usage of the system unless they include X-Slitheen headers in their upstream requests. To alert the client of the fact that their decoy routing session has not been safely established, we propose a slight modification to the TLS handshake in which the Slitheen station adds an additional input, seeded from the client's tag, to the TLS Finished hash sent to the client, after the station has verified that the Finished messages are correct. When the client receives the Finished message, they will verify it using the additional seeded input to determine whether the decoy session has been established. If it has, the client proceeds to include X-Slitheen headers with upstream data. If it has not, the client will verify the Finished message the traditional way and continue with a regular (non-decoy) fetch of the page. A man-in-the-middle capable of viewing the plaintext will therefore detect no unusual behaviour from the client.

Server Collusion. In previous systems, the censor could collude with or set up an overt destination server to entrap clients that use that server for decoy routing purposes. In Slitheen, the client's behaviour from the overt site's perspective will be identical to regular use with the exception of an X-Ignore header containing garbage bytes. If the existence of the X-Ignore header is a concern, the relay station can instead replace it with a common but mostly unused header. However, a censor that monitors information leaving the overt destination can compare ciphertexts to detect content replacement. In fact, no existing decoy routing system can completely defend against an adversary that has a complete view of packets entering and leaving both the client and the overt site. Our system increases the work of the adversary from previous systems by requiring the colluding parties to compare ciphertexts as opposed to metadata.

4.4 Routing-Capable Attacks

Routing-capable attacks were introduced by Schuchard et al. [18] and rely on the censor's ability to route packets through either a tainted path (i.e., one on which a Slitheen station resides between the client and the overt destination), or a clean path (i.e., a path with no Slitheen station between the client and the overt site). While a censor may not always be able to find a clean path to the overt destination, our system defends against an adversary that does have this ability. We also note that, as in Telex, the location of relay stations can be public knowledge, and therefore routing-capable attacks to determine whether a network path contains a relay station (as opposed to whether a *particular flow is using* a relay station) do not affect the security of our system.

TCP Replay Attack. In a TCP replay attack, the censor attempts to identify the use of decoy routing by testing whether the client has a TCP connection with the overt site. The censor can replay a TCP packet sent by the client on a clean path. In TapDance and first-generation decoy routing systems, the connection between the client and the overt site has been severed or abandoned and the overt site will issue a TCP RST packet or a stale TCP sequence number, signaling to the censor the usage of decoy routing. Note that in TapDance, the adversary does not need to find a clean path, but can inject a TCP packet into the stream. Since the TapDance station does not perform in-line blocking, the packet will be forwarded to the overt destination despite the fact that it traverses a tainted path.

Our system maintains a TCP connection to the overt destination, providing a defense against this type of replay attack. Every TCP packet sent by the client is received by the overt site throughout the duration of the decoy-routed connection. Upon the receipt of a replayed packet, the server will send a duplicate acknowledgement in an identical manner to a regular connection.

Crazy Ivan. The Crazy Ivan attack involves a censor with the ability to control the path a client's packets take to their destination to detect the usage of, or deny availability to, decoy routing. The censor allows a client to connect to the overt site through a tainted path, and waits until the TLS session has been established to redirect the flow down a clean path.

In previous systems, this attack gives the censor overwhelming evidence of decoy routing. Systems such as Telex, Cirripede, Curveball, and TapDance that sever or abandon the connection between the client and the overt destination will be unable to block packets sent down the new clean path, resulting in TCP RST packets from the overt site. By keeping the connection between the client and the overt site active, both Slitheen and Rebound offer a defense against this type of detection attack. Packets sent down a clean path will be received by the overt destination in the usual manner, prompting the server to send the requested resource (in

Table 1: Slitheen is the first decoy routing system to defend against latency analysis and website fingerprinting attacks. Our method provides strong defenses against both active and passive attacks, at the cost of requiring symmetric flows and in-line blocking. Although the latter requirements have been previously thought to be barriers to deployment, we argue that increasingly popular traffic shaping tools provide ISPs with an easy way to block and redirect traffic, opening avenues for easier deployments. The half-circle indicates that while Rebound stores a queue of downstream data at the relay to decrease latency, it also increases the latency of their system by sending unusually high amounts of upstream data.

	Telex [26]	Cirripede [12]	Decoy Routing [14]	TapDance [25]	Rebound [7]	Slitheen
No inline blocking	○	○	○	●	○	○
Handles asymmetry	○	●	●	●	●	○
Resistant to replay attacks	●	○	○	○	●	●
Resistant to latency analysis	○	○	○	○	◑	●
Website fingerprinting defense	○	○	○	○	○	●
Protocol fingerprinting defense	○	○	○	○	●	●

Slitheen) or an HTTP error message with the invalid request (in Rebound). The TCP sequence and acknowledgement numbers will match those that the censor expects. The censor, unable to decrypt these packets, will see no difference in the traffic. However, if the client is compressing HTTP GET requests to provide more upstream bandwidth, the server will respond to such a request sent down a clean path with an HTTP error messages, raising the suspicion of the censor. To defend against this attack, the client can replace or compress only non-essential headers, taking a loss to upstream bandwidth.

Forced Clean Paths. An adversary with the ability to chose between clean and tainted paths may route around a Slitheen station altogether. This would prevent the client from ever coming into contact with a participating ISP. Although the consequences of this attack would result in a complete loss of availability to the decoy system, Houmansadr et al. [13] show that this attack is too expensive for realistic censors, and very unlikely. We also note that this attack is only a denial of service, and will not leak information about whether the client is using or has used Slitheen.

4.5 Comparison to Existing Systems

A large advantage of Slitheen over existing systems is its resistance to latency analysis and website fingerprinting attacks. We not only eliminate the ability to use latencies to fingerprint the censored webpage accessed through a decoy routing session, but also minimize the latencies caused by the station itself. We give an overview of the comparison between Slitheen and previous systems in Table 1.

While Rebound also takes steps to minimize latency by storing a queue of downstream data from the covert destination, the use of their system is trivial to detect by a minimally capable passive adversary. Rebound traffic differs radically from typical web-browsing traffic in both the amount of upstream data sent by the client to provide space for inserted downstream data, and also the amount of HTTP error messages. Slitheen relays information to and from the covert destination in a way that does not deviate at all from a typical access to the specific overt site in use, providing a much more secure defense against passive attacks.

Our system defends against active attacks as well as, or better, than all existing decoy routing systems. Rebound is the only other system that *actively* maintains the connection between the client and the overt site, defending against the routing-capable attacks meant to unveil the true TCP state between the client and the overt site. Rebound, of course, requires the client to send upstream data in an equal amount to the downstream data she wishes to receive in order to maintain a consistent TCP state. In addition to being unusual web-browsing behaviour, this approach is also extremely inefficient. ISPs often offer clients much lower bandwidth for upstream than downstream data.

Although Slitheen does not allow asymmetric flows, and continues to require in-line blocking of downstream data, we argue that these requirements are growing less prohibitive towards the deployment of decoy routing stations as the popularity of specialized traffic shaping tools increases. Companies such as Sandvine[2] have developed highly efficient DPI boxes that would allow a participating ISP to detect tagged flows and easily redirect them to a Slitheen relay. They also provide service providers with the ability to force flow symmetry or share a flow's state between multiple DPI boxes [17], as long as the traffic in both directions crosses their area of influence. We note that this is very likely if the stations are deployed in close proximity to the overt sites.

5. IMPLEMENTATION

We developed a proof-of-concept implementation of our system and tested the relay station and client on desktop machines running Ubuntu 14.04. This implementation serves to demonstrate that our design behaves as expected, and provides a basis for our evaluations in the following section. Our code is available online for reuse and analysis.[3]

5.1 Client

We implemented the client as two distinct parts: the overt user simulator (OUS) that repeatedly connects to overt sites, and a SOCKS proxy frontend that relays SOCKS connection requests and data between the client's browser and the OUS. The OUS takes data from the SOCKS frontend and inserts it into X-Slitheen headers of outgoing HTTP requests. It then takes downstream data from the received resources of content type "slitheen" and returns this to the SOCKS frontend to be delivered to the browser. To allow the browser to send multiple simultaneous requests, we assign a stream ID to each connection. When the relay station receives downstream data for a particular stream, it includes the stream ID along with the data in the replaced resource, allowing the SOCKS frontend at the client side to demultiplex streams from the data received from the OUS.

For the tagging procedure, we modified OpenSSL[4] to allow the client to specify the value of the random nonce in the ClientHello

[2]https://www.sandvine.com/

[3]https://crysp.uwaterloo.ca/software/slitheen/

[4]http://openssl.org/

Table 2: Comparison of the total amount of content (in bytes, with means and standard deviations over 100 samples indicated) in a page, the amount of potentially replaceable leaf content, and the amount of leaf content actually replaced by the Slitheen station. Sites such as Facebook offer a significant amount of leaf content, but large TLS records prevent Slitheen from replacing any leaf content with downstream data from the censored site. Clients should select overt destinations that not only have a large amount of leaf content, but also perform well in practice. In the last two columns, we give the percentage of leaf content that was actually replaced by the relay station and the percentage of total content that was replaced by the relay station.

Site name	Total content (bytes)	Leaf content (bytes)	Replaceable content (bytes)	% leaf content replaced	% total replaced
Facebook	600000 ± 200000	40000 ± 10000	0 ± 0	0 ± 0	0 ± 0
Gmail	40000 ± 10000	8800 ± 100	7700 ± 100	87.7 ± 0.2	23 ± 9
Wikipedia	74000 ± 8000	24000 ± 2000	24000 ± 2000	100 ± 0	33 ± 4
Yahoo	1300000 ± 700000	400000 ± 100000	400000 ± 100000	100.0 ± 0.2	40 ± 20

message, as well as supply a given value for the client DH parameter. Although there are many algorithms available to negotiate a TLS master secret, our proof-of-concept implementation only allows the use of the DH key exchange methods (including the popular ECDHE). Other methods could easily be added to our system to expand the range of overt sites used by the client. Our modifications to OpenSSL take the form of user-defined callbacks, minimizing both the lines of code we had to alter in the source, as well as reducing any unintended consequences from our modifications. To send an untagged flow, the client can simply refrain from setting the provided callbacks, resulting in OpenSSL's default behaviour. Our modifications consisted of additions only, introducing 46 lines of code to OpenSSL. The callback functions and helper code for generating the Telex-style tags are about 2200 lines of C code.

We used the PhantomJS headless browser[5] as the basis for our OUS, although more common browsers such as Firefox[6] or Chrome[7] could be adapted for use as an OUS as well. To tag the TLS sessions established by the OUS, we made a few modifications to the PhantomJS source code to add options that set the OpenSSL ClientHello and ClientKeyExchange callbacks to the previously described functions. Our modifications consisted only of additions and introduced 43 lines of C++ code to PhantomJS and 56 lines of code to its version of Qtbase. We wrote a PhantomJS script consisting of 55 lines of Javascript to read data from the SOCKS frontend and add it to outgoing request headers. When the script receives a resource, it sends the contents of all resources of content type "slitheen" to the SOCKS frontend through a WebSocket [8].

The SOCKS frontend receives connection requests and data from the browser and writes it to a named pipe for the OUS to process. It assigns each new connection a stream ID and sends that along with the data to the OUS to send to the relay station. The SOCKS frontend reads downstream data from the OUS and first demultiplexes it by stream ID before sending it to the browser. We wrote the SOCKS frontend in approximately 500 lines of C code.

5.2 Slitheen Relay Station

We implemented the Slitheen station in approximately 3200 lines of C code. The station is responsible for recognizing and processing tagged TLS handshakes, proxying data to censored sites, and monitoring and replacing upstream and downstream application data to overt sites. When the station detects a tagged flow, it saves the source and destination addresses and ports in a flow table, to later identify packets in the same decoy routing session. The station continues to passively observe the remainder of the TLS handshake and then uses the tag-derived client secret and observed server handshake messages to compute the TLS master secret for the session, saving it in the flow table.

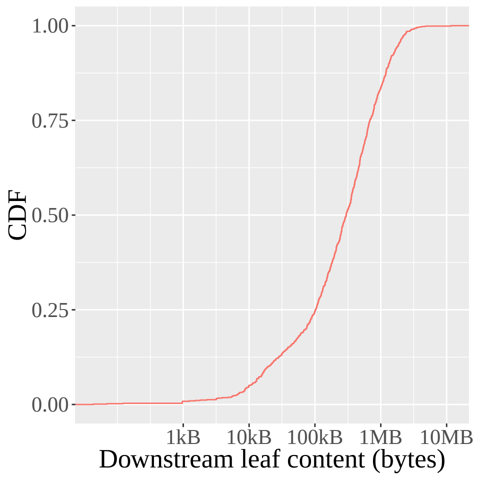


Figure 4: Cumulative distribution function of downstream bandwidth for proxied data provided by the Alexa top 1,000 sites

After verifying the TLS Finished message from both sides of the connection, the Slitheen station begins to monitor HTTP GET requests from the client for upstream data, and stores any downstream response from the blocked server in a censored content queue. Once the station receives the client's Slitheen ID, it saves this information in the flow table in order to later identify the stream IDs that can replace downstream resources. As Slitheen reuses the TCP/IP headers from the overt site, we have no need to modify kernel code to set up a forged TCP state (as Telex requires). Application data is simply swapped into TCP segments as they are read from the interface, and sent back out to their destination with a recomputed TCP checksum.

Once the client has terminated the connection with the overt site by sending a TCP FIN packet, the station removes the flow from the table. We save session tickets and session IDs to allow clients to resume TLS sessions for subsequent requests to the same server.

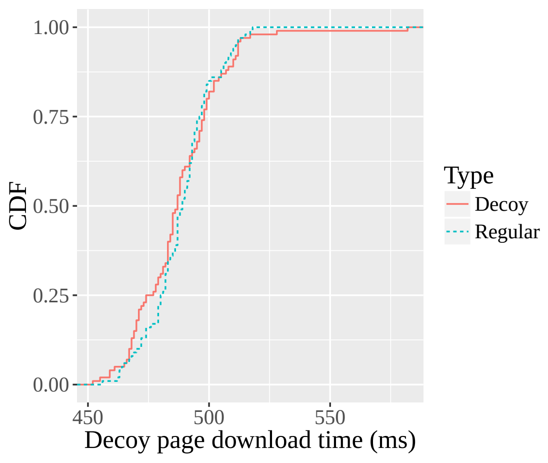
6. EVALUATION

6.1 Bandwidth

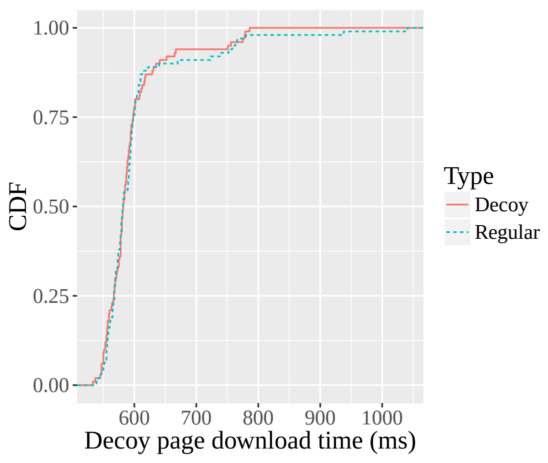
The amount of downstream data a client can receive in a single page load is dependent on the amount and size of the leaf resources of the overt page. We visited the Alexa[8] top 1000 sites and mea-

[5]http://phantomjs.org/
[6]https://www.mozilla.org/firefox/
[7]https://www.google.com/chrome/

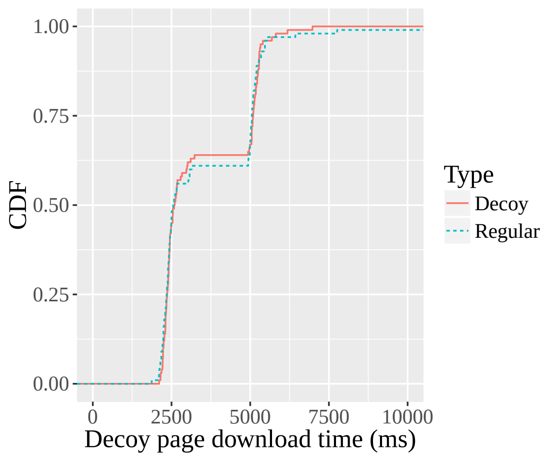
[8]www.alexa.com/



(a) Latency measurements for www.wikipedia.org



(b) Latency measurements for gmail.com



(c) Latency measurements for www.yahoo.com

Figure 5: Cumulative distribution functions of the page load time of three overt destinations as both a decoy access and a regular access. The CDF shows a minimal difference in the latency distributions of the two types of access, and a K-S test fails to find a difference in the latency distributions due to the Slitheen station replacement and processing.

sured the amount of downstream data available to a Slitheen client when using the site as an overt destination. To collect these results, we used PhantomJS to capture the size of HTTP responses from each site. We then measured the total amount of possible downstream bandwidth from each site as the sum of the sizes of all replaceable resources for the page in bytes. We note that this is the maximum amount of potential downstream bandwidth as some resources may not be accessible on a path that contains the decoy router, and not all leaf HTTP responses will be replaceable if the headers are contained in a TLS record that is split across multiple packets.

We give a cumulative distribution function of the amount of leaf content in bytes for the top 1,000 sites in Figure 4. About 25% of sites offer 1 MB or more of potentially replaceable content, giving us a fair amount of downstream bandwidth for relaying content from censored sites to the client. In total, we found that about 40% of the content from all 1,000 sites was leaf content. This suggests that censored sites will load in a little more than twice the amount of time they normally would.

To test the actual replaceability of data, we accessed four different overt sites: www.facebook.com, gmail.com, www.wikipedia.org, and www.yahoo.com as overt destinations and counted the total bytes received, the total amount of leaf content, and the amount of leaf content that was replaced by the system. We give the results in Table 2. We found that the relay station's ability to replace leaf resources varied significantly among sites. Facebook was the most extreme example, where none of the leaf content was replaceable by the system. Upon further investigation, we noticed that all leaf content response headers from this site were encrypted in a large record along with the response body and spread across multiple packets, making its content type immutable by the Slitheen station. For Wikipedia and Yahoo, on the other hand, almost all of the leaf content in our 100 trials was replaced by the Slitheen station. While the station missed some of Gmail's leaf content, most leaf resources were replaced by the station without a loss of HTTP state. These findings suggest that the selection of an overt destination should depend on both the amount of leaf content on the site as well as the amount of replaceable content determined through use.

6.2 Latency Measurements

As mentioned in the previous section, Slitheen reduces latency from two sources. By queueing downstream data from the covert destination, Slitheen removes the latency that comes from fetching content from a possible distant censored site. We also minimize the latency introduced by the relay station itself by not waiting for data to be proxied to the covert destination, but rather queueing up previously collected proxy data for the client to be replaced as soon as incoming packets from the overt site arrive. Our results show that the encryption and replacement procedures do not add enough latency to identify the usage of Slitheen.

We measured the time it took to fully load the overt destination both as an overt site for tagged flows whose leaf resources were replaced with proxied data, and from a regular, untagged access. We tested three different overt destinations: www.wikipedia.org, gmail.com, and www.yahoo.com 100 times each and give the CDF of these load times in Figure 5. We performed a two-sided Kolmogorov-Smirnov test on the collected data to determine whether the relay station induced a different latency distribution for decoy accesses and measured a D-value of 0.11 and a p-value of 0.58 for www.wikipedia.org, a D-value of 0.12 and a p-value of 0.47 for gmail.com, and a D-value of 0.07 and a p-value of 0.97 for www.yahoo.com. These results indicate that the K-S test fails to find any significant difference in the latency distributions of the overt destination between its use as a regular or an overt site in a decoy routing session.

7. CONCLUSION

Slitheen is the first decoy routing system to provide a defense against latency analysis and website fingerprinting attacks. We mimic the packet sequence of a regular visit to the overt site by replacing the site's actual packets in response to valid resource requests with downstream data from the covert destination, forcing the covert datastream into the shape of the overt datastream. Slitheen eliminates latency from the censored site by building up a queue of downstream data and replacing the contents of TCP segments as soon as they arrive at the station. We also eliminate the censor's ability to use latencies or packet sequences to fingerprint the covert site accessed through the decoy routing system.

While our system does not support asymmetric flows and requires the in-line blocking of downstream data, we argue that taking the side of security in this trade-off is reasonable given the increased capabilities of technology available to participant ISPs. In return, Slitheen provides stronger defenses to known attacks than any previous decoy routing system.

8. REFERENCES

[1] S. Aryan, H. Aryan, and J. A. Halderman. Internet censorship in Iran: A first look. In *3rd USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2013.

[2] M. Dewhirst. Censorship in Russia, 1991 and 2001. *The Journal of Communist Studies and Transition Politics*, 18(1):21–34, 2002.

[3] R. Dingledine. Obfsproxy: The next step in the censorship arms race. https://blog.torproject.org/blog/obfsproxy-next-step-censorship-arms-race, February 2012. [Online; accessed 29-February-2016].

[4] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *13th USENIX Security Symposium*, pages 303–320, 2004.

[5] K. P. Dyer, S. E. Coull, and T. Shrimpton. Marionette: A programmable network-traffic obfuscation system. In *24th USENIX Security Symposium*, pages 367–382, 2015.

[6] E. L. Eisenstein. *The printing press as an agent of change*, volume 1. Cambridge University Press, 1980.

[7] D. Ellard, C. Jones, V. Manfredi, W. Strayer, B. Thapa, M. Van Welie, and A. Jackson. Rebound: Decoy routing on asymmetric routes via error messages. In *Local Computer Networks (LCN), 2015 IEEE 40th Conference on*, pages 91–99, Oct 2015.

[8] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, December 2011.

[9] R. Fielding, J. Gettys, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[10] D. Fifield, C. Lan, R. Hynes, P. Wegmann, and V. Paxson. Blocking-resistant communication through domain fronting. *Proceedings on Privacy Enhancing Technologies*, 2015(2):46–64, 2015.

[11] D. Herrmann, R. Wendolsky, and H. Federrath. Website fingerprinting: Attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 31–42, 2009.

[12] A. Houmansadr, G. T. Nguyen, M. Caesar, and N. Borisov. Cirripede: Circumvention infrastructure using router redirection with plausible deniability. In *18th ACM Conference on Computer and Communications Security*, CCS '11, pages 187–200, 2011.

[13] A. Houmansadr, E. L. Wong, and V. Shmatikov. No direction home: The true cost of routing around decoys. In *2014 Network and Distributed System Security (NDSS) Symposium*, 2014.

[14] 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 *USENIX workshop on free and open communications on the Internet*, 2011.

[15] H. Mohajeri Moghaddam, B. Li, M. Derakhshani, and I. Goldberg. Skypemorph: Protocol obfuscation for Tor bridges. In *2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 97–108, 2012.

[16] Z. Nabi. The anatomy of web censorship in Pakistan. In *3rd USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2013.

[17] Sandvine. White paper: Applying network policy control to asymmetric traffic: Considerations and solutions. https://www.sandvine.com/downloads/general/whitepapers/applying-network-policy-control-to-asymmetric-traffic.pdf.

[18] M. Schuchard, J. Geddes, C. Thompson, and N. Hopper. Routing around decoys. In *2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 85–96, 2012.

[19] The Tor Project. Tor: Bridges. https://www.torproject.org/docs/bridges. [Online; accessed 27-February-2016].

[20] The Tor Project. Torproject.org blocked by GFW in China: Sooner or later? https://blog.torproject.org/blog/tor-partially-blocked-china, September 2009. [Online; accessed 27-February-2016].

[21] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective attacks and provable defenses for website fingerprinting. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 143–157, 2014.

[22] Z. Weinberg, J. Wang, V. Yegneswaran, L. Briesemeister, S. Cheung, F. Wang, and D. Boneh. StegoTorus: A camouflage proxy for the Tor anonymity system. In *2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 109–120, 2012.

[23] P. Winter and S. Lindskog. How the great firewall of china is blocking Tor. In *2nd USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2012.

[24] P. Winter, T. Pulls, and J. Fuss. ScrambleSuit: A polymorphic network protocol to circumvent censorship. In *12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, pages 213–224, 2013.

[25] E. Wustrow, C. M. Swanson, and J. A. Halderman. Tapdance: End-to-middle anticensorship without flow blocking. In *23rd USENIX Security Symposium*, pages 159–174, 2014.

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