TapDance: End-to-Middle Anticensorship without Flow Blocking

Abstract

In response to increasingly sophisticated state-sponsored Internet censorship, recent research has proposed a new approach to censorship resistance: end-to-middle proxying. This concept, developed in systems such as Telex, Decoy Routing, and Cirripede, moves anticensorship technology into the core of the network, at large ISPs outside the censoring country. In this paper, we focus on two technical barriers to the deployment of end-to-middle proxy designs — the need to selectively block flows, and the need to observe both directions of a connection — and we propose a new construction, TapDance, that avoids these shortcomings. To accomplish this, we employ a novel TCP-level technique that allows the anticensorship station at an ISP to function as a passive network tap, without an inline blocking component. We also apply a novel steganographic encoding to embed control messages in TLS ciphertext, allowing us to operate on HTTPS connections even with asymmetric flows. We implement and evaluate a proof-of-concept prototype of TapDance with the goal of functioning with minimal impact on normal ISP operations.

1 Introduction

Internet censorship by repressive governments is reaching new levels of technical sophistication. In order to circumvent these measures, many users in censored countries employ proxy-based systems, such as VPNs, open HTTP proxies, and a variety purpose-built anticensorship proxies [1, 22]. However, state-level censors are increasingly able to block many of these systems, by discovering and banning the IP addresses of the servers on which they rely. Some proxies attempt to remain unblocked by frequently changing their IP addresses, but they face a tension between the desire to make their network locations known to would-be users and the need to keep the same information secret from the censor.

To avoid this tension and escape the cat-and-mouse game that results between censors and anticensorship tools, researchers have recently introduced a new anticensorship approach called end-to-middle (E2M) proxying [18, 24, 43]. E2M systems place proxies at Internet Service Providers (ISPs) outside censoring countries. As users in the censored countries make connections that pass through these ISPs on their way to unblocked, decoy destinations, a device at the ISP (the "station") is able to detect covertly tagged flows from anticensorship users and divert them to otherwise censored destinations. This technique is more difficult for a censor to block, because E2M proxy connections look similar to normal requests for uncensored content. In order to block them, the censor would have to block *all* connections that pass through ISPs running stations. If stations were widely deployed at major ISPs, this would block a prohibitive amount of innocent traffic as well, potentially interfering with desirable commerce or communication outside the country.

While E2M approaches appear promising compared to traditional proxies, they face technical hurdles that have thus far prevented any of them from being deployed at an ISP. All three existing schemes assume that the station has the ability both to monitor traffic traveling through an ISP and to selectively block flows between users and decoy sites. Unfortunately, this requires introducing new hardware in-line with ISP links, which adds latency and introduces a possible point of failure. ISPs typically have service level agreements (SLAs) with their customers and peers that govern performance and reliability, and adding in-line blocking components may violate their contractual obligations. Additionally, adding in-line components increases the number of possible points of failure to check when a failure does occur, even in unrelated parts of the ISP. For these reasons, ISPs are extremely reluctant to add in-line elements to their networks.

Furthermore, some existing schemes assume that the station sees traffic in both directions, client-decoy and decoy-client. While this might be true when the station is immediately upstream from the decoy server, it does not generally hold farther away. IP flows are often asymmetric through ISPs, such that the route taken by packets from source to destination may be different from the reverse path. This asymmetry limits an ISP to observing only one side of a connection. The amount of asymmetry experienced is ISP-dependent, but in general, tier-2 ISPs see lower amounts of asymmetry (around 25% of packets) than tier-1 ISPs, where up to 90% of packets can be part of asymmetric flows [23]. This severely constrains where in the network stations can be deployed.

In this paper, we propose TapDance, a novel end-to-middle proxy approach that overcomes these deployment barriers, at the cost of a moderate increase in its susceptibility to active attacks by the censor. TapDance is the first E2M proxy that works without an inline-blocking or redirecting element at an ISP. Instead, our design requires only a passive tap that observes traffic transiting the ISP,

and the ability to inject new packets into the network. TapDance also includes a novel connection tagging mechanism that embeds steganographic tags into the ciphertext of a TLS connection. We make use of this to allow the system to support asymmetric flows and to efficiently include large steganographic payloads in a single packet.

Although TapDance appears to be more feasible to deploy than previous E2M designs, it is not without tradeoffs. There are several active attacks, discussed in Section 5, that a censor could perform on live flows in order to distinguish between TapDance connections and normal traffic. (However, we note that each of the previous designs is also vulnerable to some of these attacks, including replay and preplay attacks.) As a countermeasure, we introduce *active defense* mechanisms that could be deployed in response to certain active attacks.

Despite these tradeoffs, TapDance represents a path towards deployment for E2M proxy schemes. Given the choice between the previous systems, which appear not to be practically fieldable, and our system, which better satisfies the constraints of real ISPs but also requires a careful defense strategy, we believe TapDance is the more viable approach to building anticensorship into the Internet's core.

Organization The remainder of this paper is organized as follows. In Section 2, we review the three existing E2M proposals and briefly survey other related work. Section 3 introduces our ciphertext-based covert channel mechanism, and Section 4 explains the rest of the TapDance architecture. In Section 5, we perform a security analysis of our scheme, compare its attack surface to previous E2M approaches, and discuss active defense strategies. We describe our proof-of-concept implementation in Section 6 and evaluate its performance in Section 7. We discuss future work in Section 8, and conclude in Section 9.

2 Background and Related Work

There are three original publications on end-to-middle proxying: Telex [43], Decoy Routing [24] and Cirripede [18]. The designs for these three systems are largely similar, although some notable differences exist. Figure 1 shows an overview of the Telex design for reference.

In each design, a client wishes to reach a censored website. To do so, the client creates a connection to an unblocked *decoy* server, with the connection to this website passing through a cooperating ISP outside the censored country deploying an *ISP station*. The decoy can be any server and is oblivious to the operation of the anticensorship system. The ISP station determines that a particular client wishes to be proxied by recognizing a *tag*. In Telex, this is a public-key steganographic tag placed in the random nonce in the ClientHello message of a Transport Layer Security (TLS) connection [10]. In Cirripede, users register their IP address with a registration server by making a series of TCP connections, encoding a similar tag in the sequence numbers. In Decoy Routing, the tag is placed in the TLS client nonce as in Telex, but the client and the ISP station are assumed to have a shared secret established out of band.

In Telex and Cirripede, the tag encoded consists of an elliptic curve Diffie-Hellman (ECDH) public key point and a hash of the ECDH shared secret. In other words, for an elliptic curve generator point G, Telex publishes the ECDH public key point $P = dG$, and keeps the private key d secret. The client then generates an ECDH public key point $Q = eG$ and the shared secret $S = eP = dQ$. The tag is then simply the concatenation of the client's public point and a hash of the shared secret: $Q||H(S)$. The station then receives potential tags, extracts point Q, generates the shared secret $S = dQ$, and checks whether the rest of the tag is $H(S)$. In Decoy Routing, it is assumed that the client and station have a previously shared secret key, and the tag consists of an HMAC of the shared key, the current hour, and a per-hour sequence number.

Once the station has determined that a particular flow should be proxied, all three designs employ an inline blocking component at the ISP to block further communication between the client and the decoy destination. At this point, the station impersonates the server, receiving packets to and spoofing from its IP address. In Telex, the station uses the tag in the TLS client nonce to compute a shared secret with the client, which the client uses to seed its secret keys during the key exchange with the server. Using this seed and the ability to observe both sides of the TLS handshake, Telex derives the master secret under which the TLS client-server communication is encrypted under. In Cirripede, each new HTTPS connection to the overt destination is transparently proxied by the service proxy located at the friendly ISP. In Decoy Routing, the client and station are assumed to have a shared secret. After the client's first encrypted data message (which the decoy router cannot decrypt), the decoy router encrypts a Hello message to the client-decoy router shared key and sends it back.

In all cases, the station blocks the server from receiving additional packets from the client, and continues to use the connection to communicate with the client. In Telex, the station is able to derive the existing client-server shared master secret, and continues to use this without an additional change in keys. In both Cirripede and Decoy Routing, the station changes the key stream to be encrypted under the derived or previously shared secret.

Changing the communication to a new shared secret opens Cirripede and Decoy Routing to replay and preplay active attacks by the adversary. If an adversary suspects a user is accessing these proxies, it can create a new connection that replays parts from the suspected connection and receive confirmation that a particular flow uses the

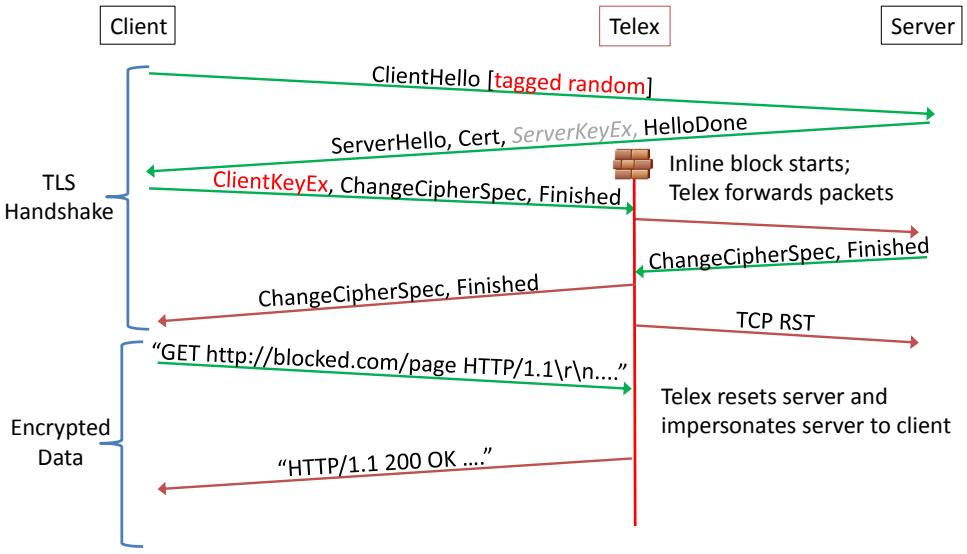

Figure 1: Telex end-to-middle scheme — To establish communication with an ISP-deployed Telex station, the client performs a TLS connection with an unblocked decoy server. In the client's hello, it replaces the TLS random nonce with a public-key steganographic tag that can be observed by the Telex station at the on-path ISP outside the censored country. When the station detects this tag with its private key, it blocks the true connection using an inline-blocking component, and forwards packets for the remainder of the handshake. Once the handshake is complete, Telex stops forwarding, and begins to spoof packets from the server in order to communicate with the client. While here we only show the details of Telex, all of the first generation ISP proxies (Telex, Cirripede, and Decoy Routing) are similar in architecture; we note differences in Section 2.

proxy. For example, in Decoy Routing, the adversary can simply use the suspected connection's TLS client nonce in a new connection and send a request. If the first response is not able to be decrypted with the client-server shared secret, it confirms that the particular nonce was a tagged sentinel. For Cirripede, a similar replay of the tagged TCP SYN packets will register the adversary's client, and a connection to the overt destination over TLS will confirm this: if the adversary can decrypt the TLS response with the established master secret, the adversary is not registered with Cirripede, indicating that the TCP SYN packets were not a secret Cirripede tag. Otherwise, if the adversary cannot decrypt the response, this indicates that the SYN packets were indeed a Cirripede tag.

Telex is not vulnerable to either of these attacks, because the client uses the client-station shared secret to seed its half of the key exchange. This allows the station to also compute the client-server shared master secret, and verify that the client has knowledge of the client-station shared secret by verifying the TLS finished messages. If an adversary attempted to replay the client random in a new connection, Telex would be able to determine that the user (in this case, the adversary) did not have knowledge of the client-station shared secret, because the user did not originally generate the Diffie-Hellman tag. Thus, Telex is unable to decrypt and verify the TLS finished messages as expected, and will not spoof messages from the server.

Other anticensorship schemes Besides end-to-middle proxies, previous anticensorship approaches, including Collage [8] and Message in a Bottle [21], have leveraged using user-generated content on websites to bootstrap communication between censored users and a centrally-operated proxy. However, these designs are not intended to work with low-latency applications such as web browsing. SkypeMorph [28], FreeWave [19], CensorSpoofer [39] and StegoTorus [40] are proxies or proxy-transports that attempt to mimic other protocols, such as Skype, VoIP, or HTTP in order to avoid censorship by traffic fingerprinting. However, recent work appears to suggest that such mimicry may be detectable under certain circumstances by an adversary [15, 17]. Finally, browser-based proxies work by running a small flash proxy inside non-censored users browsers (for example, when they visit a website), and serve as short-lived proxies for censored users [14]. These rapidly changing proxies can be difficult for a censor to block in practice, though it is essentially a more fast-paced version of the traditional censor cat-and-mouse game.

3 Ciphertext Covert Channel

In our setting, previous end-to-middle covert channels have been limited in size, forcing implementations to use small payloads, or several flows in order to steganographically communicate information to the ISP station.

However, because TapDance does not depend on inline-flow blocking and must work with asymmetric flows, we need a way to communicate the client's request directly to the TapDance station, while maintaining a valid TLS session between client and server. We therefore introduce a novel technique that allows us to encode a much higher bandwidth steganographic payload in the ciphertexts of legitimate (i.e., censor-allowed) TLS traffic.

The classic problem in steganography is known as the *prisoners' problem*, formulated by Simmons [38]: two prisoners, Alice and Bob, wish to send hidden messages in the presence of a jailer. These messages are disguised in legitimate, public communication between Alice and Bob in such a way that the jailer cannot detect their presence. Many traditional steganographic techniques focus on embedding hidden messages in non-uniform cover channels such as images or text [3]; in the network setting, each layer of the OSI model may provide potential cover traffic [16] of varying bandwidths. To avoid detection, these channels must not alter the expected distribution of cover traffic [30]. In addition, use of header fields in network protocols for steganographic cover limits the carrying capacity of the covert channel.

We observe it is possible for the sender to use stream ciphers and CBC-mode ciphers as steganographic channels. This allows a sender Alice to embed an arbitrary hidden message to a *third party*, Bob, inside a *valid* ciphertext for Cathy. That is, Bob will be able to extract the hidden message and Cathy will be able to decrypt the ciphertext, without alerting outside entities (or, indeed, Cathy, subject to certain assumptions) to the presence of the steganographic messages.

Moreover, through this technique, we can place limited constraints on the plaintext (such as requiring it be valid base64 or numeric characters), while encoding arbitrary data in the corresponding ciphertext. This allows us to ensure that Cathy can not only decrypt the received ciphertext, but also that the plaintext is consistent with the protocol used. Note that this is a departure from the original prisoners' problem, as we assume Alice is allowed to securely communicate with Cathy, so long as this communication looks legitimate to outside entities.

As our technique works both with stream ciphers and CBC-mode ciphers, which are the two most common modes used in TLS [26], we will use this building block to encode steganographic tags and payloads in the ciphertext of TLS requests. From the server's perspective, the ciphertext received is valid and decrypts to a protocol-conformant plaintext.

3.1 Encoding technique description

To describe our technique, we start with a stream cipher in counter mode. The key observation is that counter mode ciphers, even with authentication tags, have ciphertexts

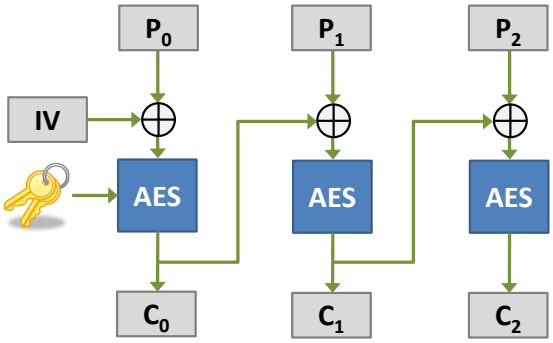

Figure 2: CBC review — In CBC-mode encryption, each ciphertext block C_n is the encryption of an intermediary $C_{n-1} \oplus P_n$ (or $IV \oplus P_0$ for the first block). Given a fixed intermediary, we can choose a mixture of bits in the plaintext block P_n and previous ciphertext block C_{n-1} such that the resulting xor is the fixed intermediary value. By choosing some of the bits in the ciphertext blocks, we can encode arbitrary data in CBC-mode ciphertext while still allowing constraints on the plaintext.

that are *malleable* from the perspective of the sender, Alice. That is, stream ciphers have the general property of *ciphertext malleability*, in that flipping a single bit in the ciphertext flips a single corresponding bit in the decrypted plaintext. Alice can likewise change bits in the plaintext to effect specific bits in the corresponding ciphertext. Since Alice knows the keystream for the stream cipher, she can choose an arbitrary string that she would like to appear in the ciphertext, and compute (decrypt) the corresponding plaintext. Note that this does not invalidate the MAC or authentication tag used in addition to this cipher, because Alice first computes a valid plaintext, and then encrypts and MACs it using the standard library, resulting in ciphertext that contains her chosen steganographic data.

Furthermore, Alice can "fix" particular bits in the plaintext and allow the remaining bits to be determined by the data encoded in the ciphertext. For example, Alice could require that each plaintext byte starts with 5 bits set to 00110, and allow the remaining 3 bits to be chosen by the ciphertext. In this way, the plaintext will always be an ASCII character from the set "01234567" and the ciphertext has a steganographic "carrying capacity" to encode 3 bits per byte.

While it seems intuitive that Alice can limit plaintext bits for stream ciphers, it may not be as intuitive to see how this is also possible for CBC-mode ciphers. However, while the ciphertext malleability of stream ciphers allows Alice partial control over the resulting plaintext, we show that it is also possible to use this technique in other cipher modes, with equal control over the plaintext values. For reference, Figure 2 shows an overview of CBC-mode encryption.

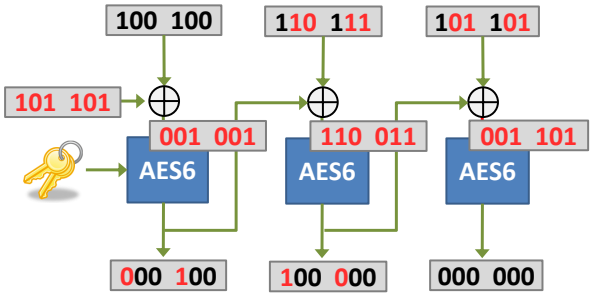

Figure 3: CBC chosen ciphertext example — In this example, bits chosen by the encoding are in black, while bits "forced" by computation are red. For example, we choose all 6-bits to be 0 in the last ciphertext block. This forces the block's intermediary to be "forced" to a value beyond our control; in this case 001101. To obtain this value, we can choose a mixture of bits in the plaintext, which forces the corresponding bits in the previous ciphertext block. In this example, we choose the plaintext block to be of the form $1xx1xx$, allowing us to choose 4-bits in the ciphertext, which we choose to be 0s. Thus, the ciphertext has the form $x00x00$. We solve for the unknown bits in the ciphertext and plaintext ($1xx1xx \oplus x00x00 = 001101$) to fill in the missing "fixed" values. We can repeat this process backward until the first block, where we simply compute the IV in order to allow choosing all the bits in the first plaintext block.

In this mode, it is possible to choose the value of an arbitrary ciphertext block (e.g., C_2), and decrypt it to compute an intermediary result. This intermediary result must also be the result of the current plaintext block (P_2) xored with the previous ciphertext block (C_1) in order to encrypt to the chosen ciphertext value. This means that, given a ciphertext block, we can choose either the plaintext value (P_2), or the previous ciphertext block (C_1), and compute the other. However, we can also choose a mixture of the two; that is, for each bit we pick in the plaintext, we are "forced" to choose that corresponding bit in the previous plaintext block and vise-versa. Choosing any bits in a ciphertext block (C_1) will force us to repeat this operation for the previous plaintext block (P_1) and twice previous ciphertext block (C_0). We can choose to pick the value of plaintext blocks (fixing the corresponding ciphertext blocks), all the way back to the first plaintext block, where we are left to decide if we want to choose the value of the first plaintext block or the Initialization Vector (IV) value. At this point, fixing the IV is the natural choice, as this leaves us greater control over the first plaintext block. Figure 3 shows an example of this backpropagation, encoding a total of 4-bits per 6-bit ciphertext block (plus a full final block).

This scheme allows us to restrict plaintexts encrypted with CBC to the same ASCII range as before, while still allowing us to encode arbitrary-length messages in the ciphertext.

While the sender can encode any value in the ciphertext in this manner, we do not wish to change the expected ciphertext distribution. The counter and CBC modes of encryption both satisfy indistinguishability from random bits [35], so encoding anything that is distinguishable from a uniform random string would allow third parties (e.g., a network censor) to detect this covert channel. To prevent this, Alice encrypts her hidden message if necessary, using an encryption scheme that produces ciphertexts indistinguishable from random bits. The resulting ciphertext for Bob is then encoded in the CBC or stream-cipher ciphertext as outlined above. To an outside adversary, this resulting "ciphertext-in-ciphertext" should still be a string indistinguishable from random, as expected.

3.2 Related techniques

Other techniques [2,4,29] leverage pseudorandom public-key encryption (i.e., encryption that produces ciphertext indistinguishable from random bits) in order to solve the classic prisoners' problem. These techniques allow protocol participants to produce messages that mimic the distribution of an "innocent-looking" communication channel. The problem setting differs from ours, however, and the encoding of hidden messages inside an allowed encrypted channel (as valid ciphertexts) is not considered.

Dyer et al. [13] introduce a related technique called *format transforming encryption (FTE)*, which disguises encrypted application-layer traffic to look like an innocent, allowed protocol from the perspective of deep packet inspection (DPI) technologies. The basic notion is to transform ciphertexts to match an expected format; as DPI technologies typically use membership in a regular language to classify application-layer traffic, FTE works by using a (bijective) encoding function that maps a ciphertext to a member of a pre-specified language. This steganographic technique differs significantly from ours, in that we do not attempt to disguise the use of a particular internet protocol itself (i.e., TLS), but rather ensure that our encoded ciphertext does not alter the expected distribution of the selected protocol traffic (i.e., TLS ciphertexts, in our system design).

4 TapDance Architecture

4.1 Inline blocking alternative

As outlined in Section 1, one of the largest challenges to deploying E2M proxies at ISPs is the inline flow-blocking component. TapDance gets rid of this requirement, and instead requires only a passive network tap and traffic injection capability. To accomplish this, we utilize several tricks gleaned from a close reading of the TCP specification [33] to allow the station to impersonate the server, without blocking traffic between client and server.

First, the client establishes a normal TLS connection to the decoy web server. Once this handshake is complete,

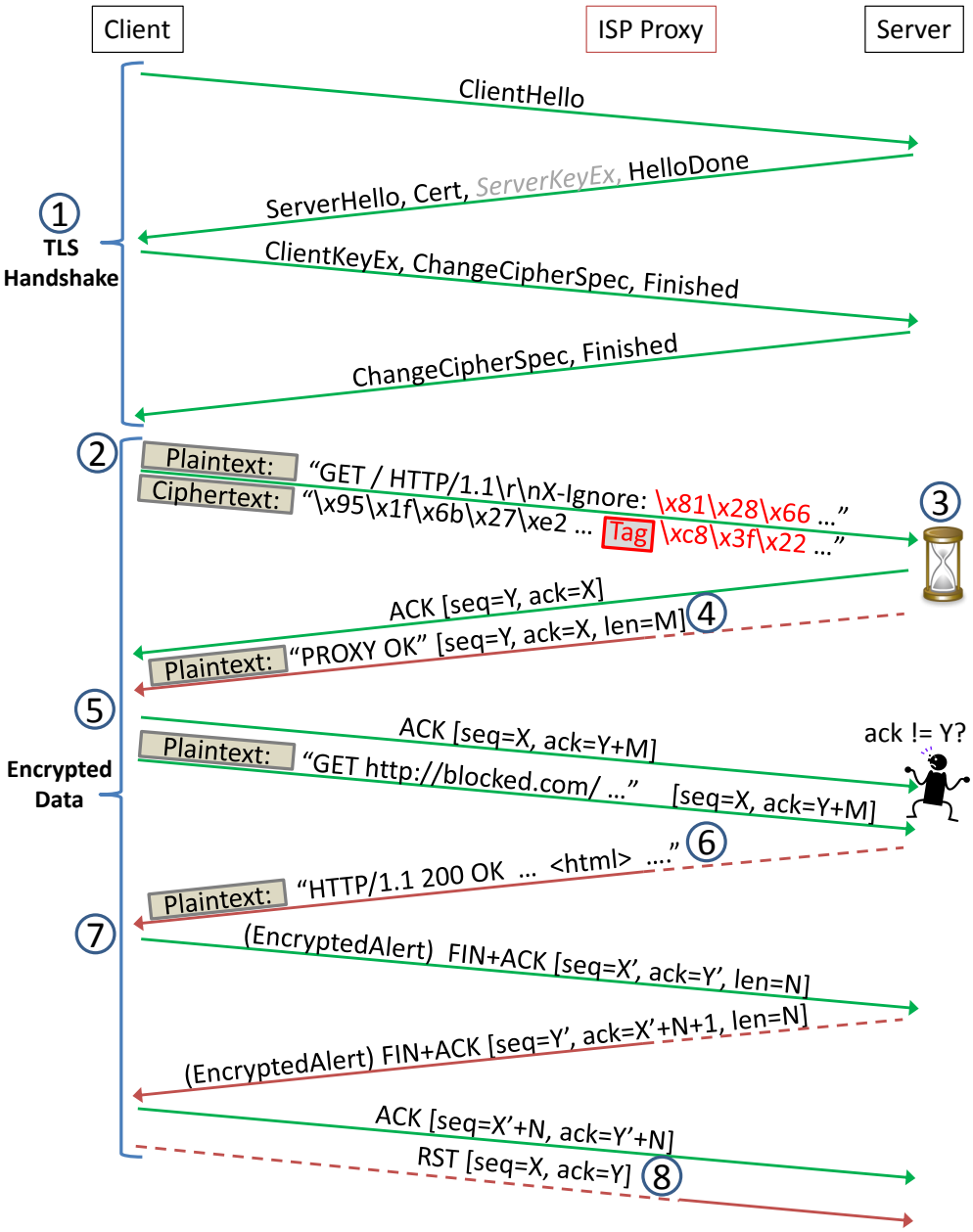

Figure 4: TapDance overview — Our approach requires no inline blocking element, making it feasible to deploy at an ISP. (1) The client performs a normal TLS handshake with an unblocked decoy server. (2) The client sends an *incomplete* HTTP request through the connection, and encodes a steganographic tag in the *ciphertext* of the request, using a novel encoding scheme (Section 4.2). (3) The server sends a TCP ACK message in response, and wait for the rest of the client's request until it times out. (4) The ISP-located station observes and extracts the client's tag, and spoofs a response to the client from the server, encrypted under the client-server session secret, and indicating the station's precense to the client. (5) The client sends a TCP ACK (for the spoofed data), and its real request. The server ignores both of these, because the TCP acknowledgement field is higher than the server's TCP SND.NXT. (6) The ISP station sends back a spoofed response for the real page. (7) When finished, the client and ISP station simulate a standard TCP/TLS authenticated shutdown, which is again ignored by the true server. (8) After the connection is terminated by the client, the ISP station sends a TCP RST packet that is valid for the server's SND.NXT, silently closing its end of the connection before its timeout expires.

the client and server share a master secret, which they use to generate encryption keys, MAC keys, and initialization vector or sequence state. Next, the client sends an *incomplete* HTTP request through the encrypted channel. This request is formatted in a way such that the web server will not respond with data to the client, for example by simply withholding the two consecutive line breaks that mark the end of an HTTP request. The server will acknowledge this data only at the TCP level by sending a TCP ACK packet. The request is also formatted in a way to contain a tag visible only to the station. The station extracts this tag (and the contained master secret) from the encrypted request (more details follow in Section 4.2), and injects a spoofed encrypted response from the server to the client.

At the TCP level, the client will similarly acknowledge this spoofed data with a TCP ACK packet, and because there is no inline-blocking between it and the server, the ACK will reach the server. However, because the acknowledgement number is above the server's $SND.NXT$, the server will not respond. Similarly, if the client responds with additional data, the acknowledgement field contained in those TCP packets will also be beyond what the server has sent. This allows the station to continue to impersonate the server, acknowledging data the client sends, and sending its own data in response, without interference from the real server.

The main advantage to this approach is that it removes our dependence on an inline blocking element at the ISP.

4.2 Tag format

Previous proposals of E2M proxies have used TCP sequence numbers and TLS ClientHello random nonces to hide their steganographic tag. These fields are useful because these strings should be uniformly random for legitimate connections, providing a good cover for the tag that replaces them, provided it is indistinguishable from random (except to the owner of the ISP station's private key).

However, both of these fields are fixed size; each TLS nonce can be replaced with a 224-bit uniform random tag, and each TCP sequence number with only 24 bits of a tag. Cirripede, which encodes its tag into TCP sequence numbers, uses multiple TCP connections to convey the full tag to the station. Telex and Decoy Routing both use a single TLS nonce to encode their tag. Given their limited size, they can only convey short secrets, while the rest of the payload (such as the request for a blocked website) must take place in a future packet.

As described in Section 3, we observe that it is also possible to encode steganographic tags in the *ciphertext* of a TLS connection, without invalidating the TLS session. Encoding the tag in the ciphertext has several advantages. First, the tag is no longer constrained to a fixed field size of either 24 or 224 bits, allowing us to encode more information in each tag, and use larger and more secure elliptic curves. Second, because the ciphertext is sent after the TLS handshake has completed, it is possible to encode the connection's master secret in this tag, allowing the station to decrypt the TLS session from a single packet, and without requiring the station to observe packets from the server.

Limiting plaintext range Because the server will ultimately decrypt and parse the tag-containing ciphertext, it is important to limit the character set of the plaintext to prevent errors. For example, plaintext that contains line breaks or other arbitrary binary data may cause the HTTP server to send an error in response, breaking the purpose of the incomplete header sent in Section 4.1. We utilize the plaintext-limiting as described in Section 3 to maintain some control over the plaintext that the server receives. Specifically, we split the tag into 6-bit chunks, and encode each chunk in the low order bits of a ciphertext byte. This allows the two most significant bits to be chosen freely in the plaintext (i.e. not decided by the decryption of the tag-containing ciphertext). We choose these two bits so that the plaintext always falls within the ASCII range 0x40 to 0x7f. We verified that Apache was capable of handling this range of characters in a header line without triggering an error.

Elliptic curve encoding In order to conserve bandwidth, our ISP station proxy design, like previous schemes, uses elliptic curve cryptography to perform a Diffie-Hellman handshake between client and station. However, traditional encoding of elliptic curve points is distinguishable from random for a few reasons.

Fix a prime p and an elliptic curve $E: y^2 = x^3 + Ax + b$ over the field $\mathbb{F}_p$. First, transmitting both the x and y-coordinates allows an adversary to test whether a suspected point lies on the curve E. The client can encode only the x-coordinate (i.e., "compressed" format), but an adversary can still test whether $x^3 + Ax + b$ is a square over the given curve's field; since this happens with probability $1/2$ for a string picked uniformly at random, an adversary can distinguish elliptic curve points from random strings over time. To address this concern, both Telex and Cirripede employ a second, related elliptic curve, called the *twist*, which has the property that all elements of the field $\mathbb{F}_p$ appear as x-coordinates on either the main curve, E, or its twist. The client then chooses randomly between these two curves when selecting a point, and the station must test candidate tags on both curves.

In TapDance, we take advantage of recent work due to Bernstein et al., named Elligator [6]. Elligator 1 and 2 are efficient encoding functions that transform, for certain forms of elliptic curves, exactly half of the points on the curve to strings that are indistinguishable from uniform random strings. Elligator may therefore be used

in the context of *re-randomizable* protocols, in order to ensure that any points sent are encodable; protocol participants should, on average, have to try twice in order to choose an encodable point. In our case, we use Elligator 2 on curve25519 [5] in conjunction with Diffie-Hellman; curve25519 is a high-security general-purpose elliptic curve at security level 2^{128}. This allows us to use only a single curve, rather than both the curve and its twist, which simplifies checking for tags at the station and decreases the size of public keys shipped to clients.

5 Security Analysis

Our threat model is similar to that of previous end-to-middle designs. We assume an adversarial censor that can observe, alter, block or inject network traffic within their domain or geographic region (i.e., country). The censor can block its own citizens' access to websites it finds objectionable, proxies, or other communication endpoints it chooses, using IP blocking, DNS blacklists, and deep-packet inspection. We assume the censor uses a blacklist-approach to blocking resources and that the censor does not wish to block legitimate websites, or otherwise cut themselves off from the rest of the Internet, which may inhibit desirable commerce or communication.

We also assume that the censor allows end-to-end encrypted communication, specifically TLS communication. As websites increasingly support HTTPS, censors face increasing pressures against preventing TLS communication [12].

Despite control over its network infrastructure, we assume the censor does not have control over end-users' computers, such as the ability to install arbitrary programs or Trojans. However, we do assume the censor can gain access to resources outside of their country, such as VPNs or private servers, by leasing them from providers.

While the threat model for TapDance is similar to those assumed by prior end-to-middle schemes, our fundamentally new design has a different attack surface than the others. We perform a security analysis of TapDance and compare it to the previous generation designs.

5.1 Passive attacks

TCP/IP protocol fingerprinting The adversary could attempt to observe packets coming from potential decoy servers and build profiles for each server, including the set of TCP options supported by the server, IP TTL values, TCP window sizes, and TCP timestamp slope and offset. If these values ever change, particularly in the middle of a connection (and only for that connection), it could be a strong indication of a particular flow using a proxy at an on-path ISP. To prevent this attack, the station also needs to build these profiles for servers, either by actively collecting this profile from potential servers, or passively observing the server's responses to non-proxy connections and extracting the parameters. Alternatively, the client can signal to the station some of the parameters. Previous generations varied in defense for this type of attack, for example, Telex's implementation is able to infer and mimic all of these parameters from observing the servers' responses, although Telex requires a symmetric path in order to accomplish this. In general, parameters that the adversary can measure for fingerprinting can also be measured by the station and mimicked.

TLS handshake TLS allows implementations to support many different extensions and cipher suites. As a result, implementations can be easy to differentiate based on the ciphers and extensions they claim to support in their ClientHello or ServerHello messages. In order to prevent this from being used to locate suspicious implementations, our proxy must blend in to or mimic another popular client TLS implementation. For example, we could support the same set of ciphers and extensions as Chrome on whatever platform the user is running the client from. Currently, our client mimics Chrome's cipher suite list for Linux.

Cryptographic attacks A computationally powerful adversary could attempt to derive the station's private key from the public key. However, our use of ECC curve25519 should resist even the most powerful computation attacks using known discrete logarithm algorithms. The largest publicly known ECC key to be broken is only 112 bits, broken over 6 months in 2009 on a 200-PlayStation3 cluster [7]. TapDance also supports increasing the key size as needed, as we are not limited to a fixed field size for our tag. In contrast, previous designs including Telex and Cirripede use fixed fields (TCP sequence numbers and TLS ClientHello nonces) and cannot as easily expand their payloads to match higher security levels.

Forward secrecy An adversary who compromises an ISP station or otherwise obtains a station's private key can use it to trivially detect both future and previously recorded flows in order to tell if they were proxy flows. Additionally, they can use the key to decrypt the user's request (and proxy's response), learning the censored website users have visited. To address the first problem, we can use a technique suggested in Telex [43], where the ISP station generates many private keys ahead of time, and stores them in either a hardware security module or offline storage, and provides all of the public keys to the clients. Clients can then cycle through the public keys they use based on a course-grained time (e.g., hours or days). The proxy could also cycle through keys, destroying expired keys and limiting access to future ones.

To address the second problem, TapDance is compatible with existing forward secure protocols. For example, for each new connection it receives, the TapDance station

can generate a new ECDH point randomly, and establish a new shared secret between this new point and the original point sent by the client in the connection tag. The station sends its new ECDH public point to the client in its hello message, and the remainder of the connection is encrypted under the new shared secret. This scheme has the advantage that it adds no new round-trips to the scheme, and only 32-bytes to the original ISP station's ClientHello message.

Packet timing and length The censor could passively measure the normal round-trip time between potential servers, and observe the set of packet lengths of encrypted data that a website typically returns. During a proxy connection, the round-trip time or the packet lengths of the apparent server may change for an observant censor, as the station may be closer or have more processing delay than the true server. This attack is possible on all three of the previous generations of E2M proxies, as performed in detail in [37]. However, such an attack at the application level may be difficult to carry out in practice, as larger, legitimate websites may have many member-only pages that contain different payload lengths and different processing overhead. The censor must be able to distinguish between "blind pages" it cannot confirm are part of the legitimate site, and decoy proxy connections. We note that this is difficult at the application level, but TCP round-trip times may have a more consistent and distinguishable difference.

Lack of server response If the TapDance station fails to detect a client's flow, it will not respond to the client. This may appear suspicious to a censor, as the client sends a request, but there is no response at the application level from the server. This scenario could occur for three reasons. First, the censor may disrupt the path between client and TapDance station in order to cause such a timeout, using one of the active attacks below (such as the routing-around attack), to confirm a particular flow is attempting to use TapDance. Second, such false pickups may happen intermittently (due to ISP station failure). Finally, a client may attempt to find new TapDance stations by probing many potential decoy servers with tagged TLS connections. Paths that do not contain ISP-hosted proxies will have suspiciously long server response times.

To address the last issue, probing clients could send complete requests and tag their requests with a secret nonce. The station could record these secret nonces, and at a later time (out of band, or through a different TapDance station), the client can query the proxy for the secret nonces it sent. In this way, the client learns new servers for which the ISP station is willing to proxy, without revealing the probing pattern. To address the first two problems, we could have clients commonly use servers that support long-polling HTTP push notification. In these services, normal requests can go unanswered at the application layer as long as the server does not have data to send to the client, such as in online-gaming or XMPP servers. Another defense is to have the client send complete requests that force the server to keep the connection alive for additional requests, and have the TapDance station inject additional data *after* the server's initial response. This requires careful knowledge of the timing and length of the server's initial response, which could either be provided by active probing from the station, or information given by the client.

5.2 Active attacks

TLS attacks The censor may issue fake TLS certificates from a Certificate Authority under its control and then target TLS sessions with a man-in-the-middle attack. While TapDance and previous designs are vulnerable to this attack, there may be external political pressure that discourages a censor from this attack, as it may be disruptive to foreign e-commerce in particular. We also argue that as the number of sites using TLS continues to increase, this attack becomes more expensive for the censor to perform without impacting performance. Finally, decoy servers that use certificate pinning or other CA-protection mechanisms such as Perspectives [41] or CAge [25] can potentially avoid such attacks.

Packet injection Because TapDance does not block packets from the client to the true server, it is possible for the censor to inject spoofed probes from the client that will reach the server. If the censor can craft a probe that will result in the server generating a response that reveals the server's true TCP state, the censor will be able to use this response to differentiate real connections from proxy connections. While the previous designs also faced this threat [37], the censor had to inject the spoofed packet in a way that bypassed the station's ISP inline blocking element. In TapDance, there is no blocking element, and so the censor is able to simply send it without any routing tricks. An example of this attack is the censor sending a TCP ACK packet with a stale sequence number, or one for data outside the server's receive window. The server will respond to this packet with an ACK containing the server's TCP state (sequence and acknowledgement), which will be smaller than the last sequence and/or acknowledgements sent by the station.

There are a few ways to deal with this attack if the censor employs it. First, we can simply limit each proxy connection to a single request from the client, and a response from the station, followed immediately by a connection close. This will dramatically increase the overhead of the system, but will remove the potential for the adversary to use injected packets and their responses to differentiate between normal and proxy connections. This is because

the TCP state between the station and real server will not diverge until the station has sent its response, leaving only a very small window where the censor can probe the real server for its state and get a different response.

Active Defense Alternatively, in order to frustrate the censor from performing packet injection attacks, we can perform *active defense*, where the station observes active probes such as the TCP ACK, and responds to them in a way that would "reveal" a proxy connection, even for flows that are not proxy connections. To the censor, this would make even legitimate non-proxy connections to the server appear as if they were proxy connections. As an example, consider a censor that injects a stale ACK for suspected proxy connections. Connections that are actually proxy connections will respond with a stale ACK from the server, revealing the connection to the censor. However, the station could detect the original probe, and if it is not a proxied connection, respond with a stale ACK so as to make it appear to the censor as if it were. In this way, for every probe the censor makes, they will "learn", sometimes incorrectly, that the connection was a proxy connection.

Replay attacks The censor could capture suspected tags and attempt to replay them in a new connection, to determine if the station responds to the tag. To specifically attack TapDance, the adversary could replay the client's tag-containing request packet after the connection has closed and observe if the the station appears to send back a response. We note that both Cirripede and Decoy Routing are also vulnerable to tag replay attacks, although Telex provides some limited protection from them. To protect against duplicated tags, the station could record previous tags and refuse to respond to a repeated tag. To avoid saving all tags, the station could require clients to include a recent timestamp in the encrypted payload[1].

However, such a defense may enable a denial of service attack: the censor could delay the true request of a suspected client, and send it in a different connection first. In this preplay version of the attack, the censor is also able to observe if the station responds with the ClientHello message. If it does, the censor will know the suspected request contained a tag.

Denial of Service The censor could attempt to exhaust the station's resources, by creating lots of proxy connections, or sending a large volume of traffic that the ISP station will have to check for tags using an expensive ECC function. We estimate that a single ISP station deployment of our implementation on a 16-core machine could be overwhelmed if an attacker sends approximately 1.2 Gbps of pure TLS application data packets past it. This type of attack is feasible for an attack with a small botnet, or even a few well-connected servers. Because ISPs commonly perform load balancing by flow-based hashing, we can scale our deployment linearly to multiple branches of machines and use standard Intrusion Detection Systems to detect and ignore packets that do not appear to belong to valid connections, spoofed or blacklisted sources, or match specific rules [32].

Routing around the proxy A recent paper by Schuchard et al. details a novel attack against our and previous designs [37]. In this attack, the censor is able to change local routing policy in a way that redirects outbound flows around potential proxy-deploying ISPs, while still allowing them to reach their destination. This prevents the ISP station from being able to observe the tagged flows and thus from being a proxy for the clients. However, Houmansadr et al. investigate the cost to the censor of performing such an attack, and find it to be prohibitively expensive to implement [20]. Although both of these papers ultimately contribute to deciding which ISPs should deploy proxies in order to be most resistant to routing attacks, we consider such a discussion outside the scope of this paper.

Tunneling around the proxy A more conceptually simple attack is for the censor to transparently tunnel specific suspected flows around the ISP station. For example, the censor could rent a VPN or VPS outside the country and send specific flows through them to avoid their paths crossing the ISP station. This attack is expensive for the adversary to perform, and so could not reasonably be performed for an entire country. However, it could be performed for particular targets, and combined with previous passive detection attacks to aid the censor in confirming if particular users are tagging their flows.

Complicit servers A censor may be able to compromise, coerce, or host websites that can act as servers for decoy connections. The vantage point from a server allows them to observe incomplete requests from clients, including the plaintext that the client mangled in order to produce the tag in the ciphertext. This allows the censor to both observe specific clients using the ISP station and also disrupt use of the proxy with the particular server. There is little TapDance or previous designs can do to avoid cooperation between servers and the censor, as the two can simply compare traffic received and detect proxy flows as ones that have different data at the two vantage points. However, using this vantage point to disrupt proxy use could be detected by clients, and the server avoided (and potentially notified in the case of a compromise).

5.3 Summary

The only attacks unique to TapDance include the lack of server response and packet injection attacks. Besides

[1]The client random which is sent in the encrypted payload already contains a timestamp for the first 4 bytes

these, we find our design to add no additional vulnerabilities that all previous designs were immune from. While these two attacks do pose a threat to TapDance, benefits of a practical ISP station deployment—combined with our defenses—may outweigh the potential risk.

6 Implementation

We have implemented TapDance in two parts: a client that acts as a local HTTP proxy for a user's browser, and a station that observes a packet tap at an ISP and injects traffic when it detects tagged connections. Our station code is written in approximately 1,300 lines of C, using libevent, OpenSSL, PF_RING [31], and forge_socket[2].

6.1 Client implementation

Our client is written in approximately 1,000 lines of C using libevent [27] and OpenSSL [34]. The client currently takes the domain name of the decoy server as a command line argument, and for each new local connection from the browser, creates a TLS connection to the decoy server. Once the handshake completes, the client sends the incomplete response to prevent the server from sending additional data, and to encode the secret tag in the ciphertext as specified in Section 4.2. The request is simply an HTTP request with a valid HTTP request line, "Host" header, and an "X-Ignore" header that precedes the "garbage" plaintext that will be computed to result in the chosen tag to appear in the ciphertext. We have implemented our ciphertext encoding for AES_128_GCM [36], although it also works without modification for AES_256_GCM cipher suites. We have implemented Elligator 2 to work with curve25519, in order to encode the client's public point in the ciphertext as a string that is indistinguishable from uniform random. After this 32-byte encoded point, the client places a 144-byte encrypted payload. This payload is encrypted using a SHA256 hash of the 32-byte shared secret (derived from the client's secret and station's public point) using AES128 in CBC mode[3]. The payload contains an 8-byte magic value, the 48-byte TLS master secret, 32-byte client random, 32-byte server random, and a 16-byte randomized connection ID.

6.2 Station implementation

Our TapDance station consists of a 16-core Supermicro server connected over a gigabit Ethernet to a mirror port on an HP 6600-24G-4XG switch in front of a well-used Tor exit node generating about 160 mb/s of traffic. The station uses PF_RING, a fast packet capture Linux kernel module, to read packets from the mirror interface. In addition to decreasing packet capture overhead, PF_RING supports flow clustering, allowing our implementation to spread TCP flow capture across multiple processes. Using this library, our station can have several processes on separate cores share the aggregate load.

For each unique flow (4-tuple), we keep a small amount of state whether we have seen an Application Data packet for the flow yet. If we have not, we verify the current packet's TCP checksum, and inspect the packet to determine if it is an Application Data packet. If it is, we mark this flow as having been inspected, and pass the packet ciphertext to the tag extractor function. This function extracts the potential tag from the ciphertext, decoding the client's public point using Elligator 2, generating the shared secret using curve25519, and hashing it to get the AES decryption key for the payload. The extractor decrypts the 144-byte payload included by the client, and verifies that the first 8 bytes are the expected magic value. If it is, the station knows this is a tagged flow, and uses the master secret and nonces extracted from the encrypted payload to compute the key block, which contains encryption and decryption keys, sequence numbers or IVs, and MAC keys (if not using authenticated encryption) for the TLS session between the client and server.

This "ciphertext-in-ciphertext" is indistinguishable from random to everyone except the client and station. The 144-byte payload is encrypted using a strong symmetric block cipher (AES128) in CBC mode, whose key is derrived from the client-station shared secret. The remainder of the tag is the client's ECDH public point, encoded using Elligator 2 [6] over curve25519 [5]. The encoded point is indistinguishable from uniform random due to the properties of the Elligator 2 encoding function.

Once the station has determined the connection is a tagged flow, it sets up a socket in the kernel to allow it to spoof packets from and receive packets for the server using the forge_socket kernel module. The station makes this socket non-blocking, and attaches an SSL object initialized with the extracted key block to it. The station then sends a response to the client over this channel, containing a confirmation that the station has picked up, and the number of bytes that the client is allowed to send toward this station before it must create a new connection.

6.3 Connection limits

Because the server's connection with the client remains open, the server receives packets from the client, including data and acknowledgments for the station's data. The server will initially ignore these messages, however there are two instances where the server will send data. When it does so, the censor would be able to see this anomalous behavior, because the server will send data with stale sequence numbers and different payloads from what the station sent.

The first instance of the server sending data is when the server times out the connection at the application level.

[2] https://github.com/ewust/forge_socket/

[3] We use the first 16-bytes of the shared secret hash as the key, and the last 16 bytes as the initialization vector (IV)

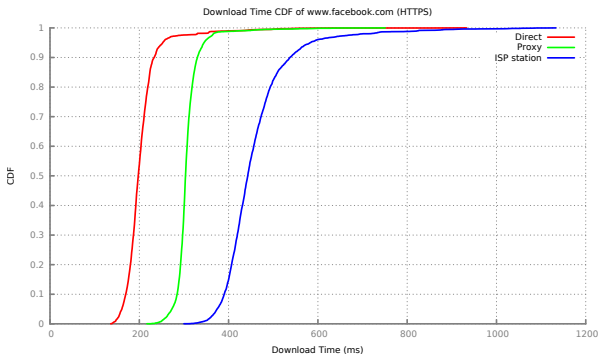

Figure 5: Download times through proxy — We used Apache Benchmark to download www.facebook.com 5000 times with a concurrency of 100 over HTTPS directly, through a single-hop proxy, an through our proof-of-concept end-to-middle proxy.

For example, web servers can be configured to timeout incomplete requests after a certain time, by using the mod_reqtimeout[4] module in Apache. We found through our development and testing the shortest timeout was 20 seconds, although most servers had much longer timeouts, such as Nginx with a default of over 5 minutes.

The second reason a server will send observable packets back to the client is if the client sends it a sequence number that is outside of the server's current TCP receive window. This happens when the client has sent more than a window's worth of data to the station, at which point the server will respond with a TCP ACK packet containing the server's stale sequence and acknowledgement numbers, alerting an observant censor to the anomaly.

To prevent both of these instances from occurring in our implementation, we limit the connection duration to less than the server's timeout, and we limit the number of bytes that a client can send to the station to up to the server's receive window size. Receive window sizes after the TLS handshake completes are typically above about 16 KB. We note that the station is not limited to the number of bytes it can send to the client per connection, making the 16 KB limit have minimal impact on most download-heavy connections.

In the event that the client wants to maintain its connection for longer than the duration or send more than 16 KB, the client can reuse the 16-byte connection ID in a new E2M TLS connection to the server. The station will decode the connection ID and reconnect the new flow to the old proxy connection seamlessly. This allows the browser to communicate to the HTTP proxy indefinitely, without having to deal with the limitations of the underlying decoy connection.

7 Evaluation

Throughout our evaluation, we used a client running Ubuntu 13.10 connected to a university network over gigabit Ethernet. For our decoy server, we used a Tor exit server at our institution, with a gigabit upstream through an HP 6600-24G-4XG switch. For our ISP station, we used a 16-core Supermicro server with 64 GB of RAM, connected via gigabit NICs to an upstream and to a mirror port from the HP switch. Our ISP station is therefore able to observe (but not block) packets to the Tor exit server, which provides a reasonable amount of background traffic on the order of 160 mb/s. In our tests, the Tor exit node generates a modest amount of realistic user traffic. Although not anywhere near the bandwidth of a Tier-1 ISP, Tor exit nodes generate a greater ratio of HTTPS flows than a typical ISP (due to the Tor browser's inclusion of the HTTPS Everywhere plugin), and we can use this microbenchmark to perform a back-of-the-envelope calculation to the loads we would see at a 40 Gbps Transit ISP tap.

We evaluate our proof-of-concept implementation with the goal of demonstrating that our system operates as described, and that our implementation is able to function within the constraints of our mock-ISP. To demonstrate that our system operates as described, we set Firefox to use our client as a proxy, and browsed several websites while capturing packets on the client and the decoy server. We then manually inspected the recorded packets to confirm that there were no additional packets sent by the server that would reveal our connections to be proxied connections. Empirically, we note that we are able to easily browse the Internet through this proxy, for example watching high-definition YouTube videos.

To evaluate the performance of our system, we created 8 proxy processes on our ISP station, using the same PF_RING cluster ID in order to share the load across 8 cores. The background traffic from the Tor exit server does not appear to have a significant impact on the proxy's load: each process handles between 20 and 50 flows at a given time, comprising up to 35 mb/s of TLS traffic. The CPU load during this time was less than 1%.

We used Apache Benchmark[5] in order to issue 5,000 requests through our station proxy, with a concurrency of 100, and compared the performance for fetching a simple page over HTTP and over HTTPS. We also compare fetching the same pages directly from the server and through a single-hop proxy. Figure 5 shows the cumulative distribution function for the total time to download the page. Although there is a modest overhead for end-to-middle proxy connections compared to direct or simple proxies, the overhead is not prohibitive to web browsing habits;

[4] http://httpd.apache.org/docs/2.2/mod/mod_reqtimeout.html
[5] http://httpd.apache.org/docs/2.2/programs/ab.html

users are still able to interact with the page, and pages can be expected to load in a reasonable time period. In particular, our proxy adds a median latency of 270 milliseconds to a page download in our tests when compared with a direct download.

We find that the CPU performance is bottlenecked by our single-threaded client. During our tests, the client consumes 100% CPU on a single core, while each of the 8 processes on the ISP station consume between 4-7% CPU. We also observe that a majority of the download time is spent waiting for the connection handshake to complete with the server. To improve this performance, we could speculatively maintain a connection pool in order to decrease the wait-time between requests. However, care must be taken in order to mimic the same connection pool behaviors that a browser might exhibit.

We also note that although the distribution of download times appear different for ISP station vs. normal connections, this does not necessarily indicate an observable feature for a censor. This is because our download involves a second round trip between client and server before the data reaches the client. The censor would still have to distinguish between this type of connection behavior and innocuous HTTP pipelined connections. It still may be possible for the censor to distinguish, however, as we discussed in Section 5, traffic analysis is an open problem for existing network proxies, and outside the scope of this paper.

Tag creation and processing In order to evaluate the overhead of creating and checking for tags, we timed the creation and evaluation of 10,000 tags. We were able to create over 2,400 tags/second on our client and verify over 12,000 tags/second on a single core of our ISP station. We find that the majority of time (approximately 80%) during tag creation is spent performing the expected three ECC point multiplications (an expected two to generate the client's Elligator-encodable public point and one to generate the shared secret). Similarly, during tag checking, nearly 90% of the computation time is spent on the single ECC point multiplication. Faster ECC implementations (such as tuned-assembly or ASICs) could have a significant impact toward improving the performance of tag verification on the ISP station.

8 Future Work

The long term-goal of end-to-middle proxies is to be implemented and deployed in a way that effectively combats censorship. While we have suggested a design that we believe is more feasible than previous work, more engineering must be done to bring it to maturity.

One open research question is where specifically in the network such proxies should be deployed. Previously, "Routing around Decoys" [37] outlined several novel attacks that a censor could perform in order to circumvent many anticensorship deployment strategies. There is ongoing discussion in the literature about the practical costs of these attacks, and practical countermeasures deployments could take to protect against them [9, 20].

As mentioned in Section 5, traffic fingerprinting is a concern for all proxies, and remains an open problem. Previous work has discussed these attacks as they apply to ISP-located proxies [37] and other covert channel proxies [15, 17]. Future work in this direction could provide insight into how to generate or mimic network traffic and protocols.

Finally, there is room to explore more active defense techniques, as outlined in Section 5. As end-to-middle proxies become more prominent, this is likely to become an important problem, as China has already started to employ active attacks in order to detect and censor Tor bridge relays [11, 42]. Collaborating with ISPs will allow us to explore the the technical capabilities and policies that would permit active defense against these attacks.

9 Conclusion

End-to-middle proxies are a promising concept that may help tilt the balance of power from censors to citizens. Although previous designs including Telex, Cirripede, and Decoy Routing have layed the ground for this new direction, there are several problems when it comes to deploying any of these designs in practice. Previous designs have required inline blocking elements and sometimes assumed symmetric network paths. To address these concerns, we have developed TapDance, a novel end-to-middle proxy that operates without the need for inline flow blocking. We also described a novel way to support asymmetric flows without inline-flow blocking, by encoding arbitrary-length steganographic payloads in ciphertext. This covert channel may be independently useful for future E2M schemes and other censorship resistance applications.

Ultimately, anticensorship proxies are only useful if they are actually deployed. We hope that removing these barriers to end-to-middle proxying is a step towards that goal.

References

[1] Ultrasurf. https://ultrasurf.us/.

[2] Luis Ahn and NicholasJ. Hopper. Public-key steganography. In Christian Cachin and JanL. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 323–341. Springer Berlin Heidelberg, 2004.

[3] R. J. Anderson and F. A.P. Petitcolas. On the limits of steganography. *IEEE J.Sel. A. Commun.*, 16(4):474–481, September 2006.

[4] Michael Backes and Christian Cachin. Public-key steganography with active attacks. In *Proceedings of the*

Second International Conference on Theory of Cryptography, TCC'05, pages 210–226, Berlin, Heidelberg, 2005. Springer-Verlag.

[5] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *Public Key Cryptography-PKC 2006*, pages 207–228. Springer, 2006.

[6] Daniel J Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: Elliptic-curve points indistinguishable from uniform random strings. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 967–980. ACM, 2013.

[7] Joppe W. Bos, Marcelo E. Kaihara1, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. Playstation 3 computing breaks 2^{60} barrier 112-bit prime ecdlp solved. http://lacal.epfl.ch/112bit_prime, 2009.

[8] Sam Burnett, Nick Feamster, and Santosh Vempala. Chipping away at censorship firewalls with user-generated content. In *USENIX Security Symposium*, pages 463–468. Washington, DC, 2010.

[9] Jacopo Cesareo, Josh Karlin, J Rexford, and M Schapira. Optimizing the placement of implicit proxies, 2012.

[10] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[11] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[12] Zakir Durumeric, James Kasten, Michael Bailey, and J Alex Halderman. Analysis of the https certificate ecosystem. In *Internet Measurement Conference*, 2013.

[13] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 61–72, New York, NY, USA, 2013. ACM.

[14] David Fifield, Nate Hardison, Jonathan Ellithorpe, Emily Stark, Dan Boneh, Roger Dingledine, and Phil Porras. Evading censorship with browser-based proxies. In *Privacy Enhancing Technologies*, pages 239–258. Springer, 2012.

[15] John Geddes, Max Schuchard, and Nicholas Hopper. Cover your acks: pitfalls of covert channel censorship circumvention. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 361–372. ACM, 2013.

[16] Theodore G. Handel and Maxwell T. Sandford, II. Hiding data in the OSI network model. In *Proceedings of the First International Workshop on Information Hiding*, pages 23–38, London, UK, UK, 1996. Springer-Verlag.

[17] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The parrot is dead: Observing unobservable network communications. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 65–79. IEEE, 2013.

[18] Amir Houmansadr, Giang TK Nguyen, Matthew Caesar, and Nikita Borisov. Cirripede: circumvention infrastructure using router redirection with plausible deniability. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 187–200. ACM, 2011.

[19] Amir Houmansadr, Thomas Riedl, Nikita Borisov, and Andrew Singer. I want my voice to be heard: Ip over voice-over-ip for unobservable censorship circumvention. In *The 20th Annual Network and Distributed System Security Symposium (NDSS)*, 2013.

[20] Amir Houmansadr, Edmund L Wong, and Vitaly Shmatikov. No direction home: The true cost of routing around decoys. 2014.

[21] Luca Invernizzi, Christopher Kruegel, and Giovanni Vigna. Message in a bottle: sailing past censorship. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 39–48. ACM, 2013.

[22] J Jia and P Smith. Psiphon: Analysis and estimation, 2004.

[23] W. John, M. Dusi, and k. claffy. Estimating Routing Symmetry on Single Links by Passive Flow Measurements. Technical report, ACM 1st International Workshop on TRaffic Analysis and Classification (TRAC), Mar 2010.

[24] Josh Karlin, Daniel Ellard, Alden W Jackson, Christine E Jones, Greg Lauer, David P Mankins, and W Timothy Strayer. Decoy routing: Toward unblockable internet communication. In *USENIX Workshop on Free and Open Communications on the Internet*, 2011.

[25] James Kasten, Eric Wustrow, and J Alex Halderman. CAge: Taming certificate authorities by inferring restricted scopes. In *Financial Cryptography and Data Security*, pages 329–337. Springer, 2013.

[26] Adam Langley. Tls symmetric crypto. https://www.imperialviolet.org/2014/02/27/tlssymmetriccrypto.html, February 2014.

[27] Nick Mathewson and Niels Provos. libevent: an event notification library. http://libevent.org/.

[28] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. Skypemorph: Protocol obfuscation for tor bridges. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 97–108. ACM, 2012.

[29] Bodo Möller. A public-key encryption sesocheme with pseudo-random ciphertexts. In Pierangela Samarati, Peter Ryan, Dieter Gollmann, and Refik Molva, editors, *Computer Security – ESORICS 2004*, volume 3193 of *Lecture Notes in Computer Science*, pages 335–351. Springer Berlin Heidelberg, 2004.

[30] Steven J. Murdoch and Stephen Lewis. Embedding covert channels into TCP/IP. In Mauro Barni, Jordi Herrera-Joancomartí, Stefan Katzenbeisser, and Fernando Pérez-González, editors, *Information Hiding*, volume 3727 of *Lecture Notes in Computer Science*, pages 247–261. Springer Berlin Heidelberg, 2005.

[31] ntop.org. PF_RING: High-speed packet capture, filtering and analysis. http://www.ntop.org/products/pf_ring/.

[32] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.

[33] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[34] OpenSSL Project. OpenSSL: Cryptography and SSL/TLS toolkit. http://www.openssl.org/.

[35] Phillip Rogaway. Evaluation of some blockcipher modes of operation. *Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan*, 2011.

[36] J. Salowey, A. Choudhury, and D. McGrew. AES Galois Counter Mode (GCM) Cipher Suites for TLS. RFC 5288 (Proposed Standard), August 2008.

[37] Max Schuchard, John Geddes, Christopher Thompson, and Nicholas Hopper. Routing around decoys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 85–96. ACM, 2012.

[38] Gustavus J. Simmons. The prisoners' problem and the subliminal channel. In David Chaum, editor, *Advances in Cryptology*, pages 51–67. Springer US, 1984.

[39] Qiyan Wang, Xun Gong, Giang TK Nguyen, Amir Houmansadr, and Nikita Borisov. Censorspoofer: asymmetric communication using ip spoofing for censorship-resistant web browsing. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 121–132. ACM, 2012.

[40] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. StegoTorus: a camouflage proxy for the Tor anonymity system. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 109–120. ACM, 2012.

[41] Dan Wendlandt, David G Andersen, and Adrian Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference*, pages 321–334, 2008.

[42] T. Wilde. Great Firewall Tor probing. https://gist.github.com/twilde/da3c7a9af01d74cd7de7, 2012.

[43] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. Telex: Anticensorship in the network infrastructure. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.