Metadata Protection Considerations for TLS Present and Future (draft)

Bryan Ford
EPFL

Abstract

TLS 1.3 takes important steps to improve both performance and security, so far offers little protection against traffic analysis or fingerprinting using unencrypted metadata or other side-channels such as transmission lengths and timings. This paper explores metadata protection mechanisms for TLS, including already-included provisions (*e.g.*, record padding), provisions not yet included but potentially feasible in TLS 1.3 (*e.g.*, optional or encrypted headers), and provisions that are likely too ambitious to achieve in TLS 1.3 but may be worth considering for a future "TLS 2.0" (*e.g.*, fully encrypted and authenticated negotiation/handshaking). In addition, we briefly explore how these metadata protection provisions might apply to the datagram-oriented DTLS, or to a version of TLS supporting out-of-order delivery atop TCP.

1 Introduction

The goal of SSL/TLS [5] has traditionally been to protect the integrity and privacy of the *content* communicated between two endpoints, most commonly between a web browser and web server. However, there are many increasingly well-known ways in which attackers can use *traffic analysis* to learn potentially sensitive information about a TLS communication session.

For example, an eavesdropper who can passively monitor a user's TLS communication stream, *e.g.*, by listening "promiscuously" on a public or common WiFi network, might use website fingerprinting [9, 13, 4, 11, 20] to determine what websites another user is visiting, or even which known page on those websites. Related attacks include user fingerprinting [14, 19], and application fingerprinting [22]. The attacks can be effective even against traffic on encrypted WiFi networks [16], as well as traffic protected by anonymous communication systems such as Tor [12, 13]. In general, protecting communication against traffic analysis is extremely difficult, and this paper makes no pretense at solving this problem in general. Even just the approximate total number of bytes exchanged between a client and a server, for example, can uniquely distinguish among at least small, closed sets of websites [6].

The goal of this paper is *not* to identify new fingerprinting or side-channel attacks or to attempt to determine the "best" traffic analysis countermeasures. Instead, the focus of this paper is primarily on the *protocol design* of TLS 1.3, and specifically on ways that TLS 1.3 might be made more *amenable* to the implementation of traffic analysis countermeasures. That is, the topic of this paper is not how best to protect against traffic analysis, but what *tools* or useful properties TLS could readily provide that might make implementation of countermeasures more readily feasible, more effective, and/or more efficient, based on our current knowledge of traffic analysis attacks and defenses. Some of the potential provisions discussed here seem likely to be feasible to include in TLS 1.3; others represent more significant and ambitious changes that may more realistically need to be deferred until a more fundamental TLS overhaul occurs (*e.g.*, "TLS 2.0").

This paper restricts focus only to metadata protection and traffic analysis considerations that a pairwise encrypted communication layer such as TLS can hope to address without changing the actual communication parties or routes. Thus, traffic analysis countermeasures that require adding new communication parties, such as onion routing as implemented via Tor [17], we leave out of scope.

We first briefly explore multiple relevant attacks and threat models in Section 2, then Section 3 offers an informal taxonomy of the different ways TLS currently leaks metadata information usable in traffic analysis. Section 4 next

summarizes several classes of countermeasures, each with different strengths and weaknesses, some of which are potentially complementary to each other. Finally, Section 5 covers features or protocol design provisions that could be incorporated into TLS 1.3 to make traffic analysis countermeasures more efficient and/or effective.

2 Threat Models

There are a wide variety of possible traffic analysis attacks against TLS and other encrypted communication protocols, and we make no attempt to enumerate them exhaustively. However, we outline several important classes of attackers with varying strengths and capabilities.

While there can be a temptation to consider only "the strongest" realistically envisionable attacker, this paper takes the position that there is value in considering *multiple* attacker strengths, in part because weaker attackers are likely to be far more numerous in practice than extremely strong attackers, so a relatively cheap and easy-to-implement countermeasure that frustrates only weaker attackers can provide a significant security gain even if that countermeasure is completely ineffective against the (far fewer) strong attackers.

In this paper we assume the attacker *cannot* break TLS's encryption or authentication functions; *i.e.*, cryptographic or PKI-related attacks are out of scope.

2.1 Attacker Capabilities

The two most obvious, classic attack models "passive" and "active" adversaries in the network. Based on lessons we have learned in recent years about the traffic analysis attacks and capabilities of real-world attackers, we find value in identifying a slightly more nuanced categorization of attackers, as follows:

Passive eavesdropper (EVE) The passive eavesdropper can intercept and view packets between two communicating TLS endpoints, but cannot inject or modify those packets. One realistic example of a passive eavesdropper is a person or organization with access to a network monitoring tap, a feature common in network routers that sends a copy of all received traffic of a certain class to a special monitoring port, without delaying or otherwise affecting the traffic's normal delivery.

Man-on-the-Side (MOTS) A MOTS attacker can not only monitor the TLS connection, but also inject packets that falsely appear to come from either endpoint. The MOTS attacker cannot prevent either endpoint's legitimate traffic from being delivered to the other endpoint, however, and hence is not truly "in the middle" of the connection. One important, low-cost and hence potentially pervasive manifestation of this attacker is the "WiFi snooper": a malicious user on the same (public or private) WiFi network as one of the TLS endpoints. A second real-world example is the NSA's "Quantum Insert" [**?**], which passively watches for a connection from a target of interest and attempts to inject a connection-hijacking packet that arrives before the legitimate version of the same packet. It is important in each of these cases that the attacker cannot readily block or delay legitimate traffic, which would typically require much more invasive (and perhaps expensive) access to the network path.

Man-in-the-Middle (MITM) The full MITM attacker can monitor the connection, inject traffic, and selectively block or delay legitimate traffic. In practice, any routers, firewalls, and other middleboxes directly on the path can perform such attacks – though such attack capabilities often come at higher cost to the attacker and may run greater risk of detection.

Man-on-the-Inside (MOTI) The MOTI attacker can not only monitor and perhaps manipulate the TLS traffic from within the network, but also can exert at least limited control over the traffic flowing *inside* the TLS connection. For example, a single persistent TLS connection often carries HTTP traffic associated with multiple distinct security contexts (*i.e.*, "origins"). An attacker who controls one such origin, such as a malicious online advertisement, can send JavaScript to the user's browser that conspires with the attacker's server to "shape" the traffic within the TLS tunnel to make traffic analysis or side-channel attacks easier against *other* traffic from different security contexts whose traffic

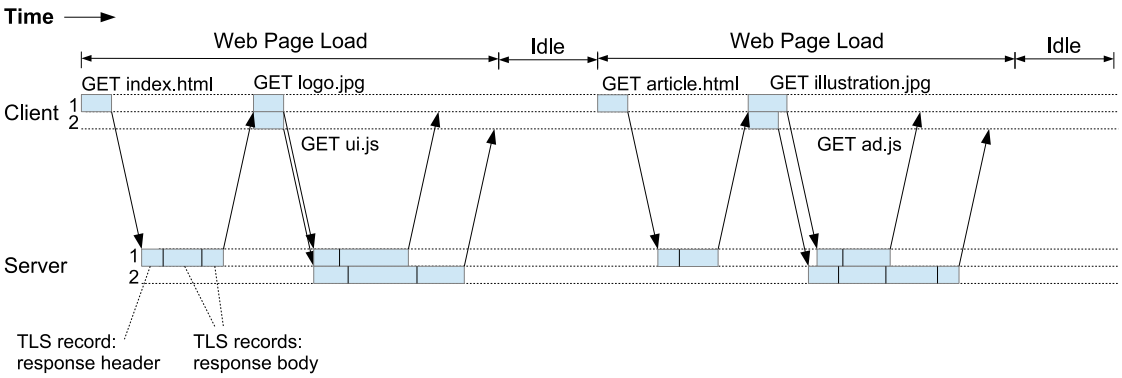

Figure 1: Illustration of the traffic pattern of basic Web browsing behavior visible to traffic analysis.

shares the same TLS tunnel. The CRIME attack [8] against the closely-related SPDY protocol [1] was one example of such an attack, in which the attacker's JavaScript would "prime" shared data compression state in order to confirm guesses of password characters a user enters into a different origin.

2.2 Attacker Objectives

Attackers may have many different objectives in mounting traffic analysis attacks; we mention only a few of particular interest:

- Fingerprinting users: *e.g.*, determining which TLS connections emerging from a corporate NAT/firewall over time belong to the same user [19], or tracking a mobile user as he/she moves from one Internet access point to another.
- Fingerprinting user's communication behaviors or targets: *e.g.*, website, which page on website, specific product being viewed or purchased, etc.
- Side-channel information theft: attempting to learn secret information leaked via side-channels in the observable TLS traffic pattern, as in the CRIME attack for example.

Finally, some attackers worry about their actions being detected and are likely to avoid attack modes that risk detection (*e.g.*, the NSA); other attackers often do not care and are perfectly willing to mount readily detectable and even "obvious" attacks (*e.g.*, China's Great Firewall). Some potential traffic analysis countermeasures may be effective only against attackers who wish to avoid detection; such countermeasures can still be valuable and increase net security even if they deter only some but not all attackers.

3 Modes of Leakage

Figure 1 shows a timeline illustrating the structure of a basic Web browsing activity. Web browsers typically open several parallel TLS connections to the same server for parallelism (illustrated as only two connections for simplicity, though in practice usually more than two). Loading a page or clicking a link typically causes the browser first to load one object, typically an HTML file, which contains many embedded objects that the browser then loads in subsequent requests over all available TLS connections. After a page loads, the connections typically go idle. This idle time might end either when the user clicks the next link, or when JavaScript running in the client's browser initiates communication, to check periodically for updates to a social networking feed for example.

We now briefly summarize some of the (many) ways a typical Web browsing activity such as this can leak information usable for traffic analysis, even if the attacker cannot see the content of the requests and replies:

Total Size Measures The most obvious and coarse-grained information that any interactive activity leaks is its "total size." Further, a traffic analysis attacker has multiple metrics available by which to measure an activity's size: *e.g.*, number of bytes transmitted in either direction, number of TCP segments, number of TLS records, or simply the total time taken by a burst of activity such as a page load. Even these extremely coarse-grained statistics alone can allow an attacker to distinguish quite reliably among a limited set of known web sites [6]. Such coarse-grained information alone may be less uniquely distinguishing among the "open universe" of the large number of sites on the actual Web. If a particularly popular Web page is known to require approximately some number of bytes to load, however, an attacker may still identify loads of that page with fair certainty, at the risk of false positives when users occasionally load less-popular sites that happen to have similar aggregate metrics.

Size and Timing of Transmission Bursts As illustrated in Figure 1, a single user-initiated "activity" such as the load of a web page can involve multiple bursts of TLS communication in each direction: most commonly, one burst in each direction to fetch the main HTML file, and one or more subsequent bursts to fetch the (often many) embedded objects such as images and JavaScript libraries comprising typical Web pages. Each such communication burst within an activity potentially gives an attacker more detailed, fine-grained information with which to fingerprint the activity or exploit side-channels.

For example, an attacker might discover that a particular page on a bank's website displays sensitive balance information, and that the size in bytes of one particular object on the page (*e.g.*, the Web server's response to a particular JavaScript-initiated request) corresponds exactly to the number of digits in the user's bank balance. If the attacker can fingerprint that particular page accurately enough to be able to tell when one of a set of surveilled users loads that page, *and* reliably differentiate between the "leaky" object of interest and all of the other embedded objects in the page, then the attacker can effectively exploit this side-channel leak. However, if the attacker had only aggregate information about the total size of all the objects comprising the page – some of which may be highly "noisy" advertisements and other dynamic content – then it might be impractical to isolate and exploit object-granularity leaks.

In the case of HTTP, one potentially important factor is whether the connection uses pipelining. Without pipelining, each client request is typically a single upstream burst, followed by an idle period as the client waits for the server's downstream response to that (single) request, followed by another upstream request, etc. With pipelining, the client may submit multiple requests "back-to-back" without waiting for the first response, and the corresponding responses are similarly likely to arrive "back-to-back," making it harder for traffic analysis to differentiate individual objects. Thus, we can think of traffic analysis as a weak but still useful traffic analysis countermeasure, independent of its primary purpose of improving performance. Unfortunately, while HTTP/1.1 has long supported pipelining [7], most Web clients have never enabled it for backward compatibility reasons. HTTP/2 [2], however, supports not only pipelining but more general multi-streaming – multiple requests concurrently multiplexed over one TLS connection – and thus we can expect batches of pipelined requests and replies to become the norm.

TLS Record Structure TLS streams consist of records, each of which traditionally contains an unencrypted (but integrity-checked) header followed by an encrypted payload and MAC tag. While all the content of each record is encrypted, any eavesdropper can observe the precise size of each record, which can reveal even more fine-grained information usable by an attacker for traffic analysis.

In particular, TLS record boundaries often reflect the logical boundaries of data written by the application using TLS. For example, when an HTTP request or reply header is followed by a body, the header is typically written as one record, followed by one or more records containing the body, depending on the length of the body. This makes it particularly trivial for any eavesdropper to tell (a) whether or not a request or reply has a body at all, (b) what the length of the header is independent of the length of the body, and (c) what the length of the body is independent of the length of the header.

All of these pieces of information can be useful to an attacker. For example, one recent replay attack against HTTPS [18] leverages the fact that POST requests are often trivially distinguishable from GET requests, by virtue of POST requests generally have a body whereas GET requests have none. To detect the difference, therefore, an attacker simply checks whether the upstream burst consists of one TLS record or several.

Further, since much of a typical request header is predictable, an attacker may be able to work back from the size of the header (represented by the first TLS record in a burst) to deduce the exact length of the URL being requested or

posted to. If the attacker knows or has already deduced through more coarse-grained fingerprinting techniques which web site the user is visiting, knowing the length of the specific URL can potentially provide the attacker a fairly unique fingerprint of the precise page on that site. The length of such a URL might also reveal other personal information: e.g., a variable-length field such as '`?gender=male`' or '`?gender=female`' may immediately reveal the user's gender if the rest of the URL has a fixed or predictable size.

With pipelined HTTP/1.2 or multi-streaming HTTP/2, the record boundaries that TLS reveals can negate the potential traffic analysis protection advantages of multi-streaming. If an HTTP/2 client sends several GET request in immediate succession to request multiple embedded objects within a page, for example, then it is natural for HTTP implementations to make one write per GET request, translating to one TLS record per GET request. Thus, by examining the TLS record boundaries, an eavesdropper can determine not just the total size of all the GET requests the client sends in a burst, but rather: (a) the *individual* size of each GET request, (b) the *number* of GET requests submitted, and (c) the *order* in which the client submitted those requests. All of this can provide the attacker a much higher-rate leak containing much more fine-grained information to be potentially exploited. For example, if the URL size of one embedded object on a known web page leaks sensitive information but other URLs loaded by the same page are have unpredictable sizes, then an attacker who can observe the size of each TLS record may be able to extract that sensitive information reliably, whereas an attacker who only observed the total size of a burst of GET requests for all the page's embedded objects could not.

TCP Segment Structure Finally, the length and timings of TCP segments transmitted can reveal fine-grained information similar to that leaked by the TLS record boundaries. TCP segment boundaries need not correspond to TLS record boundaries, but often do, if the TCP stack "pushes" TCP data to the network immediately when the application (in this case TLS) writes it. TCP segment boundaries can be de-synchronized from TLS boundaries for a number of reasons, however: *e.g.*, if the TCP stack already had data in its write buffer waiting to be transmitted due to congestion control, then the TCP stack may simply append further written data directly onto the existing write buffer, causing a TCP segment to be subsequently transmitted containing the end of one TLS record and the beginning of another. Further, middleboxes in the network can and do re-segment TCP streams, *e.g.*, when acting as an invisible TCP-level proxy by terminating the TCP connection arriving from one endpoint and forwarding its traffic to a separate TCP connection with the other remote endpoint. Thus, while TCP segments can potentially reveal the same fine-grained structure information that TLS records do, TCP segment boundaries may be "noisier" for a variety of reasons and hence potentially less reliably useful to attackers.

4 Potential Countermeasures for TLS

We now explore several classes of potential countermeasures and how they might affect considerations for TLS 1.3's design. Any countermeasures that a TLS implementation might deploy have their limits, exhibiting varying costs and benefits, and some countermeasures are complementary to each other. The countermeasures discussed here are not intended to be either particularly novel or an exhaustive treatment in general; for this purpose we refer to the extensive literature on traffic analysis attacks and defenses in general [6, 4, 20].

4.1 Padding TLS Records

The TLS 1.3 specification draft already includes a new facility allowing TLS ciphertexts to be padded, allowing the coupling between plaintext and ciphertext lengths to be loosened. Possible approaches to padding that have been suggested include (a) padding all records to a fixed length as is done in Tor [17], (b) adding a random amount of padding to each record as done in some versions of GnuTLS [1], or (c) picking a single random padding amount each session and consistently adding that amount of padding to every record [6].

The main benefit of padding records is to make it more difficult for a traffic analysis attacker to determine the exact or near-exact, byte-granularity length of an encrypted payload – information that can be used as discussed above to

[1] `http://gnutls.org/manual/html_node/On-Record-Padding.html`

extract sensitive information in some circumstances. Different approaches to padding exhibit different strengths and weaknesses, however, and each can "fail" in some scenarios.

With randomized padding, for example, if a TLS conversation contains many similar GET requests whose lengths consistently depend on some sensitive data, then the attacker can statistically average over many records to eliminate the effects of the random padding. Adding a *consistent* random amount of padding eliminates this weakness, but creates a different one: if an attacker can correctly guess the correct size of even just one record in a stream – *e.g.*, because he knows that the first (or last) record *always* has a particular well-known size for example – then he can eliminate the effects of padding from *all* records.

Padding all records to a fixed length effectively scrubs all length information from small plaintexts (*e.g.*, GET requests that are smaller than the padding length). However, if the attacker is able to manipulate the lengths of requests – via a variable-length, user-defined field that gets incorporated into sensitive URLs for example – then the attacker might be able to arrange for requests either to "just barely" fit in one padded record or (when just one byte larger) to overflow into the next padded record, depending on sensitive information in the request. In this way, the attacker might be able to arrange for a one-byte difference in plaintext length to manifest as an equally reliable, *one-record* difference in actual traffic.

Finally, padding individual records of course does not tend to obscure aggregate traffic features such as total number of bytes or records in a burst [6]. Thus, while a carefully designed padding mechanism is almost certainly one important element in traffic analysis defense, it is not the whole story.

4.2 Hiding TLS Record Boundaries

While TLS leaves its record headers in cleartext, this is by no means essential: for example, the SSH protocol [21] in contrast encryptes record headers as well as payloads, producing a cryptographically uniform stream of random-looking bytes. As a result, a passive attacker observing the stream of encrypted bytes cannot necessarily tell readily how many records a stream contains and the individual sizes of each.

In the context of HTTP/1.1 with pipelining disabled as is commonplace, hiding TLS record boundaries could have the useful but limited effect of making it harder for an attacker to tell whether a given request or response has a body (*e.g.*, GET versus POST), or to infer the size of the header independently from that of the body. In the transition to HTTP/2, however, hiding TLS record headers could offer much more significant protection, by making a whole burst of consecutive upstream requests, and their pipelined and/or intermingled downstream responses, appear to be just one large pair of transmission bursts rather than many individual smaller request/response pairs. If the length of one object embedded in a web page depends on sensitive data, this fact may be far less readily exploitable under HTTP/2 over TLS 1.3 if the sensitive request cryptographically blends into a burst of requests and responses that includes other noise-inducing dynamic content such as advertisements.

As noted earlier, hiding TLS record boundaries may not be effective if TCP segment boundaries reveal the same information. However, there are multiple ways to decouple TCP segment boundaries, either from within the TLS implementation, the TCP stack, or even in network middleboxes such as corporate firewalls. For example, the TLS sender might buffer records to be transmitted and send them out in fixed-length writes (*e.g.*, MTU-size) that deliberately fail to correspond to TLS record boundaries. Alternatively, some TCP stacks such as Linux's offer this functionality "built-in" for any application that enables it. In particular, if the TLS implementation sets the `TCP_CORK` socket option, the TCP stack will normally transmit *only* MTU-size TCP segments, keeping short segments buffered locally until more data is available to send or a timeout expires (*e.g.*, 200ms). Alternatively, the TLS implementation can write records using `send()` instead of `write()` and specify the Linux `MSG_MORE` flag on all but the last record in a burst of records to send, to exercise more precise control over when the sequence of MTU-sized segment transmissions concludes.

4.3 Limiting Leakage via Aggregate Metrics

TLS record padding can reduce *byte-granularity* information leakage through record length, and hiding TLS record boundaries can reduce *record-granularity* information leakage within a transmission burst. This still leaves information leakage through the total sizes of bursts, however, which could be measured in bytes, segments, time, or other metrics.

The only *provably* strong defenses we are aware of against fingerprinting attacks using aggregate information such as this [20] involve the definition or synthesis of generic "traffic profiles" for different categories of web sites or activities, and actively shaping and constraining communication activity to fit one of those traffic profiles. This ensures that an eavesdropper at least cannot distinguish between different activities assigned to the same traffic profile: *e.g.*, between web sites with similar aggregate size metrics. However, actually defining (and potentially standardizing) a suitable body of such traffic profiles for TLS applications, presumably based on some form of systematic analysis of common traffic profiles of popular real-world Web sites, would present major practical challenges.

As a slightly weaker but perhaps more readily implementable and deployable alternative, therefore, we suggest a countermeasure that focuses not on bucketing web sites into traffic profiles but instead on bucketing *transmission burst lengths*. The goal is simply to limit the effective amount of information that the length of any single TLS transmission burst may leak, by allowing bursts to have only a restricted set of lengths and hence forcing many similar-length bursts into equivalence classes in which they become cryptographically indistinguishable and hence "anonymous" within that equivalence class.

Since a communication activity may involve multiple transmission bursts, an attacker who observes a succession of bursts can still potentially learn more fingerprintable and analyzable information across bursts, so this is by no means a perfect countermeasure. However, by combining TLS record padding and boundary hiding as discussed above with the effects of HTTP/2 multi-streaming, many web-page loads might soon start to take the general shape of just one or two fully uniform-looking (to an attacker) transmission bursts in each direction.

4.3.1 Burst Length Bucketing and Binary Magnitudes

There are many ways burst lengths could be bucketed into equivalence classes: for example, we could simply require that the length of all bursts be rounded up in size to the next power of two. Because a power of two can be represented merely as an integer exponent, this implies that transmission bursts of length up to N bytes can leak at most $O(\log \log N)$ bits of information per burst, in contrast to the $O(\log N)$ bits of information that the burst's "precise" plaintext length could leak. Thus, if we view the process of encryption as reducing maximum leakage of a plaintext up to N bits long from $O(N)$ down to $O(\log N)$ bits, padding the length to the next power of two offers the attractive asymptotic "next step" reduction from $O(\log N)$ leakage to $O(\log \log N)$. The cost, of course, is that this padding policy may incur up to almost 100% bandwidth overhead on a given burst.

A more nuanced approach we suggest instead pads lengths to the next higher *binary magnitude* or **mag**, as defined informally below:

> A binary magnitude or **mag** is a positive integer that is exactly representable in a binary floating-point representation whose mantissa has no more significant bits than its exponent.

For example, lengths between 128 and 255 would be represented in binary floating-point notation with an exponent of 7 ($2^7 = 128$), which can be expressed in three bits (binary 111). Therefore, an integer between 128 and 255 is a **mag** if and only if can be expressed in the form $1.bbb \times 2^7$, where bbb is a three-bit mantissa no larger than the three-bit exponent. Thus, integers in this range are **mag**s only if they are divisible by 16: *i.e.*, $128, 144, 160, \ldots, 224, 240, 256$.

The key benefit of rounding lengths to the next **mag** instead of the next power of two is that maximum bandwidth overhead, expressed as a percentage of wasted over "useful" bandwidth, is never more than about 11% and gradually decreases with larger sizes. For example, for sizes between 256 bytes and 64KB, padding waste is less than 7%, and for bursts between 64KB and 4GB waste is less than 4%.

Nevertheless, the maximum amount of information a **mag** can leak in an information-theoretic sense is still asymptotically $O(\log \log N)$, just like a power-of-two. This is because adding the mantissa to the **mag**'s representation (as opposed to keeping only the exponent) by definition increases by only a constant factor (namely $2\times$) the size of the representation in bits and hence the maximum possible information its value can leak to an adversary.

As an added bonus, a **mag** can be represented extremely compactly in protocols when needed. For example, we can split a single byte into a 4-bit exponent and 4-bit mantissa to represent any **mag** between 1 and 63488 (binary 1.1111×2^{15}), large enough for the maximum-size TLS records. More strikingly, two bytes (a one-byte exponent and one-byte mantissa) can represent any **mag** between 1 and nearly 2^{256}.

5 TLS 1.3 Design Considerations

While the above discussion barely scratches the surface of potential traffic analysis countermeasures that might in principle be applied in the context of TLS, it hopefully illustrates the multiple levels and granularities of information leakage that traffic analysis can utilize, potentially requiring different and complementary sets of countermeasures. We now consider what these considerations might mean for TLS's ongoing design.

Fortunately, most of the traffic analysis countermeasures we envision implementations *might* want to implement do not require any special provisions in the TLS protocol specification, but can be implemented essentially independently as one of many valid implementations of the same protocol. For example, while protecting aggregate measures such as total burst length as discussed above is probably an important part of any systematic defense against traffic analysis, it is not clear that this presents any immediate demands on the protocol specification from a wire-format perspective.

5.1 Padding and Record Boundary Hiding

The TLS 1.3 draft already includes a provision for record padding, and there are several ways record boundary hiding could be added, as recently discussed on the TLS mailing list[2]. The main technical challenge is that the TLS receiver needs to know how large a record to read from the stream and pass to the authenticated encryption cipher (AEAD), before it has actually decrypted that record.

The simplest currently apparent approach is to extend the one-byte encrypted content-type trailer, which TLS 1.3 already adds to the plaintext of each record before encryption, with an additional field optionally indicating the *next* record's length. If this **nextRecordLength** field is zero, the subsequent TLS record includes the usual 5-byte cleartext header; otherwise the subsequent TLS record is transmitted with *no* header and its length is specified by this **nextRecordLength** value.

This approach to record boundary hiding has the main advantage of requiring no special support from the AEAD cipher interface or other special cryptographic mechanisms. It also has the benefit of ensuring that all *nextRecordLength* fields are fully integrity-checked (as part of the previous record) before they are processed at all – in contrast with the length field in the cleartext TLS header, whose value must be used (at least to read the rest of the record) before it has been integrity-checked by the AEAD. As a result, protecting record lengths also fixes a minor known vulnerability in which a MITM or MOTS attacker can corrupt the length field to indicate that the last record in a communication will be large, potentially causing the receiver to hang for an extended period attempting to read a large record that will never arrive, causing an eventual timeout instead of promptly noticing a MAC check failure.

This approach to record boundary hiding has the main downside of requiring the sender to "commit" to the ciphertext length of the next record before finalizing the current record. When the sender is issuing a series of records as part of a large write (*e.g.*, a multi-record HTTP payload), determining an appropriate size for the next record is not typically a problem – except in the case of the last record in a write or transmission burst. In this case, the sender may not yet know the length of the next TLS record to be sent, and hence may simply have to "pick something." The sender might pick a smallest-possible next record length (*e.g.*, just one or even no "useful" payload bytes) as a starting point to "bootstrap" the next burst. Better, a sender that is also padding records to a fixed minimum length (*e.g.*, 256 bytes) can pick that as the default next record length.

Record boundary hiding can also complement record padding by making padding more bandwidth-efficient. In particular, to prevent record-granularity information leakage, a padding implementation of TLS without record boundary hiding must generally pad every record to either a fixed or a close to uniformly random size. With record boundary hiding, however, an attacker cannot cryptographically differentiate between N consecutive P-byte records and a single $(N \times P)$-byte record: thus, the sender has the opportunity to send larger records in multiples of the standard padded record size, to save both per-record bandwidth overhead and per-record processing costs when handling bulk data. If $P = 256$ bytes, for example, per-record overhead for AES-GCM is over 10%, but drops to a negligible 0.1% in a 16,384-byte packet (64×256).

[2] `https://www.ietf.org/mail-archive/web/tls/current/maillist.html`

5.2 Encryption of Handshaking Metadata

While we discussed above the hiding of TLS record boundaries, record headers are not the only part of the protocol that TLS traditionally leaves unencrypted and vulnerable to traffic analysis. In particular, initial ciphersuite negotiation, handshaking, and key exchange are normally also performed unencrypted. Thus, an adversary who can monitor a connection "from the start" has an opportunity to learn a lot about the connection and its endpoints before the first encrypted data even begins flowing. The fact that negotiations are performed in the clear and not (yet) cryptographically integrity-checked has also resulted in downgrade attack vulnerabilities, for which TLS 1.3 includes important but unfortunately somewhat ad hoc workarounds.

One important piece of information that this unencrypted handshaking process leaks is the Server Name Indication (SNI), which an eavesdropper can often easily use to determine which specific Web site a client is connecting to among a possibly large number of virtual web sites hosted behind a single large Web server front-end. Hence, encrypting the SNI would be quite valuable from a perspective of defense against traffic analysis.

However, from a longer-term perspective, SNI is just one of many parts of the handshake process that could benefit from being encrypted "from the start." For example, the unencrypted handshake informs an attacker exactly what ciphersuite was negotiated: therefore, if some cipher or MAC has become known to have a weakness the attacker knows exactly which TLS streams to invest in attempting to crack and which to ignore (or save for the future). In principle, if TLS could negotiate the ciphersuite *under encryption*, an adversary attempting to decide what to do with a TLS stream might have to invest significantly greater computational resources attempting to attack a much larger number of opaque TLS streams, even to determine whether the stream is encrypted using the known-weak algorithm. However, negotiating ciphersuites under encryption presents a fairly obvious "chicken-and-egg" problem; this challenge can be surmounted in some contexts, but perhaps not realistically in a timeframe suitable for TLS 1.3.

One special-case scenario where "encrypted negotiation" seems potentially more realistic in the near term is in the new 0-RTT handshake mechanism. This mechanism, already introduced in TLS 1.3, allows a client who has previously talked with a server, and already has some information cached about that server, to resume communication with no extra round-trips. This mechanism fundamentally depends on the client's prior knowledge – or at least "educated guess" – at the ciphersuite(s) and keys supported by the server. However, that same knowledge of the server could in principle allow the client to initiate encrypted communication with the server starting from the very first byte of communication, *i.e.*, leaving *no* metadata unencrypted even at the start of the connection. Negotiating a new session with no round-trip latencies and fully under encryption is likely to require uniform-encoding techniques as in Elligator [3], which add significant complexity but are known to be feasible.

More generally, encrypted-from-the-start negotiation could be extended even to the case where a client has not previously communicated with the server, if the client can obtain cryptographic "bootstrap" information in some other way such as via DANE [10], or using a similar directory service as in MinimaLT [15]. However, such goals, however worthwhile, seem almost certainly too ambitious for TLS 1.3 and more suitable to consider for a future "TLS 2.0."

References

[1] SPDY: An experimental protocol for a faster Web. `http://www.chromium.org/spdy/spdy-whitepaper`.

[2] M. Belshe, R. Peon, and M. Thomson, Ed. Hypertext transfer protocol version 2 (HTTP/2), May 2015. RFC 7540.

[3] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange. Elligator: Elliptic-curve points indistinguishable from uniform random strings. In *ACM CCS*, Nov. 2013.

[4] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *ACM CCS*, Oct. 2012.

[5] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2, Aug. 2008. RFC 5246.

[6] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *IEEE Symposium on Security and Privacy*, May 2012.

[7] R. Fielding et al. Hypertext transfer protocol – HTTP/1.1, June 1999. RFC 2616.

[8] D. Goodin. Crack in Internets foundation of trust allows HTTPS session hijacking. *ars technica*, Sept. 2012.

[9] D. Herrmann, R. Wendolsky, and H. Federrath. Website fingerprinting: Attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *CCSW*, Nov. 2009.

[10] P. Hoffman and J. Schlyter. The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA, Aug. 2012. RFC 6698.

[11] B. Miller, L. H. andA.D.Joseph, and J. Tygar. I know why you went to the clinic: Risks and realization of HTTPS traffic analysis. In *PETS*, July 2014.

[12] P. Mittal, A. Khurshid, J. Juen, M. Caesar, and N. Borisov. Stealthy traffic analysis of low-latency anonymous communication using throughput fingerprinting. In *18th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2011.

[13] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website fingerprinting in onion routing based anonymization networks. In *Workshop on Privacy in the Electronic Society (WPES)*, Oct. 2011.

[14] J. Pang, B. Greenstein, R. Gummadi, S. Seshan, and D. Wetherall. 802.11 user fingerprinting. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, pages 99–110, 2007.

[15] W. M. Petullo, X. Zhang, J. A. Solworth, D. J. Bernstein, and T. Lange. MinimaLT: Minimal-latency Networking Through Better Security. In *ACM CCS*, Nov. 2013.

[16] V. Srinivasan, J. Stankovic, and K. Whitehouse. Protecting your daily in-home activity information from a wireless snooping attack. In *UbiComp*, Sept. 2008.

[17] Tor: Anonymity Online. `https://www.torproject.org`.

[18] T. Valverde. Bad life advice – Replay attacks against HTTPS, Dec. 2015. Blog.

[19] N. V. Verde, G. Ateniese, E. Gabrielli, L. V. Mancini, and A. Spognardi. "no nat'd user left behind": Fingerprinting users behind NAT from NetFlow records alone. In *34th International Conference on Distributed Computing Systems (ICDCS)*, July 2014.

[20] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective attacks and provable defenses for website fingerprinting. In *USENIX Security Symposium*, Aug. 2014.

[21] T. Ylonen and C. Lonvick, Ed. The secure shell (SSH) transport layer protocol, Jan. 2006. RFC 4253.

[22] F. Zhang, W. He, X. Liu, and P. G. Bridges. Inferring users' online activities through traffic analysis. In *WiSec*, June 2011.