Lucky Thirteen: Breaking the TLS and DTLS Record Protocols

Nadhem J. AlFardan and Kenneth G. Paterson*
Information Security Group
Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK
{nadhem.alfardan.2009, kenny.paterson}@rhul.ac.uk

27th February 2013

Abstract

The Transport Layer Security (TLS) protocol aims to provide confidentiality and integrity of data in transit across untrusted networks. TLS has become the de facto *secure protocol of choice for Internet and mobile applications. DTLS is a variant of TLS that is growing in importance. In this paper, we present distinguishing and plaintext recovery attacks against TLS and DTLS. The attacks are based on a delicate timing analysis of decryption processing in the two protocols. We include experimental results demonstrating the feasibility of the attacks in realistic network environments for several different implementations of TLS and DTLS, including the leading OpenSSL implementations. We provide countermeasures for the attacks. Finally, we discuss the wider implications of our attacks for the cryptographic design used by TLS and DTLS.*

Keywords TLS, DTLS, CBC-mode encryption, timing attack, plaintext recovery

1 Introduction

TLS is arguably the most widely-used secure communications protocol on the Internet today. Starting life as SSL, the protocol was adopted by the IETF and specified as TLS 1.0 [10]. It has since evolved through TLS 1.1 [11] to the current version TLS 1.2 [12]. Various other RFCs define additional TLS cryptographic algorithms and extensions. TLS is now used for securing a wide variety of application-level traffic and has become a serious rival to IPsec for general VPN usage. It is widely supported in client and server software and in cryptographic libraries for embedded systems, mobile devices, and web application frameworks. Open-source implementations of TLS and DTLS include OpenSSL, GnuTLS, PolarSSL and CyaSSL.

The DTLS protocol is a close relative of TLS, developed from TLS by making minimal changes so as to allow it to operate over UDP instead of TCP [25]. This makes DTLS suitable for use where the costs of TCP connection establishment and TCP retransmissions are not warranted, for example, in voice and gaming applications. DTLS exists in two versions, DTLS 1.0 [31], which roughly matches TLS 1.1 and DTLS 1.2 [32], which aligns with TLS 1.2.

Both TLS and DTLS are actually protocol suites, rather than single protocols. The main component of (D)TLS that concerns us here is the Record Protocol, which uses symmetric key cryptography (block ciphers, stream ciphers and MAC algorithms) in combination with sequence numbers to build a secure channel for transporting application-layer data. Other major components are the (D)TLS Handshake Protocol, which is responsible for authentication, session key establishment and ciphersuite negotiation, and the TLS Alert Protocol, which carries error messages and management traffic. Setting aside dedicated authenticated encryption algorithms (which are yet to see widespread support in TLS or DTLS implementations), the (D)TLS Record Protocol uses a MAC-Encode-Encrypt (MEE) construction. Here, the plaintext data to be transported is first passed through a MAC algorithm (along with certain header bytes) to create a MAC tag. The supported MAC algorithms are all HMAC-based, with MD5, SHA-1 and SHA-256 being the allowed hash algorithms in TLS 1.2 [12]. Then an encoding step takes place. For the RC4 stream cipher, this just involves concatenation of the plaintext and the MAC tag, while for CBC-mode encryption (the other possible option), the plaintext, MAC tag, and some encryption padding of a specified format are concatenated. In the encryption step, the encoded plaintext is encrypted with the selected cipher. In the case where CBC-mode is selected, the block cipher is DES, 3DES or AES (with DES being deprecated in TLS 1.2). Following [28], we refer to this MEE construction as MEE-TLS-CBC. We provide greater detail on its operation in the (D)TLS Record Protocol in Section 2.

The widespread use of TLS (and the increasing use of DTLS) makes the continued study of the security of these protocols of great importance. Indeed, the evolution of the TLS Record Protocol has largely been driven by cryptographic attacks that have been discovered against it, including those in [37, 6, 26, 2, 3, 13, 28, 1].

Of particular interest lately have been attacks based on the use of chained initialisation vectors (IVs) for CBC-mode in SSL and TLS 1.0, in particular, the so-called BEAST attack [13] which has its roots in [35, 26, 2, 3]. This attack achieved full plaintext recovery against TLS, but only in scenarios where an attacker can gain access to a chosen plaintext capability, per-

*This author's research supported by an EPSRC Leadership Fellowship, EP/H005455/1.

haps by inducing the user to first download malicious javascript code into his browser. Despite this strong requirement, the BEAST attack attracted significant industry and media attention in 2011. Amongst the possible countermeasures are upgrading to TLS 1.1 or 1.2, the inclusion of a dummy zero-length message prior to each real TLS message, or the abandonment of CBC-mode encryption in favour of RC4 or an authenticated encryption algorithm.

The other major line of attacks against the TLS Record Protocol comprises [37, 6, 26, 28, 1] and relates to how the padding that is required in MEE-TLS-CBC is handled during decryption. The problems here all stem from the fact that the padding is added *after* the MAC has been computed and so forms unauthenticated data in the encoded plaintext. Taken altogether, the attacks in [37, 6, 26, 28, 1] show that handling padding arising during decryption processing is a delicate and complex issue for MEE-TLS-CBC.

It is the case that all these attacks on CBC-mode in TLS could be avoided by adopting RC4 or a dedicated authenticated encryption mode, or perhaps by redesigning (D)TLS to use only an Encrypt-then-MAC construction. However, RC4 is not an option for DTLS, and not NIST-recommended for TLS [7]; meanwhile authenticated encryption modes are only available in TLS 1.2, which is not yet widely supported.[1] Redesigning (D)TLS would require even more radical changes than adopting TLS 1.2. So it would be fanciful to "wish away" MEE-TLS-CBC, and all the complexity that this entails: this is an option that is firmly embedded in the TLS and DTLS RFCs, in widespread use, and will remain so for the foreseeable future. On the other hand, we might hope that after more than a decade of intensive study, we would have arrived at a point where we understand how to implement MEE-TLS-CBC securely. In this paper, we show that this is not the case.

1.1 Our Results

We present a family of attacks that apply to CBC-mode in all TLS and DTLS implementations that are compliant with TLS 1.1 or 1.2, or with DTLS 1.0 or 1.2. They also apply to implementations of SSL 3.0 and TLS 1.0 that incorporate padding oracle attack countermeasures (implementations that do not are of course already vulnerable to known attacks).

The attacks come in various distinguishing, partial plaintext recovery, and full plaintext recovery flavours. For the plaintext recovery attacks, no chosen-plaintext capability is needed, in contrast to the BEAST attacks: the attacks can be mounted by a standard man-in-the-middle (MITM) attacker who sees only ciphertext and can inject ciphertexts of his own composition into the network. The details of which specific attacks are possible depends on the exact size of MAC tags output by the MAC algorithm negotiated by the Handshake Protocol, and also on the fact that the exactly 13 bytes of header data are incorporated in the MAC calculation (hence our title).

The applicability of the attacks is also implementation-dependent, because of the manner in which different implementations interpret the RFCs. We have investigated several different open-source implementations of TLS and DTLS, and found all of them to be vulnerable to our new attacks or variants of them (or even old attacks in one case). We also found basic coding errors in the security-critical decryption function of one popular implementation, GnuTLS. In view of the amount of variation we have seen in open-source code and our success in devising variant attacks, we expect *all* implementations – whether open or closed – to be vulnerable to our attacks to some extent.

We have implemented a selection of the attacks in an experimental setting. As with earlier attacks, completely breaking TLS is challenging because the attacks create "broken" TLS records and so consume many TLS sessions. Nevertheless, our basic attack can extract full plaintext for the current OpenSSL implementation of TLS assuming the attacker is located, say, in the same LAN segment as the targeted TLS client or server, using roughly 2^{23} TLS sessions to reliably recover a block of plaintext in a multi-session attack scenario like that considered in [6]. Such a scenario is applicable when, for example, an application protocol performs automatic TLS reconnection and password retransmission. Given its complexity, this basic attack would seem to present only a theoretical threat. However, variants of it are much more effective:

- The distinguishing attacks against TLS are quite practical for OpenSSL, requiring just a handful of sessions in order to reliably tell apart the encryptions of chosen messages.
- Breaking DTLS implementations is fully practical even for a remote attacker, since we can exploit the fact that DTLS errors are non-fatal to mount the attacks in a single session, and reuse the amplification techniques from [1] to boost the delicate timing signals on which our attacks depend.
- We also have more efficient *partial* plaintext recovery attacks on TLS and DTLS. For example, against OpenSSL TLS, an attacker who knows one byte of a block in either of the last two byte positions can reliably recover each of the remaining bytes in that block using 2^{16} sessions.
- The complexity of all our attacks can be reduced using language models and sequential statistical techniques as in [6, 13]. As a simple example, if the plaintext is base64 encoded, as is the case for HTTP basic access authentication and cookies, then the number of TLS sessions needed to recover a block reduces from roughly 2^{23} to 2^{19}.
- In the web setting, our techniques can be combined with those used in the BEAST attack [13]: client-side malware running in the browser can be used to initiate all the needed TLS sessions, with an HTTP cookie being automatically injected by the browser in a predictable location in the plaintext stream in each session. The malware can also control the location of the cookie such that there is only one unknown byte in the target block at each stage of the attack. The attacker then combines the "one known byte" variant of our attack and the base64 optimisation above (assuming the sensitive part of the cookie is base64 encoded). Putting all of these improvements together, we estimate that HTTP cookies can be recovered using 2^{13}

[1] SSL Pulse (`https://www.trustworthyinternet.org/ssl-pulse/`) reported that only 11.4% of 200,000 websites surveyed support TLS 1.2 in January 2013; most major browsers currently do not support TLS 1.2.

sessions per byte of cookie (with all the sessions being automatically generated). Note that the malware does not need the ability to inject chosen plaintext into an existing TLS session for this attack.

How the attacks work: Our new attacks exploit the fact that, when badly formatted padding is encountered during decryption, a MAC check must still be performed on *some* data to prevent the known timing attacks. But what data should be used for that calculation? The TLS 1.1 and 1.2 RFCs recommend checking the MAC as if there was a zero-length pad. As noted in those RFCs:

> *This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.*

We confirm that there are indeed small timing differences, but, contrary to what is written in the RFCs, they *can* be exploited. In short, provided there is a fortuitous alignment of various factors such as the size of MAC tags, the block cipher's block size, and the number of header bytes, then there will be a time difference in the time that it takes to process TLS records having good and bad padding, and this difference will show up in the time at which error messages appear on the network. This timing side-channel can then be "wrangled" into revealing plaintext data via careful statistical analysis of multiple timing samples. As we shall show, other natural methods for handling MAC checking in the event of bad padding also lead to exploitable timing differences.

It is not clear to us whether the attacks we present here were already known to the TLS community. We suspect not, in view of the attacks' complexity and the state-of-the-art in attacks at the time of writing of the TLS 1.1 RFC. However, this question seems moot in view of the fact that attacks exist for RFC-compliant implementations and present a threat to the security of TLS and DTLS.

Our new attacks demonstrate that properly implementing MEE-TLS-CBC so as to avoid all exploitable timing differences is in fact quite difficult, and is not achieved by any of the implementations we examined. A complicating factor, in addition to dealing with padding, is the need for careful sanity checking of various fields during decryption. We provide a detailed prescription for dealing with these issues. We also discuss other, more easily-implemented countermeasures.

1.2 Disclosure (as at 27/02/2013)

Given the large number of affected implementations, we first notified the IETF TLS Working Group chairs, the IETF Security Area directors and the IRTF Crypto Forum Research Group (CFRG) chairs of our attacks in November 2012. We then began the process of contacting individual vendors:

OpenSSL addressed the attacks in versions 1.0.1d, 1.0.0k and 0.9.8y, released 05/02/2013. See `http://www.openssl.org/news/secadv_20130205.txt` for further details.

NSS addressed the attacks in version 3.14.3, released 15/02/2013. See `https://developer.mozilla.org/en-US/docs/NSS/NSS_3.14.3_release_notes` for further details.

Microsoft performed an investigation and determined that the issue had been adequately addressed in previous modifications to their TLS and DTLS implementations

Apple were notified of our attacks in December 2012. The status of patch development by Apple is currently unknown.

GnuTLS corrected the programming errors in decryption that we identified in version 3.1.6 (released 02/01/2013) and addressed the attacks in versions 2.12.23, 3.0.28 and 3.1.7, released 04/02/13.

PolarSSL addressed the attacks in version 1.2.5, released 03/02/13.

CyaSSL addressed the attacks in CyaSSL version 2.5.0, released 04/02/2013.

MatrixSSL addressed the attacks in version 3.4.1, released 06/02/13.

Opera addressed the attacks in Opera version 12.13, released 30/01/2013. For further details, see `www.opera.com/docs/changelogs/unified/1213/`.

F5 were notified of the attacks in December 2012. They have informed us that their TLS dataplane traffic is not vulnerable due to cryptographic offload, but that local management ports and virtual editions may be vulnerable. For further details, see `http://support.f5.com/kb/en-us/solutions/public/14000/100/sol14190.html`.

BouncyCastle addressed the attacks in version 1.48 of the Java library, released 10/02/2013. The C# version of BouncyCastle was fixed in CVS at a similar time, and will be included in release 1.8 at a later date.

Oracle (Java) addressed the attacks as part of a special critical patch update of JavaSE, released 19/02/2012.

In addition, a number of other companies and organisations were given advance notice of the attacks prior to them being made public.

We will continue to update this section as the disclosure process progresses.

1.3 Further Details on Related Work

TLS, and in particular the TLS Handshake Protocol, has been the subject of much analysis using a variety of security paradigms, see for example [29, 18, 27, 5]. In general, these analyses are at too high a level of abstraction to capture our attacks.

Padding oracle attacks began with Vaudenay [37], who showed that the presence of a *padding oracle*, that is, an oracle telling an attacker whether the padding was correctly formatted or not, could be leveraged to build a decryption capability. Canvel *et al.* [6] showed that such an oracle could be obtained for the then-current version of OpenSSL by exploiting a timing difference in TLS decryption processing. In essence, in OpenSSL, if the padding was incorrectly formatted, then no MAC check was performed, while if the padding was correct, then the MAC check was done. In turn, this meant faster production of an error message in the "invalid padding" case than in the "valid padding" case. Thus the padding oracle was revealed through a timing side-channel. A complication for full

plaintext recovery is that in TLS, the corresponding error messages are fatal, leading to the termination of the TLS session. To overcome this, Canvel *et al.* considered the multi-session setting, wherein it is assumed that the same plaintext is transmitted in the same position in the ciphertext in many sessions. In this setting, they were able to mount full plaintext recovery attacks, recovering, for example, TLS-protected Microsoft Outlook passwords in a LAN. Moeller [26] subsequently pointed out that *not* doing padding format checks is not an option, since this enables even simpler attacks. The correct solution, as advocated in TLS 1.1 and TLS 1.2, is to check the padding format carefully, report a single error message for padding and MAC failures, and to make the record processing time essentially the same whether or not the padding is correct. However, even this is not enough: Paterson *et al.* recently showed that distinguishing attacks would still be possible against TLS if the short MAC extension advocated in [15] were to be implemented in combination with the variable length padding specified in TLS 1.0 and up. Their attack, whilst not a padding oracle attack, does exploit the padding format. Most recently, in [1], we showed that the OpenSSL implementation of DTLS did not adopt the known attack countermeasures. We also introduced novel timing amplification techniques to build full plaintext recovery attacks against this implementation of DTLS, even though DTLS has no explicit error messages to time.

Theoretical support for the MEE construction used in (D)TLS can be found in [20, 22, 28]. In particular, Paterson *et al.* [28] gave the first positive security results for a fully accurate model MEE-TLS-CBC that includes all the details of the CBC-mode encoding step (which incorporates padding), proving that MEE-TLS-CBC provides Length Hiding Authenticated-Encryption security, provided that its MAC and CBC-mode block cipher components satisfy natural security properties, that the MAC tags are long enough, and that it is implemented so that decryption does not reveal the cause of any failures. The latter is modelled by having indistinguishable error messages in the security model. Our attacks exploit the fact that implementations of (D)TLS fail to meet this last assumption, and so the attacks do not contradict the result of [28], but instead relativize its applicability to practice.

Other recent work on the security of TLS implementations includes [16, 17, 30]. In particular, in independent work, Pironti *et al.* [30] identify effectively the same timing channel in TLS that we exploit. However they dismiss it as being “too small to be measured over the network” and instead focus on using it to recover information about message lengths. The recent CRIME attack exploits the optional use of compression in TLS in combination with a chosen plaintext capability to mount a plaintext recovery attack.

Other work showing that implementing MAC-then-encrypt securely can be difficult is given in [9] in the context of IPsec. That this is so, and that encrypt-then-MAC is the preferable construction, has been known in a theoretical sense since at least [4, 20]. Interesting padding oracle (and related) attacks abound in the literature, see for example [8, 34, 14, 19].

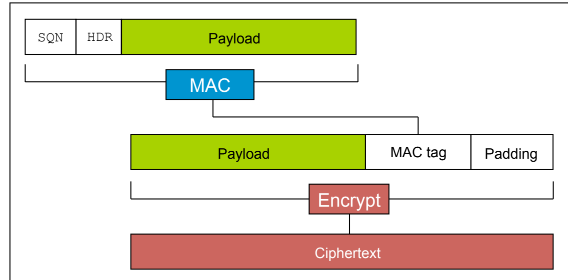

Figure 1: D(TLS) encryption process

1.4 Paper Organisation

Section 2 provides further background on the (D)TLS Record Protocol and the MEE-TLS-CBC construction. Section 3 presents the basic distinguishing attack against RFC-compliant implementations of TLS and DTLS, while Section 4 describes our plaintext recovery attacks in the context of TLS and explains how to modify them to apply to DTLS. In Section 5 we report on the experimental validation of our attacks for the OpenSSL implementation, and in Section 6 we describe the modifications needed to make our attacks applicable to other implementations, including GnuTLS, CyaSSL and PolarSSL. Section 7 discusses countermeasures to our attacks, giving guidance on how to implement MEE-TLS-CBC so as to avoid the attacks. Finally, Section 8 concludes with a recap of the main issues raised by our work.

2 The (D)TLS Record Protocol

We focus on the cryptographic operation of the TLS and DTLS Record Protocols in the case of CBC-mode encryption. The core encryption process is illustrated in Figure 1 and explained in more detail below.

Data to be protected by TLS or DTLS is received from the application and may be fragmented and compressed before further processing. An individual record R (viewed as a byte sequence of length at least zero) is then processed as follows. The sender maintains an 8-byte sequence number $\texttt{SQN}$ which is incremented for each record sent[2], and forms a 5-byte field $\texttt{HDR}$ consisting of a 2-byte version field, a 1-byte type field, and a 2-byte length field. It then calculates a MAC over the bytes $\texttt{SQN}||\texttt{HDR}||R$; let T denote the resulting MAC tag. Note that exactly 13 bytes of data are prepended to the record R here before the MAC is computed. The size of the MAC tag is 16 bytes (HMAC-MD5), 20 bytes (HMAC-SHA-1), or 32 bytes (HMAC-SHA-256). We let t denote this size in bytes.

The record is then encoded to create the plaintext P by setting $P = R||T||\texttt{pad}$. Here $\texttt{pad}$ is a sequence of padding bytes chosen such that the length of P in bytes is a multiple of b, where b is the block-size of the selected block cipher (so $b = 8$ for 3DES and $b = 16$ for AES). In all versions of TLS and DTLS, the padding must consist of $p + 1$ copies of some byte value p, where $0 \leq p \leq 255$. In particular, at

[2] In fact, in DTLS, this 8-byte field is composed from a 16-bit *epoch* number and a 48-bit sequence number. We will abuse terminology and refer throughout to the 8-byte field as being the sequence number for both TLS and DTLS.

least one byte of padding must always be added. So examples of valid byte sequences for pad are: “0x00”, “0x01||0x01” and “0x02||0x02||0x02”. The padding may extend over multiple blocks, and receivers must support the removal of such extended padding.

In the encryption step, the encoded record P is encrypted using CBC-mode of the selected block cipher. TLS 1.1 and 1.2 and both versions of DTLS mandate an explicit IV, which should be randomly generated. TLS 1.0 and SSL use a chained IV; our attacks work for either option. Thus, the ciphertext blocks are computed as:

$$C_j = E_{K_e}(P_j \oplus C_{j-1})$$

where P_i are the blocks of P, C_0 is the IV, and K_e is the key for the block cipher E. For TLS (and SSL), the data transmitted over the wire then has the form:

$$\texttt{HDR}||C$$

where C is the concatenation of the ciphertext blocks C_i (including or excluding the IV depending on the particular SSL or TLS version). Note that the sequence number is not transmitted as part of the message. In DTLS, the data transmitted over the wire is the same as in TLS, except that SQN is included as part of the record header and the CBC-mode IV is always explicit.

Simplistically, the decryption process reverses this sequence of steps: first the ciphertext is decrypted block by block to recover the plaintext blocks:

$$P_j = D_{K_e}(C_j) \oplus C_{j-1},$$

where D denotes the decryption algorithm of the block cipher. Then the padding is removed, and finally, the MAC is checked, using the header information (and, in TLS, a version of the sequence number that is maintained at the receiver). Finally, in DTLS, the sequence number is optionally checked for replays.

In reality, much more sophisticated processing than this is needed. The receiver should check that the ciphertext size is a multiple of the block size and is large enough to contain at least a zero-length record, a MAC tag of the required size, and at least one byte of padding. After decryption, the receiver should check that the format of the padding is one of the possible patterns when removing it, otherwise attacks are possible [26] (SSL allows a loose padding format, while no specific padding checks are enforced during decryption in TLS 1.0, so both are potentially vulnerable to the attacks in [26]). Typically this is done by examining the last byte of the plaintext, treating it as a padding length byte padlen, and using this to dictate how many additional bytes of padding should be removed. But care is needed here, since blindly removing bytes could result in an underflow condition: there needs to be sufficient bytes in the plaintext to remove a total of padlen+1 bytes and leave enough bytes for at least zero-length record and a MAC tag.

If all this succeeds, then the MAC can be recomputed and compared to the MAC tag in the plaintext. If the padding fails to be correctly formatted, then implementations should continue to perform a MAC check anyway, to avoid providing a timing side-channel of the type exploited in [6]. But since the padding format is incorrect in this case, it’s not immediately clear where the padding ends and the MAC tag is located: in effect, the plaintext is now unparseable. The solution recommended in TLS 1.1 and 1.2 (and by extension, also in DTLS 1.0 and 1.2) is to assume zero-length padding, interpret the last t bytes of the plaintext as a MAC tag, interpret the remainder as the record R and run MAC verification on $\texttt{SQN}||\texttt{HDR}||R$. This has been adopted in OpenSSL and elsewhere; GnuTLS on the other hand removes $\texttt{padlen} + 1$ bytes from the end of the plaintext, takes the next t bytes as the MAC, interprets what is left as R and then runs MAC verification on $\texttt{SQN}||\texttt{HDR}||R$.

For TLS, any error arising during decryption should be treated as fatal, meaning an encrypted error message is sent to the sender and the session terminated with all keys and other cryptographic material being disposed of. For DTLS, such errors may be rated non-fatal and the session would proceed to process the next ciphertext.

It should now be apparent that implementing the basic decryption processing of TLS and DTLS requires some care in implementation, with there being significant room for coding errors and inadequate parsing. Moreover, this should all be implemented in such a way that the processing time does not leak anything about the plaintext (including the padding bytes). As we shall see, this has proved to be a challenge for implementers: no implementation we examined gets it completely correct, and the advice from TLS 1.1 and 1.2 that one should extract and check the MAC tag as if the padding were of zero-length leaves an exploitable timing side-channel.

2.1 Details of HMAC

As mentioned above, TLS and DTLS exclusively use the HMAC algorithm [21], with HMAC-MD5, HMAC-SHA-1, and HMAC-SHA-256 being supported in TLS 1.2.[3] To compute the MAC tag T for a message M with key K_a, HMAC applies the specified hash algorithm H twice, in an iterated fashion:

$$T = H((K_a \oplus \texttt{opad})||H((K_a \oplus \texttt{ipad})||M)).$$

Here opad and ipad are specific 64-byte values, and the key K_a is zero-padded to bring it up to 64 bytes before the XOR operations are performed. For all the hash functions H used in TLS, the application of H itself uses an encoding step called *Merkle-Damgård strengthening*. Here, an 8-byte length field followed by padding of a specified byte format are appended to the message M to be hashed. The padding is at least 1 byte in length and aligns the data on a 64-byte boundary. The relevant hash functions also have an iterated structure, processing messages in chunks of 64 bytes (512 bits) using a compression function, with the output of each compression step being chained into the next step. The compression function in turn involves a complex round structure, with many basic arithmetic operations on data being involved in each round.

In combination, these features mean that HMAC implementations for MD5, SHA-1 and SHA-256 have a distinctive timing profile. Messages M of length up to 55 bytes can be encoded into a single 64-byte block, meaning that the first, inner hash

[3]TLS ciphersuites using HMAC with SHA-384 are specified in RFC 5289 (ECC cipher suites for SHA256/SHA384) and RFC 5487 (Pre-Shared Keys SHA384/AES) but we do not consider this algorithm further here.

operation in HMAC is done in 2 compression function evaluations, with 2 more being required for the outer hash operation, for a total of 4 compression function evaluations. Messages M containing from 56 up to $64 + 55 = 119$ bytes can be encoded in two 64-byte blocks, meaning that the inner hash is done in 3 compression function evaluations, with 2 more being required for the outer operation, for a total of 5. In general, an extra compression function evaluation is needed for each additional 64 bytes of message data, with the exact number needed being given by the formula $\lceil \frac{\ell - 55}{64} \rceil + 4$, where ℓ is the message length in bytes. A single compression function evaluation takes typically around 500 to 1000 hardware cycles (depending on the hash function and details of the implementation), giving a time in the sub-μs range for modern processors.

Recall that in TLS the MAC is computed on plaintext *after* removing padding. Hence, one might expect the total running time for decryption processing to reveal some information about the size of the depadded plaintext, perhaps up to a resolution of 64 bytes in view of the above discussion. Our distinguishing attack exploits this, but we will show that much more is possible.

3 A Distinguishing Attack

In this section we describe a simple distinguishing attack against the MEE-TLS-CBC construction as used in TLS. This is a warm-up to our plaintext recovery attacks, but we note that even a distinguishing attack against such an important protocol would usually be regarded as a significant weakness.

Recall that in a distinguishing attack, the attacker gets to choose pairs of messages (M_0, M_1). One of these is encrypted, M_d, say, and the resulting ciphertext is given to the attacker. The attacker's task is to decide the value of the bit d. To prevent the attacker from winning trivially, we require that M_0 and M_1 have the same length.

We focus on the case where $b = 16$, i.e. the block cipher is AES. A variant of the attack works for $b = 8$. Suppose the MAC algorithm is HMAC-H where H is either MD5, SHA-1 or SHA-256. Let M_0 consist of 32 arbitrary bytes followed by 256 copies of 0xFF. Let M_1 consist of 287 arbitrary bytes followed by 0x00. Note that both messages have 288 bytes, and hence fit exactly into 18 plaintext blocks. Our attacker submits the pair (M_0, M_1) for encryption and receives a MEE-TLS-CBC ciphertext $\mathtt{HDR}||C$. Now C consists of a CBC-mode encryption of an encoded version of M_d, where the encoding step adds a MAC tag T and some padding $\mathtt{pad}$. Because the end of M_d aligns with a block boundary, the additional bytes $T||\mathtt{pad}$ are encrypted in separate blocks from M_d. The attacker now forms a new ciphertext $\mathtt{HDR}||C'$ in which C' keeps the same 16-byte IV as C (if explicit IVs are being used), but truncates the non-IV part of C to 288 bytes. This has the effect of removing those blocks of C that contain $T||\mathtt{pad}$.

Now the attacker submits $\mathtt{HDR}||C'$ for decryption. If the record underlying C was M_0, then the plaintext P' corresponding to C' appears to end with the valid 256-byte padding pattern 0xFF...0xFF. In this case, all of these bytes are removed, and the remaining 32 bytes of plaintext are interpreted as a short message and a MAC tag. For example, if H is SHA-1, then we have a 12-byte message and a 20-byte MAC tag. The MAC verification fails (with overwhelming probability), and an error message is returned to the attacker. If the underlying record was M_1, then P' appears to end with the valid 1-byte padding pattern 0x00. In this case, a single byte is removed, and the remaining 287 bytes of plaintext are interpreted as a long message and a MAC tag. Again, the MAC verification fails and an error message is returned to the attacker.

Notice that when $d = 0$, so C encrypts M_0, a short message consisting of 13 bytes of header plus at most 16 bytes of message (when the hash algorithm is MD5) is passed through the MAC algorithm. To calculate the MAC requires 4 evaluations of H's compression function. On the other hand, when $d = 1$, C encrypts M_1, and a long message consisting of 13 bytes of header plus at least 255 bytes of message is passed through the MAC algorithm. Then to calculate the MAC requires at least 8 evaluations of H's compression function, at least 4 more than for the $d = 0$ case. Hence, we expect the time it takes to produce the error message on decryption failure to be somewhat larger if $d = 1$ than when $d = 0$, on the order of a couple of μs for a modern processor. This timing difference then allows, in theory, a distinguishing attack on the MEE-TLS-CBC construction used in TLS.

3.1 Practical Considerations

In describing the attack, we have ignored the time taken to remove padding. This is different for the two messages being processed, and the difference is opposite to that for MAC checking in that padding removal for M_0 takes longer than for M_1. Similarly, we have ignored any other timing differences that might arise during other processing steps. In practice, as we will see in Section 5, these differences turn out to be smaller than the MAC timing difference.

The attack exploits the requirement from the (D)TLS RFCs that implementations be able to properly decrypt records having variable length padding, but does not require implementations to actually send records containing such padding. A variant attack is possible in case only minimum-length padding is supported, but involves a smaller timing signal.

In TLS, the error messages are sent over the network, and so can easily be detected by the attacker. However, these messages are subject to network jitter, and this may be large enough to swamp the timing difference arising from the 4 extra compression function evaluations. On the other hand, the timing signal may be quite large when the cryptographic processing is performed in a constrained environment, e.g. on an 8-bit or 16-bit processor, or even on a smartphone. Furthermore, the jitter may be significantly reduced when the adversary runs as a separate process on the machine performing TLS decryption. This may be possible in virtualised environments, e.g. in a cloud scenario as explored in [33]. The attack also destroys the TLS session, since in TLS such errors are fatal. The attack can be iterated across L sessions, with M_d being encrypted in each session, and statistical processing used to extract the timing signal.

In DTLS, there are no error messages, but the techniques of [1] can be applied to solve this problem. There, the authors send a packet containing a ciphertext C closely followed by a DTLS message, with the latter always provoking a response

message. Any timing difference arising from the decryption of C then shows up as a difference in the arrival time of the response messages. The signal amplification techniques from [1] can also be used to boost the timing difference – here, the idea is to send multiple packets all containing C in quick succession, to create a cumulative timing difference (since each time C is processed, it will be processed in the same way).

In the attack as described, we have used 288 byte messages. This ensured that there were sufficient bytes left after the removal of padding to leave room for a message (possibly of zero length) and a MAC tag. This ensures that C' passes any sanity checks that might be applied during decryption. However, these sanity checks might be exploitable in variants of our basic attack. For example, an implementation that finds it does not have enough bytes left to contain a MAC after depadding may choose to skip MAC verification altogether, leading to an increased timing difference.

Note that the attack would still work as described if the truncated MACs specifed for TLS in [15] were used, since the full HMAC-H computation is still performed but only certain bytes of the computed tag are compared to bytes of the plaintext.

We report on the successful implementation of this attack in Section 5.

4 Plaintext Recovery Attacks

4.1 General Approach

As we have seen in the previous section, the processing time for a (D)TLS record (and therefore the appearance time of error messages) will depend on the amount of padding that the receiver interprets the encoded plaintext as containing. However, by placing a target ciphertext block at the end of the encrypted record, an attacker can arrange that the plaintext block corresponding to this block is interpreted as padding, and hence make the processing time depend on plaintext bytes. But, it seems that large amounts of valid padding are needed to create a significant timing difference, and this is difficult to arrange in a plaintext recovery attack. We show that this barrier to plaintext recovery can be overcome under certain circumstances.

Let C^* be any ciphertext block whose corresponding plaintext P^* the attacker wishes to recover. Let C' denote the ciphertext block preceding C^*. Note that C' may be the IV or the last block of the preceding ciphertext if C^* is the first block of a ciphertext. We have:

$$P^* = D_{K_e}(C^*) \oplus C'.$$

For any block B of plaintext or ciphertext, we write $B = [B_0 B_1 \dots B_{b-1}]$, where B_i denote the bytes of B. In particular, we have $P^* = [P^*_0 P^*_1 \dots P^*_{b-1}]$.

As usual, we assume that the attacker is capable of eavesdropping on the (D)TLS-protected communications and of injecting messages of his choice into the network. For TLS, or DTLS with sequence number checking disabled, we do not need the ability to prevent messages from reaching their destination. Nor do we require a chosen-plaintext capability.

4.2 Full Plaintext Recovery

For simplicity of presentation, in what follows, we assume the CBC-mode IVs are explicit (as in TLS 1.1, 1.2 and DTLS 1.0, 1.2). We also assume that $b = 16$ (so our block cipher is AES). It is easy to construct variants of our attacks for implicit IVs and for $b = 8$. We begin by considering only TLS, with details for DTLS to follow. We also assume that the TLS implementation follows the advice in the TLS 1.1 and 1.2 RFCs about checking the MAC as if there was a zero-length pad when the padding is incorrectly formatted. We will examine the security of other implementation options in Section 6. Most importantly, and for reasons that will become clear, we assume for the moment that $t = 20$ (so that the MAC algorithm is HMAC-SHA-1). We consider $t = 16$ and $t = 32$ (HMAC-MD5 and HMAC-SHA-256) shortly.

Let Δ be a block of 16 bytes and consider the decryption of a ciphertext $C^{\text{att}}(\Delta)$ of the form

$$C^{\text{att}}(\Delta) = \texttt{HDR}||C_0||C_1||C_2||C' \oplus \Delta||C^*$$

in which there are 4 non-IV ciphertext blocks, the penultimate block $C' \oplus \Delta$ is an XOR-masked version of C' and the last block is C^*. The corresponding 64-byte plaintext is $P = P_1||P_2||P_3||P_4$ in which

$$\begin{aligned} P_4 &= D_{K_e}(C^*) \oplus (C' \oplus \Delta) \\ &= P^* \oplus \Delta. \end{aligned}$$

Notice that P_4 is closely related to the unknown, target plaintext block P^*. We consider 3 distinct cases, which between them cover all possibilities for what can happen during decryption of $C^{\text{att}}(\Delta)$:

1. P_4 ends with a 0x00 byte: in this case, a single byte of padding is removed, the next 20 bytes are interpreted as a MAC tag T, and the remaining $64 - 21 = 43$ bytes of plaintext are taken as the record R. MAC verification is then performed on a $13 + 43 = 56$-byte message $\texttt{SQN}||\texttt{HDR}||R$.

2. P_4 ends with a valid padding pattern of length at least 2 bytes: in this case, at least 2 bytes of padding are removed, and the next 20 bytes are interpreted as a MAC tag T. This leaves a record R of length at most 42 bytes, meaning that MAC verification is then performed on a message of length at most 55 bytes.

3. P_4 ends with any other byte pattern: in this case, the byte pattern does not correspond to valid padding. Following the prescription in the TLS 1.1 and 1.2 RFCs, the plaintext is treated as if it contains no bytes of padding, so the last 20 bytes are interpreted as a MAC tag T, and the remaining 44 bytes of plaintext are taken as the record R. MAC verification is then performed on a 57-byte message.

In all cases, the MAC verification will fail (with overwhelming probability) and an error message produced. Notice that, in accordance with the discussion in Section 2.1, in Cases 1 and 3, the MAC verification will involve 5 evaluations of the compression function for SHA-1, while Case 2 only requires 4

evaluations. Therefore, we can hope to distinguish Case 2 from Cases 1 and 3 by timing the appearance of the error message on the network. Here the timing difference is that needed for a single SHA-1 compression function evaluation (compared to 4 such evaluations in our distinguishing attack). Notice that the size of the header, 13 bytes, in conjunction with the MAC tag size, 20 bytes, are critical in generating this distinctive timing behaviour.

In Case 2, assuming that the plaintext has no special structure, the most likely padding pattern to arise is the one of length 2, namely 0x01||0x01, with all longer padding patterns being roughly 256 times less likely. Thus, if the attacker selects a mask Δ in such a way that he detects Case 2 after submitting $C^{\text{att}}(\Delta)$ for decryption, then he can infer that P_4 ends with 0x01||0x01, and, using the equation $P_4 = P^* \oplus \Delta$, can now recover the last 2 bytes of P^*. (In fact, by repeating the attack with a mask Δ' that is modified from Δ in the third-to-last byte, the attacker can easily separate the case of a length 2 padding pattern from all longer patterns.)

The question remains: how does the attacker trigger Case 2, so that he can extract the last 2 bytes of P^*? Recall that the attacker has the freedom to select Δ. By injecting a sequence of ciphertexts $C^{\text{att}}(\Delta)$ with values of Δ that vary over all possible values in the last 2 bytes Δ_{14}, Δ_{15}, then (in the worst case) after 2^{16} trials, the attacker will surely select a value for Δ such that $C^{\text{att}}(\Delta)$ triggers Case 2.

Once the last 2 bytes of P^* have been extracted, the attacker can more efficiently recover the remaining bytes of P^*, working from right to left. This phase is essentially identical to Vaudenay's original padding oracle attack [37]. For example, to extract the third-to-last byte, the attacker can use his new knowledge of the last two bytes of P^* to now set Δ_{14}, Δ_{15} so that P_4 ends with 0x02||0x02. Then he generates candidates $C^{\text{att}}(\Delta)$ as before, but modifying Δ_{13} only. After at most 2^8 trials, he will produce a ciphertext which falls into case 2 again, which reveals he has managed to set a value 0x02 in the third-to-last byte of $P_4 = P^* \oplus \Delta$. From this, he can recover P^*_{13}. Recovery of each subsequent byte in P^* requires at most 2^8 trials, giving a total of $14 \cdot 2^8$ trials to complete the extraction of P^*.

Practical considerations: In practice, for TLS, there are two severe complications. Firstly, the TLS session is destroyed as soon as the attacker submits his very first attack ciphertext. Secondly, the timing difference between the cases is very small, and so likely to be hidden by network jitter and other sources of timing difference.

The first problem can be overcome for TLS by mounting a multi-session attack, wherein we suppose that the same plaintext is repeated in the same position over many sessions (as in [6], for example). We have used masks Δ in such a way that no further modification to the attack is needed to cater for this setting – of course blocks C' and C^* change for each session.

The second problem can be overcome in the same multi-session setting by iterating the attack many times for each Δ value and then performing statistical processing of the recorded times to estimate which value of Δ is most likely to correspond to Case 2. In practice, we have found that a basic percentile test (and even averaging) works well – see Section 5 for further details. Assuming that L trials are used for each Δ value, the attack as described consumes roughly $L \cdot 2^{16}$ sessions, with one ciphertext $C^{\text{att}}(\Delta)$ being tried in each session.

More efficient variants: The attack complexity can be significantly reduced by assuming that the language from which plaintexts are drawn can be modelled using a finite-length Markov chain. This is a fair assumption for natural languages, as well as application-layer protocol messages such as HTML, XML etc. This model can be used to drive the selection of candidate plaintext bytes in order of decreasing likelihood, and from this, determine the bytes of Δ needed to test whether a guess for the plaintext bytes leads to valid padding or not. Similar techniques were used in [6, 13] in combination with sequential statistical techniques to reduce the complexity of recovering low-entropy plaintexts. Note that this approach does not work well if TLS's optional compression is used. Another possibility is that the plaintext bytes are drawn from a reduced space of possibilities. For example, in HTTP basic access authentication, the username and password are Base64 encoded, meaning that each byte of plaintext has only 64 possible values. Similar restrictions often apply to the sensitive parts of HTTP cookies.

In a related attack scenario, if the attacker already knows one of the last two bytes of P^*, he can recover the other byte with much lower complexity than our analysis so far would suggest. This is then a plaintext recovery attack with partially-known-plaintext. For example, suppose the attacker knows the value of the byte P^*_{14}. Then he sets the starting value of Δ such that $\Delta_{14} = P^*_{14} \oplus$ 0x01, so that when $C^{\text{att}}(\Delta)$ is decrypted, the second-to-last byte of P_4 already equals 0x01. Then he iterates over the 2^8 possible values for Δ_{15}, eventually finding one such that P_4 has its last two bytes equal to 0x01||0x01, triggering Case 2. He can then proceed to recover the rest of P^* with the same complexity as before. Overall, this attack, which recovers 15 bytes of plaintext with 1-out-of-2 of the last bytes of the target block known, consumes only $15L \cdot 2^8$ sessions, where L is the number of trials used for each Δ value in each byte position. This can be further reduced by combining the two variants. For example, for base64 encoded plaintext, only $15L \cdot 2^6$ sessions are needed to decrypt a block.

Combining Lucky 13 with the BEAST: A significant limitation of our attacks as described so far is their consumption of many TLS sessions. This limitation can be overcome by combining our attacks with techniques from the BEAST attack [13] to target TLS-protected HTTP cookies.

Specifically, in the context of a web browser communicating with a web server over TLS, the user can be induced into downloading malware into his browser from a rogue website. This malware, perhaps implemented in Javascript, can then initiate all the TLS sessions need for our attack, with the browser automatically appending the targetted HTTP cookie to the browser's initial HTTP request. Furthermore, by adjusting the length of that initial HTTP request, the malware can ensure that there is only one unknown byte of HTTP cookie plaintext in each target ciphertext block. This allows our remote attacker to carry out the variant attack described immediately above. Assuming the targeted part of the cookie is base64 encoded, the attack consumes $L \cdot 2^6$ sessions per byte of HTTP cookie. As we

will discuss in more detail in Section 5, we found that setting $L = 2^7$ yields reliable plaintext recovery in our experimental set-up, giving us an attack that recovers HTTP cookies using roughly 2^{13} sessions per unknown byte of cookie.

4.3 Plaintext Recovery for Other MAC Algorithms

A critical feature of our attack above is the relationship between the size of the header included in the MAC calculation (fixed at $h = 13$ bytes), the MAC tag size t, and the block size b. For example, if TLS happened to be designed such that $h = 12$, then, with $t = 20$ and $b = 16$, a similar case analysis as before shows that our ciphertext $C^{\text{att}}(\Delta)$ would have the property of having faster MAC verification if P_4 also ends with the single byte 0x00 (the valid padding pattern of length 1). This would allow an improved 2^8 attack against TLS with CBC-mode and HMAC-SHA-1. In some sense, 13 *is* lucky, but 12 would have been luckier!

Similarly, we have (less efficient) variants of our attacks for HMAC-MD5 and HMAC-SHA-256, where the tag sizes t are 16 and 32 bytes, respectively. In fact, because here t is a multiple of b, the analysis is largely the same in both cases, and we consider only HMAC-MD5 in detail. This time $C^{\text{att}}(\Delta)$ is such that we fall into Case 2 (valid padding with a message of size at most 55 bytes, giving fast MAC verification) only if $P_4 = P^* \oplus \Delta$ ends with a valid padding of length 6 or more. With no additional information on P^* the attacker would need (worst case) 2^{48} attempts to construct the correct Δ so as to trigger this case; detecting that he had done so would be more difficult in view of the large number of candidate Δ values. This is not an attractive attack, especially in view of the practical considerations for TLS mentioned above. On the other hand, we do have attractive partially-known-plaintext attacks for HMAC-MD5 and HMAC-SHA-256. For example, if any 5 out of the last 6 bytes of P^* are known, we can recover the remaining 11 bytes using $11L \cdot 2^8$ sessions. The attack can also be made more efficient if the plaintext has low entropy, by trying candidates for the last 6 bytes of P^* in order of decreasing probability and then recovering the remaining bytes of P^* once the right 6-byte candidate is found. This would be an good option for password recovery, for example.

A similar analysis can be carried out for truncated MAC algorithms, as per [15]. For example, for an 80-bit (10-byte) MAC tag, if any 11 out of the last 12 bytes of P^* are known, we can recover the remaining 5 bytes using $5L \cdot 2^8$ sessions.

Finally, we note that the "Lucky 13 + BEAST" attacks work equally well, no matter what the MAC tag size is.

4.4 Applying the Attacks to DTLS

So far we have focussed on TLS. The changes needed to handle DTLS are the same as for our distinguishing attack in Section 3: we can use the techniques of [1] to amplify the timing differences and to emulate TLS's error messages. The amplification capability reduces the attack complexity dramatically: essentially, we can accurately test each Δ value using just a few packet trains instead of requiring L trials.

There is one further critical difference that we wish to emphasise: as already noted, DTLS does not treat errors arising during decryption as being fatal. This means that the entire attack against DTLS can be carried out in a *single* session, that is, without requiring the same plaintext to be repeated in the same position in the plaintext across multiple sessions, and without waiting for the Handshake Protocol to rerun for each session.

These differences brings our attack well within the bounds of practicality for DTLS. This is particularly so if DTLS's optional checking of sequence numbers is disabled. Even if this is not the case, the attacks are quite feasible in practice, provided enough DTLS messages are available, or if the upper layer protocol being protected by DTLS produces replies to sent messages in a consistent manner. These issues are discussed at greater length in [1] and the next section, where we report on the successful implementation of our attacks for the OpenSSL implementation of TLS and DTLS.

5 Experimental Results for OpenSSL

5.1 Experimental Setup

We ran version 1.0.1 of OpenSSL on the client and the server. In our laboratory set-up, a client, the attacker and the targeted server are all connected to the same VLAN on a 100Mbps Ethernet switch. The targeted server was running on a single core processor machine operating at 1.87 GHz with 1 GByte of RAM, while the attacker was running on a dual core processor machine operating at 3.4 GHz, with 2 GByte of RAM.

To simulate the (D)TLS client, we made use of `s_client`, a generic tool that is available as part of the OpenSSL distribution package. We modified `s_client`'s source code to satisfy our testing requirements. We also developed a basic Python script that calls `s_client` when needed. Our attack code is written in C and is capable of capturing, manipulating and injecting packets of choice into the network.

In the case of TLS, the attacker captures the "targeted" packet, manipulates it and then sends the crafted version to the targeted server causing the TLS session to terminate. This crafted packet forces the client and the targeted server to lose TCP synchronization, causing delay in the TCP connection shutdown. To speed up the TCP connection tear down, the attacker sends spoofed RST packets to the client and the targeted system upon detecting the TLS encrypted alert message, forcing both systems to independently destroy the underlying TCP structure associated with the terminated TLS session.

All the timing values presented in the paper are based on hardware cycles, which are specific to processor speed. For example, 187 hardware cycles on our targeted server operating at speed of 1.87 GHz translate to an absolute timing of 100 ns. To count the hardware cycles, we made use of an existing C library licensed under GNU GPL v3[4].

5.2 Statistical Analysis

The network timings we collect in each experiment are from skewed distribution(s) with long tails and many outliers. However, we found that using basic statistical techniques (medians

[4] `code.google.com/p/fau-timer`

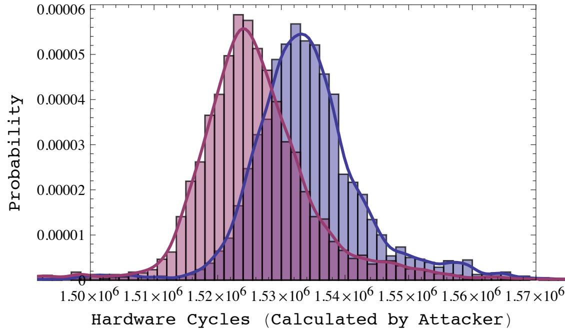

Figure 2: Distribution of timing values (outliers removed) for distinguishing attack on OpenSSL TLS, showing faster processing time in the case of M_0 (in red) compared to M_1 (in blue).

L	Success Probability
1	0.756
2	0.769
4	0.858
8	0.914
16	0.951
32	0.983
64	0.992
128	1

Table 1: OpenSSL TLS distinguishing attack success probabilities.

and, more generally, percentiles) was sufficient to analyse our data.

5.3 Distinguishing Attack for OpenSSL TLS

Figure 2 shows the experimental distribution of timing values for the TLS distinguishing attack described in Section 3. The figure indicates that, with enough samples, it should be possible to distinguish encryptions of message M_0 (consisting of 32 arbitrary bytes followed by 256 copies of 0xFF) from encryptions of message M_1 (consisting of 287 arbitrary bytes followed by 0x00).

We used a simple threshold test to build a concrete attack: we calculate a threshold value T based on profiling, gather L timing samples, filter outliers, calculate the median of the remaining timing samples, and then output 1 if the median value is greater than T and 0 if it is less. Table 1 shows the success probabilities for this concrete distinguishing attack; it is evident that the attack is reliable even if only a moderate number of samples are available. The attack already has a significant advantage over guessing when $L = 1$, i.e. when only one sample is available.

5.4 Plaintext Recovery Attacks for OpenSSL TLS

Partial plaintext recovery: Section 4 describes an attack where byte P^*_{15} can be recovered when P^*_{14} is known. This involves setting Δ_{14} to force $P^*_{14} \oplus \Delta_{14}$ to equal 0x01, and then trying all possible values of Δ_{15}, identifying which one forces $P^*_{15} \oplus \Delta_{15}$ to also equal 0x01. Figure 3 shows the median server-side decryption time as a function of Δ_{15} for the particular values of $P^*_{14} = $ 0x01 (so $\Delta_{14} = $ 0x00) and $P^*_{15} = $ 0xFF. A

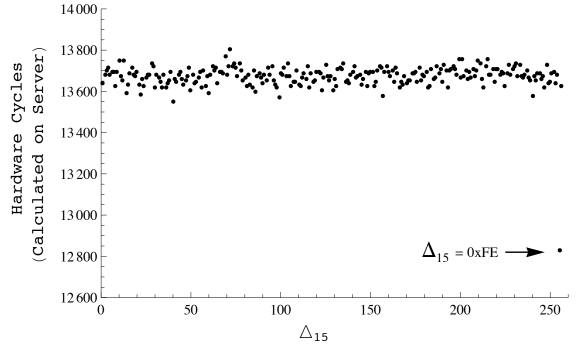

Figure 3: OpenSSL TLS median server timings (in hardware cycles) when $P^*_{14} = $ 0x01 and $P^*_{15} = $ 0xFF. As expected, $\Delta_{15} = $ 0xFE leads to faster processing time.

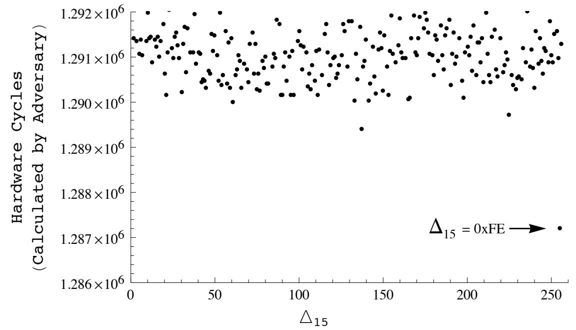

Figure 4: OpenSSL TLS median network timings in terms of hardware cycles when $P^*_{14} = $ 0x01 and $P^*_{15} = $ 0xFF. As expected $\Delta_{15} = $ 0xFE leads to faster processing time.

clear reduction in processing time can be seen for the expected value of Δ_{15}, namely $\Delta_{15} = $ 0xFE. Also notable is the stability in the processing time for other byte values. These server-side times indicate that an attack based on timing error message on the network has some prospect of success. Figure 4 shows the corresponding distribution of median network timings in our experimental setup. Clearly, the data is noisier, but the "dip" at $\Delta_{15} = $ 0xFE is clearly distinguishable.

Figure 5 shows success probabilities for the attack. Each data-point in the figure is based on at least 64 experiments. Each curve in the figure represents a different number of total sessions consumed in the attack (corresponding to different values for L, the number of trials for each Δ value). The x-axis represents the percentile used in our statistical test: if the percentile value is p, then we take as the correct value for Δ_{15} the one for which the p-th percentile value of the timing distribution (measured over L samples) is *minimised*. It is evident that a range of percentiles work well, including the median. As expected, the success probability of the attack increases as L increases. We already reach a success probability of 1 when $L = 2^8$, where the total number of sessions needed is 2^{16}. Similarly, we have a success probability of 0.93 when $L = 2^7$, where the total number of sessions is 2^{15}.

Given these results, we anticipate that the attack would extend easily to recovering 15 unknown bytes from a block, given one of the last two bytes. We have not implemented this variant.

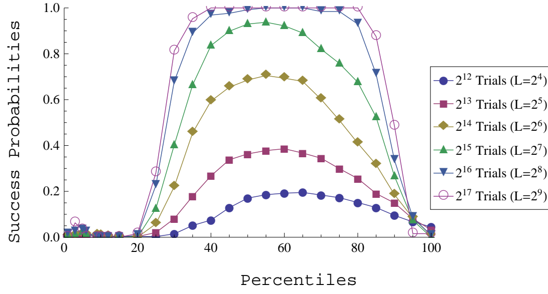

Figure 5: OpenSSL TLS partial plaintext recovery: percentile-based success probabilities for recovering P^*_{15} assuming P^*_{14} known.

Full plaintext recovery: The next step would be to perform the full plaintext recovery attack from Section 4. In this case, the attacker would need a total of $L \cdot 2^{16}$ trials to discover which mask value triggers Case 2. In the case of TLS, this takes a considerable amount of time due to the underlying TCP and TLS connection set-up and tear-down times. For example, with $L = 2^7$ we estimate that the 2^{23} sessions would take around 64 hours in our setup. However, once the last two bytes of a block have been successfully recovered, then the remaining bytes in that block can be recovered in a much shorter time. We have not implemented the full plaintext recovery attack for TLS. Our results below for DTLS strongly indicate that the full attack would work for TLS with $L = 2^7$, albeit slowly.

5.5 Plaintext Recovery Attacks for OpenSSL DTLS

As explained in Section 4.4, we can use the timing and amplification techniques from [1] in combination with the previously described attacks to attack DTLS. Now the attacker sends a number (n) of crafted packets, followed by a DTLS Heartbeat request and waits for the corresponding Heartbeat reply. This process is repeated L times for each mask value. The attacker selects n and L in order to trade-off the attack success probability and the total number of packets injected. We have found experimentally that $n = 10$ is a good choice for achieving stable timing values. On the other hand, $n = 1$ is indicative of what might be expected to happen with TLS but without enduring the overhead of TCP and TLS connection setups (note that the noise levels for DTLS are generally somewhat higher since we depend on an application-layer error message rather than a native TLS error message). Higher values of n could be used if the attacker is remote from the server.

Figure 6 shows the percentile-based success probabilities for recovering P^*_{15} assuming that P^*_{14} is known, for $n = 10$. It can be seen that the attack is very effective, reliably recovering the unknown plaintext byte with only 2^{11} trials ($L = 2^3$). Even for 2^8 trials ($L = 1$), the success probability is 0.266.

We also conducted a 2-byte recovery attack against OpenSSL DTLS; this attack is effectively the first step of the full plaintext recovery attack described in Section 4. Figure 7 shows the success probabilities for recovering P^*_{14} and P^*_{15} when $n = 10$. Again, the attack is very effective, recovering both bytes with success probability 0.93 for 2^{19} trials ($L = 2^3$). The quality of these results is evidence that the attack should extend easily to a full plaintext recovery attack. Figure 8 shows

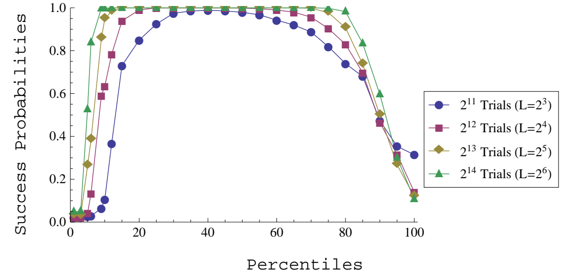

Figure 6: OpenSSL DTLS partial plaintext recovery: percentile-based success probabilities for recovering P^*_{15} with P^*_{14} known, $n = 10$.

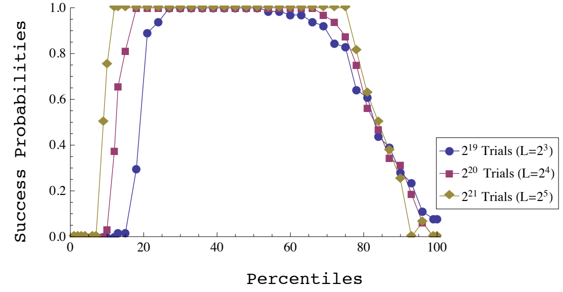

Figure 7: OpenSSL DTLS 2-byte recovery: percentile-based success probabilities for recovering P^*_{14} and P^*_{15}, $n = 10$.

our results for $n = 1$, which we recall serves as an experimental model for TLS. We see that 2-byte recovery is reliable given 2^{23} trials ($L = 2^7$); we already reach more than 80% success rate using 2^{22} trials.

5.6 More Challenging Network Environments

We have not conducted experiments where the attacker is not situated in the same LAN as the server. Given the small timing differences involved, we would expect the attacks to fail when the attacker is remote, i.e. more than a couple of hops away from the server, or that very large numbers of sessions would be needed to get reliable results. Nevertheless, there are realistic scenarios where the proximity requirement can be met, for example when a hostile network service provider attacks its customers, or in cloud computing environments. For DTLS, the timing signals can be amplified, effectively by an arbitrary amount, and so we would expect to be able to mount the attacks remotely.

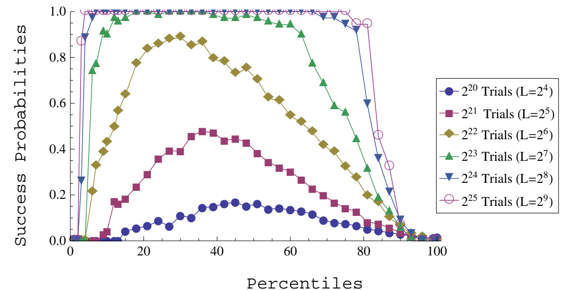

Figure 8: OpenSSL DTLS 2-byte recovery: percentile-based success probabilities for recovering P^*_{14} and P^*_{15}, $n = 1$.

6 Other Implementations of TLS

6.1 GnuTLS

The GnuTLS[5] implementation of MEE-TLS-CBC deals with bad padding in a different way to that recommended in the RFCs: instead of assuming zero-length padding, it uses the last byte of plaintext to determine how many plaintext bytes to remove (whether or not those bytes are correctly formatted padding). More precisely, GnuTLS sets a variable `pad` as:

```
pad = ciphertext->data[ciphertext->size - 1] + 1
```

and then, after doing some basic sanity checking on the value of `pad`, subtracts `pad` bytes from the length field:

```
length = ciphertext->size - tag_size - pad
```

The GnuTLS code then proceeds to check the padding bytes, but the value of `length` stays the same for the remainder of the processing whether the padding check succeeds or fails. This variable dictates the number of record bytes involved in the MAC verification.

Since this approach is a natural alternative to the RFCs' advice for handling bad padding, we analyse it in detail, first for HMAC-SHA-1 as the MAC algorithm, and then in brief for other MAC algorithms. As before, we assume that our block cipher is AES and that IVs are explicit, with obvious modifications for other cases. We focus on TLS, but our attacks apply equally to DTLS. We then report experimental results.

GnuTLS + HMAC-SHA-1: Firstly, we point out that GnuTLS-style processing is just as vulnerable to distinguishing attacks as RFC-compliant processing. Indeed, the attack described in Section 3 will work just as before[6]. We next present an attack that recovers the rightmost byte of plaintext in any target block for GnuTLS-style padding processing.

Let C^* denoting the target ciphertext block, C' denote the previous ciphertext block and Δ denote a mask block of 16 bytes. We consider the decryption of a ciphertext $C^{\text{att}}(\Delta)$ of the form:

$$C^{\text{att}}(\Delta) = \texttt{HDR}||C_0||C_1||C_2||\dots||C_{18}||C' \oplus \Delta||C^*$$

in which there are 20 non-IV ciphertext blocks, the penultimate block is an XOR-masked version of C' and the last block is C^*, the target ciphertext block. The corresponding 320-byte plaintext is $P = P_1||P_2||\dots||P_{19}||P_{20}$ in which

$$\begin{aligned} P_{20} &= D_{K_e}(C^*) \oplus (C' \oplus \Delta) \\ &= P^* \oplus \Delta. \end{aligned}$$

Now we need consider only two distinct cases, which between them cover all possibilities:

1. P_{20} ends with a 0x00 byte: in this case, a single byte of padding is removed, the next 20 bytes are interpreted as a MAC tag T, and the remaining $320 - 21 = 299$ bytes of plaintext are taken as the record R. MAC verification is then performed on a $13 + 299 = 312$-byte message $\texttt{SQN}||\texttt{HDR}||R$.
2. P_{20} ends with any other byte value: in this case, at least two bytes of "padding" are removed, the next 20 bytes are interpreted as a MAC tag T, and the remaining bytes of plaintext are taken as the record R. Because the starting message length, at 320 bytes, is long enough to allow for the removal of 256 bytes of padding and a 20-byte MAC whilst still leaving a non-null record, no length sanity tests will fail. MAC verification is then performed on a message $\texttt{SQN}||\texttt{HDR}||R$ that contains *at most* 311 bytes.

In both cases, the MAC verification will fail (with overwhelming probability) and an error message produced. Notice that, in accordance with the discussion in Section 2.1, in Case 1, the MAC verification will involve 9 evaluations of the compression function for SHA-1, while Case 2 requires at most 8 evaluations. Therefore, we can hope to distinguish the two cases by careful timing, as previously.

Now the single-byte plaintext recovery attack is straightforward: the attacker injects a sequence of ciphertexts $C^{\text{att}}(\Delta)$ with values of Δ that vary over all possible values in the last byte Δ_{15}, then (in the worst case) after 2^8 trials, the attacker will surely select a value for Δ such that $C^{\text{att}}(\Delta)$ triggers Case 1. When this is detected, he knows that P_{20} ends with a 0x00 byte and can infer the value of the last byte of P^* via the block-wise equation $P_{20} = P^* \oplus \Delta$.

This basic attack can be further improved. The 2 most significant bits of the last byte of P^* can be extracted using 4 trials by simply examining the time taken to produce an error message when ciphertexts $C^{\text{att}}(\Delta)$ are injected for values Δ which vary in the 2 most significant bits of Δ_{15}: the maximum running time is produced when the last byte of P_{20} is set to have bits 00 in the most significant positions. The remaining 6 bits can then be extracted using a further 64 trials to find the value of Δ_{15} which triggers Case 1. Thus an enhanced version of the attack only needs 68 trials to recover the last byte of the target block.

For TLS, the usual problems of fatal errors and noisy timing information can be overcome in a multi-session attack. For DTLS, we can use the techniques of [1] to amplify the timing differences and overcome the lack of error messages.

GnuTLS + HMAC-MD5/HMAC-SHA-256: For HMAC-MD5 and HMAC-SHA-256, a similar analysis as before shows that the ciphertext $C^{\text{att}}(\Delta)$ triggers "slow" MAC evaluation (9 compression function evaluations) if P_{20} has last byte that is any of the 5 possibilities 0x00, 0x01, 0x02, 0x03, 0x04, while all other values for the last byte of P_{20} result in "fast" MAC evaluation (at most 8 evaluations). These 5 byte values correspond to bit patterns $000, 001, 010, 011, 100$ in the 3 least significant bits. Exploiting this, we can build an attack using even fewer trials than previously. In the worst case, the attacker needs $24L$ trials to recover all the bits of the last byte of P^*. For TLS, we will need a multi-session attack, but note that the parameter L can be quite small since we only need to distinguish between a few possibilities (at most 16) in each phase of the attack. We omit the details.

[5] `www.gnu.org/software/gnutls/`

[6] In fact, since the attack only involves plaintexts which are correctly padded, it will work for *any* correct decryption algorithm.

Interestingly, the attacks for HMAC-MD5 and HMAC-SHA-256 are much more efficient for GnuTLS-style processing than they are for RFC-compliant processing. This is opposite to the situation for HMAC-SHA-1. We note that we have not found attacks for GnuTLS-style processing that can extract more than the last byte of the target block. This is not surprising in view of the fact that the decryption time for GnuTLS-style processing depends only on the last byte of plaintext.

Attack implementation for GnuTLS: We worked with version 3.0.21 of GnuTLS to implement the above attacks. In doing so, we found some subtle coding errors.

Firstly, the variable `pad` is defined as being of type `uint8`. In the code:

```
pad = ciphertext->data[ciphertext->size - 1] + 1
```

this has the unintended action of setting `pad` to zero when the last byte of plaintext equals 0xFF instead of the desired value of 256, meaning that no bytes of padding are removed in this case instead of 256 bytes. As a consequence, GnuTLS does not properly support variable length padding during decryption, and the TLS session would be terminated if the encrypting party ever uses 0xFF padding.

This coding error is easily patched, but means that our attacks do not quite work as described, since now 2 byte values (0x00 and 0xFF) in the last byte of P_{20} lead to slow MAC verification (in the HMAC-SHA-1 case). In fact, this does not present a serious barrier to our attack, and there is a variant using at most 66 trials to recover the last byte of P^*. We omit the details.

The second coding error we found relates to the implementation of the padding check. This uses the following `for` loop:

```
for (i = 2; i < pad; i++)
    {
      if (ciphertext->data[ciphertext->size - i] !=
          ciphertext->data[ciphertext->size - 1])
        pad_failed = GNUTLS_E_DECRYPTION_FAILED;
    }
```

It is not hard to see that this loop should also cover the edge case `i=pad` in order to carry out a full padding check. This means that one byte of what should be padding actually has a free format. This would enable, for example, a variant of the short MAC attack of [28] even if variable length padding was not supported. This coding error does not affect our attack. Notice also that the number of iterations in the loop depends on `pad`, which is plaintext-dependent.

Experimental Results for GnuTLS: By default, GnuTLS adds random length padding to every TLS record it sends (including alerts), subject to constraints imposed by the TLS specification. The time required to encrypt that random padding disrupts the timing signal that our attacks attempt to detect. For the purposes of experimental validation, we disabled GnuTLS's random padding. Note, however, that the attacks would still be effective even if the the random padding were to be reactivated, since the error messages can be grouped according to their lengths, and the time difference attributable to adding extra padding can be profiled and subtracted for each group.

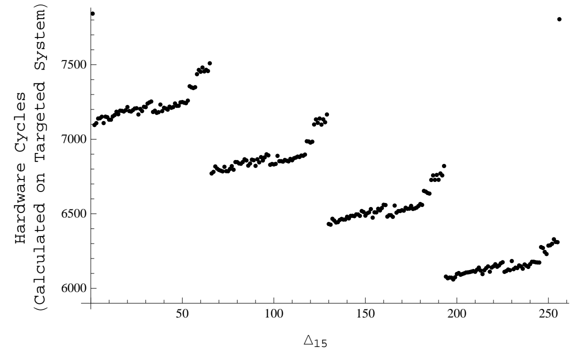

Figure 9: GnuTLS TLS median server timings (in hardware cycles) for varying values of Δ_{15} and $P^*_{15} = 0\text{x}00$.

We began by measuring the time (in hardware cycles) taken by the GnuTLS server to perform the padding check, MAC verification and other associated operations as a function of the value of Δ_{15}, for ciphertexts containing 20 non-IV blocks and with the last byte of P^* equal to 0x00. Figure 9 shows the results. The expected behaviour is observed: byte values 0x00 and 0xFF have similar, long processing times. Moreover, there are four "blocks" of timings, corresponding to the reducing number of compression function evaluations needed as the byte value $\Delta_{15} \oplus P^*_{15}$ increases. (Here, P^*_{15} denotes the last byte of the target plaintext block P^*.) Within these blocks, the trend is upwards, and this is attributable to the increasing amount of time needed for the padding check as the value of `pad` increases.

Our next step was to gather timing of error messages from the network. Figure 10 shows median network timings for the same ciphertext structure. It is evident that there are anomalies at byte values 0x01, 0x11, ..., 0xF1 (with 16 byte increments). In further testing, we discovered that their positions did not depend on the plaintext byte P^*_{15}. This phenomenon was subsequently explained to us [23] as arising from the way in which GnuTLS's random number generator updates its state (when generating CBC-mode IVs for TLS's encrypted error messages). We handled this in our attack by setting the timing values for these mask values to the average value of the neighbouring bytes.

The data is clearly very noisy, and the distinct pattern exhibited in the server timings in Figure 9 is not immediately evident in Figure 10. However, a zoomed view (see Figure 11) shows that an overall descending pattern is evident. Further analysis using linear regression showed that the ascending pattern within each of the 4 blocks is weakly preserved in the network timings. We could not reliably distinguish the values 0x00 and 0xFF needed for the attack mentioned above; however, we are able to reliably extract the 4 most significant bits (MSBs) of P^*_{15}, as we explain briefly next.

Extracting 4 bits of P^*_{15}: To extract the 2 MSBs of P^*_{15}, the attacker focusses on the overall downward trend in the processing time (as a function of $\Delta_{15} \oplus P^*_{15}$) exhibited in Figure 11. Let $\delta_7\delta_6 \ldots \delta_0$ denote the bits of Δ_{15}. By setting $\delta_7 = 0$ and then $\delta_7 = 1$, the attacker has 2 sets each containing 128 masks; he gathers timings for each of these two sets; if larger timings

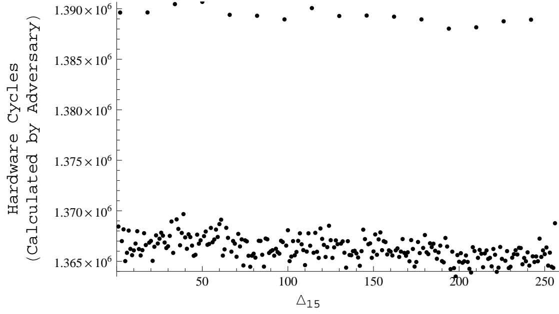

Figure 10: GnuTLS TLS median network timings (in hardware cycles) for varying values of Δ_{15} and $P^*_{15} = 0\text{x}00$.

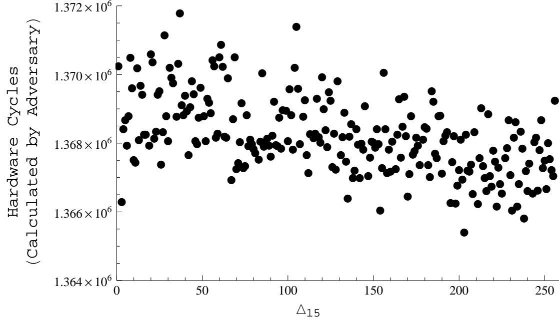

Figure 11: Zoomed view of GnuTLS TLS network timings.

are obtained on average when $\delta_7 = 0$, then the attacker deduces that the MSB of P^*_{15} is a 0; otherwise he guesses that the MSB is 1. The attacker can also use a reduced set of masks, and collect multiple timing samples for each mask that he tries. Thus we have two parameters: the total number of masks S that he uses across the two sets, and the number of timing samples L for each mask. The second MSB of P^*_{15} is extracted in the same way: now we consider masks for $\delta_6 = 0$ and then $\delta_6 = 1$. In principle, we have S as large as 256 again, by varying δ_7 as well as the other 6 bits of Δ_{15}. In practice, we just set $\delta_7 = 0$ when extracting the second MSB. The third and fourth MSBs are extracted in roughly the same way, but now we reverse the test, setting the targeted bit to 1 if larger timings are obtained on average when $\delta_5 = 0$ or $\delta_4 = 0$, respectively. This change reflects the ascending trend within the 4 blocks observed in Figure 9.

Success probabilities for this attack are shown in Table 2. We tried to recover the remaining bits, but did not obtain significant success probabilities. Whilst extracting less plaintext than our OpenSSL attack, far fewer TLS sessions are required in this attack on GnuTLS. This indicates that ignoring the recommendations of the RFCs can have severe security consequences.

6.2 Further Implementations

NSS: Network Security Services (NSS)[7] is an open-source set of libraries implementing, amongst other things, TLS. It is widely used, including in Mozilla client products and Google Chrome.

In the decryption code[8] the variable `plaintext->len` is reduced by the assumed amount of padding `(padding_length + 1)` *before* the padding is checked for correctness. This is the same approach as taken in GnuTLS, potentially rendering the code vulnerable to an attack recovering a single byte of plaintext per block. The sanity check performed at the beginning of the decryption code is also problematic, since it leaves `plaintext->len` unmodified if the check fails, meaning that MAC verification may take longer than when the check passes.

PolarSSL: We also examined the PolarSSL[9] implementation of TLS. The code[10] behaves in much the same way as OpenSSL, setting a variable `padlen` to 0 if the padding check fails, and then verifying the MAC on a record stripped of `padlen` bytes. This would render it vulnerable to the attacks described in Section 4.

S \ L	4	8	16	32	64	128
4	0.575	0.662	0.746	0.828	0.875	0.937
8	0.516	0.615	0.781	0.836	0.844	1
16	0.531	0.609	0.766	0.852	0.969	1
32	0.536	0.596	0.750	0.898	0.984	1
64	0.544	0.596	0.781	0.937	0.984	1
128	0.555	0.627	0.812	0.977	1	1
256	0.593	0.635	0.859	1	1	1

MSB

S \ L	4	8	16	32	64	128
4	0.511	0.580	0.629	0.687	0.656	0.812
8	0.513	0.576	0.695	0.789	0.812	0.812
16	0.515	0.564	0.637	0.742	0.734	0.844
32	0.509	0.549	0.617	0.734	0.766	0.844
64	0.519	0.570	0.656	0.859	0.953	0.969
128	0.544	0.557	0.557	0.914	1	1

Second MSB

S \ L	4	8	16	32	64	128
4	0.486	0.451	0.418	0.391	0.422	0.375
8	0.522	0.508	0.523	0.500	0.531	0.625
16	0.537	0.555	0.598	0.625	0.625	0.781
32	0.543	0.572	0.609	0.609	0.609	0.609
64	0.528	0.541	0.602	0.758	0.758	1

Third MSB

S \ L	4	8	16	32	64	128
4	0.456	0.434	0.363	0.336	0.312	0.25
8	0.487	0.484	0.445	0.477	0.484	0.375
16	0.495	0.531	0.539	0.570	0.594	0.687
32	0.506	0.520	0.566	0.695	0.828	0.812

Fourth MSB

Table 2: GnuTLS success probabilities for recovering the four MSBs of P^*_{15}.

[7] `http://www.mozilla.org/projects/security/pki/nss`

[8] We worked with version 3.13.6 available at `https://ftp.mozilla.org/pub/mozilla.org/security/nss/releases/NSS_3_13_6_RTM/src/`.

[9] `polarssl.org/`

[10] We worked with version 1.1.4 available at `polarssl.org/trac/browser/trunk/library/ssl_tls.c`.

In fact, this implementation has other problems too. The code does not sanity check `padlen` before running the padding check, meaning that out-of-bounds comparisons may be made if the value of `padlen` exceeds the plaintext length. It does sanity check `padlen` *after* the padding check, checking that the plaintext is big enough to contain both the expected amount of padding and the MAC tag. However, it does not perform any MAC check if this sanity check fails, but instead exits immediately. This would render the implementation vulnerable to a simple timing-based distinguishing attack as follows: M_0 consists of 256 copies of 0xFF, while M_1 consists of 255 arbitrary bytes followed by 0x00; as in the attack of Section 3, the encrypted version C of one of these is received; the attacker truncates C so that the underlying plaintext has 256 bytes; if the message was M_0, then the padding is good, but the post-padding sanity check fails and no MAC computation is performed; if the message was M_1, then the padding is also good, but now the post-padding sanity check passes and a MAC computation is performed. This attack produces a larger timing difference than our previous distinguishing attack and illustrates the role that careful sanity checking plays in preventing attacks.

However, none of these attacks would work in practice, since in its default configuration, PolarSSL does not send any TLS alert messages when decryption errors are encountered. This means that PolarSSL is *not* RFC-compliant in this aspect, since such alerts are a required part of TLS implementations.

yaSSL: The yaSSL[11] embedded SSL library, CyaSSL, is targetted at embedded and real-time operating system environments. It appears to have rather few known vulnerabilities, with only 5 being reported in the CVE database[12] since 2005. The CyaSSL code[13] does not perform proper padding checks, but instead just examines the last byte of plaintext and uses this to determine how many bytes to remove. This can be seen in the following CyaSSL code extract:

```
if (ssl->specs.cipher_type == block) {
 if (ssl->options.tls1_1)
    ivExtra = ssl->specs.block_size;
 pad = *(input + idx + msgSz - ivExtra - 1);
 padByte = 1;
}

dataSz = msgSz - ivExtra - digestSz - pad - padByte;
if (dataSz < 0) {
 CYASSL_MSG("App data buffer error, malicious input?");
 return BUFFER_ERROR;
}
```

This approach renders the code vulnerable to the old attack from [26] which recovers one byte of plaintext per block. This was the only implementation that we found that still contains this basic flaw. Note also that the sanity checking represented by the last 3 lines of code above would render the code vulnerable to other plaintext recovery attacks even if the padding check was done properly, since it exits the code without performing a MAC check if the tested condition (which depends on the byte `pad` extracted from the plaintext) is violated.

[11] yassl.com/yaSSL/Home.html

[12] www.cvedetails.com/vulnerability-list/vendor_id-3485/Yassl.html

[13] We worked with version 2.3.0 available at yassl.com/yaSSL/Source/output/src/internal.c.html.

Java: We have examined the BouncyCastle[14] and OpenJDK[15] Java implementations of TLS.

The BouncyCastle code does careful sanity checking of the padding length (as indicated by the last byte of plaintext) but treats the padding as having length 1 if the padding format, when checked, is found to be incorrect (a variable `paddingsize` is set to 0, but then the plaintext size is reduced by an amount `paddingsize+minLength` where `minLength` is set to be 1 larger than the MAC tag size). This deviates slightly from the recommendation of the RFCs to treat the padding as having length zero, but still allows our attacks in Sections 3 and 4 to be applied (for Case 3 of the main plaintext recovery attack in Section 4, MAC verification ends up being performed on a 56-byte message, but this will still involve 5 evaluations of the compression function for SHA-1).

The OpenJDK code appears follow the recommendation of the RFCs in treating the padding as having zero length if the padding format, when checked, is found to be incorrect. This is because this case is trapped by exception handling, during which the variable defining the plaintext length is not changed. This potentially renders it vulnerable to our attacks in Sections 3 and 4.

Other implementations: There are further open-source and many closed-source implementations of (D)TLS. We have not conducted any further testing to see if these are vulnerable to any of our attacks. However, we expect that any RFC-compliant implementation will be vulnerable. We also expect that all implementations will be vulnerable to simple variants of our attacks, unless the implementers have taken great care to ensure that the decryption processing time is uniform, or nearly so. Our experiences in investigating open-source implementations suggests this is unlikely.

7 Countermeasures

Add Random Time Delays: A natural reaction to timing-based attacks is to add random time delays to the decryption process to frustrate statistical analysis. In fact, this countermeasure is surprisingly *in*effective, as we explain next.

Consider our distinguishing attack: this attack involves distinguishing two distributions X, Y, where X has mean μ and Y has mean $\mu+4$, where we measure time in units of compression function evaluations. Suppose X, Y both have variance σ^2. Now suppose we add a random delay that is uniformly chosen from the interval $[0, T]$ to the decryption process. Then we obtain distributions X', Y' with means $\mu+T/2$ and $\mu+4+T/2$ and variance $\sigma^2 + T^2/12$. Now consider the random variables $V_L = \sum_{i=1}^{L} X'_i/L$ and $W_L = \sum_{i=1}^{L} Y'_i/L$ obtained from averaging L samples of X', Y', respectively. Treating these samples as being independent, the Central Limit Theorem guarantees

[14] www.bouncycastle.org/viewcvs/viewcvs.cgi/java/crypto/src/org/bouncycastle/crypto/tls/TlsBlockCipher.java?view=markup

[15] hg.openjdk.java.net/jdk7/l10n/jdk/file/3598d6eb087c/src/share/classes/sun/security/ssl/SSLSocketImpl.java and hg.openjdk.java.net/jdk7/2d/jdk/file/85fe3cd9d6f9/src/share/classes/sun/security/ssl/CipherBox.java

that V_L, W_L are approximately Normal with means $\mu + T/2$, $\mu + 4 + T/2$ and equal variance $\tau^2 = (\sigma^2 + T^2/12)/L$. Note that the difference between the means of V_L, W_L is 4; now, using standard results about the Normal distribution, it is easy to see that if $4\tau \leq 4$, then the distributions of V_L, W_L are sufficiently "tight" about their means that a simple statistical test based on taking means of L samples will be 90% accurate. Solving for L, we see that we need

$$L \geq \sigma^2 + T^2/12$$

and it is apparent that the effect of adding the random time delay is to increase the number of samples needed from σ^2 to $\sigma^2 + T^2/12$. From our experiments for OpenSSL, we estimate that $\sigma \approx 10$; then taking $T = 50$ only increases the number of samples needed for a 90% success rate from 100 to about 300, at the cost of increasing the average decryption time by 25 compression function evaluations. This does not seem like a good trade-off between security and performance.

Use RC4: The simplest countermeasure for TLS is to switch to using the RC4 stream cipher in place of CBC-mode encryption. However, this is not an option for DTLS. When a stream cipher is used in TLS, no padding is required. Consequently none of the attacks in this paper will work. RC4 is widely supported in implementations of TLS, the same countermeasure is effective against the BEAST attack, and was fairly widely adopted in response to BEAST (e.g. by Google and Facebook). The use of a stream cipher in a MEE construction is well-supported by theory [20]. There are two potential drawbacks of making this switch. Firstly, the use of variable length padding in CBC-mode allows for a modicum of plaintext length hiding, and this is no longer possible when using a stream cipher. Secondly, and more importantly, the first bytes of keystream output by the RC4 generator have certain small biases, and TLS does not seem to discard these before starting encryption.

Use Authenticated Encryption: Another possibility is to switch from MEE-TLS-CBC to using a dedicated authenticated encryption algorithm, such as AES-GCM or AES-CCM which were standardised for use in TLS in RFCs 5288 [36] and 6655 [24], respectively. In theory, this should obviate all attacks based on weaknesses in the MEE construction. However, we cannot rule out implementation errors, and we are not aware of any detailed analysis of implementations of these algorithms in (D)TLS for potential side-channels. A further issue is that authenticated encryption was only added in TLS 1.2, and this version of TLS is not yet widely supported in implementations. Finally, the current authenticated encryption algorithms do not offer any length-hiding facility.

Careful implementation of MEE-TLS-CBC decryption: Our final option is to encourage more careful implementation of MEE-TLS-CBC decryption. However, we believe that implementers will find it difficult to do this in a way that eliminates all significant timing channels (especially for DTLS).

The key requirement is to ensure uniform processing time for all MEE-TLS-CBC ciphertexts of a given size. That is, the total processing time should depend only on the ciphertext size, and not on any characteristics of the underlying plaintext (including padding). The basic principle to be followed in achieving this is quite simple: since the major timing differences arise from MAC processing, implementations should make sure the same amount of MAC processing is carried out no matter what the underlying plaintext indicates the message length to be.

However, this simple principle is complicated by the need to also perform careful sanity checking on the underlying plaintext whilst avoiding the introduction of yet more timing side-channels, and to make sure appropriate amounts of MAC processing are performed even when these checks fail.

A further complication arises because the number of bytes to be examined in the padding check depends on the last byte of the last plaintext block, and so, even if the MAC processing is made uniform, the running time of the padding check may still leak a small amount of information about the plaintext. This can be seen for GnuTLS in Figure 9: notice that the maximum difference in the running time for the padding check is more than 1000 hardware cycles for this implementation. For example, then, distinguishing attacks would require a timing resolution of around 1000 hardware cycles, while a timing resolution of 250 cycles would be sufficient to allow an attack recovering 2 bits of plaintext per block for this implementation.

With these remarks in mind, we now proceed to give a detailed prescription of how to achieve constant-time processing of MEE-TLS-CBC ciphertexts, incorporating suitable sanity checking. In what follows, we let `plen` denote the length (in bytes) of the plaintext P obtained immediately after CBC-mode decryption of the ciphertext, `padlen` denote the last byte of that plaintext interpreted as an integer between 0 and 255, and t denote the length of the MAC tags (in bytes). Also, let `HDR`, `SQN` denote the (D)TLS record header and the expected value of the sequence number for this record. Our recommended procedure is then as follows:

1. First sanity check the ciphertext: check that its length in bytes is a multiple of the block-size b and is at least $\max\{b, t+1\}$ (for chained IVs) or $b + \max\{b, t+1\}$ (for explicit IVs). If these conditions are not met, then return `fatal error`.
2. Decrypt the ciphertext to obtain plaintext P; now `plen` will be a multiple of b and at least $\max\{b, t+1\}$.
3. If $t + \texttt{padlen} + 1 > \texttt{plen}$, then the plaintext is not long enough to contain the padding (as indicated by the last byte of plaintext) plus a MAC tag. In this case, run a loop as if there were 256 bytes of padding, with a dummy check in each iteration. Then let P' denote the first $\texttt{plen} - t$ bytes of P, compute a MAC on $\texttt{SQN}||\texttt{HDR}||P'$ and do a constant-time comparison of the computed MAC with the last t bytes of P. Return `fatal error`.
4. Otherwise (when $t + \texttt{padlen} + 1 \leq \texttt{plen}$), check the last $\texttt{padlen} + 1$ bytes of P to ensure they are all equal (to the last byte of P), ensuring that the loop does check all the bytes (and does not stop as soon as the first mismatch is detected). If this fails, then run a loop as if there were $256 - \texttt{padlen} - 1$ bytes of padding, with a dummy check in each iteration, and then do a MAC check as in the previous step. Return `fatal error`.

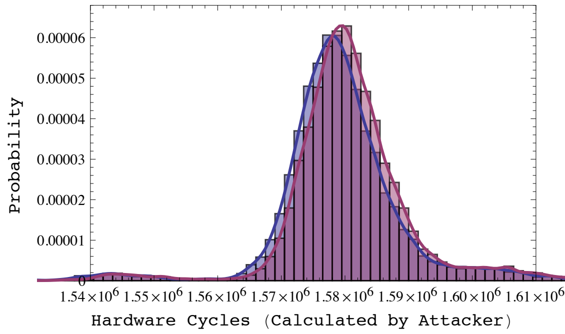

Figure 12: Distribution of timing values (outliers removed) for distinguishing attack on OpenSSL TLS, using our decryption procedure.

5. Otherwise (the padding is now correctly formatted) run a loop as if there were $256 - \texttt{padlen} - 1$ bytes of padding, doing a dummy check in each iteration. Then let P' denote the first $\texttt{plen} - \texttt{padlen} - 1 - t$ bytes of P, and let T denote the next t bytes of P (the remainder of P is valid padding). Run the MAC computation on $\texttt{SQN}||\texttt{HDR}||P'$ to obtain a MAC tag T'. Then set $L_1 = 13 + \texttt{plen} - t$, $L_2 = 13 + \texttt{plen} - \texttt{padlen} - 1 - t$, and perform an additional $\lceil \frac{L_1-55}{64} \rceil - \lceil \frac{L_2-55}{64} \rceil$ MAC compression function evaluations (on dummy data). Finally, do a constant-time comparison of T and T'. If these are equal, then return P'. Otherwise, return `fatal error`.

When implementing the above procedure, it would be tempting to omit seemingly unnecessary computations that are performed, for example when $t + \texttt{padlen} + 1 > \texttt{plen}$. However, these are needed to prevent other timing side-channels like those reported in [1] for the GnuTLS implementation of DTLS. Notice also that the dummy computations performed in the last step are compression function evaluations and not full MAC computations. These give a MAC computation time that is the same irrespective of how much padding is removed (and equal to that carried out in earlier steps). Finally, note that some adjustments to this procedure would be needed when SHA-384 is used as the hash function in HMAC: SHA-384 operates on 128-byte blocks and uses a 16-byte encoding for message length.

We have implemented the above procedure by modifying OpenSSL version 1.0.1, the same version used for our attacks. We modified the code in files `ssl/s3_pkt.c` and `ssl/t1_enc.c` to perform the required sanity checks, dummy padding checks, and dummy MAC compression function evaluations. In `ssl/s3_pkt.c`, we make a single call to OpenSSL's `SHA1_Update` function using a message size that will invoke the required number of dummy MAC compression function evaluations. Our call to `SHA1_Update` happens before OpenSSL's actual MAC calculation and comparison operations.

We then ran our distinguishing attack from Section 3 against the modified OpenSSL code. Each packet in the attack passes the padding check, but fails MAC verification, causing the server to close the TLS session and send an encrypted alert message. Figure 12 shows the distribution of timing values (in hardware cycles) after implementing our procedure. This figure should be compared with Figure 2: visual inspection alone shows that the timing difference is substantially reduced. In fact, the separation between the medians of the two distributions is reduced from about 8500 to about 1100 hardware cycles (from around 2.5μs to 0.32μs). In turn, this small separation means that 128 sessions are needed to achieve a distinguishing success probability of 0.68, whereas, prior to our modifications, just 1 session was enough to give a success probability of 0.756. For the plaintext recovery attack, the adversary will have access to timing differences roughly one quarter of this, i.e. roughly 80ns on our hardware. Notice also that the two distributions are reversed compared to Figure 2, i.e. processing 0xFF packets now takes longer, on average, than for 0x00 packets. We believe that this is caused by overhead introduced by the `SHA1_Update` function call that occurs for 0xFF packets but not 0x00 packets.

To achieve further reductions in timing difference would require a more sophisticated "constant time" programming approach. The OpenSSL patch addressing the attacks in this paper provides an exemplar of how to do this. The complexity of the OpenSSL patch is notable, with around 500 lines of new 'C' code being required. For further discussion and explanation, see `www.imperialviolet.org/2013/02/04/luckythirteen.html`.

8 Discussion

We have demonstrated a variety of attacks against implementations of (D)TLS. We reiterate that the attacks are ciphertext-only, and so can be carried out by the standard MITM attacker, *without* a chosen-plaintext capability. The attacks that are possible depend crucially on low-level implementation details, as well as factors such as the relationship between the MAC tag size t and the block size b. All implementations we examined were vulnerable to one or more attacks.

For TLS, we need a multi-session attack, with, in some cases, many sessions. This limits the practicality of the attacks, but note that they be further improved using standard techniques such as language models and sequential estimation. They can also be enhanced in a BEAST-style attack to enable efficient recovery of HTTP cookies. The timing differences we must detect are close to or below the levels of jitter one typically finds in real networks. In particular, our attacker needs to be positioned relatively close (in terms of network hops) to the machine being attacked. Still, the attacks should be considered as a realistic threat to TLS, and we have described a range of suitable countermeasures. The attacks are much more serious for DTLS, because of this protocol's tolerance of errors and because of the availability of timing amplification techniques from [1]. Very careful implementation of the MEE-TLS-CBC decryption algorithm is needed to thwart these amplification techniques. In view of this, we highly recommend the use of a suitable authenticated encryption algorithm in preference to CBC-mode for DTLS. The contrast between the security of TLS and DTLS reaffirms one of the main messages from [1].

More generally, our attacks illustrate the difficulty of implementing MEE securely. Similar issues were identified for MEE configurations of IPsec in [8]. We encourage protocol designers in general, and the IETF TLS working group in particular, to move away from using MEE. None of the attacks on TLS

presented here would have been possible with an Encrypt-then-MAC approach, for example. A more realistic solution for TLS is to move as quickly as possible to TLS 1.2 and adopt its authenticated encryption algorithms.

Acknowledgements

We thank Xuelei Fan, David McGrew, Adam Langley, Brad Wetmore and the anonymous reviewers for useful feedback. We also thank Eric Rescorla for pointing out that our attacks can be enhanced in the web setting using BEAST-like techniques.

References

[1] N. AlFardan and K. G. Paterson. Plaintext-recovery attacks against Datagram TLS. In *NDSS*, 2012.

[2] G. V. Bard. The vulnerability of SSL to chosen plaintext attack. *IACR Cryptology ePrint Archive*, 2004:111, 2004.

[3] G. V. Bard. A challenging but feasible blockwise-adaptive chosen-plaintext attack on SSL. In *SECRYPT*, pages 99–109, 2006.

[4] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT*, pages 531–545, 2000.

[5] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu. Verified cryptographic implementations for TLS. *ACM TISSEC*, 15(1):3, 2012.

[6] B. Canvel, A. P. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password Interception in a SSL/TLS Channel. In D. Boneh, editor, *CRYPTO*, volume 2729 of *LNCS*, pages 583–599. Springer, 2003. ISBN 3-540-40674-3.

[7] C. M. Chernick, C. Edington III, M. J. Fanto, and R. Rosenthal. Guidelines for the Selection and Use of Transport Layer Security (TLS) Implementations. In *NIST Special Publication 800-52, June 2005, National Institute of Standards and Technology. Available at* `http://csrc.nist.gov/publications/nistpubs/800-52/SP-800-52.pdf`, 2005.

[8] J. P. Degabriele and K. G. Paterson. Attacking the IPsec standards in encryption-only configurations. In *IEEE S&P*, pages 335–349, 2007.

[9] J. P. Degabriele and K. G. Paterson. On the (in)security of IPsec in MAC-then-encrypt configurations. In *ACM CCS*, pages 493–504, 2010.

[10] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246, Internet Engineering Task Force, 1999.

[11] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Internet Engineering Task Force, 2006.

[12] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Internet Engineering Task Force, 2008.

[13] T. Duong and J. Rizzo. Here come the $\oplus$ Ninjas. Unpublished manuscript, 2011.

[14] T. Duong and J. Rizzo. Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET. In *IEEE S&P*, May 2011.

[15] D. Eastlake 3rd. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, 2011.

[16] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *ACM CCS*, 2012.

[17] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *ACM CCS*, 2012.

[18] C. He, M. Sundararajan, A. Datta, A. Derek, and J. C. Mitchell. A modular correctness proof of IEEE 802.11i and TLS. In *ACM CCS*, pages 2–15, 2005.

[19] T. Jager and J. Somorovsky. How to break XML encryption. In *ACM CCS*, pages 413–422, 2011.

[20] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In *CRYPTO*, pages 310–331, 2001.

[21] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), 1997.

[22] U. Maurer and B. Tackmann. On the soundness of authenticate-then-encrypt: formalizing the malleability of symmetric encryption. In *ACM CCS*, pages 505–515, 2010.

[23] N. Mavrogiannopoulos. Personal communication, January 2013.

[24] D. McGrew and D. Bailey. AES-CCM Cipher Suites for Transport Layer Security (TLS). RFC 6655 (Proposed Standard), 2012.

[25] N. Modadugu and E. Rescorla. The Design and Implementation of Datagram TLS. In *NDSS*, 2004.

[26] B. Moeller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures, 2004. `http://www.openssl.org/~bodo/tls-cbc.txt`.

[27] P. Morrissey, N. P. Smart, and B. Warinschi. The TLS Handshake Protocol: A modular analysis. *J. Cryptology*, 23(2):187–223, 2010.

[28] K. G. Paterson, T. Ristenpart, and T. Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *ASIACRYPT*, pages 372–389, 2011.

[29] L. C. Paulson. Inductive analysis of the internet protocol TLS. *ACM TISSEC*, 2(3):332–351, 1999.

[30] A. Pironti, P.-Y. Strub, and K. Bhargavan. Identifying website users by TLS traffic analysis: New attacks and effective countermeasures. Technical Report 8067, INRIA, September 2012.

[31] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347, Internet Engineering Task Force, 2006.

[32] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, Internet Engineering Task Force, 2012.

[33] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, *ACM Conference on Computer and Communications Security*, pages 199–212. ACM, 2009.

[34] J. Rizzo and T. Duong. Practical padding oracle attacks. In *WOOT*, pages 1–8, 2010.

[35] P. Rogaway. Problems with proposed IP cryptography. Unpublished manuscript, 1995. `http://www.cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt`.

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

[37] S. Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ... In *EUROCRYPT*, pages 534–546, 2002.