Lucky Microseconds: A Timing Attack on Amazon's *s2n* Implementation of TLS

Martin R. Albrecht[⋆] and Kenneth G. Paterson[⋆⋆]

Information Security Group
Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK
{martin.albrecht, kenny.paterson}@rhul.ac.uk

Abstract. *s2n* is an implementation of the TLS protocol that was released in late June 2015 by Amazon. It is implemented in around 6,000 lines of C99 code. By comparison, OpenSSL needs around 70,000 lines of code to implement the protocol. At the time of its release, Amazon announced that *s2n* had undergone three external security evaluations and penetration tests. We show that, despite this, *s2n* — as initially released — was vulnerable to a timing attack in the case of CBC-mode ciphersuites, which could be extended to complete plaintext recovery in some settings. Our attack has two components. The first part is a novel variant of the Lucky 13 attack that works even though protections against Lucky 13 were implemented in *s2n*. The second part deals with the randomised delays that were put in place in *s2n* as an *additional* countermeasure to Lucky 13. Our work highlights the challenges of protecting implementations against sophisticated timing attacks. It also illustrates that standard code audits are insufficient to uncover all cryptographic attack vectors.

Keywords TLS, CBC-mode encryption, timing attack, plaintext recovery, Lucky 13, s2n.

1 Introduction

In late June 2015, Amazon announced a new implementation of TLS (and SSLv3), called *s2n* [Lab15,Sch15]. A particular feature of *s2n* is its small code-base: while *s2n* relies on OpenSSL or any of its forks for low-level cryptographic processing the core of the TLS protocol implementation is written in around 6,000 lines of C99. This is intended to make *s2n* easier to audit. Indeed, Amazon also announced that *s2n* had undergone three

[⋆] This author's research supported by EPSRC grant EP/L018543/1.

[⋆⋆] This author's research supported by EPSRC grants EP/L018543/1 and EP/M013472/1, and a research programme funded by Huawei Technologies and delivered through the Institute for Cyber Security Innovation at Royal Holloway, University of London.

external security evaluations and penetration tests prior to release. No details of these audits appear to be in the public domain at the time of writing. Given the recent travails of SSL/TLS in general and the OpenSSL implementation in particular, *s2n* generated significant interest in the security community and technical press.[1]

We show that *s2n* — as initially released — was vulnerable to a timing attack on its implementation of CBC-mode ciphersuites. Specifically, we show that the two levels of protection offered against the Lucky 13 attack [AP13] in *s2n* at the time of first release were imperfect, and that a novel variant of the Lucky 13 attack could be mounted against *s2n*.

The attack is particularly powerful in the web setting, where an attack involving malicious client-side Javascript (as per BEAST, POODLE [MDK14] and Lucky 13) results in the complete recovery of HTTP session cookies, and user credentials such as BasicAuth passwords. In this setting, an adversary runs malicious JavaScript in a victim's browser and additionally performs a Person-in-the-Middle attack. We note, though, that many modern browsers prefer TLS 1.2 AEAD cipher suites avoiding CBC-mode, making them immune to the attack described in this work if the sever also supports TLS 1.2 cipher suites as *s2n* does. The issues identified in this work have since been addressed in *s2n*, partly in response to this work, and current versions are no longer vulnerable to the attacks described in this work.

We stress that the problem we identify in *s2n* does not arise from reusing OpenSSL's crypto code, but rather from *s2n*'s own attempt to protect itself against the Lucky 13 attack when processing incoming TLS records. It does this in two steps: (1) using additional cryptographic operations, to equalise the running time of the record processing; and (2) introducing random waiting periods in case of an error such as a MAC failure.

Step (1) involves calls to a function `s2n_hmac_update`, which in turn makes hash compression function calls to, for example, OpenSSL or LibreSSL. The designers of *s2n* chose to draw a line above which to start their implementation, roughly aligned at the boundary between low-level crypto functions and the protocol itself. The first part of our attack is focused at the lowest level above that line. Specifically, we show that the desired additional cryptographic operations may not be carried out as anticipated: while *s2n* always fed the same number of *bytes* to `s2n_hmac_update`,

[1] See for example `http://www.theregister.co.uk/2015/07/01/amazon_s2n_tls_library/`, `http://www.securityweek.com/amazon-releases-new-open-source-implementation-tls-protocol`.

to defeat timing attacks, this need not result in the same number of *compression function calls* of the underlying hash function. Indeed this latter number may vary depending on the padding length byte which controls after how many bytes `s2n_hmac_digest` is called, this call producing a digest over all data submitted so far. We can also arrange that subsequent calls to `s2n_hmac_update` do not trigger any compression function calls at all. This has the effect of removing the timing equalisation and reopening the window for an attack in the style of Lucky 13.

The second part of our attack is focussed on step (2), the random waiting periods introduced in *s2n* as an additional protection against timing attacks. The authors of [AP13] showed that adding random delays as a countermeasure to Lucky 13 would be ineffective if the maximum delay was too small. The *s2n* code had a maximum waiting period that is enormous relative to the processing time for a TLS record, 10s compared to around 1 μs, putting the attack techniques of [AP13] well out of contention. However, the initial release of *s2n* used timing delays generated by calls to `sleep` and `usleep`, giving them a granularity much greater than the timing differences arising from the failure to equalise the running time in step (1). Consequently, at a high level, we were able to bypass step (2) by "mod-ing out" the timing delays provided by `sleep` and `usleep`. However, the reality is slightly more complex than this simple description would suggest, because those functions do not provide delays that are exact multiples of 1 μs but instead themselves have distributions that need to be taken into account in our statistical analysis. Weaknesses in random delays as countermeasures to timing side-channels have been point out before, cf. [CK10]. In contrast to previous work, though, here the source of timing differences was not close enough to uniform, allowing our analysis of the low-level code to "leak through" the random timing delays, despite them being very large.

Our attack illustrates that protecting TLS's CBC construction against attacks in the style of Lucky 13 is hard (cf. [AIES15]). It also shows that standard code audits may be insufficient to uncover all cryptographic attack vectors.

Our attack can be prevented by more carefully implementing countermeasures to the Lucky 13 attack that were presented in [AP13]. A fully constant time/constant memory access patch can be found in the OpenSSL implementation; its complexity is such that around 500 lines of new code were required to implement it, and it is arguable whether the code would be understandable by all but a few crypto-expert developers. It is worth noting that the countermeasure against Lucky 13 in

OpenSSL does not respect the separation adopted in the *s2n* design, i.e. it avoids higher-level interfaces to HMAC but makes hash compression function calls directly on manually constructed blocks.[2] The *s2n* code was patched to prevent our attacks using a different strategy, (mostly) maintaining the above-mentioned separation. At a high-level, the first step of our attacks exploits that *s2n* counted bytes submitted to HMAC instead of compression function calls. In response, *s2n* now counts the number of compression function calls. Furthermore, the second *s2n* countermeasure was strengthened by switching from using `usleep` to using `nanosleep`.

1.1 Disclosure and Remediation

We notified Amazon of the issue in step (1) of their countermeasures, in the function `s2n_verify_cbc` in *s2n* on 5th July 2015. Subsequently and in response, this function was revised to address the issue reported. This issue in itself does not constitute a successful attack because *s2n* also implemented step (2), the randomised waiting period, as was pointed out to us by the developers of *s2n*. This countermeasure has since been strengthened by switching to the use of `nanosleep` to implement randomised wait periods. This transition was already planned by the developers of *s2n* prior to learning about our work, but the change was accelerated in response to it. Our work shows that the switch to using `nanosleep` was a good decision because this step prevents the attacks described in this work.[3]

1.2 Lucky 13 Remedies in other Libraries

As mentioned above OpenSSL prevents the Lucky 13 attack in 500 lines of code which achieves fully constant time/memory access [Lan13]. GnuTLS does not completely eliminate all potential sources of timing differences, but makes sure the number of compression function calls is constant and other major sources of timing differences are eliminated. As reported in [Mav13] this results in timing differences in the tens of nanonseconds, likely too small to be exploited in practice. In contrast, GoTLS as of now does not implement any countermeasure to Lucky 13. However, a patch

[2] See [Lan13] for a detailed description of the patch.

[3] We also note that the first fix was still vulnerable to a timing attack in step (1), as reported in [ABBD15]. This further highlights the delicacy of protecting against timing side-channel attacks and that the move towards using `nanosleep` was a good decision.

is currently under review to equalise the number of compression function calls regardless of padding value [VF15]. This fix does not promise constant time/memory access. Botan does not implement any countermeasure to Lucky 13.[4] WolfSSL implements the recommended countermeasures to Lucky 13 from [AP13].[5]

2 The TLS Record Protocol and s2n

The main component of TLS of interest 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. In SSL and versions of TLS prior to TLS 1.2, the only encryption option uses a MAC-Encode-Encrypt (MEE) construction. Here, the plaintext data to be transported is first passed through a MAC algorithm (along with a group of 13 header bytes) to create a MAC tag. The supported MAC algorithms are all HMAC-based, with MD5, SHA-1 and SHA-256 being typical hash algorithms. 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). The *s2n* implementation supports 3DES and AES. Following [PRS11], we refer to this MEE construction as MEE-TLS-CBC.

The MEE construction used in the TLS has been the source of many security issues and attacks [Vau02,CHVV03,Moe04,PRS11,AP12,AP13]. These all stem from how the padding that is required in MEE-TLS-CBC is handled during decryption, specifically the fact that the padding is added *after* the MAC has been computed and so forms unauthenticated data in the encoded plaintext. This long sequence of attacks shows that handling padding arising during decryption processing is a delicate and complex issue for MEE-TLS-CBC. It, along with the attacks on RC4 in TLS [ABP$^+$13], has been an important spur in the TLS community's push to using TLS 1.2 and its Authenticated Encryption modes. AES-GCM is now widely supported in implementations. However, the MEE

[4] https://github.com/randombit/botan/blob/master/src/lib/tls/tls_record.cpp#L398

[5] http://www.yassl.com/forums/topic328-wolfssl-releases-protocol-fix-for-lucky-thirteen-attack.html

construction is still in widespread use, as highlighted by the fact that Amazon chose to support it in its minimal TLS implementation *s2n*.

2.1 MEE-TLS-CBC

We now explain the core encryption process for MEE-TLS-CBC in more detail.

Data to be protected by TLS 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, 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. The sender 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, the padding must consist of $p + 1$ copies of some byte value p, where $0 \leq p \leq 255$. In particular, at least one byte of padding must always be added. The padding may extend over multiple blocks, and receivers must support the removal of such extended padding. In SSL the padding format is not so strictly specified: it is only required that the last byte of padding must indicate the total number of additional padding bytes. The attack on *s2n* that we present works irrespective of whether the padding format follows the SSL or the TLS specification.

In the encryption step, the encoded record P is encrypted using CBC-mode of the selected block cipher. TLS 1.1 and 1.2 mandate an explicit IV, which should be randomly generated. TLS 1.0 and SSL use a chained IV; our attack works 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 ciphertext data transmitted over the wire then has the form:

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

where C is the concatenation of the 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.

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, with the check including the header information and a version of the sequence number that is maintained at the receiver.

However, in order to avoid a variety of known attacks, these operations must be performed without leaking any information about what the composition of the plaintext blocks is in terms of message, MAC field and padding, and indeed whether the format is even valid. The difficulties and dangers inherent in this are explained at length in [AP13].

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.

2.2 Details of HMAC

As mentioned above, TLS exclusively uses the HMAC algorithm [KBC97], with HMAC-MD5, HMAC-SHA-1, and HMAC-SHA-256 being supported in TLS 1.2.[6] 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. All the hash functions H used in TLS 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. Also, for all relevant hash functions used in TLS, an 8-byte length field followed by padding of a specified byte format are appended

[6] 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 the SHA-384 algorithm further here.

to the message M to be hashed. The padding is at least 1 byte in length and extends the data to a (56 mod 64)-byte boundary.

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 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. A single compression function evaluation takes typically a few hundred clock cycles.[7]

Implementations typically implement HMAC via an "IUF" interface, meaning that the computation is first initialised (I), then the computation is updated (U) as many times as are needed with each update involving the buffering and/or hashing of further message bytes. When the complete message has been processed, a finalisation (F) step is performed. In *s2n*, OpenSSL or any of its forks is used to implement HMAC. The initialisation step `s2n_hmac_init` carries out a compression function call on the 64-byte string $K_a \oplus \texttt{ipad}$. The update step `s2n_hmac_update` involves buffering of message bytes and calls to the compression function on buffered 64-byte chunks of message. Note that no compression function call will be made until at least 64 bytes have been buffered. The finalisation step `s2n_hmac_digest` consists of adding the length encoding and padding, performing final compression function calls to compute the inner hash and then performing the outer hash operation (itself involving 2 compression function evaluations).

2.3 HMAC Computations after Decryption in *s2n*

The *s2n* implementation uses the code in Figure 1 to check the MAC on a record in the function `s2n_verify_cbc`. This code is followed by a constant-time padding check that need not concern us here (except to note that the fact that it is constant time helps our attack, since it enables us to isolate timing differences coming from this code fragment). In Figure 1, the content of buffer `decrypted->data` is the plaintext after

[7] For example, SHA-256 takes about 550 cycles per block on one of our test systems, an Intel Core i7–4850HQ CPU @ 2.30GHz, whereas SHA-1 takes about 300 cycles.

CBC-mode decryption. The header SQN||HDR of 13 bytes is dealt with by the calling function.

Notice how the code first computes, using the last byte of plaintext, a value for `padding_length`, the presumed length of padding that should be removed (excluding the pad length byte). Arithmetic is then performed to find `payload_length`, the presumed length of the remaining payload over which the HMAC computation is to be done. The actual HMAC computation is performed via an initialise call (not shown), and then the code in line 78 (update via the function `s2n_hmac_update`) and line 84 (finalise via the function `s2n_hmac_digest`). Line 86 compares the computed HMAC value with that contained in the plaintext, and sets a flag `mismatches` if they do not match as expected.

Line 79 copies the HMAC state to a dummy state, so that line 89 can perform a dummy `s2n_hmac_update` computation on data from the plaintext buffer. This attempts to ensure that the number of hash computations carried out is the same, irrespective of the amount of padding that should be removed. This is in an effort to remove the timing channel exploited in the Lucky 13 attack. The number of bytes over which the update is performed is equal to `decrypted->size - payload_length - mac_digest_size - 1`, which is one less than the number of bytes in the plaintext buffer excluding the 13 bytes of SQN||HDR, the message, and the MAC value. Recall, however, that this update operation may not actually result in any compression function computations being carried out. What happens depends on exactly how many bytes are already sitting unprocessed in the internal buffer and how many are added to it in the call.

2.4 Randomised Waiting Period

In order to additionally protect against attacks exploiting timing side-channels, *s2n* implements the following countermeasure: whenever an error occurs, the implementation waits for a random period of time before sending an error message. We reproduce the relevant code excerpts in Figure 2; at a high level, when a MAC failure occurs, the following steps are taken:

- All available data is erased. Depending on the amount of buffered data, the time this takes may vary.
- All connection data is wiped, which may also introduce a timing difference.

```
int payload_and_padding_size = decrypted->size - mac_digest_size;

/* Determine what the padding length is */
uint8_t padding_length = decrypted->data[decrypted->size - 1];

int payload_length = payload_and_padding_size - padding_length \
  - 1;
if (payload_length < 0) {
     payload_length = 0;
}

/* Update the MAC */
GUARD(s2n_hmac_update(hmac, decrypted->data, payload_length));
GUARD(s2n_hmac_copy(&copy, hmac));

/* Check the MAC */
uint8_t check_digest[S2N_MAX_DIGEST_LEN];
lte_check(mac_digest_size, sizeof(check_digest));
GUARD(s2n_hmac_digest(hmac, check_digest, mac_digest_size));

int mismatches = s2n_constant_time_equals(decrypted->data +
                                          payload_length,
                                          check_digest,
                                          mac_digest_size) ^ 1;

/* Compute a MAC on the rest of the data so that we perform
   the same number of hash operations */
GUARD(s2n_hmac_update(&copy, decrypted->data + payload_length +
                      mac_digest_size,
                      decrypted->size - payload_length -
                      mac_digest_size - 1));
```

Fig. 1. Excerpt from `s2n_verify_cbc`, *s2n*'s code for checking the MAC on a TLS record

- A random integer x between 1,000 and 10,001,000 is requested. Since rejection sampling is used to generate x, this might also introduce some timing variation.
- This random integer is then fed to `usleep` and `sleep` calls (after the appropriate scaling), causing a random delay of at least x μs.

We note that this countermeasure, which is activated by default, is designed as an API mode which can in principle be disabled. This is to support implementations which provide their own timing channel countermeasures. If the variable `blinding` is not equal to `S2N_BUILT_IN_BLINDING` then none of the countermeasure code is run.[8] Since this countermeasure

[8] However, we note that a bug in the version of *s2n* that we studied prevented this from ever happening, because the call to wipe the connection data erased this configuration flag as well.

```
s2n_record_read.c
int s2n_record_parse(struct s2n_connection *conn)
...
    /* Padding */
    if (cipher_suite->cipher->type == S2N_CBC) {
        if (s2n_verify_cbc(conn, mac, &en) < 0) {
            GUARD(s2n_stuffer_wipe(&conn->in));
            S2N_ERROR(S2N_ERR_BAD_MESSAGE);
            return -1;
        }

s2n_recv.c
int s2n_read_full_record(struct s2n_connection *conn, \
                         uint8_t *record_type, int *isSSLv2)
    /* Decrypt and parse the record */
    if (s2n_record_parse(conn) < 0) {
        GUARD(s2n_connection_wipe(conn));
        if (conn->blinding == S2N_BUILT_IN_BLINDING) {
            int delay;
            GUARD(delay = s2n_connection_get_delay(conn));
            GUARD(sleep(delay / 1000000));
            GUARD(usleep(delay % 1000000));
        }
        return -1;
    }
```

Fig. 2. Excerpts from `s2n_record_read.c` and `s2n_recv.c`, *s2n*'s code for adding a random waiting period

introduces a delay of up to 10s in case of an error, it might be tempting for some application developers to disable it. However, note that the *s2n* documentation strongly advises against disabling this counter measure without replacing it by an equivalent one on the application level.

3 The Attack without the Random Waiting Period Countermeasure

We first describe our variant of the Lucky 13 attack against *s2n* assuming the random waiting period countermeasure is not present. We show how to deal with this additional countermeasure in Section 4.

For simplicity of presentation, in what follows, we assume the CBC-mode IVs are explicit (as in TLS 1.1 and 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 $b = 8$. The MAC algorithm is HMAC-H where H is either MD5, SHA-1 or SHA-256. We focus at first on the case where the MAC algorithm is HMAC-SHA-256, so that $t = 32$. We explain below how the attack can be adapted to $t = 16$ and $t = 20$ (HMAC-MD5 and HMAC-SHA-1, respectively).

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:

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

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

$$C^{\text{att}}(\Delta) = \texttt{HDR}||C_0||C_1||C_2||C_3||C' \oplus \Delta||C^*$$

consisting of a header field `HDR` containing an appropriate value in the length field, an IV block, and 5 non-IV blocks. The IV block and the first 3 non-IV blocks are arbitrary, the penultimate block $C_4 = C' \oplus \Delta$ is an XOR-masked version of C' and the last block is $C_5 = C^*$. The corresponding 80-byte plaintext is $P = P_1||P_2||P_3||P_4||P_5$ in which

$$\begin{aligned} P_5 &= D_{K_e}(C^*) \oplus (C' \oplus \Delta) \\ &= P^* \oplus \Delta. \end{aligned}$$

Notice that P_5 is closely related to the unknown, target plaintext block P^*. Notice also that, via line 67 of the code in Figure 1, the variable `payload_and_padding_size` is set to $80 - 32 = 48$ (recall that the 13-byte string `SQN`||`HDR` was fed to HMAC by the calling function and is buffered but otherwise unprocessed at this point). We now consider 2 distinct cases:

1. Suppose P_5 ends with a byte value from the set $\{\texttt{0x00}, \ldots, \texttt{0x04}\}$. In this case, the code sets `padding_length` to be at most 4 and then, at line 72, `payload_length` is set to a value that is at least $48 - 4 - 1 = 43$ (and at most 47). This means that when the HMAC computation is performed in lines 78 (update) and 84 (finalise), the internal buffer contains at least 56 bytes (because 13 bytes were already buffered by the calling function) and exactly 5 calls to the compression function will be made, including one call that initialises HMAC and 2 that finalises it. The time equalising code at line 89 adds between 0 and 4 bytes to the internal buffer, which still holds the previous message bytes. However, because of the short length of our chosen ciphertext, the buffer ends up being exactly 60 bytes in

size. This number is obtained by considering the 13 bytes of SQN||HDR, the `payload_length` bytes added to the buffer at line 78 and the `decrypted->size - payload_length - mac_digest_size - 1` bytes added to the buffer at line 89. Combining these, one arrives at there being 12 + `decrypted->size - mac_digest_size` bytes in the buffer. This evaluates to 60 for the particular values in the attack. Notably, this number is independent of `payload_length` and `padding_length`. The call at line 89 is to the update function rather than the finalise function, so at least 64 bytes would be needed in the buffer to cause any compression function computations to be performed at this point. Thus no compression function call is made as a consequence of the call to `s2n_hmac_update` at line 89.

2. Suppose P_5 ends with a byte value from the set $\{$`0x05`$, \ldots,$ `0xff`$\}$. In this case, the code sets `padding_length` to be at least 5 and then, at line 72, `payload_length` is set to a value that is at most $48-5-1=42$ (and at least 0). This means that when the HMAC computation is performed in lines 78 (update) and 84 (finalise), the internal buffer contains at most 55 bytes and exactly 4 calls to the compression function will be made (again, including the initialisation and finalisation calls). The time equalising code at line 89 will again result in no additional calls to the compression function being made, as the internal buffer is again too small at exactly 60 bytes in size (recall that the buffer size is independent of `payload_length` and `padding_length`).

Based on this case analysis, a timing difference will arise in HMAC processing of the attack ciphertext $C^{\text{att}}(\Delta)$, according to whether the last byte of $P_5 = P^* \oplus \Delta$ is from the set $\{$`0x00`$, \ldots,$ `0x04`$\}$ or not. The difference is equal to that taken by one compression function call. This timing difference becomes evident on the network in the form of a difference in the arrival time of an error message at the man-in-the-middle attacker who injects the attack ciphertext. The difference is of the same size as that observed in the plaintext recovery attack presented in [AP13], a few hundred clock cycles on a modern processor. Of course, as in [AP13], this time difference would be affected by noise arising from network jitter, but it is sufficiently big to enable it to be detected. Furthermore, if the attacker can arrange to be co-resident with the victim in a cloud environment, a realistic prospect as shown by a line of work culminating in [VZRS15], the attacker can perform a Person-in-the-Middle attack and observe the usage of resources on the server by being co-resident.

As was the case in [AP13], the attack can be iterated as often as is desired and with different values of Δ, provided the same plaintext is repeated at a predictable location across multiple sessions. The attack as presented already takes care of the complication that each trial will involve a different key in a different TLS session; only P^* needs to be constant for it to work.

By carefully exploring the timing behaviour for different values in the last byte of Δ (each value being tried sufficiently often so as to minimise the effect of noise), the attacker can deduce the value of the last byte of P^*. For example, the attacker can try every value in the 6 most significant bits in the last byte of Δ to identify a value Δ^* for which the time taken is relatively high. This indicates that the last byte of $P^* \oplus \Delta^*$ is in the set $\{\texttt{0x00}, \ldots, \texttt{0x04}\}$; a more refined analysis can then be carried out on the 3 least significant bits of the last byte of Δ^* to identify the exact value of the last byte of P^*. The worst case cost of this version of the attack is $64 + 8 = 72$ trials (multiplied by a factor corresponding to the number of trials per Δ needed to remove noise).

The attack cost can be reduced further by using initially longer ciphertexts, because the peculiar characteristics of the *s2n* code mean that this choice results in there being a greater number of values for (the last byte of) Δ that result in a higher processing time; the precise value of the last byte of P^* can then be pinned down by using progressively shorter ciphertexts. We omit the details of this enhancement.

3.1 Extending to Full Plaintext Recovery

In the web setting, with HTTP session cookies as the target, the attack extends in a straightforward manner to full plaintext recovery using by-now-standard techniques involving malicious client-side Javascript and careful HTTP message padding. A good explanation of how this is achieved can be found in [MDK14] describing the POODLE attack on TLS. `BasicAuth` passwords also form a good target; see [GPdM15] for details.

3.2 Variants for HMAC-MD5 and HMAC-SHA-1

Assume $b = 16$ (as in AES) and consider the case of HMAC-MD5. Then, because $t = 16$ in this case, and t is still a multiple of b, the attack described above works perfectly, except that we need to use a ciphertext having 4 non-IV blocks instead of 5. The attack also works for $b = 8$ for both HMAC-MD5 and HMAC-SHA-256 by doubling the number of non-IV blocks used.

For HMAC-SHA-1, we have $t = 20$. Assume $b = 16$ (AES). Then a similar case analysis as above shows that using a ciphertext with 4 blocks result in a slow execution time if and only if the last plaintext block P_4 ends with `0x00`. This leads to a plaintext recovery attack requiring, in the worst case, 256 trials per byte. The attack adapts to the $b = 8$ case by again doubling the number of non-IV blocks used.

4 Defeating the Random Wait Period Countermeasure

As described in Section 2.4, *s2n* implemented a second countermeasure against attacks exploiting timing channels. In this section, we show how it could be defeated.

4.1 Characterising the Timing Delays

To start off, we notice that at the price of increasing the number of samples by a factor of roughly ten, we can assume that `sleep` at line 103 in the code in Figure 2 is called with parameter zero, by rejecting in an attack any sample where the overall time is more than 1s. This removes one potential source of randomness. As shown in Figure 3, calling `sleep(0)` has a rather stable timing profile.

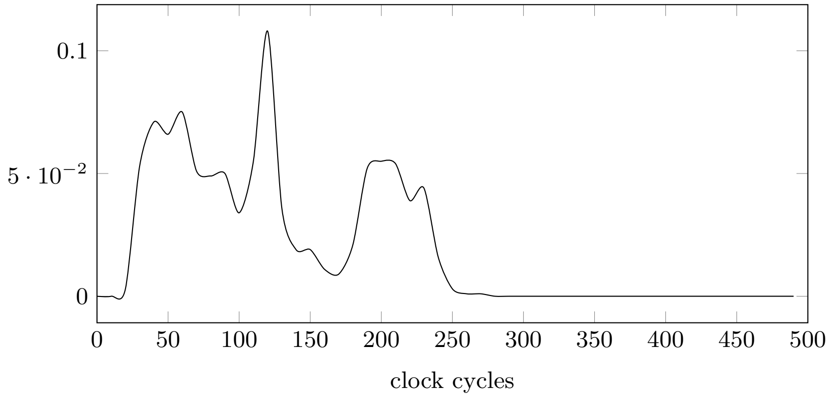

Fig. 3. Distribution of clock ticks for calling `sleep(0)` on Intel(R) Xeon(R) CPU E5-2667 v2 @ 3.30GHz.

Next, we consider calls to `usleep` with a random delay as a source of timing randomness. For this, note that `usleep` has a granularity of 1μs. On our main test machine, which is clocked at 3.3Ghz, this translates to

3,300 clock cycles.[9] From this, we might expect that if we take our timings modulo the clock ticks per μs (namely, 3,300 on our test machine), we could filter out all the additional noise contributed by the `usleep(delay)` call. However, `usleep(delay)` does not guarantee to return after *exactly* `delay` μs, or even to return after an exact number of μs. Instead, it merely guarantees that it will return after *at least* `delay` μs have elapsed. Indeed, on a typical UNIX system, waking up a process from sleep can take an unpredictable amount of time depending on global the state of the OS.

However, despite this, `usleep` does show exploitable non-uniform behaviour on the systems we tested. Figures 4 and 5 illustrate this behaviour. Figure 4 shows raw timings (in clock cycles) for `usleep`(d), normalised to remove the minimum possible delay, namely $3,300 \cdot d$ clock cycles. Figure 5 shows the distribution of timings (in clock cycles) for `usleep(delay)` with `delay` uniformly random in an interval $[0, d)$, but now taken modulo 3,300. Both figures are generated from data captured on our main test machine. They exhibit the non-uniformity needed to bypass the random waiting period countermeasure in *s2n*.

Figures 6 and 7 show that, like the call to `usleep`, the calls to the functions `s2n_stuffer_wipe` and `s2n_public_random` also do not produce timing profiles which are uniform modulo 1μs (3,300 clock cycles).

However, it is not enough to simply characterise the timing profile of the calls to `usleep`; rather it is necessary to study the distribution of the running time of the entire random timing delay code in Figure 2, in combination with the code for checking the MAC on a TLS record in Figure 1, for different values of the mask Δ in the attack in Section 3. Figure 8 brings different sources of timing difference together and shows that the timing distributions (modulo 3,300) that are obtained for different mask values are indeed still rather easily distinguishable. The figure is for samples with the maximum delay restricted to 100,000μs instead of 10s. We stress that this is a synthetic benchmark for studying the behaviour of the various sources of timing randomness and does not necessarily represent actual behaviour. See Section 5 for experiments with the actual *s2n* implementation of these countermeasures.

4.2 Distinguishing Attack

Having characterised the timing behaviour of the *s2n* code, as exemplified in Figure 8, we are now in a position to describe a statistical attack

[9] We note, however, that modern CPUs reclock their CPUs dynamically both below the base operating frequency and above it (e.g. Intel Turbo Boost). This must be taken into account when measuring time delays in elapsed clock cycles.

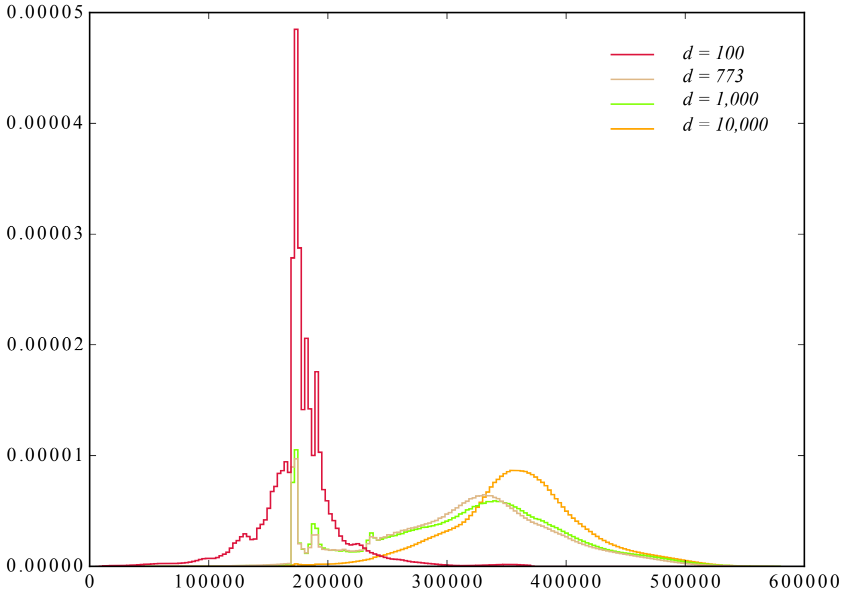

Fig. 4. Distribution of `usleep`(d)$-3,300 \cdot d$ (in clock cycles) on Intel(R) Xeon(R) CPU E5-2667 v2 @ 3.30GHz. Probability on the y axis.

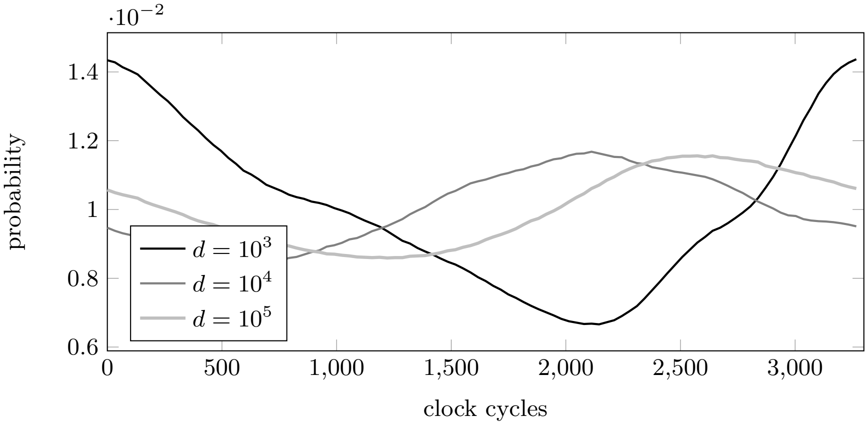

Fig. 5. Distribution of clock ticks modulo 3,300 for `usleep(delay)` with `delay` uniformly random in $[0, d)$, on Intel(R) Xeon(R) CPU E5-2667 v2 @ 3.30GHz.

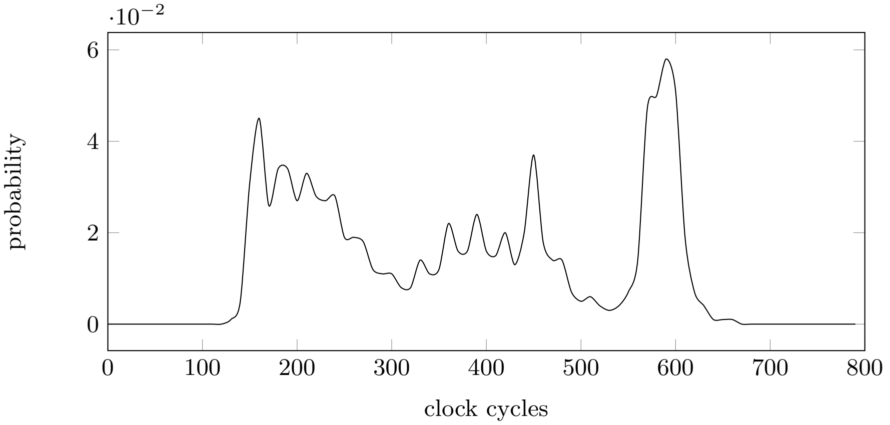

Fig. 6. Distribution of clock ticks for calling `s2n_stuffer_wipe` on Intel(R) Xeon(R) CPU E5-2667 v2 @ 3.30GHz.

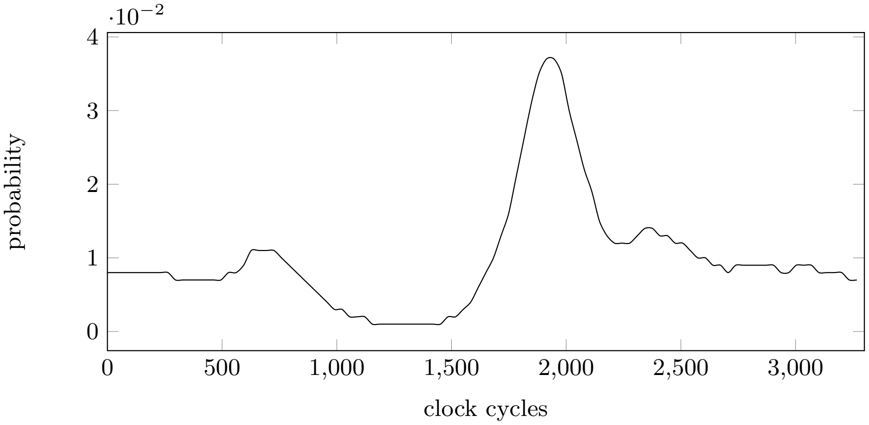

Fig. 7. Distribution of clock ticks modulo 3300 for calling `s2n_public_random` on Intel(R) Xeon(R) CPU E5-2667 v2 @ 3.30GHz.

recovering plaintext bytes and its performance. In fact, the approach is completely standard: given the preceding analysis, we expect the timing distributions modulo 1μs for ciphertexts in the attack of Section 3 to fall into two classes depending on the value of the last byte of $P^* \oplus \Delta$, one class $H = \{\texttt{0x00}, \ldots, \texttt{0x04}\}$, the other class $L = \{\texttt{0x05}, \ldots, \texttt{0xff}\}$; if the observed distributions for all values in L (resp. H) are close to each other but the Kullback-Leibler (KL) divergence between distributions from L and H is large (and equal to D, say), then, applying standard statistical machinery, we know that we will require about $1/D$ samples to distinguish

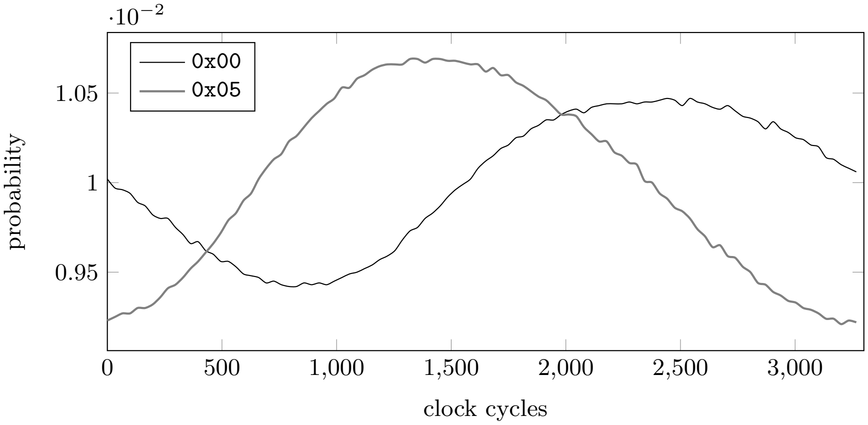

Fig. 8. Distribution of clock ticks modulo 3,300 for timing signals on Intel(R) Xeon(R) CPU E5-2667 v2 @ 3.30GHz with the maximum delay restricted to $d = 100,000$.

samples from the two distributions. As Tables 1 and 2 demonstrate, the requirements on KL divergence for values in L and H are indeed satisfied, even for relatively large values for the maximum delay.

	0x00	0x04	0x05	0x10	0x20	0x30	0x40	0x64	0xc8
0x00	.0	.7	14.1	15.1	17.7	13.2	18.4	17.4	17.6
0x04	.7	.0	15.4	16.8	19.5	15.3	20.0	18.9	19.3
0x05	14.0	15.3	.0	.1	.2	.3	.3	.2	.2
0x10	15.0	16.6	.1	.0	.1	.2	.2	.1	.1
0x20	17.4	19.2	.2	.1	.0	.5	.0	.0	.0
0x30	13.0	15.1	.3	.2	.5	.0	.7	.5	.5
0x40	18.2	19.7	.3	.2	.0	.7	.0	.0	.0
0x64	17.2	18.7	.2	.1	.0	.5	.0	.0	.0
0xc8	17.4	19.0	.2	.1	.0	.5	.0	.0	.0

Table 1. KL divergence multiplied by 1,000 of time distributions in clock cycles modulo 3,300 with the maximum delay limited to 1,000μs on Intel(R) Xeon(R) CPU E5-2667 v2 @ 3.30GHz.

For example, assuming for the sake of argument that no additional noise is introduced by network jitter or other sources, we would be able to distinguish the value `0x00` from `0xc8` in the last byte of $P^* \oplus \Delta$ with $1/(3.6/1,000) \approx 280$ TLS sessions if the maximum delay were restricted to 100,000μs. Using rejection sampling, i.e. discarding all samples with

	0x00	0x04	0x05	0x10	0x20	0x30	0x40	0x64	0xc8
0x00	.0	.0	2.4	1.9	2.3	2.0	2.8	2.1	3.6
0x04	.0	.0	2.3	1.8	2.1	2.0	2.6	1.9	3.3
0x05	2.4	2.3	.0	.0	.0	.1	.0	.0	.2
0x10	1.9	1.8	.0	.0	.1	.1	.1	.0	.3
0x20	2.3	2.1	.0	.1	.0	.2	.0	.0	.1
0x30	2.0	2.0	.1	.1	.2	.0	.3	.2	.5
0x40	2.8	2.7	.0	.1	.0	.3	.0	.1	.0
0x64	2.1	1.9	.0	.0	.0	.2	.1	.0	.2
0xc8	3.6	3.4	.2	.3	.1	.5	.0	.2	.0

Table 2. KL divergence (scaled by 1,000 for readability) of time distributions in clock cycles modulo 3,300 with the maximum delay limited to 100,000μs on Intel(R) Xeon(R) CPU E5-2667 v2 @ 3.30GHz.

a delay greater than 100,000μs from the actual distribution produced by *s2n* (where the maximum delay is 10s), this increases to roughly $28,000$ TLS sessions for a successful distinguishing attack. We stress that this estimate is optimistic because it is derived from a synthetic benchmark not the actual implementation and because the surrounding code and network jitter will introduce additional noise.

4.3 Plaintext Recovery Attack

We can extend this distinguishing attack to a plaintext recovery attack in the following (standard) way. We assume that in a characterisation step, we have obtained, for possible value x of the last byte in block P_5, a histogram of the timing distribution modulo 1μs for ciphertexts $C^{\text{att}}(\Delta)$ of the form used in the attack. We assume these timings are distributed into B equal-sized bins, and so the empirical probability of each bin $p_{x,b}$ for $0 \leq b < B$ can be calculated. (In fact, since we expect that timing behaviours for the classes H and L are similar, it is sufficient to sample for two values x, one from each class.)

Now, in the actual attack, for each value δ of the last byte of Δ, we obtain N samples for ciphertexts $C^{\text{att}}(\Delta)$ for which the timing delay is at most 100,000μs. This then requires a total of about $256 \cdot 100 \cdot N$ TLS sessions. We bin these into B bins as above, letting $n_{\delta,b}$ denote the number of values in bin b for last byte value δ. Now for each candidate value y for the last byte of P^*, we compute the log likelihood for the candidate,

Byte value	Cycles	Byte value	Cycles	Byte value	Cycles
0x00	2251.96	0x05	1746.49	...	...
0x01	2354.57	0x06	1747.65	0xfc	1640.79
0x02	2252.07	0x07	1705.62	0xfd	1634.61
0x03	2135.11	0x08	1808.73	0xfe	1648.70
0x04	2130.02	0x09	1806.50	0xff	1634.64

Table 3. Timing of function `s2n_verify_cbc` (in cycles) with $H =$ SHA-256 for different values of last byte in the `decrypted` buffer, each cycle count averaged over 2^8 trials.

using the formula:

$$LL(y) = \sum_{\delta \in \{\texttt{0x00},\ldots,\texttt{0xFF}\}} n_{\delta,b} \cdot \log(p_{\delta \oplus y,b}) \,.$$

We then output as the preferred candidate for the last plaintext byte the value y^* having the highest value of $LL(y)$ amongst all candidates.

We omit the detailed analysis of the performance of this attack, pausing only to note that it will require more samples than the distinguishing attack because the underlying statistical problem is to now separate one correct candidate from 255 wrong candidates, and this is more demanding than the basic distinguishing problem.

To wrap up, we note that `nanosleep`, which is now used in *s2n* to add a random time delay, has a granularity of nanoseconds, does not show this behaviour, and therefore thwarts the attacks described in this work.

5 Proof of Concept

We confirmed that *s2n* does indeed behave as expected using the following two experiments.

For the first experiment, we setup a `s2n_blob` buffer of length 93 and filled it with random data. Then, we assigned all possible padding length values `0x00` to `0xff` by overwriting the last byte of the buffer and timed how long the function `s2n_verify_cbc` took to return. As expected, the padding length values between `0x00` and `0x04` resulted in timings about 500–550 cycles longer than all other values. The timing difference was clear and stable. Some sample data is shown in Tables 3 and 4. We note that at present we cannot explain the variation within the second and third columns of those tables.

For the second experiment, we ran the attack against the actual *s2n* implementation instead of running a synthetic benchmark. That is, we

Byte value	Cycles	Byte value	Cycles	Byte value	Cycles
`0x00`	1333.99	`0x05`	1095.01	...	...
`0x01`	1174.29	`0x06`	1092.68	`0xfc`	1062.37
`0x02`	1178.52	`0x07`	1065.08	`0xfd`	1035.48
`0x03`	1156.56	`0x08`	1102.31	`0xfe`	1035.15
`0x04`	1140.14	`0x09`	1101.04	`0xff`	1036.02

Table 4. Timing of function `s2n_verify_cbc` (in cycles) with H = SHA-1 for different values of last byte in the `decrypted` buffer, each cycle count averaged over 2^{10} trials.

timed the execution of `s2n_recv` under the attack described in Section 3. However, to speed up execution we patched *s2n* to only sample random delays up to 10,000μs. As highlighted in Table 5, this, too, shows marked non-uniform timing behaviour modulo 1μs.

	`0x00`	`0x01`	`0x02`	`0x03`	`0x04`	`0x05`	`0x0a`	`0x10`	`0x20`
`0x00`	.0	.4	.2	.1	.4	1.7	1.6	1.9	2.2
`0x01`	.4	.0	.4	.3	.3	2.6	2.6	2.8	3.2
`0x02`	.2	.4	.0	.1	.2	2.3	2.2	2.6	2.8
`0x03`	.1	.3	.1	.0	.3	2.1	1.9	2.3	2.7
`0x04`	.4	.3	.2	.3	.0	2.6	2.6	2.9	3.2
`0x05`	1.7	2.6	2.3	2.1	2.6	.0	.1	.2	.3
`0x0a`	1.6	2.6	2.2	1.9	2.6	.1	.0	.2	.3
`0x10`	1.9	2.8	2.6	2.3	2.9	.2	.2	.0	.2
`0x20`	2.2	3.2	2.8	2.7	3.2	.3	.3	.2	.0

Table 5. KL divergence observed the full attack against actual *s2n* implementation (scaled by 10^5 for readability) using 2^{24} samples on Intel(R) Xeon(R) CPU E5-2667 v2 @ 3.30GHz.

We did not adjust our proof-of-concept code to realise a full plaintext recovery attack, because (a) *s2n* has since been patched in response to this work and because (b) the cost is somewhat dependent on the target machine and operating system. We note, though, that an attack can establish the characteristics of a target machine by establishing genuine TLS sessions (where, hence, padding bytes are known) but with some random bits flipped.

The complete source codes for our experiments (which borrow heavily from the *s2n* test suite) are available at `https://bitbucket.org/malb/research-snippets`.

6 Discussion

Our attack successfully overcomes both levels of defence against timing attacks that were instituted in *s2n*, the first level being the inclusion of extra cryptographic operations in an attempt to equalise the code's running time and the second level being the use of a random wait interval in the event of an error such as a MAC failure.

Fundamentally, the first level could be bypassed because *s2n* counted bytes going into `s2n_hmac_update` instead of computing the number of compression function calls that need to be performed as suggested in [AP13]. A call to `s2n_hmac_update` in itself will not necessarily trigger a compression function call if insufficient data for such a call is provided. A call to `s2n_hmac_digest`, however, will pad the data and trigger several compression function calls, the number also depending on the data already submitted at the time of the call. We note that in OpenSSL this issue is avoided by effectively re-implementing HMAC in the function `ssl3_cbc_digest_record`, i.e. by performing lower-level cryptographic operations within the protocol layer. In contrast, *s2n* is specifically aimed at separating those layers. In response to this work, *s2n* now sensibly counts the number of compression function calls performed, somewhat maintaining this separation.

The second level could be bypassed because, while the randomised wait periods were large, they were not sufficiently random to completely mask the timing signal remaining from the first step of our attack. Note that the analysis in [AP13] of the effectiveness of random delays in preventing the Lucky 13 attack assumed the delays were uniformly distributed; under this assumption, their analysis shows that the count measure is not effective unless the maximum delay is rather large. What the second step of our attack shows is that, even if the maximum delay is very large, non-uniformity in the distribution of the delay can be exploited. In short, it is vital to carefully study any source of timing delay to ensure it is of an appropriate quality when using it for this kind of protection.

Our experiments indicate that the distribution of `nanonsleep` as implemented on Linux is sufficiently close to uniform to thwart the attack described in this work. We note, however, that this puts a high security burden on this function which is not designed for this purpose. In particular, `nanosleep(2)` states (emphasis added): "nanosleep() suspends the execution of the calling thread until either *at least* the time specified in *req has elapsed, or the delivery of a signal that triggers the invocation of a handler in the calling thread or that terminates the process."

Finally, since randomised waiting can also have a significant performance impact, this work further highlights that MAC-then-Encrypt constructions such as MEE-TLS should be avoided where possible.

Acknowledgement

We would like to thank Colm MacCarthaigh and the rest of the *s2n* development team for pointing out the randomised waiting countermeasure and for helpful discussions on an earlier draft of this work.

References

[ABBD15] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC. *IACR Cryptology ePrint Archive*, 2015:1241, 2015.

[ABP+13] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the Security of RC4 in TLS. In Samuel T. King, editor, *Proceedings of the 22nd USENIX Security Symposium*, pages 305–320, Washington D.C., USA, August 2013. USENIX.

[AIES15] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 85–96. ACM, 2015.

[AP12] Nadhem AlFardan and Kenneth G. Paterson. Plaintext-recovery attacks against Datagram TLS. In *Network and Distributed System Security Symposium (NDSS 2012)*, 2012.

[AP13] Nadhem AlFardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In Robin Sommer, editor, *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P 2013)*, pages 526–540, San Diego, CA, USA, May 2013. IEEE Press.

[CHVV03] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password Interception in a SSL/TLS Channel. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 583–599. Springer, 2003.

[CK10] Jean-Sébastien Coron and Ilya Kizhvatov. Analysis and improvement of the random delay countermeasure of CHES 2009. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2010.

[GPdM15] Christina Garman, Kenneth G. Paterson, and Thyla Van der Merwe. Attacks only get better: Password recovery attacks against RC4 in TLS. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 113–128. USENIX Association, 2015.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997.

[Lab15] Amazon Web Services Labs. s2n : an implementation of the TLS/SSL protocols. `https://github.com/awslabs/s2n`, 2015.

[Lan13] Adam Langley. Lucky thirteen attack on TLS CBC. `https://www.imperialviolet.org/2013/02/04/luckythirteen.html`, February 2013.

[Mav13] Nikos Mavrogiannopoulos. Time is money (in CBC ciphersuites). `http://nmav.gnutls.org/2013/02/time-is-money-for-cbc-ciphersuites.html`, February 2013.

[MDK14] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: Exploiting the SSL 3.0 fallback, September 2014.

[Moe04] Bodo Moeller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures. Unpublished manuscript, May 2004. `http://www.openssl.org/~bodo/tls-cbc.txt`.

[PRS11] Kenneth G. Paterson, Thomas Ristenpart, and Thomas Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 372–389. Springer, 2011.

[Sch15] Stephen Schmidt. Introducing s2n, a new open source TLS implementation. `https://blogs.aws.amazon.com/security/post/TxCKZM94ST1S6Y/Introducing-s2n-a-New-Open-Source-TLS-Implementation`, June 2015.

[Vau02] Serge Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ... In Lars R. Knudsen, editor, *EUROCRYPT*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546. Springer, 2002.

[VF15] Filippo Valsorda and Brad Fitzpatrick. crypto/tls: implement countermeasures against CBC padding oracles. `https://go-review.googlesource.com/#/c/18130/`, December 2015.

[VZRS15] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. A placement vulnerability study in multi-tenant public clouds. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 913–928. USENIX Association, 2015.