Pseudo Constant Time Implementations of TLS Are Only Pseudo Secure

Eyal Ronen
Weizmann Institute of Science
eyal.ronen@weizmann.ac.il

Kenneth G. Paterson
Royal Holloway, University of London
kenny.paterson@rhul.ac.uk

Adi Shamir
Weizmann Institute of Science
adi.shamir@weizmann.ac.il

ABSTRACT

Today, about 10% of TLS connections are still using CBC-mode cipher suites, despite a long history of attacks and the availability of better options (e.g. AES-GCM). In this work, we present three new types of attack against four popular fully patched implementations of TLS (Amazon's s2n, GnuTLS, mbed TLS and wolfSSL) which elected to use "pseudo constant time" countermeasures against the Lucky 13 attack on CBC-mode. Our attacks combine several variants of the PRIME+PROBE cache timing technique with a new extension of the original Lucky 13 attack. They apply in a cross-VM attack setting and are capable of recovering most of the plaintext whilst requiring only a moderate number of TLS connections. Along the way, we uncovered additional serious (but easy to patch) bugs in all four of the TLS implementations that we studied; in three cases, these bugs lead to Lucky 13 style attacks that can be mounted remotely with no access to a shared cache. Our work shows that adopting pseudo constant time countermeasures is not sufficient to attain real security in TLS implementations in CBC mode.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; *Security protocols*; Symmetric cryptography and hash functions;

KEYWORDS

Lucky 13 attack, TLS, Side-channel cache attacks, Plaintext recovery

1 INTRODUCTION

"All secure implementations are alike; each insecure implementation is buggy in its own way." – after Leo Tolstoy, Anna Karenina.

1.1 Background

The celebrated Lucky 13 attack on TLS [3] builds on Vaudenay's padding oracle attack [9, 33] and exploits small timing variations present in implementations of TLS's decryption processing for CBC-mode cipher suites. The attack enables remote plaintext recovery of TLS-protected data that is sent repeatedly in predictable locations in a connection, such as HTTP cookies. The exploited timing variations were endemic in TLS implementations due to TLS's reliance on a MAC-then-pad-then-encrypt construction in CBC mode: reversing these steps requires removal of padding *before* robust integrity checks have been performed. An attacker could exploit CBC-mode's "cut and paste" property to place target ciphertext blocks at the end of TLS records so that they were interpreted as containing padding. The timing differences between good and bad padding could then be translated into leakage about target plaintext blocks. Under an assumption about the presence of malicious client-side JavaScript, an attacker could arrange for arbitrary plaintext bytes to be recovered. Similar techniques were exploited in the POODLE attack [29] which was specific to SSL's padding construction. Implementation-specific variants of the Lucky 13 attack were also discovered, see for example [2, 4, 5].

The TLS developer community responded in a variety of different ways to Lucky 13.[1] OpenSSL, used in Apache and NGINX, its BoringSSL fork used by Google in Chrome and server-side, as well as NSS used in Mozilla Firefox, added roughly 500 new lines of code to implement the decryption processing required in a fully constant-time, constant-memory-access fashion. The code is complex and difficult to understand for developers not fully conversant in constant-time programming techniques.[2] Even this was not fully successful at first, as a code-branch in OpenSSL taking advantage of AES hardware support was not properly patched, and an even worse attack was enabled [32].

Other implementations (e.g. Amazon's s2n, GnuTLS, wolfSSL) took an easier route by adopting "pseudo constant time" solutions to address Lucky 13. For example, s2n attempted to equalise the MAC verification time by adding dummy HMAC computations and also included a random timing delay. This kind of approach was perhaps justified given the small timing differences involved in Lucky 13 (on the order of 1 microsecond, making the attack difficult to mount in practice, especially remotely) and the complexity of the OpenSSL patch. However, soon after, Irazoqui *et al.* [5] showed how to re-enable the Lucky 13 attack in a cross-VM setting, by presenting cache-based "FLUSH+RELOAD" attacks that detect the dummy function calls that only occur when bad padding is encountered. Their attacks work on deduplication-enabled platforms (e.g. those implementing Kernel SamePage Merging, KSM, or related technologies). In this setting, their attacks apply to those implementations which take the simplest approach to remediation, that of adding dummy computations via new function calls. Irazoqui *et al.* showed that the PolarSSL (now mbed TLS), GnuTLS and CyaSSL (now wolfSSL) implementations were all vulnerable to attack in their specific deduplication-enabled, cross-VM setting. PolarSSL patched against this attack, but GnuTLS and wolfSSL chose not to. However, as deduplication is currently disabled across different VMs by Infrastructure-as-a-service (IaaS) providers [23], no practical cross-VM attack against any implementation is currently known.[3]

[1] A partial list of vendor responses can be found at http://www.isg.rhul.ac.uk/tls/Lucky13.html.
[2] See https://www.imperialviolet.org/2013/02/04/luckythirteen.html for a detailed discussion of this patch.
[3] More recently Xiao *et al.* [35] used an automated differential analysis framework to find cache-based side channels to re-enable the Lucky 13 attack against GnuTLS and mbed TLS code that runs directly inside an Intel Software Guard Extension (SGX) secure enclave. However, they require root permissions for their "man in the kernel" attack.

ECDHE_RSA_AES_256_CBC_SHA384	4.4%
RSA_AES_256_CBC_SHA	1.7%
RSA_AES_128_CBC_SHA	1.2%
ECDHE_RSA_AES_256_CBC_SHA	1.2%
ECDHE_RSA_AES_128_CBC_SHA	1.1%
ECDHE_RSA_AES_128_CBC_SHA256	1.1%

Table 1: Distribution of CBC-mode TLS cipher suites. Source: ICSI Certificate Notary, 24/04/2018 [1]

More broadly, the Lucky 13 attack, and the vulnerabilities in the alternative RC4-based cipher suites discovered around the same time, nudged developers into finally implementing and deploying TLS 1.2 with its support for more modern AES-GCM-based cipher suites. These have risen in popularity to the point today where more than 80% of TLS connections rely on AES-GCM. Yet still today, more than 10% of TLS traffic is protected with CBC-mode cipher suites in the original MAC-then-pad-then-encrypt construction, see Table 1.[4] Notably, CBC-mode cipher suites relying on HMAC with SHA-384 for integrity have risen in popularity over SHA-256 and SHA-1 (this might be due to the fact that on modern 64-bit CPUs, SHA-384 is faster per byte than SHA-256). This 10% figure makes the security of CBC-mode cipher suites of enduring interest and means that their continued study (and elimination in the event of new vulnerabilities being found) is still of considerable value.

1.2 Our contributions

Our main contribution is to present novel cache timing attacks on a representative set of implementations of TLS that did *not* adopt the fully constant-time/constant-memory-access approach to address Lucky 13, but which instead used pseudo constant time fixes. We are thus able to mount practical attacks on TLS implementations that have been fully patched against all previously known variants of Lucky 13, including previous cache-based attacks such as [5].

As usual for such attacks, we assume the existence of a co-located adversary running on the same CPU as the victim's process or VM, and a shared cache. However, in contrast to [5], we do not rely on memory deduplication technologies like KSM, that are currently disabled across different VMs by IaaS providers [23]. Instead, we only assume a shared Last Level Cache (LLC) side-channel as in [23].

Our attacks are capable of plaintext recovery with low complexity. We use 3 different LLC side-channel based attack techniques to target the s2n, GnuTLS, mbed TLS and wolfSSL implementations, giving for each technique a proof of concept (PoC) attack.

The attacks were developed through manual code inspection of the different implementations. We show that each implementation provides some leakage to the adversary about the amount of TLS padding present in a plaintext underlying a chosen ciphertext. Using by-now standard "JavaScript in the browser" methods (see, for example, [3]), such leakage can be leveraged to perform plaintext recovery for TLS cookies, for example. Naive exploitation of some of the leakages requires a large number of TLS connections, but we show how to fine-tune them to improve their performance by three orders of magnitude. We also introduce a novel variant of Lucky 13 that uses long TLS padding patterns. These enhancements should be of independent interest.

1.2.1 Implementation Bugs in Lucky13 Countermeasures. As a secondary contribution, we point out that all the reviewed pseudo constant time implementations of TLS (s2n, GnuTLS, mbed TLS, wolfSSL) have bugs in their pseudo-constant-time code that come into play when SHA-384 is selected as the hash algorithm in HMAC. These bugs are easy to fix by changing some constants related to the SHA-384 hash size, but render the decryption operations non-constant time and therefore vulnerable to relatively simple plaintext recovery attacks. Moreover, we show that GnuTLS is still vulnerable to a novel variant of the original Lucky 13 attack even for SHA-256, despite having been specifically patched against Lucky 13.

1.2.2 New variants of the PRIME+PROBE attack:

(1) The synchronized probe PRIME+PROBE attack can sense the time between an event controlled by the attacker (e.g. sending a message to the target) and a non-constant-time memory access. We send the message at time t_{send} and assume that the memory access will occur either at time $t_{\text{send}} + t_1$ or $t_{\text{send}} + t_2$ depending on some secret value. We synchronize the cache probing to occur at time $t_{\text{send}} + t_{\text{probe}}$ (where $t_1 < t_{\text{probe}} < t_2$). We create a "race condition" between our probe and the target's memory access. From the result of the cache probing, we get a timing oracle. This attack technique is especially useful in LLC side-channels like those in [23], in a scenario where the side-channel probing resolution is not high enough to time the event by continuous probing. The delay t_{probe} is both code and machine specific.
(2) The synchronized prime PRIME+PROBE attack can distinguish between two different memory access patterns after an event controlled by the attacker (e.g. sending a message to the target). We send the message at time t_{send} and assume that the target code will access the memory at time $t_{\text{send}} + t_1$. Depending on some secret value, the target code might access the memory again at time $t_{\text{send}} + t_2$. We synchronize the cache priming to occur at time $t_{\text{send}} + t_{\text{prime}}$ (where $t_1 < t_{\text{prime}} < t_2$). We probe the cache at some time $t_{\text{probe}} > t_2$. From the result of the cache probing, we get an oracle for the secret value. Again this attack technique is especially useful a scenario where the side-channel probing resolution is not high enough to perform multiple measurements in the interval of length $t_1 - t_2$. The delay t_{prime} is both code and machine specific.
(3) The "*PostFetch*" attack can help to overcome large cache lines that reduce the cache attack resolution. We would like to distinguish between the cases of accessing just the first few bytes of an array inside a cache line, and accessing most of the array. However, due to the cache line size, the whole array will be read into the cache in both cases. In some scenarios, if a large part of a cache line is accessed (near the cache line boundary), then the next cache line will also be read into the cache. This can be caused by either hardware memory prefetching or by speculative execution. In those scenarios, we can distinguish between the two types of access by probing the cache line that contains the bytes *after* the array.

[4] RFC 7366 [18] specifies an alternative "Encrypt-then-MAC" construction, but figures obtained from the ICSI Certificate Notary indicate that it is barely used.

1.2.3 Implications of Our Results. We consider our complete set of results surprising in the light of the huge amount of effort spent on correcting and verifying CBC-mode and HMAC implementations in TLS over the last 5 years. For example, s2n was repeatedly patched in response to Lucky 13 style attacks [2, 4]. Its principal author, Colm MacCarthaigh wrote a detailed and thoughtful blogpost explaining AWS's selected approach to defending against this kind of attack [24], focussing on the argument that a balance needs to be struck between code simplicity and security. Moreover, the vulnerable s2n HMAC code had also passed formal verification [10, 13]. At its core, our work shows that nothing short of the full "belt and braces" approach adopted in OpenSSL is sufficient to provide a robust defence against Lucky 13 style attacks in all their forms, and in fact the approach taken in OpenSSL is immune to our attacks. While our cache-based timing attacks are different from the methods used in previous attacks against s2n [2, 4] and mbed TLS [5], cache-based attack scenarios have been well-known and broadly accepted as being realistic in the security community for some years. In retrospect, the developers of the TLS implementations which we target in this work might have better invested their code development effort in adopting fully robust approaches from the beginning, rather than being forced to incrementally patch against each new generation of attack (or to have to expend energy defending the decision not to patch at all).

1.3 Disclosure

We have disclosed the vulnerabilities to all vendors mentioned in the paper, and suggested a coordinated public disclosure on the 25th of July 2018. The status of these disclosures at the time of writing is as follows:

- The wolfSSL team followed our recommendation and switched to the full constant time solution in release 3.15.3[5] (released 20th June 2018).
- The mbed TLS team released a security advisory[6] on July 25th 2018. CVEs 2018-0497 and 2018-0497 were assigned to the SHA-384 bug and to the cache-based timing attacks, respectively. Both CVEs were rated "high severity" and users were advised to upgrade to new releases of the code, or to disable the CBC-mode cipher suite if this is not possibles. Our understanding is that the new releases provide interim fixies, with a full solution to follow in due course.
- The GnuTLS team made a number of changes to their code on June 12th 2018 and then in releases 3.6.3, 3.5.19 and 3.3.30 on July 16th 2018. These changes address the bugs in SHA-384 constants and adopt a new variant of the pseudo constant time approach, roughly equalising the running time of decryption processing by ensuring a constant number of hash compression function calls is made. However, we believe that the GnuTLS code is still vulnerable to variants of the attacks presented in our paper due to its padding-dependent memory accesses. We notified the GnuTLS team of our concerns about this on June 13th 2018. Our understanding is that the GnuTLS team does not plan to address the issues, but prefers to promote the use of Encrypt-then-MAC (as specified in RFC 7366) when legacy cipher suites are required. Red Hat assigned CVEs 2018-10844, 2018-10845 and 2018-10846 to the issues.
- Amazon's s2n team plans to remove CBC-mode cipher suites from their list of preferred ciphers, and will replace their implementation of CBC-mode decryption with the fully constant time one from BoringSSL.

[5] https://www.wolfssl.com/docs/wolfssl-changelog/.
[6] https://tls.mbed.org/tech-updates/security-advisories/mbedtls-security-advisory-2018-02.

1.4 Paper Structure

Section 2 gives further background on the Lucky 13 attack and cache attacks, and Section 3 describes the bugs we found in the various implementations of the lucky 13 countermeasures. Next we describe the main contribution of our paper: Section 4 describes our synchronized probe PRIME+PROBE attack on Amazon's s2n implementation and Section 5 provides details on how to optimize the full byte plaintext recovery. Section 6 describes our synchronized prime PRIME+PROBE attack on mbed TLS, GnuTLS and wolfSSL. In Section 7 we introduce our novel "*PostFetch*" attack on the mbed TLS implementation. Finally, Section 8 discusses the results and raises some open questions.

2 FURTHER BACKGROUND

2.1 TLS Record Processing and the Lucky 13 Attack

For a detailed account of how TLS is processing records in CBC-mode cipher suites and how this enables the Lucky 13 attack, see [2, 3]. We present here a highly compressed version of this information heavily based on [2] in order to make the paper self-contained.

A TLS record R (viewed as a byte sequence) is processed as follows. The sender has an 8-byte per-record sequence number SQN, and forms a 5-byte field 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 $\mathsf{SQN}||\mathsf{HDR}||R$; let T denote the resulting MAC tag. The size t of the MAC tag depends on the hash function specified for use in HMAC in the cipher suite.

The record is encoded by setting $P = R||T||\mathsf{pad}$. Here pad is a sequence of padding bytes chosen such that the length of P in bytes is a multiple of the block-size b of the selected block cipher ($b = 16$ for AES). In TLS, the padding must consist of $p + 1$ copies of some byte value p, where $0 \leq p \leq 255$. Implementations typically use the last byte of pad as an indicator of the padding length to determine how many padding bytes should be present in a record and what values those bytes should take.

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. 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. The final ciphertext data has the form:

$$\mathsf{HDR}||C$$

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.

At a high level, 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 checked and removed, and finally, the MAC is checked. However, these operations must be performed without leaking any information about what the make-up of the plaintext blocks is in terms of message, MAC field and padding, and whether the format is valid. Prior literature including [2–5, 9] illustrates the difficulties of doing this securely.

As a flavour of what can go wrong, consider an attacker that wishes to decrypt a target ciphertext block C^*; let C^*_{-1} denote the preceding block in the sequence of ciphertext blocks. The attacker intercepts a ciphertext $\mathtt{HDR}||C$ and injects $\mathtt{HDR'}||C||C^*_{-1} \oplus \Delta||C^*$ so that it is received in the sequence of TLS records. Here $\mathtt{HDR'}$ is a modified header containing the correct length field and Δ is a block-size mask. A naive implementation might treat the last block of this ciphertext as containing padding, check its validity, and then either send a padding error message or extract and verify the MAC. By construction, the last block is equal to $P^* \oplus \Delta$, where P^* is the (unknown) target plaintext block. Whether or not the padding is valid therefore leaks information about $P^* \oplus \Delta$, and thence about P^*. By varying the value of Δ across different injected TLS records, the attacker can gradually build up information about P^*, possibly recovering it in its entirety.

In reality, attacks against CBC-mode in TLS are more complex than this:

- First, all errors are fatal, meaning that the connection is terminated and the key is thrown away. However, an attacker can aim to recover plaintext blocks that are repeated in predictable locations over many connections, e.g. HTTP cookies, with client-side malicious JavaScript being used to initiate the required connections and the cookies being automatically injected into connections by the victim's web browser.
- Second, the error messages are encrypted, so the attacker cannot directly learn whether or not the padding is valid. Instead, the leakage typically comes from timing information. For example, in the above discussion, we assumed that the MAC was only checked if the padding is good; of course the MAC verification will fail with overwhelming probability, and the error condition will then leak through the timing of the error message, which can be measured by an attacker located on the network.
- Third, such large timing differences are no longer present in implementations, due to patching. In particular, in view of the attacks of [9, 33], the TLS 1.1 and 1.2 specifications recommend checking the MAC even if the padding is bad, and doing so on a synthetic message whose length is equal to that of the plaintext (i.e. as if the padding had zero length). This reduces, but does not completely eliminate the timing differences; the remaining timing variation was exploited in Lucky 13 [3].
- Fourth, as the timing differences have become smaller through patching, so network noise has made mounting the attacks remotely progressively harder. This in part motivates cache-based attacks with a co-located attacker, like those in [5] and here.

2.2 Cache attacks

Cache attacks have become one of the most prolific types of attack against cryptographic primitives, using different techniques for measuring leakage of secret values (e.g. [16, 23, 30, 38]). Those different techniques were used to break real world cryptographic implementations (e.g. [6, 8, 15, 17, 19, 37, 39]). The assumption that the attacker can run code on the same platform as the target's process is now widely accepted and used, including in the recent Meltdown [22] and Spectre [20] attacks.

Some cache attacks (e.g. [38] required shared memory between the attacker and target processes. Memory might be shared between different processes or even VMs due to memory deduplication. Memory deduplication optimizations (e.g. KSM), allow two or more processes or VMs to share identical memory pages (e.g. shared library code or constants). However, due to the discovered security implications, today they are disabled between different VMs by IaaS providers [23]. Using more advanced techniques such as those in [16, 21, 23] cross-VM attacks are now practical even when memory deduplication is disabled.

Our cache attack techniques are based on the PRIME+PROBE [30] attack variant of Liu *et al.* [23] that allows cross-VM attacks. The Mastik [36] toolkit contains an implementation of this attack. We will give a short description of the general PRIME+PROBE attack (for a detailed account of the techniques see [23]). The main idea is that the access time to data that is stored in the cache is much smaller than for data that is stored in main memory. In the first PRIME phase of the attack, the attacker fills the part of the cache that will hold the target's data by accessing its own data in specific memory locations. In the second PROBE phase, the attacker tests if part of its data was evicted from the cache by measuring the access time to its own data. If all of the data is still in the cache, the target's data was not accessed. Otherwise, either the target's data was accessed, or some other code forced the eviction of the attacker's data. If the target's code access pattern to its data is determined by some secret value, the attacker can learn this value.

3 IMPLEMENTATION BUGS IN LUCKY13 COUNTERMEASURES

Pseudo constant time countermeasures are very hard to get right and maintain over time. This is due both to the possibility of finding novel variants of the original attacks, and the need to manually check the timing implications of adding new features. In contrast, real constant time implementations are more robust against novel attack variants, and bugs created by supporting new features will likely be found by unit-testing. TLS 1.2 [12] added new ciphers suites based on CBC-mode for encryption and HMAC-SHA-384 for integrity. The SHA-384 hash function is considered more secure, and also has better performance on 64-bit processors, than the previously supported SHA-1 and SHA-256 algorithms. We tested if TLS implementations supporting HMAC-SHA-384 are vulnerable to timing attacks similar to the one described in [2]. All of the constant-time implementations that we checked (OpenSSL, BoringSSL, NSS) were secure. However, all of the "pseudo" constant time implementations (i.e. those only ensuring a constant number of compression function calls) had bugs making them vulnerable to attack. The reason for the bugs is that, although the SHA-384

cipher suites were added, the code responsible for adding dummy compression function calls was not updated correctly. Specifically, SHA-384 has a 128-byte block size (compared to the 64-byte blocks of SHA-256), and encodes the message length using 16 bytes (compared to 8 bytes in SHA-256). All of the extra compression function call calculations have hard-coded values appropriate for SHA-256 but not SHA-384, resulting in them using a non-constant number of calls to the SHA-384 compression function. We explain in more detail below for each of the four "pseudo" constant time implementations we studied; since the bugs are easily fixed, we do not go into great detail on how each bug leads to a plaintext recovery attack.

3.1 GnuTLS Implementation

Although the function `dummy_wait` (see Listing 3 in Appendix A) uses the correct hash block size, it also uses the hard-coded number "9". This comes from at least 1 byte for the the hash function padding and the 8 bytes used to encode the hashed message length for SHA-256. However, in SHA-384, the message length is encoded using 16 bytes, and so the correct value should be "17" rather than "9". The code includes a comment warning that this is a hash-specific fix, but it was apparently not corrected when the SHA-384 cipher suites were added.

Even more surprisingly, we discovered that the GnuTLS Implementation is vulnerable to a timing attack when SHA-256 is selected as the hash algorithm in HMAC, despite this having been patched in response to Lucky 13 [27]. The function `dummy_wait` can add at most one call to the hash compression function. However, the attack described in Section 6.3 creates a padding oracle that distinguishes between a valid pad of a large length (PadLen $>$ 240) and an invalid padding (PadLen = 0). In that case, for GnuTLS, there will be a timing difference of 3 calls to the compression function of SHA-256, that is 3 times larger than the timing difference in the original Lucky 13 attack [3].

3.2 mbed TLS Implementation

The function `ssl_decrypt_buf` (see Listing 8 in Appendix A) uses the hard-coded value "64" (for block size) and "8" (for message length encoding). These should be "128" and "16", respectively, for SHA-384. For example, HMAC verification of a decrypted TLS record of length 512 and valid padding of length in the range PadLen = 229 will result in **3 more** compression functions call than same length TLS record with invalid padding (PadLen = 0). Again we can use the attack described in Section 6.3 to create a padding oracle to distinguish between the two cases, resulting in a timing difference much larger than the one in the original Lucky 13 attack [3].

3.3 WolfSSL Implementation

The function `GetRounds` (see Listing 5 in Appendix A) uses the hard-coded numbers "64" and "55" (64-8-1). These should be "128" and "111", respectively, for SHA-384. The same attack described in Section 3.2can also be used against the WolfSSL implementation.

3.4 Amazon's s2n Implementation

The `s2n_hmac_digest_two_compression_rounds` function (see Listing 1) can add one dummy compression function call. The calculation of the condition uses the hard-coded number 9 as the minimal number of bytes to add, whereas 17 would be appropriate for SHA-384. This bug was not detected during the formal verification of the HMAC code carried out by Galois [13]. However, unlike our new cache attack for s2n presented in Section 4, the attack arising from this bug (modeled on that in [2]) is likely to be impractical due to the random delay protection in s2n.

4 A CACHE-BASED PADDING ORACLE IN AMAZON'S S2N IMPLEMENTATION

Amazon's s2n TLS implementation is responsible for protecting all of the traffic to Amazon's S3 cloud storage service [31]. This implementation was previously analysed by Albrecht and Paterson [2] and found vulnerable to a variant of the Lucky 13 attack. The current protection includes a pseudo constant time implementation, and the inclusion of a very high resolution and large random delay after detecting any error in TLS decryption. This causes previous timing attacks to become impractical. Moreover, the correctness of s2n's patched HMAC implementation was formally verified [13].

However, as we will see, the memory access pattern in s2n depends on the padding length byte (i.e. the last byte of the decrypted TLS record). We will use a PRIME+PROBE [30] cache attack to build a new padding oracle for s2n. We assume a cache side-channel as in Liu *et al.* [23], and describe two versions of our attack on s2n: In a simplified version, we target the specific code written to block the attack in [2]. However, this attack is not practical due to an *ad hoc* programming decision in s2n. In our full synchronized probe PRIME+PROBE attack, we exploit the same programming decision, but using the probability of cache hits and misses as an indicator of padding length. Our full attack works on HMAC using both SHA-384 and SHA-256, even if the simple bug described in Section 3.4 is fixed.

4.1 Attack Preliminaries

In both attacks the cache side-channel arises from the access pattern to a dynamically allocated memory location, more specifically a buffer used to store part of the key in the HMAC calculation. We first have to find the mapping of this location to the right cache set, by exploiting a design decision of the s2n developers: All the structures and memory buffers required for a specific connection in s2n are allocated in the handshake phase and are reused for all messages. We can then find the right cache set in the same manner as in [23]. For each handshake, we trace the cache set while processing valid messages, and find the cache set exhibiting the activity pattern we expect for the HMAC code.

4.2 Simplified Attack

The attack described in [2] is based on the following fact: If we split a message into two parts, and hash each of them separately, the number of calls to the internal hash compression function might vary depending on the split point. This is due to the padding and length bytes added internally by the hash function. A new function `s2n_hmac_digest_two_compression_rounds` (see Listing 1 in Appendix A) was added to the HMAC API in s2n to block this attack vector. This function makes two calls to the internal hash compression function, even if the hash padding doesn't necessitate it. The function checks if the hash padding will require another

Algorithm 1 s2n Simplified Attack

```
1:  function SimplifiedS2NPadOracle(valid_msg,attack_msg)
2:      xor_pad ← FindXorPadCache(valid_msg)
3:      Prime(xor_pad)                    ▷ evict xor_pad set from cache
4:      Send attacker's TLS record to target
5:      Wait for verification error
6:      if Probe(xor_pad) then
7:          return 1                          ▷ buffer was accessed
8:      else
9:          return 0                      ▷ buffer was not accessed
10:     end if
11: end function
```

compression call. If not, it will reset the hash context and call another update function. In that case, the buffer that is sent to the update function is the HMAC state buffer called `xor_pad`. The only other place this buffer is used is in the HMAC initialization function, and that is called only once, in the TLS handshake.

Our attack (see Algorithm 1) is now straightforward. The attack code runs on a different VM on the same CPU socket. We start by finding the cache set of the *xor_pad* buffer. This is done by sending valid TLS records to the target (for example, by using malicious JavaScript running in a remote victim's browser), and using the PRIME+PROBE technique of [23] to find the correct cache set. We then prime the *xor_pad* buffer, and send the attacker-constructed TLS record for decryption. After the TLS record is rejected (due to a MAC failure) we probe the cache set. If we get a cache miss, we assume that with high probability the *xor_pad* buffer was accessed, and so the TLS record's padding length (as determined by s2n from PadLen, the value of the last byte of the decrypted TLS record $P = R||T||\text{pad}$) is such that the length of R mod `hash_block_size` is between `hash_block_size` $- 9 - 13$ and `hash_block_size` $- 13$ (here, 13 comes from the TLS header length and "9" would be replaced by "17" for SHA-384 if the bug identified in Section 3.4 were to be fixed). By setting the attacker's TLS record so that its last blocks are $C^*_{-1} \oplus \Delta||C^*$, as in Section 2.1, information about the value of the last byte of $P^* \oplus \Delta$ is thereby leaked to the attacker.

4.3 Full synchronized probe PRIME+PROBE Attack

The simplified attack is not practical due to the following *ad hoc* programming decision in the verification function s2n_verify_cbc (see Listing 2 in Appendix A). After finishing the HMAC calculation, the s2n code hashes the rest of the TLS record padding bytes, to ensure a constant number of compression function calls. In this specific solution it is required to "remember" the number of bytes digested up to this point. To allow this, the function `s2n_hmac_copy` was added to the HMAC API. This function (used only in CBC-mode processing) copies all of the state buffers of the HMAC calculation, so that the copy can be used to digest the remaining padding bytes. The copy function also copies the `xor_pad` buffer (although it is not required for the calculation), and so it accidentally causes it to be read into the cache. For the simplified attack to work, we would need to arrange for the probing of the `xor_pad` buffer to happen exactly in-between the call to `s2n_hmac_copy` and `s2n_hmac_digest_two_compression_rounds`. This requires too fine a control over timing.

Algorithm 2 s2n synchronized probe PRIME+PROBE Attack

```
1:  function S2NPadOracle(valid_msg,attack_msg)
2:      copy_xor_pad ← FindCopyXorPadCache(valid_msg)
3:      Prime(copy_xor_pad) ▷ evict copy_xor_pad set from cache
4:      Send attacker's TLS record to target
5:      Delay to synchronize the probe
6:      if Probe(copy_xor_pad) then
7:          return 1                          ▷ buffer was accessed
8:      else
9:          return 0                      ▷ buffer was not accessed
10:     end if
11: end function
```

However, we can use the fact that the HMAC copy buffer is only accessed in the `s2n_hmac_copy` function, just after finishing the HMAC calculation over the message. The time elapsed until this buffer is accessed is actually the same as the time taken for HMAC verification in the Lucky 13 attack.

The full synchronized probe PRIME+PROBE attack (see Algorithm 2) tries to approximate the HMAC execution time by using a "race condition" between the message verification and the cache probing. We start again by finding the correct cache set for the *copy_xor_pad*, using valid messages. Then we prime the cache set, and send the TLS record for decryption. We use a short delay to synchronize the attack, as our probing step should run at approximately the same time as the HMAC verification code would finish hashing a short message (corresponding to a long TLS padding pattern).

We assume that there will be small timing variations due to the behaviour of the operating system. So we run the attack multiple times, and use the probability of a cache miss as an indicator of the HMAC execution time. If the probability of a cache miss is high (so Algorithm 2 outputs "0" frequently), it indicates that we have probed before the call to `s2n_hmac_copy`. This means that the HMAC execution took longer, and so we can infer that PadLen was small. On the other hand, if the probability of a cache hit is high, it indicates that PadLen was large. We will shortly make this analysis more precise.

4.4 s2n's Timing Blinding Mitigation

Amazon added a general mitigation to protect s2n from attacks targeting their non-constant time implementation. In the event of a decryption error, the function `s2n_connection_kill` adds a very large random time delay before killing the connection and sending the error message. This is supposed to add a large amount of noise to any timing-based attack, making it impractical [24]. While this random delay can indeed block regular timing attacks, it offers no protection against our cache-based attack, as the cache access that we target is made before the random delay is added. Moreover, since a server running s2n can support many concurrent connections, the random delay does not significantly slow down the rate at which we can send attack TLS records.

4.5 s2n Proof of Concept

We experimentally implemented a PoC for the attack. We ran our attack on an Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz running Ubuntu 17.10. We used the code from the master branch of the

Hash function	Message Length	PadLen range	Cache hit probability
SHA-384	720	$0-44$	≈ 0.68
SHA-384	720	$45-172$	≈ 0.75
SHA-384	720	$173-255$	≈ 0.97
SHA-256	576	$0-44$	≈ 0.63
SHA-256	576	$45-108$	≈ 0.65
SHA-256	576	$109-172$	≈ 0.66
SHA-256	576	$173-236$	≈ 0.90
SHA-256	576	$237-255$	≈ 0.95

Table 2: Cache hit probabilities for s2n attack

official s2n git repository on 14/2/2018 (commit hash f742802), and compiled with the provided make files and GCC version 7.2.0. We targeted the code of the `s2n_verify_cbc` function (including the code for hashing the header and sequence number called before the function). The function is called to verify multiple messages that differ only in the last byte of the decrypted TLS record (which is used to set PadLen). Another thread was run in parallel to evaluate the cache hit/miss probability.

We studied HMAC verification for both SHA-384 and SHA-256. We chose a random 720-byte string (576 bytes for SHA-256) as our decrypted TLS record (the record length must be a multiple of the cipher block-size, and these sizes are optimal for our attack), and ran 2000 trials of the attack for all 256 possible padding length values in the last byte. As expected the hit probability changes on the 128-byte boundaries of the hashed data (64-byte boundaries for SHA-256). Since the number of bytes hashed until the call to `s2n_hmac_copy` is made is:

$$\begin{aligned}\text{HashLen} &= \text{InnerHashKeyLen} + \text{SeqNumLen} + \text{HdrLen} \\ &\quad + \text{DecMsgLen} - \text{MacLen} - \text{PadLen} - 1, \quad \text{then} \\ \text{HashLen}_{\text{SHA-384}} &= 128 + 8 + 5 + 720 - 48 - \text{PadLen} - 1 \\ &= 812 - \text{PadLen}, \quad \text{and} \\ \text{HashLen}_{\text{SHA-256}} &= 64 + 8 + 5 + 576 - 32 - \text{PadLen} - 1 \\ &= 620 - \text{PadLen}\end{aligned}$$

Due to the inclusion of the underlying hash length field and padding in this hashing operation, we expect to see an increase in the cache hit probability when PadLen $\equiv 45 \bmod 128$ ($45 \bmod 64$ for SHA-256). The experimental results in Table 2 show these expected changes. We synchronized the attack so that the cache probe is expected to happen after ≈ 5 calls to the SHA-384 compression function (≈ 7 for SHA-256). So we should get a cache hit with high probability if the value of PadLen is greater than 172, and with low probability otherwise. This can be seen in the table. There is also a smaller change in the probability at value 44 (48, 108 and 236 for SHA-256). When the pad length is less than 44, there is one less compression function call, which causes the probability of a cache hit to be even lower.

4.6 Creating the Padding Oracle

Our experiments show that s2n permits a single-bit oracle that can distinguish if PadLen > 172 or not. Recall that PadLen is set from the last byte of the decrypted TLS record $P = R||T||\mathsf{pad}$, and the method described in Section 2.1 can use such an oracle to learn information about the value of the last byte of $P^* \oplus \Delta$ for attacker-controlled values Δ. We build this oracle by repeatedly running Algorithm 2 and estimating the cache hit probability p. We then use this estimate to decide whether PadLen > 172 or not. The accuracy of this process is determined by the difference in probabilities and the number of iterations n of Algorithm 2 that we perform; we are effectively trying to distinguish between two binomial distributions, one with $p = 0.75$ ($p = 0.66$ for SHA-256) and the other with $p = 0.97$ ($p = 0.90$ for SHA-256). We set a threshold probability of $p_t = 0.86$ ($p_t = 0.78$ for SHA-256) and set a desired error probability of β. We can then calculate n, the required number of iterations, such that $\Pr(\text{Bin}(n, 0.75) > 0.86n) < \beta$ and $\Pr(\text{Bin}(n, 0.97) < 0.86n) < \beta$. We can calculate the required n for any chosen β by using the CDF of the binomial distribution.

In fact, we can generate different oracles by changing the length of the TLS records that we send to our target. The body of the records must have lengths that are multiples of the block-size. This allows us to obtain oracles of the form PadLen $> 12 + 16k \bmod 256$ for any k. In Section 5 we will show how to choose n, β and the optimal k to use in the oracle for achieving full plaintext recovery.

5 FROM S2N PADDING ORACLE TO FULL PLAINTEXT RECOVERY

The attack in Section 4 on s2n provides us with one-bit linear condition oracles on the padding length byte that is located at the end of the decrypted TLS record.[7]

However, our goal is to recover multiple, full bytes of plaintext. Fortunately, we can select the last blocks of the attacker's TLS record as $C^*_{-1} \oplus \Delta||C^*$ for different values of mask Δ, and post-process the results to gain information about the values of plaintext block P^*; more precisely, since the linear conditions are always on the very last byte p^* of P^*, we need only vary Δ in its last byte position δ, and we can only gain information about p^*. In this section, we describe two strategies for selecting the different single-byte masks δ to try: a naive approach, and a more sophisticated one.

We remark here that going from single-byte recovery to many-byte recovery can be achieved in the main application scenario of recovering HTTP cookies. The idea is to use progressively longer padding of pathnames in HTTP requests to move the target HTTP cookie bytes one-by-one into the last position p^* in the target block P^*. The HTTP requests are produced by malicious JavaScript running in the victim's browser; the browser automatically generates the required TLS connections in response to the requests. This is a known technique that we borrow from the literature on TLS attacks [3, 14, 29], and we do not comment on it further here.

5.1 Naive Algorithm

The naive algorithm is described in Algorithm 3. The algorithm receives the following parameters:

(1) OracleFunc(δ): a function that implements a padding oracle attack for mask value δ. This function carries out one of the padding oracle attacks from the previous sections, targeting a particular fixed byte p^* in the last position in some target plaintext block P^*. This involves repeatedly intercepting TLS

[7] We actually have several different conditions and oracles, depending on the attacker-controlled TLS record length.

Algorithm 3 Padding Oracle to Plaintext Byte — Naive Algorithm

```
1: function NaiveOracleToByte(OracleFunc, OracleCondition)
2:     ValList ← [0..255]                ▷ all possible values for PadLen
3:     MaskList ← [0..255] ▷ all possible one-byte mask values δ
4:     for all δ in MaskList do
5:         OracleRes ← OracleFunc(δ)
6:         for all val in ValList do
7:             ValRes ← OracleCondition(val ⊕ δ)
8:             if ValRes ≠ OracleRes then
9:                 remove val from ValList
10:            end if
11:        end for
12:        if Length(ValList)==1 then
13:            return ValList[0]
14:        end if
15:    end for
16: end function
```

records containing the target byte/block p^*/P^* in ciphertext block C^*, building fixed-length TLS records ending with $C^*_{-1} \oplus \Delta || C^*$ where the last byte of Δ is set to δ, and recovering the result of evaluating the oracle's one-bit linear condition on input $p^* \oplus \delta$. We assume that the oracle has error probability β when n iterations are carried out.

(2) OracleCondition(PadLen): returns the result of the linear condition executed by the above oracle, assuming that the value PadLen is used as input (along with some fixed TLS record length). For example, the oracle may return the value of the predicate "PadLen > 172".

The algorithm starts by initializing two byte-lists ValList and MaskList with all possible byte values. We iterate over all possible single-byte mask values δ. For each possible δ, we get the result of the oracle by running OracleFunc(δ). We then iterate over the possible values in ValList. For each possible val, we check if the linear condition on val $\oplus$ δ is equal to the result of the oracle. If not, the value is discarded (and this will be a correct decision with probability $1 - \beta$). The algorithm ends when there is only one possible value remaining in ValList.

The complexity of the algorithm is dominated by the number of calls to the OracleFunc($\cdot$) function. The expected number is 128, with a worst case of 256. Recall, however, that each call to this oracle involves some number n executions of the underlying cache timing attack, giving an average of $128n$ executions of the cache timing attack. Moreover, each execution of the underlying attack consumes a TLS connection (since the attacker's constructed ciphertext will always fail HMAC verification). For example, as we will see in Section 5.3, this results in roughly 13000 runs of the synchronized probe PRIME+PROBE attack in Algorithm 2 for the attack on s2n. This figure of $128n$ might make the attack impractical. For this reason, we developed an improved greedy algorithm which can reduce the attack complexity by a factor of more than 50. We present this next.

5.2 Greedy Algorithm

Our greedy algorithm optimizes the way in which we choose the masks δ that we use in our oracle calls. Instead of iterating over all possible values, in each iteration of the attack we choose as the next mask the one that will give us the most information (maximizing the entropy of each oracle call). Algorithm 4 chooses the "best" mask in a greedy manner. It takes as input a list of all remaining possible byte values (ValList) and mask values (MaskList). For each mask it simulates all the oracle responses on the possible values remaining, and chooses the mask that maximizes the entropy of this experiment. Since we only consider single bytes at a time, the algorithm is efficient. In a further optimization, we can also remove the masks with zero entropy from MaskList in each iteration.

Algorithm 4 Padding Oracle to Plaintext Byte — Greedy Algorithm

```
1: function GetBestMask(MaskList, ValList, OracleCondition)
2:     HalfLenValList ← Length(ValList)/2
3:     MinMaskCount ← 256              ▷ maximum possible value
4:     BestMask ← 0
5:     for all δ in MaskList do
6:         OneCount ← 0
7:         for all val in ValList do
8:             ValRes ← OracleCondition(val ⊕ δ)
9:             if ValRes == 1 then
10:                OneCount ← OneCount + 1
11:            end if
12:        end for
13:        Count = |HalfLenValList − OneCount|   ▷ how far are we from half the values returning 1 and half returning 0?
14:        if Count ≤ MinMaskCount then
15:            BestMask ← δ
16:            MinMaskCount ← Count
17:        end if
18:    end for
19: end function
```

5.3 Application to Amazon s2n

In our attack we can obtain oracles of the form "PadLen > $12 + 16k \bmod 256$" for any k. We will analyze the run time complexity of the SHA-384 version of the attack.

5.3.1 Attack complexity of the naive algorithm. For the naive algorithm, the expected number of runs is 128, with a worst case of 256. We focus for the moment on successfully recovering a single byte of plaintext with probability $p_B > 0.5$ (a standard requirement for success probability in cryptanalysis). In the worst case we will require 256 correct calls to the oracle. So we would need each oracle call to return the correct result with $\beta = 0.5^{1/256} = 0.9973$. Based on our experiments, we model the s2n SHA-384 attack cache hit distribution as a binomial distribution with $p = 0.75$ for PadLen $\leq$ 172 and $p = 0.97$ for PadLen $>$ 172. We would then need $n = 100$ executions of the cache attack to distinguish between the two distributions with probability larger than $\beta = 0.9973$. The expected total number of cache attack executions needed for the naive attack is then 12800 with a worst case of 25600.

5.3.2 Attack complexity of the greedy algorithm. We simulated the complexity of the greedy algorithm for all oracles of the form PadLen > $12 + 16k \bmod 256$. The oracle with the lowest complexity is the one with $k = 10$, where the condition is PadLen $>$ 172, having an expected number of 8.5 runs and a worst case of 11.

This is very close to the information theoretical lower bound of 8 runs.. To achieve $p_B > 0.5$ this requires $\beta = 0.5^{1/11} = 0.939$, which translates to $n = 28$. So the expected total number of cache attack executions needed for the greedy attack is only 238, with a worst case of 308. This is more than 50 times less than needed in the naive attack.

5.3.3 Recovering multiple bytes. We are typically interested in recovering multiple plaintext bytes, e.g. an entire 16-byte cookie, with good probability. To achieve success probability greater than 0.5 across 16 bytes, we need a per-byte probability of $p_B \approx 0.96$, which in the greedy attack yields $\beta = 0.5^{1/11 \cdot 16} = 0.996$. In turn, this translates to $n = 92$ and an expected total number of TLS connections of 782 per byte. For comparison, the naive algorithm requires an expected total of 21900 TLS connections per byte.

6 A PADDING ORACLE BASED ON TLS RECORD CACHE ACCESS PATTERN

The original Lucky 13 attack [3] exploited the time difference of the TLS record verification process for valid and invalid padding. As a mitigation, all the pseudo constant time TLS implementations added dummy compression function calls that cause the total number of compression function calls to be independent of the padding length. However, unlike proper constant time TLS implementations, the cache access pattern to the data structure holding the TLS record is still dependent on the padding length. We can exploit this cache access pattern with our novel synchronized prime PRIME+PROBE attack to restore the original padding oracle of [3] in several TLS implementations – mbed TLS , GnuTLS and wolfSSL . All implementations were patched against Lucky 13 [3]. Moreover, mbed TLS was also patched against a second cache-based attack by Irazoqui *et al.* [5], targeting the code of the first patch. We will again use a PRIME+PROBE [30] cache attack, assuming a cache side channel as in Liu et.al. [23]. Our attack works on HMAC using both SHA-384 and SHA-256, even if all the bugs in Section 3were fixed, and all the previous variants of Lucky 13 were patched correctly.

6.1 Attack Preliminaries

All of the vulnerable implementations follow this general code flow for constant time decryption:

(1) Decrypt the message, accessing all the bytes in the TLS record.
(2) Perform constant time checking of the TLS record padding, assuming zero-length padding if the padding is not valid. All of the final 256 bytes of the TLS record are accessed.
(3) Calculate HMAC on the decrypted TLS record payload (excluding the padding). All bytes in the decrypted TLS record are accessed, except for the padding bytes at the end.
(4) Add extra dummy compression function calls to make the number of calls the same in every case. The data input to these function calls is obtained from the start of the TLS record or from a dummy memory buffer. The padding bytes of the TLS record are not accessed (except for messages that are shorter than the hash block size).

In our attack, we will try to distinguish between two cases: long valid padding and long invalid padding. We will explain in Section 6.3 how an oracle yielding this information can be used to recover plaintext bytes. Consider the cache access pattern from the beginning of the HMAC verification. In the case of invalid padding, the code typically assumes zero-length padding, and the HMAC verification will access all of the TLS record bytes (possibly excluding the last byte). However, if the padding is valid and long (e.g. PadLen = 255, in which case there are 256 bytes of padding), the HMAC verification will not access the last PadLen + 1 bytes of the TLS record.

Algorithm 5 Message Access Attack

```
1:  function MessageAccessPadOracle(Valid TLS records, Attack TLS record)
2:      LastBytesCache ← FindPtrCache(Valid TLS record[End])
3:      Send attacker's TLS record to target
4:      Delay to synchronize to the start of the HMAC verification
5:      Prime(MsgCache)              ▷ evict end of record from cache
6:      Delay till maximum time for HMAC calculation
7:      if Probe(MsgCache) then      ▷ end of record was accessed
8:          return 0                 ▷ padding was invalid
9:      else
10:         return 1                 ▷ padding was valid
11:     end if
12: end function
```

6.2 synchronized prime PRIME+PROBE Attack Description

Our synchronized prime PRIME+PROBE attack exploits this difference in the access pattern using a PRIME+PROBE [30] cache attack. We synchronize the PRIME part of the attack to run in parallel to the HMAC verification process, that is, after the padding check but before the HMAC verification is done. The maximum TLS record size is ca. 2^{14} bytes, corresponding to about 2^8 compression function calls for a 64-byte hash block size. By working with ciphertexts of this size, we can force the HMAC verification to take a long time to complete. This makes the synchronization of the attack relatively easy.

The attack (described in Algorithm 5) has four main parts:

(1) Finding the cache sets containing the last bytes of the TLS record.
(2) Sending the attack TLS record. The TLS record is constructed to have long valid padding, except possibly in the first padding byte. This is the byte we try to recover in the attack.
(3) Delaying till the HMAC verification begins. This occurs after the decryption and padding check is finished (and takes a constant amount of time regardless of the padding).
(4) In parallel to the HMAC verification, we Prime the end of the TLS record to evict it from the cache.
(5) After the end of the HMAC verification calculation, we probe the cache set that contains the last few bytes of the TLS record. If it was accessed, then with high probability the padding was invalid; otherwise it was valid.

6.3 Constructing an Attack on TLS Records

It remains to explain how we construct the TLS records used in the attack, and how we use the results of the oracle to recover plaintext bytes (HTTP cookie bytes in this case). We rely on techniques first

explained in [14]: we use HTTP pathname padding and the ability to choose plaintext bytes that are placed after the cookie in the HTTP request to ensure that the plaintext in the TLS record contains 16 consecutive blocks in which the first block has the form:

$$p^*||\text{“\r”}||\text{“\n”}||\texttt{0xFF}||\ldots||\texttt{0xFF},$$

and the remaining 15 blocks consist solely of values 0xFF. Here p^* is the last byte of the cookie and the target of the first step of the attack, while “\r”, “\n” represent ASCII characters inserted after the cookie by HTTP. Note that these plaintext blocks are *almost* correct padding of maximum length; of course they are incorporated into a TLS record containing an HMAC tag and correct TLS padding. Let $C^*_0, \ldots, C^*_{15}$ denote the matching ciphertext blocks in the resulting TLS record, and let C^*_{-1} denote the preceding ciphertext block.

The attack TLS record is then constructed as:

$$\texttt{HDR}||L||C^*_{-1} \oplus \Delta||C^*_0||\ldots||C^*_{15}$$

where HDR is a suitable header, L is a long random block sequence that brings the TLS record up to the maximum size, and Δ is a mask with bytes:

$$\delta||(\text{“\r”} \oplus \texttt{0xFF})||(\text{“\n”} \oplus \texttt{0xFF})||\texttt{0x00}||\ldots||\texttt{0x00}.$$

Here, the first mask byte creates a value $p^* \oplus \delta$ in the first position of the block decrypting C^*_0, while the second and third mask bytes force values 0xFF in the corresponding positions. Clearly, when decrypted, this TLS record will have correct padding of length 256 if and only if $p^* \oplus \delta = \texttt{0xFF}$. The attacker then uses TLS records of this form with distinct values of δ in the attack of Algorithm 5; after 128 attempts on average and 256 in the worst case, the value of δ producing correct padding will be identified.

This description explains how to recover the last byte of the cookie. Further bytes can be recovered by shifting the position of the cookie by altering the length of the pathname in the HTTP request so that the last 2, 3, ... bytes are present at the start of the block underlying C^*_0. We also update Δ as needed to force correct padding 0xFF in all but the first byte of this block. This approach will recover up to 14 bytes of the cookie; the remaining bytes seems to remain inaccessible using these techniques (trying to extend further would push the “\r” and “\n” characters into the next block, where they could not be turned into correct padding by XOR masking).

We close this description by noting that the above attack with long padding patterns can be applied to the original Lucky 13 setting, quadrupling the timing differences there and so making them substantially easier to detect (at the cost of limiting how much plaintext can be recovered). This enhancement to Lucky 13 seems to have been missed by the authors of [3], though they used a similar idea in their distinguishing attack.

6.4 Proof of Concept for synchronized prime PRIME+PROBE attack

We implemented a PoC for the above attack for wolfSSL, to verify the presence of the cache side-channel. We ran our attack on an Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz running Ubuntu 17.10. We used the version 3.14 code taken from the master branch of the official git repository on 20/4/2018 (commit hash 7d425a5c), and compiled with the provided make files and GCC version 7.2.0.

6.4.1 wolfSSL code. The TLS record verification is done in function TimingPadVerify (see Listing 5 in Appendix A). First, the padding is checked by the function PadCheck. If the padding check fails, the branch taken by the code implicitly assumes PadLen = 0 and HMAC is calculated over the whole TLS record excluding the last byte (that is assumed to be the minimal length padding). However, if the padding is valid, HMAC is calculated on the TLS record excluding all of the padding bytes. To achieve constant time, the extra compression function CompressRounds is called. However, this is done with the dummy array, which points to the start of the ssl context. So in case of valid padding the last bytes of the TLS record are not accessed.

6.4.2 Attack results. We prepared two types of decrypted TLS records. The first one with 256 bytes of valid padding and the second one being identical, except for the first byte of padding which was changed to a different value. We called the TimingPadVerify function multiple times with both types of data. Before each call to the function, another “attack” thread was run in parallel to perform the cache priming during the HMAC verification. After the function returned, we checked if the cache line of the last bytes in the TLS record is in the cache or not. For TLS records with valid padding we saw a hit probability of ≈ 0.025. For TLS records with invalid padding we saw a hit probability of ≈ 0.998. Using the same calculations as in Section 5.3.3 this translates to $n = 4$ and an expected total number of TLS connections of 512 per byte.

6.4.3 mbed TLS. The same vulnerability also applies to the mbed TLS code in function ssl_decrypt_buf (see Listing 8 in Appendix A). If the padding is invalid the function sets the variable padlen to 0. For constant time, the extra compression function mbedtls_md_process is called multiple times with pointer in_msg pointing to the start of the TLS record.

6.4.4 GnuTLS. The same vulnerability also applies to the GnuTLS code in function decrypt_packet (see Listing 4 in Appendix A). If the padding is invalid the function sets pad to 0. For constant time, extra compression functions are executed via dummy_wait (see Listing 3 in Appendix A). If the padding was invalid, the function does nothing and returns. If the padding was valid, but the HMAC verification fails, the extra compression function _gnutls_auth_cipher_add_auth is called multiple times with the pointer data, pointing to the start of the TLS record.

Note that unlike other implementations, the extra compression functions are only called when the verification process fails, so the decryption time on valid messages is not constant. This may leak the real size of the encrypted messages, but cannot be used to recover plaintext bytes.

7 A CACHE-BASED PADDING ORACLE IN THE MBED TLS IMPLEMENTATION

We will describe another novel attack on mbed TLS targeting the inner hash function execution in HMAC. This attack is more robust than the one described in Section 6 as it does not require the synchronization between the attack and target code. At first we will describe a simplified version that assumes a small cache line size. The full “*PostFetch*” attack will show how we can deal with modern cache line sizes and memory prefetching. Our attack works on

HMAC with both SHA-384 and SHA-256, even if the bug described in Section 3.2is fixed.

7.1 Attack Preliminaries

HMAC makes two hashing passes over its input message, which we refer to as the *inner* and *outer* hashes. The inner hash processes a string of the form $K_1||M$ to produce a hash value h; the outer hash processes an input $K_2||h$. Here K_1, K_2 are keys derived from a single key by XOR offsets and M is the message input. The hash functions used in HMAC in TLS are based on the Merkle-Damgård construction [28]. This construction pads the message being processed to a multiple of the hash function's block size. The usual hash function padding scheme is to always add the byte $0x80$ and then zero bytes up to the required length.[8]

In mbed TLS the hash padding is implemented by defining a constant array containing the maximum possible length hash padding pattern, and passing this array with the required padding length to the hash update function (see Listing 7 in Appendix A; SHA-384 is simply a truncated output of SHA512). Our cache attack targets the access pattern to this constant array to create a padding oracle. A maximum hash padding length will cause the entire array to be saved in the cache, while a short hash padding will cause only the beginning of the array to be saved in the cache. To check if parts of the array are in the cache or not we can use cache attacks that exploit shared memory pages. If the attack code runs on the same core as the target code we can use a simple PRIME+PROBE attack on the L1 cache [30]. However, if the code runs on different cores (as is the case in most cross-VM attacks) we can use more advanced cross-core attacks [16, 21, 23]. For brevity and without loss of generality we will use the PRIME+PROBE notation.

7.2 Hash Padding Length for SHA-384

For the inner hash calculation of HMAC-SHA-384, the length of the hash padding for an encrypted message with length EncMsgLen is calculated in the following way:

$$\begin{aligned}\text{HashLen} &= \text{InnerHashKeyLen} + \text{SeqNumLen} + \text{HdrLen} \\ &\quad + \text{EncMsgLen} - \text{MacLen} - \text{IVLen} - \text{PadLen} - 1 \\ &= 128 + 8 + 5 + \text{EncMsgLen} - 48 - 16 - \text{PadLen} - 1 \\ &= \text{EncMsgLen} + 76 - \text{PadLen}, \quad \text{hence:} \\ \text{HashPadLen} &= 112 - \text{HashLen} \mod 128 \\ &= 36 - \text{EncMsgLen} + \text{PadLen} \bmod 128 \end{aligned} \tag{1}$$

Note that if $\text{HashPadLen} = 0 \bmod 128$ then $\text{HashPadLen} = 128$. The length of the hash padding for the outer hash calculation in HMAC-SHA-384 is calculated via:

$$\begin{aligned}\text{HashLen} &= \text{OuterHashKeyLen} + \text{HashLen} = 128 + 48 = 176 \\ \text{HashPadLen} &= 112 - \text{HashLen} \bmod 128 = 64\end{aligned}$$

So the the number of hash padding bytes accessed in the HMAC calculation is given by:

$$\text{HashPadLen} = \max(36 - \text{EncMsgLen} + \text{PadLen} \bmod 128, 64)$$

[8] This hash padding is distinct from the padding added by TLS in CBC-mode and which is actually transmitted as part of TLS records.

Algorithm 6 mbed TLS – Simplified Attack

```
1:  function SimplifiedMbedPadOracle(attack_msg)
2:      ProbeOffset ← 64                          ▷ 32 for SHA-256
3:      Prime shaX_padding + ProbeOffset          ▷ evict from cache
4:      Send attacker's TLS record to target
5:      if Probe(shaX_padding + ProbeOffset) then
6:          return 1          ▷ last part of the array was accessed
7:      else
8:          return 0      ▷ last part of the array was not accessed
9:      end if
10: end function
```

7.3 Hash Padding Length for SHA-256

For the inner hash calculation of HMAC based on SHA-256, the length of the hash padding for an encrypted message with length *EncMsgLen* is calculated in the following way:

$$\begin{aligned}\text{HashLen} &= \text{InnerHashKeyLen} + \text{SeqNumLen} + \text{HdrLen} \\ &\quad + \text{EncMsgLen} - \text{MacLen} - \text{IVLen} - \text{PadLen} - 1 \\ &= 64 + 8 + 5 + \text{EncMsgLen} - 32 - 16 - \text{PadLen} - 1 \\ &= \text{EncMsgLen} + 28 - \text{PadLen}, \quad \text{hence:} \\ \text{HashPadLen} &= 56 - \text{HashLen} \mod 64 \\ &= 28 - \text{EncMsgLen} + \text{PadLen} \bmod 64 \end{aligned} \tag{2}$$

Note that if $\text{HashPadLen} = 0 \bmod 64$ then $\text{HashPadLen} = 64$. For the outer hash calculation of HMAC-SHA-256, we have:

$$\begin{aligned}\text{HashLen} &= \text{OuterHashKeyLen} + \text{HashLen} = 64 + 32 = 96 \\ \text{HashPadLen} &= 56 - \text{HashLen} \bmod 64 = 24\end{aligned}$$

So the number of hash padding bytes accessed in the HMAC calculation is given by:

$$\text{HashPadLen} = \max(28 - \text{EncMsgLen} + \text{PadLen} \bmod 64, 24)$$

7.4 Simplified Attack

The simplified attack on mbed TLS is described in Algorithm 6. The start of the pad array is always accessed by the outer hash calculation of HMAC. We prime a cache set that contains the array at an offset, targeting the first cache line that is not always accessed (offset of 64 for SHA-384 and 32 for SHA-256). We send the attacker's TLS record to the target, and then probe the cache set. If the cache set was accessed, then with high probability $\text{HashPadLen} > 64$ (32 for SHA-256). Otherwise we know that $\text{HashPadLen} \leq 63$ (31 for SHA-256). From this we can infer a possible range for the value of PadLen. Using the attack described in Section 6.3 we can create a padding oracle to distinguish between invalid padding of length $\text{PadLen} = 0$, and a large padding value.

For this simplified attack to work, we need the following assumptions to hold:

(1) The cache line size is 32.
(2) The padding array is aligned with the cache line.
(3) There are no prefetching optimizations used.

Clearly these assumptions are unrealistic, and we show next how they can be relaxed.

7.5 Full "*PostFetch*" Attack

The full "*PostFetch*" attack is the same as the one described in Algorithm 6, but the way that we choose the values of ProbeOffset and the resulting condition on HashPadLen are different. This is due to the following real world conditions:

(1) In most modern CPUs, the cache line size is 64 bytes. This makes the simplified attack on SHA-256 impractical.
(2) The padding array is not always aligned with a 64-byte cache line. As the alignment keyword is not used in the array declaration, it can vary from compilation to compilation. On our test platform, the padding arrays were either aligned to a cache line, or had a 32-byte offset. The alignment was changed between compilations by minor code changes (e.g. adding or removing a `printf` function call).
(3) In the hash implementation, the padding array is always copied to the hash function's internal buffer using the `memcpy` function. Due to the cache line size the array is read into the cache in cache line resolution even if just a single byte in the cache line has been accessed. On our test platform, we observed that the bytes after the array are also read into to cache (the next cache line), if we read a large enough part of the array (near the end of the cache line). For example, if the SHA-384 padding array is aligned to the cache line, the outer hash call of HMAC (that uses a hash padding of length 64) will cause a cache hit on the next cache line, so the entire 128 bytes padding array will be in the cache regardless of the TLS record padding length. This causes the simplified attack on SHA-384 to also become impractical.

Although each of the above conditions can cause our attack to fail, the combination of these conditions actually allows the attacks to work! Instead of probing the cache line at an offset ProbeOffset = 64 (32 for SHA-256), we probe the *next* cache line using ProbeOffset = 128 (64 for SHA-256). In fact we probe a memory location that is just after the padding array itself. In some cases this memory location will be read into the cache due to either hardware memory prefetching mechanism or speculative execution. For our attack to work we require that the probed memory location is not accessed by any other code in the verification process. As we will show, this is indeed the case in mbed TLS (see Section 7.7).

In case the hash padding array is aligned to the cache line, the last cache line for the array will be always accessed due to the `memcpy` call in the outer hash of HMAC. However the cache line *after* that will not be accessed, unless we read most of the bytes of the padding array (a very large value of HashPadLen). In case the array has a 32-byte offset to the cache line, the cache line at location ProbeOffset = 128 (64 for SHA-256), includes both the end of the array and some data that is stored afterwards. This cache line will only be accessed if HashPadLen is large.

7.6 Proof of Concept

We implemented a PoC for the above attack, to verify the presence of the cache side-channel. We ran our attack on an Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz running Ubuntu 17.10. We use the version 2.7 code taken from the mbedtls-2.7 branch of the official git repository on 4/4/2018 (commit hash be97c9cc), and compiled with the provided makefiles and GCC version 7.2.0. We targeted the HMAC function call sequence that is used in the `ssl_decrypt_buf` function (see Listing 8 in Appendix A). This HMAC code is the latest pseudo constant time version, designed to protect against previous timing and cache attacks.

Hash function	Array offset from cache line	Hash Padding length range	Cache Hit probability
SHA-384	32	$1-72$	≈ 0.026
SHA-384	32	$73-128$	≈ 0.998
SHA-256	32	$1-32$	≈ 0.002
SHA-256	32	$32-64$	≈ 0.998
SHA-384	0	$1-104$	≈ 0.028
SHA-384	0	$105-128$	≈ 0.999
SHA-256	0	$1-64$	≈ 0.999

Table 3: Cache hit probabilities for mbed TLS attack

We ran the code multiple times with different lengths for the decrypted TLS record. We primed the memory before the HMAC execution, and probed it afterwards. We attacked both the SHA-384 and SHA-256 implementations, with the hash padding array both aligned and not aligned (i.e. with a 32-byte offset) relative to the cache line. The experimentally obtained cache hit probabilities are given in Table 3. Because of the differing probabilities, we obtain reliable hash padding length oracles for three out of the four combinations, the exception being when SHA-256 is combined with a cache-aligned padding array.

7.7 Analysis of the Proof of Concept

As we described in Section 7.5, our full cache attack targets a cache line that contains the data stored just after the hash padding array. For this attack to work, the data after the array must not be accessed by the targeted code (otherwise we will always get a cache hit). When analyzing the PoC results we can see that this requirement is indeed fulfilled, except for the case of SHA-256 with a cache-aligned array. By analyzing the compiled program in this case, we discovered that the data the compiler stores just after the array is an array of round constants used in the SHA-256/SHA-384 compression function. This function is called in the finalization of the hash calculation just after accessing the hash padding array. In theory, our attack shouldn't work on this specific build of the code.

To get a better understanding we looked at a dump of the compiled assembly code of the compression function, taken from the mbed TLS server example program `ssl_server2` (see Listing 9 in Appendix A). In both the SHA-384 and SHA-256 code, the programmer unrolled the first 8 rounds of the compression function. To optimise performance, the GCC compiler uses hard-coded assembly `movabs` commands to push the first 8 round constants into registers. For the remaining rounds, the code uses the constants stored in the array. So although the first 8 round constants are stored in the array, they are never accessed. Since SHA-384 constants are 64 bits each, storing the first 8 round constants requires 64 bytes. So the 64 bytes after the hash padding array are never accessed. This means that, regardless of the array alignment, the cache line we target is only accessed due to the hash padding array and the attack can work. However, in the SHA-256 case, the round constants are only 32 bits long. So storing the first 8 constants requires just 32 bytes. The attack then works when the hash padding array has a 32-byte

offset to the cache line. In this case, the targeted cache line holds the end of the hash padding array, and the first 8 round constants; then the cache line is only accessed due to the hash padding array and our attack succeeds. In the other case, the hash padding array is aligned with the cache line and the targeted cache line holds the first 16 round constants. Since the constants of round 9 to 16 are accessed by the compression function, this cache line is always accessed and our attack fails.

7.8 Creating the Padding Oracle

Combining the results from Table 3 and equations 1 and 2, we obtain a CBC-mode padding oracle in the mbed TLS implementation of the form PadLen $> 4 + 16k \bmod 128$ for any k (mod64 for SHA-256). We use this padding oracle with the attack described in Section 6.3 to get a much more robust attack. Using the same calculations as in Section 5.3.3 we get an expected total number of cache attack executions (and TLS connections) of 384 per byte for SHA-256.

8 CONCLUSION

We have conducted an in-depth analysis of the security of pseudo constant time countermeasures to the Lucky 13 attack on CBC-mode in TLS. We examined a representative set of implementations, and found them all to be vulnerable to cache timing attacks. We developed three new techniques for exploiting leakage from cache timing and access patterns, and a novel variant of Lucky 13 with increased timing differences. These ideas may be applicable in attacking other cryptographic schemes. We produced PoCs for most of the attacks and evaluated the number of iterations of the basic cache timing step (and consequently the number of TLS connections) needed for the attacks to succeed. The requirements of the attacks are modest, especially in view of our novel greedy algorithm for selecting which mask value to use at each stage.

The main takeaway from our work is encapsulated in the title of our paper: pseudo constant time protections only give "pseudo security". CBC-mode in TLS seems destined to stay with us for some years to come, despite the growth in usage of AES-GCM and the impending arrival of TLS 1.3, due to the need to support legacy code and devices. The “Encrypt-then-MAC” countermeasure from RFC 7366 is supported in mbed TLS and GnuTLS, but requires client-side support and has seen little uptake elsewhere (e.g. neither Firefox nor Chrome supports the EtM extension). We suggest that all the pseudo constant time implementations should seriously consider adopting a fully constant time, constant memory access approach to defending against Lucky 13 and its variants – only this can provide robust security across a broad range of deployment (and thereby attack) scenarios.

The paper opens up several avenues for future work. Our greedy algorithm for selecting masks has good performance (coming close to the information theoretic lower bound in some cases), but it would be of interest to seek optimal algorithms. These may be of independent interest in other areas of cryptanalysis. We use only a single oracle condition, whereas we can often obtain multiple conditions by carefully varying the length of ciphertexts. It may be possible to exploit the availability of multiple conditions to further reduce the number of TLS connections needed. Our results with a single condition in combination with our greedy algorithm already illustrate the dangers of settling for pseudo constant time code. Finally, when considering the recovery of multiple plaintext bytes, we used a simple byte-by-byte analysis to estimate plaintext recovery rates. A more sophisticated approach would be to design attacks that produce likelihood values for each plaintext byte candidate, and then combine these across multiple bytes using enumeration techniques from the side-channel literature [7, 11, 25, 26, 34].

9 ACKNOWLEDGMENTS

The authors would like to thank Yuval Yarom for his helpful comments and insights. The authors would also like to thank the anonymous reviewers for their constructive suggestions that greatly improved the paper.

REFERENCES

[1] 2017. ICSI Certificate Notary. (2017). https://notary.icsi.berkeley.edu/
[2] Martin R. Albrecht and Kenneth G. Paterson. 2016. Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS. In *Advances in Cryptology – EUROCRYPT 2016, Part I (Lecture Notes in Computer Science)*, Marc Fischlin and Jean-Sébastien Coron (Eds.), Vol. 9665. Springer, Heidelberg, 622–643. https://doi.org/10.1007/978-3-662-49890-3_24
[3] Nadhem J. AlFardan and Kenneth G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 526–540.
[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers (Lecture Notes in Computer Science)*, Thomas Peyrin (Ed.), Vol. 9783. Springer, 163–184. https://doi.org/10.1007/978-3-662-52993-5_9
[5] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2015. Lucky 13 Strikes Back. In *ASIACCS 15: 10th ACM Symposium on Information, Computer and Communications Security*, Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn (Eds.). ACM Press, 85–96.
[6] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. 2014. “Ooh Aah... Just a Little Bit”: A Small Amount of Side Channel Can Go a Long Way. In *Cryptographic Hardware and Embedded Systems – CHES 2014 (Lecture Notes in Computer Science)*, Lejla Batina and Matthew Robshaw (Eds.), Vol. 8731. Springer, Heidelberg, 75–92. https://doi.org/10.1007/978-3-662-44709-3_5
[7] Andrey Bogdanov, Ilya Kizhvatov, Kamran Manzoor, Elmar Tischhauser, and Marc Witteman. 2016. Fast and Memory-Efficient Key Recovery in Side-Channel Attacks. In *SAC 2015: 22nd Annual International Workshop on Selected Areas in Cryptography (Lecture Notes in Computer Science)*, Orr Dunkelman and Liam Keliher (Eds.), Vol. 9566. Springer, Heidelberg, 310–327. https://doi.org/10.1007/978-3-319-31301-6_19
[8] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. 2016. Flush, Gauss, and Reload - A Cache Attack on the BLISS Lattice-Based Signature Scheme. In *Cryptographic Hardware and Embedded Systems – CHES 2016 (Lecture Notes in Computer Science)*, Benedikt Gierlichs and Axel Y. Poschmann (Eds.), Vol. 9813. Springer, Heidelberg, 323–345. https://doi.org/10.1007/978-3-662-53140-2_16
[9] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. 2003. Password Interception in a SSL/TLS Channel. In *Advances in Cryptology – CRYPTO 2003 (Lecture Notes in Computer Science)*, Dan Boneh (Ed.), Vol. 2729. Springer, Heidelberg, 583–599.
[10] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. 2018. Continuous Formal Verification of Amazon s2n. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (Lecture Notes in Computer Science)*, Hana Chockler and Georg Weissenbacher (Eds.), Vol. 10982. Springer, 430–446. https://doi.org/10.1007/978-3-319-96142-2_26
[11] Liron David and Avishai Wool. 2017. A Bounded-Space Near-Optimal Key Enumeration Algorithm for Multi-subkey Side-Channel Attacks. In *Topics in Cryptology – CT-RSA 2017 (Lecture Notes in Computer Science)*, Helena Handschuh (Ed.), Vol. 10159. Springer, Heidelberg, 311–327.
[12] T. Dierks and E. Rescorla. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard). (Aug. 2008), 104 pages. https://doi.org/10.17487/RFC5246 Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919.

[13] Joey Dodds. 2016. Verifying s2n HMAC with SAW. (2016). https://galois.com/blog/2016/09/verifying-s2n-hmac-with-saw/

[14] Thai Duong and Juliano Rizzo. 2011. Here come the ⊕ Ninjas. Unpublished manuscript. (2011).

[15] Daniel Genkin, Luke Valenta, and Yuval Yarom. 2017. May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519. In *ACM CCS 17: 24th Conference on Computer and Communications Security*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 845–858.

[16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.), Vol. 9721. Springer, 279–299. https://doi.org/10.1007/978-3-319-40667-1_14

[17] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In *2011 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 490–505.

[18] P. Gutmann. 2014. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366 (Proposed Standard). (Sept. 2014), 7 pages. https://doi.org/10.17487/RFC7366

[19] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *Cryptographic Hardware and Embedded Systems – CHES 2016 (Lecture Notes in Computer Science)*, Benedikt Gierlichs and Axel Y. Poschmann (Eds.), Vol. 9813. Springer, Heidelberg, 368–388. https://doi.org/10.1007/978-3-662-53140-2_18

[20] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[21] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 549–564. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp

[22] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[23] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 605–622. https://doi.org/10.1109/SP.2015.43

[24] Colm MacCarthaigh. 2015. AWS Security Blog - s2n and Lucky 13. (2015). https://aws.amazon.com/blogs/security/s2n-and-lucky-13/

[25] Daniel P. Martin, Luke Mather, Elisabeth Oswald, and Martijn Stam. 2016. Characterisation and Estimation of the Key Rank Distribution in the Context of Side Channel Evaluations. In *Advances in Cryptology – ASIACRYPT 2016, Part I (Lecture Notes in Computer Science)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.), Vol. 10031. Springer, Heidelberg, 548–572. https://doi.org/10.1007/978-3-662-53887-6_20

[26] Daniel P. Martin, Jonathan F. O'Connell, Elisabeth Oswald, and Martijn Stam. 2015. Counting Keys in Parallel After a Side Channel Attack. In *Advances in Cryptology – ASIACRYPT 2015, Part II (Lecture Notes in Computer Science)*, Tetsu Iwata and Jung Hee Cheon (Eds.), Vol. 9453. Springer, Heidelberg, 313–337. https://doi.org/10.1007/978-3-662-48800-3_13

[27] Nikos Mavrogiannopoulos. 2013. Time is money (in CBC ciphersuites). (2013). https://nikmav.blogspot.co.uk/2013/02/time-is-money-for-cbc-ciphersuites.html

[28] Ralph Charles Merkle, Ralph Charles, et al. 1979. Secrecy, authentication, and public key systems. (1979).

[29] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. 2014. This POODLE Bites: Exploiting The SSL 3.0 Fallback. (September 2014). https://www.openssl.org/~bodo/ssl-poodle.pdf

[30] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006 (Lecture Notes in Computer Science)*, David Pointcheval (Ed.), Vol. 3860. Springer, Heidelberg, 1–20.

[31] Stephen Schmidt. 2017. AWS Security Blog - s2n Is Now Handling 100 Percent of SSL Traffic for Amazon S3. (2017). https://aws.amazon.com/blogs/security/s2n-is-now-handling-100-percent-of-of-ssl-traffic-for-amazon-s3/

[32] Juraj Somorovsky. 2016. Systematic Fuzzing and Testing of TLS Libraries. In *ACM CCS 16: 23rd Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 1492–1504.

[33] Serge Vaudenay. 2002. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS.... In *Advances in Cryptology – EUROCRYPT 2002 (Lecture Notes in Computer Science)*, Lars R. Knudsen (Ed.), Vol. 2332. Springer, Heidelberg, 534–546.

[34] Nicolas Veyrat-Charvillon, Benoît Gérard, Mathieu Renauld, and François-Xavier Standaert. 2013. An Optimal Key Enumeration Algorithm and Its Application to Side-Channel Attacks. In *SAC 2012: 19th Annual International Workshop on Selected Areas in Cryptography (Lecture Notes in Computer Science)*, Lars R. Knudsen and Huapeng Wu (Eds.), Vol. 7707. Springer, Heidelberg, 390–406. https://doi.org/10.1007/978-3-642-35999-6_25

[35] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. 2017. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *ACM CCS 17: 24th Conference on Computer and Communications Security*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 859–874.

[36] Yuval Yarom. 2016. Mastik: A Micro-Architectural Side-Channel Toolkit. (2016). http://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf

[37] Yuval Yarom and Naomi Benger. 2014. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. Cryptology ePrint Archive, Report 2014/140. (2014). http://eprint.iacr.org/2014/140.

[38] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[39] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *ACM CCS 12: 19th Conference on Computer and Communications Security*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM Press, 305–316.

A SOURCE CODE

Listing 1: s2n HMAC digest for CBC verify

```
int s2n_hmac_digest_two_compression_rounds(struct s2n_hmac_state *state,
    void *out, uint32_t size){
    /* Do the "real" work of this function. */
    GUARD(s2n_hmac_digest(state, out, size));

    /* If there were 9 or more bytes of space left in the current hash block
    * then the serialized length, plus an 0x80 byte, will have fit in that block.
    * If there were fewer than 9 then adding the length will have caused an extra
    * compression block round. This digest function always does two compression rounds,
    * even if there is no need for the second.
    */
    if (state->currently_in_hash_block > (state->hash_block_size - 9))
        return 0;
    /* Can't reuse a hash after it has been finalized,
    so reset and push another block in */
    GUARD(s2n_hash_reset(&state->inner));

    /* No-op s2n_hash_update to normalize timing and guard against Lucky13. This
    does not affect the value of *out. */
    return s2n_hash_update(&state->inner, state->xor_pad, state->hash_block_size);
}
```

Listing 2: s2n CBC verification function

```
int s2n_verify_cbc(struct s2n_connection *conn, struct s2n_hmac_state *hmac,
    struct s2n_blob *decrypted) {
    /* Set up MAC copy workspace */
    struct s2n_hmac_state *copy = &conn->client->record_mac_copy_workspace;
    ...
    /* 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_two_compression_rounds(hmac, check_digest, mac_digest_size));
```

Listing 3: GnuTLS's extra compression call calculation

```
static void dummy_wait(record_parameters_st * params, gnutls_datum_t * plaintext,
        unsigned pad_failed, unsigned int pad, unsigned total){
    ...
    /* This is really specific to the current hash functions.
    * It should be removed once a protocol fix is in place.
    */
    if ((pad + total) % len > len - 9 && total % len <= len - 9) {
        if (len < plaintext->size)_
            gnutls_auth_cipher_add_auth(&params->read.cipher_state,
            plaintext->data, len);
```

Listing 4: GnuTLS's pad check and HMAC verification

```
decrypt_packet(gnutls_session_t session, gnutls_datum_t * ciphertext,
    gnutls_datum_t * plain, content_type_t type, record_parameters_st * params,
    gnutls_uint64 * sequence) {
    ...
    pad = plain->data[ciphertext->size - 1];     /* pad */
    ...
    for (i = 2; i <= MIN(256, ciphertext->size); i++) {
        tmp_pad_failed |= (plain->data[ciphertext->size - i] != pad);
        pad_failed |= ((i <= (1 + pad)) & (tmp_pad_failed));
    }

    if (unlikely (pad_failed != 0 || (1 + pad > ((int) ciphertext->size - tag_size)))) {
        /* We do not fail here. We check below for the
        * the pad_failed. If zero means success.
        */
        pad_failed = 1;
        pad = 0;
    }
    length = ciphertext->size - tag_size - pad - 1;
    ...
    ret = _gnutls_auth_cipher_add_auth(&params->read.ctx.tls12, plain->data, length);
    if (unlikely(gnutls_memcmp(tag, tag_ptr, tag_size) != 0 || pad_failed != 0)) {
        /* HMAC was not the same. */
        dummy_wait(params, plain, pad_failed, pad, length + preamble_size);
```

Listing 5: WolfSSL's extra compression call calculation

```
COMPRESS_UPPER      = 55,  /* compression calc numerator */
COMPRESS_LOWER      = 64,  /* compression calc denominator */

/* get compression extra rounds */
static INLINE int GetRounds(int pLen, int padLen, int t) {
    ...
    L1 -= COMPRESS_UPPER;
    L2 -= COMPRESS_UPPER;

    if ( (L1 % COMPRESS_LOWER) == 0)
        roundL1 = 0;
    if ( (L2 % COMPRESS_LOWER) == 0)
        roundL2 = 0;
```

Listing 6: WolfSSL's pad checke and HMAC verification

```
/* timing resistant pad/verify check, return 0 on success */
static int TimingPadVerify(WOLFSSL* ssl, const byte* input, int padLen, int t,
    int pLen, int content){
    byte verify[WC_MAX_DIGEST_SIZE];
    byte dmy[sizeof(WOLFSSL) >= MAX_PAD_SIZE ? 1 : MAX_PAD_SIZE] = {0};
    byte* dummy = sizeof(dmy) < MAX_PAD_SIZE ? (byte*) ssl : dmy;
    ...
    if (PadCheck(input + pLen - (padLen + 1), (byte)padLen, padLen + 1) != 0) {
        WOLFSSL_MSG("PadCheck failed");
        PadCheck(dummy, (byte)padLen, MAX_PAD_SIZE - padLen - 1);
```

```
        ssl->hmac(ssl, verify, input, pLen - t, content, 1); /* still compare */
        ConstantCompare(verify, input + pLen - t, t);
    ...
    PadCheck(dummy, (byte)padLen, MAX_PAD_SIZE - padLen - 1);
    ret = ssl->hmac(ssl, verify, input, pLen - padLen - 1 - t, content, 1);

    CompressRounds(ssl, GetRounds(pLen, padLen, t), dummy);
    ...
}
```

Listing 7: MBedTLS's SHA512 finish function

```
static const unsigned char sha512_padding[128] = {
    0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    ...
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int mbedtls_sha512_finish_ret( mbedtls_sha512_context *ctx,
    unsigned char output[64] ){
    size_t last, padn;
    ...
    last = (size_t)( ctx->total[0] & 0x7F );
    padn = ( last < 112 ) ? ( 112 - last ) : ( 240 - last );
    if( ( ret = mbedtls_sha512_update_ret( ctx, sha512_padding, padn ) ) != 0 )
    return( ret );
```

Listing 8: MBedTLS's CBC HMAC verification

```
static int ssl_decrypt_buf( mbedtls_ssl_context *ssl ){
    ...
    padlen = 1 + ssl->in_msg[ssl->in_msglen - 1];
    ...
    for( i = 1; i <= 256; i++ ) {
        real_count &= ( i <= padlen );
        pad_count += real_count * ( ssl->in_msg[padding_idx + i] == padlen - 1 );
    }
    correct &= ( pad_count == padlen ); /* Only 1 on correct padding */
    padlen &= correct * 0x1FF;
    ...
    * Known timing attacks:
    *  - Lucky Thirteen (http://www.isg.rhul.ac.uk/tls/TLStiming.pdf)
    *
    * We use ( ( Lx + 8 ) / 64 ) to handle 'negative Lx' values
    * correctly. (We round down instead of up, so -56 is the correct
    * value for our calculations instead of -55)
    */
    size_t j, extra_run = 0;
    extra_run = ( 13 + ssl->in_msglen + padlen + 8 ) / 64 -
                ( 13 + ssl->in_msglen          + 8 ) / 64;
    ...
    mbedtls_md_hmac_update( &ssl->transform_in->md_ctx_dec, ssl->in_ctr, 8 );
    mbedtls_md_hmac_update( &ssl->transform_in->md_ctx_dec, ssl->in_hdr, 3 );
    mbedtls_md_hmac_update( &ssl->transform_in->md_ctx_dec, ssl->in_len, 2 );
    mbedtls_md_hmac_update( &ssl->transform_in->md_ctx_dec, ssl->in_msg,
```

```
        ssl->in_msglen );
    mbedtls_md_hmac_finish( &ssl->transform_in->md_ctx_dec, mac_expect );
    /* Call mbedtls_md_process at least once due to cache attacks */
    for( j = 0; j < extra_run + 1; j++ )
        mbedtls_md_process( &ssl->transform_in->md_ctx_dec, ssl->in_msg );

    mbedtls_md_hmac_reset( &ssl->transform_in->md_ctx_dec );
```

Listing 9: MBedTLS's internal SHA512 process function assmebly code

```
48690 <mbedtls_internal_sha512_process>:
    ...
    48797:  49 bf 18 81 6d da d5    movabs $0xab1c5ed5da6d8118,%r15
    4879e:  5e 1c ab
    487a1:  48 bd 2f 3b 4d ec cf    movabs $0xb5c0fbcfec4d3b2f,%rbp
    487a8:  fb c0 b5
    487ab:  4c 89 7c 24 20          mov    %r15,0x20(%rsp)
    487b0:  49 bf 9b 4f 19 af a4    movabs $0x923f82a4af194f9b,%r15
    487b7:  82 3f 92
    487ba:  49 bd cd 65 ef 23 91    movabs $0x7137449123ef65cd,%r13
    487c1:  44 37 71
    487c4:  4c 89 7c 24 18          mov    %r15,0x18(%rsp)
    487c9:  49 bf 19 d0 05 b6 f1    movabs $0x59f111f1b605d019,%r15
    487d0:  11 f1 59
    487d3:  49 bc 22 ae 28 d7 98    movabs $0x428a2f98d728ae22,%r12
    487da:  2f 8a 42
    ...
    48802:  49 bf 38 b5 48 f3 5b    movabs $0x3956c25bf348b538,%r15
    48809:  c2 56 39
    4880c:  48 89 74 24 30          mov    %rsi,0x30(%rsp)
    48811:  4c 89 7c 24 08          mov    %r15,0x8(%rsp)
    48816:  49 bf bc db 89 81 a5    movabs $0xe9b5dba58189dbbc,%r15
    4881d:  db b5 e9
```