Lucky 13 Strikes Back

Gorka Irazoqui
Worcester Polytechnic Institute
girazoki@wpi.edu

Mehmet Sinan İnci
Worcester Polytechnic Institute
msinci@wpi.edu

Thomas Eisenbarth
Worcester Polytechnic Institute
teisenbarth@wpi.edu

Berk Sunar
Worcester Polytechnic Institute
sunar@wpi.edu

ABSTRACT

In this work we show how the Lucky 13 attack can be resurrected in the cloud by gaining access to a virtual machine co-located with the target. Our version of the attack exploits distinguishable cache access times enabled by VM deduplication to detect dummy function calls that only happen in case of an incorrectly CBC-padded TLS packet. Thereby, we gain back a new covert channel not considered in the original paper that enables the Lucky 13 attack. In fact, the new side channel is significantly more accurate, thus yielding a much more effective attack. We briefly survey prominent cryptographic libraries for this vulnerability. The attack currently succeeds to compromise PolarSSL, GnuTLS and CyaSSL on deduplication enabled platforms while the Lucky 13 patches in OpenSSL, Mozilla NSS and MatrixSSL are immune to this vulnerability. We conclude that, any program that follows secret data dependent execution flow is exploitable by side-channel attacks as shown in (but not limited to) our version of the Lucky 13 attack.

Keywords

Lucky 13 attack, Cross-VM attacks, virtualization, deduplication

1. MOTIVATION

The Transport Layer Security (TLS) family of protocols ensures the security of the entire communications infrastructure by providing confidentiality and integrity services across untrusted networks. Numerous web applications rely on TLS to secure client-server data traffic. Similarly distributed applications use TLS to establish a secure channel for transporting application-layer data with centralized cloud servers. At the higher level TLS uses X.509 certificates along with public key cryptography to authenticate the exchanged symmetric encryption keys and to authenticate the server. This session key is then used to ensure the integrity and confidentiality of the data exchanged over a secure session between the TLS client and server.

Starting as Secure Sockets Layer (SSL), after adoption by the IETF TLS has undergone many changes (SSL 1.0, 2.0, 3.0, TLS 1.0, 1.1, 1.2). Many releases were motivated by attacks targeting both the protocols and the underlying cryptographic schemes [38, 8, 7, 17, 29, 5, 16, 41]. In this work we focus on attacks targeting the padding procedure in TLS's MAC-Encode-Encrypt (MEE) primitive.

Implementation Attacks on TLS. Handling CBC IVs, and paddings in cryptographic algorithms has a long history of attacks and countermeasures, and is notoriously hard to get right in implementations. As early as in 1998 Bleichenbacher pointed to vulnerabilities in SSL 3.0 stemming from leaked error messages due to incorrectly padded plaintexts. Later Vaudenay [38] presented an attack in the symmetric key setting on SSL/TLS induced by CBC mode padding. The BEAST chosen plaintext attack (Browser Exploit Against SSL/TLS) [15] exploited a long-known cipher block chaining (CBC) mode IV vulnerability in TLS 1.0 [25] to achieve full plaintext recovery. The exploit is based on the earlier work in [32, 8, 7]. The padding oracle attack is most commonly applied to CBC encryption mode, where the server leaks whether the padding of an encrypted message was correctly formed or not. Depending on the specifics of the encryption scheme and the encapsulating protocol, this side-channel leakage may be escalated to a full message recovery attack. These are collectively referred to as *padding oracle attacks*. A more recent striking application of the aforementioned padding attacks was given by Bardou et al. [9] where many cryptographic hardware tokens were determined to be vulnerable. Specifically, Bardou et al. apply Vaudenay's CBC attack and improve Bleichenbacher's attack to significantly reduce the number of decryption oracle accesses, thereby making attacks feasible on slow tokens.

Even though in the last years the padding oracle attacks were considered a fixed vulnerability in the community, in 2013 a new kind of padding oracle attack was presented by AlFardan et al. [16]. The Lucky 13 attack was proposed to recover TLS/DTLS encrypted messages by exploiting a vulnerability in the implementation of HMAC. The attack works by carefully modifying network packets during transmission, and using network timing information to recover the plaintext byte-by-byte from TLS encrypted packets. The attack received significant attention from the media and industry. A great deal of work went into fixing the TLS vulnerability. A widely applied and immediate fix—using RC4 encryption instead of a block cipher in CBC mode—turned out to be ill-advised: The attack described in [5] exploits statistical biases in the RC4 key stream to recover parts of

ASIA CCS'15, April 14–17, 2015, Singapore..

http://dx.doi.org/10.1145/2714576.2714625.

the plaintext using a large number of TLS encryptions. To fix the popular MEE mode that uses a block cipher in CBC mode, cryptographic library providers applied various techniques aimed to equalize packet processing times, e.g. by calling a dummy HMAC function. Since then modifications are being studied to solve attacks against MEE schemes and the Lucky 13 issue has been considered closed by the security community and the industry.

In this work, we revive the Lucky 13 attack on a number of prominent cryptographic libraries which have been *patched* to eliminate the network timing side-channel. We instead run our attacks in the cross-VM setting using cache access information to realize the Lucky 13 attack.

Cross-VM Attacks. Cross-VM attacks assume a co-located process running on the same physical hardware as a target process (e.g. same machine on different cores) can extract information from the target in spite of the VM sandboxing. Many side-channel attacks have been proposed that manage to recover sensitive data when a spy process is executed in the same OS as the victim. For instance, the early proposal by Bernstein [11] (and later in [12, 36, 18]) targets the time variation due to memory accesses to recover a AES encryption key. These techniques are now being moved to cloud servers to break sandboxing across virtual machines.

Cross-VM attacks assume the attacker to be able to co-locate with the victim. Co-location was considered a major obstacle until 2009 when Ristenpart et al. [31] demonstrated that it is possible to co-locate with a potential victim and extract sensitive data across VMs.This initial result fueled many other research targeting a co-located victim in a cloud system.

In 2011, Suzaki et al. [33, 34] exploited a memory saving OS-optimization called Kernel Samepage Merging (KSM) to recover data from another user and to identify a co-located user running in KVM hypervisors. Shortly later, Zhang et al. [42] used an access driven cache timing attack, namely *Prime and Probe* to recover an El Gamal decryption key from a victim process running in Xen VMs. In order to cope with multiple sources of microarchitectural noise, the authors used a hidden Markov model. In contrast to the work of Ristenpart et al. [31], the authors of [42] were able to extract fine grain information from a cryptographic implementation across VMs.

Recently the powerful *Flush+Reload* attack was used by Yarom et.al in cloud-like environments such as VMware ESXI and KVM to extract RSA [41, 10] and ECDSA keys, while Irazoqui et al. used the same detection method to recover AES keys from co-located VMware VMs [21].

1.1 Our Contribution

In this work we demonstrate that by mounting cache attacks it is possible to revive a modified Lucky 13 attack on many of the patched TLS libraries. Specifically, we show that it is possible to recover plaintexts from TLS encrypted sessions *across VM boundaries* by applying a *flush+reload* cache attack in VMware ESXi VMs. The vulnerability persists even if the VMs are running on different cores in the same machine. The attack works because some TLS libraries prevent the Lucky 13 attack by using *dummy functions* to ensure constant time executions. By monitoring the instruction cache, we detect accesses to these dummy functions and hence distinguish valid CBC-paddings, as done in the Lucky 13 attack. While requiring co-location, the cache side channel is less noisy than the network timing side channel originally exploited in [16], resulting in a more efficient attack. The effectiveness of the new attack is demonstrated on a number prominent cryptographic libraries: PolarSSL [30], GnuTLS [24], and CyaSSL [1]. Fortunately, our results also indicate that some libraries such as OpenSSL [35], Mozilla's NSS [26], and MatrixSSL [3] have been patched well and the new attack does not apply to them. These libraries feature carefully crafted constant run time execution while OpenSSL and Mozilla's NSS also ensure branch-free handling of MAC checking.

2. BACKGROUND

In this work we substitute the network timing channel with the cache timing channel as experienced in a Cross-VM setting. There is a very rich literature of cache attacks and here we only very briefly review cache timing attacks and focus on a more recent and effective cache attack variant, e.g **the *Flush+Reload* cache attack**.

Cache Architecture. The cache architecture is a set of components that reside between the CPU and the RAM. The principal function of the cache is to reduce the average access time to the main memory by exploiting spatial and temporal locality principles. When the CPU requests a memory line, the cache will be searched first to see if it is located there. If so, it is said that a cache *hit* has occurred and therefore, the access delay is much smaller. However when the data is not found in the cache, the CPU will try to find the memory line in the subsequent levels of cache or in the memory, which translates to greater delays. In this case it is said that a cache *miss* has occurred. When a cache miss occurs, the data is retrieved from the memory and a copy is stored in all levels of the cache hierarchy following both the spatial and temporal locality principles: recently accessed data and data in nearby locations are likely to be accessed soon.

Cache Side channel attacks. Cache based side channel attacks have been widely studied over the last two decades. It was in 1992 when the cache was first considered as a valid covert channel to extract sensitive information [20], and this approach was further studied theoretically later in [23, 28, 37]. In the last decade many implementations of cache based side channel attacks have been investigated. Bernstein in 2005 [11] recovered an AES keys due to microarchitectural time differences between different memory lines, whereas Osvik et al. studied the performance of different spy processes monitoring the data cache like *Prime and Probe* and *Evict+Time* on AES [27]. Only one year later, Bonneau et al. implemented a cache attack based on table look up collisions on AES [13].

Shortly later Acıiçmez showed that the instruction cache also leaks information by mounting an attack targeting RSA encryptions [4]. In a follow up work, Chen et al. improved the attack proposed in [4] and applied it in a more realistic scenario [14]. One year later, cache attacks were moved to the cloud by Zhang et al. where they managed to recover an El Gamal encryption key across XEN VMs [42].

Recently Gullasch et al. [19] demonstrated that deduplication features implemented in modern OSs can open a covert channel to recover sensitive information like AES keys with the *Flush+Reload* attack, but assuming to have control over the CFS. This approach was later followed by Yarom et al.

and Irazoqui et al. to recover RSA and AES keys respectively, even in cloud environments [41, 21]. Finally Benger et al. also showed that the security of ECDSA encryptions is compromised when the adversary is able to monitor cache accesses [10].

2.1 The Flush+Reload Technique

The *Flush+Reload* attack is a powerful cache-based side channel attack technique that checks if specific cache lines have been accessed or not by the code under attack. Gullasch et al. [18] first used this spy process on AES, although the authors did not brand their attack as *Flush+Reload* at the time. Later Yarom et al. [41, 10] used it to target specific functions instead of data. In their studies, they used the *Flush+Reload* technique to recover keys from RSA and ECDSA decryption processes. Here we briefly explain how *Flush+Reload* attack works. The attack is carried out in 3 stages:

- **Flush step:** In this stage, the attacker uses the `clflush` instruction to flush the desired memory lines from the cache and make sure that they go to the main memory. We have to remark here that the `clflush` command does not only flush the memory line from the cache hierarchy of the corresponding working core, but it flushes from all the caches of all the cores in the CPU. This is an important point: if it only flushed from the corresponding core's cache hierarchy, the attack would only work if the attacker and victim's processes were running on the same CPU core. This would have required a much stronger assumption than just being on the same physical machine.

- **Victim accessing step:** In this stage the attacker waits until the victim runs a fragment of the targeted code, which uses the memory lines that have been flushed in the first stage.

- **Reload step:** In this stage the attacker reloads the previously flushed memory lines and measures the time it takes to reload them. Depending on the reloading time, the attacker decides whether the victim accessed the memory line (in which case the memory line would be present in the cache) or if the victim did not access the corresponding memory line (in which case the memory line will not be present in the cache.) The timing difference between a cache hit and a cache miss makes this difference detectable by the attacker.

The fact that the attacker and the victim processes do not run on the same core is not a problem here. Even though there may be isolation at various levels of the cache, in most systems there is some level of cache that is shared between all the cores. Therefore, through this shared level of cache (typically the L3 cache), one can still distinguish between accesses to the main memory and accesses to the cache.

2.2 Memory Deduplication

Memory deduplication is an optimization technique that was originally introduced in Linux as KSM to improve the memory utilization by merging duplicate memory pages. KSM first appeared in Linux kernel version 2.6.32 [22, 2]. In this implementation, KSM kernel daemon `ksmd`, scans the user memory for potential pages to be shared among users [6], creating signatures for these pages. The signatures are kept in the deduplication table for matching and merging. When two or more pages with the same signature are found, they are cross-checked completely to determine if they are identical in which case they are merged with the copy-on-write tag set.

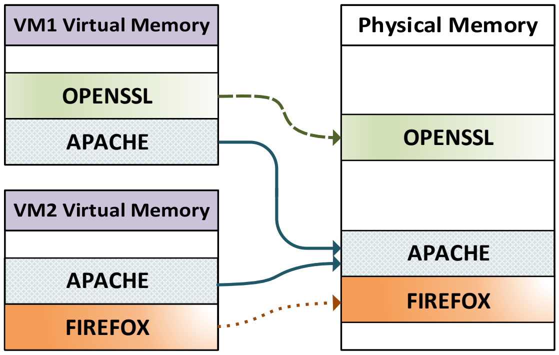

Figure 1: Memory Deduplication Scheme

Deduplication later became a standard technique for improving the memory utilization in VMMs. It is especially effective in virtual machine environments where multiple guest OSs co-reside on the same physical machine and share the physical memory. At the more abstract level, deduplication works by recognizing processes (or VMs) that place the same data in memory. This frequently happens when two processes use the same shared libraries. The deduplication scheme eliminates multiple copies from memory and allows the data to be shared between users and processes. Consequently, variations of memory deduplication techniques are now implemented in VMware ESXI [39, 40] and others such as KVM [2, 22] VMMs. Since KVM converts the Linux kernel into a hypervisor, it directly uses KSM as page sharing technique, whereas VMware uses Transparent Page Sharing (TPS).

Even though deduplication saves memory and thus allows more virtual machines to run on the host system, it also opens a door to side channel attacks. While the data in the cache cannot be modified or corrupted by an adversary, parallel access rights can be exploited to reveal secret information about processes executing in the target VM.

3. THE LUCKY 13 ATTACK

The Lucky 13 attack targets a vulnerability in the TLS (and DTLS) protocol design. The vulnerability is due to MAC-then-encrypt mode, in combination with the padding of the CBC encryption, also referred to as MEE-TLS-CBC. In the following, our description focuses on this popular mode. Vaudenay [38] showed how the CBC padding can be exploited for a message recovery attack. AlFardan et al. [16] showed—more than 10 years later—that the subsequent MAC verification introduces timing behavior that makes the message recovery attack feasible in practical settings. In fact, their work includes a comprehensive study of the vulnerability of several TLS libraries. In this section we give a brief description of the attack. For a more detailed description, please refer to the original paper [16].

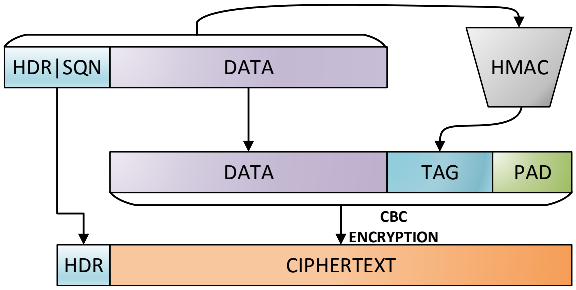

Figure 2: Encryption and authentication in the TLS record protocol when using HMAC and a block cipher in CBC mode.

3.1 The TLS Record Protocol

The TLS record protocol provides encryption and message authentication for bulk data transmitted in TLS. The basic operation of the protocol is depicted in Figure 2. When a payload is sent, a sequence number and a header are attached to it and a MAC tag is generated by any of the available HMAC choices. Once the MAC tag is generated, it is appended to the payload together with a padding. The payload, tag, and pad are then encrypted using a block cipher in CBC mode. The final message is formed by the encrypted ciphertext plus the header.

Upon receiving an encrypted packet, the receiver decrypts the ciphertext with the session key that was negotiated in the handshake process. Next, the padding and the MAC tag need to be removed. For this, first the receiver checks whether the size of the ciphertext is a multiple of the block size and makes sure that the ciphertext can accommodate minimally a zero-length record, a MAC tag, and at least one byte of padding. After decryption, the receiver checks if the recovered padding matches one of the allowed patterns. A standard way to implement this decoding step is to check the last byte of the plaintext, and to use it to determine how many of the trailing bytes belong to the padding. Once the padding is removed, and the plain payload is recovered, the receiver attaches the header and the sequence number and performs the HMAC operation. Finally, the computed tag is compared to the received tag. If they are equal, the contents of the message are concluded to be securely transmitted.

3.2 HMAC

The TLS record protocol uses the HMAC algorithm to compute the tag. The HMAC algorithm is based on a hash function H that performs the following operations:

$$\mathsf{HMAC}(K,m) = H((K \oplus opad)||H((K \oplus ipad)||M)$$

Common choices in TLS 1.2 for H are SHA-1, SHA-256 and the now defunct MD5. The message M is padded with a single 1 bit followed by zeros and an 8 byte length field. The pad aligns the data to a multiple of 64 bytes. $K \oplus opad$ already forms a 64 byte field, as well as $K \oplus ipad$. Therefore, the minimum number of compression function calls for a HMAC operation is 4. This means that depending on the number of bytes of the message, the HMAC operation is going to take more or less compression functions. To illustrate this, we are repeating the example given in [16] as follows. Assume that the plaintext size is 55 bytes. In this case an 8 byte length field is appended together with a padding of size 1, so that the total size is 64 bytes. Here in total the HMAC operation is going to take four compression function calls. However if the plaintext size is 58, an 8 byte length field is attached and 62 bytes of padding are appended to make the total size equal to 128 bytes. In this case, the total compression function calls are going to be equal to five. Distinguishing the number of performed compression function calls is the basic idea that enables the Lucky 13 attack.

3.3 CBC Encryption & Padding

Until the support of the Galois Counter Mode in TLS 1.2, block ciphers were always used in cipher block chaining (CBC) mode in TLS. Decryption of each block of a ciphertext C_i is performed as follows:

$$P_i = D_k(C_i) \oplus C_{i-1}$$

Here, P_i is the plaintext block and $D_k(\cdot)$ is the decryption under key k. For the prevalent AES, the block size is 16 bytes. The size of the message to be encrypted in CBC mode has to be indeed a multiple of the cipher block size. The TLS protocol specifies a padding as follows: the last padding byte indicates the length of the padding; the value of the remaining padding bytes is equal to the number of padding bytes needed. This means that if 3 bytes of padding is needed, the correct padding has to be 0x02|0x02|0x02. Possible TLS paddings are: 0x00, 0x01|0x01, 0x02|0x02|0x02, up to 0xff|0xff|...|0xff. Note that there are several valid paddings for each message length.

3.4 An Attack On CBC Encryption

We now discuss the basics of the Lucky 13 attack. For the purposes of this study the target cipher is going to be AES in CBC mode, as described above. Again, we are going to use the same example that AlFardan et al. gave in [16]. Assume that the sender is sending 4 non-IV blocks of 16 bytes each, one IV block, and the header number. Let's further assume that we are using SHA-1 to compute the MAC tag, in which case the digest size is 20 bytes. The header has a fixed length of 5 bytes and the sequence number would have a total size of 8 bytes. The payload would look like this:

$$\mathsf{HDR}|C_{IV}|C_1|C_2|C_3|C_4$$

Now assume that the attacker masks Δ in C_3. The decryption of C_4 is going to be as follows:

$$P_4^* = D_k(C_4) \oplus C_3 \oplus \Delta = P_4 \oplus \Delta$$

Focusing on the last two bytes $P^*_{4(14)}|P^*_{4(15)}$ three possible scenarios emerge:

Invalid padding This is the most probable case, where the plaintext ends with an invalid padding. Therefore, according to TLS protocol, this is treated as 0 padding. 20 bytes of MAC (SHA-1) are removed and the corresponding HMAC operation in the client side is performed on 44 bytes +13 bytes of header, in total 57 bytes. Therefore the HMAC evaluates 5 compression function calls.

Valid 0x00 padding If $P^*_{4(15)}$ is 0x00, this is considered as valid padding, and a single byte of padding is removed. Then the 20 bytes of digest are removed, and the HMAC operation in client side is done in 43+13 bytes, 56 in total, which takes 5 compression function calls.

Any other valid padding For instance, if we consider a valid padding of two bytes, the valid padding would be 0x01|0x01 and 2 bytes of padding are removed. Then 20 bytes of digest are removed, and the HMAC operation is performed over $42 + 13 = 55$ bytes, which means four compression function calls.

The Lucky 13 attack is based on detecting this difference between 4 and 5 compression function calls. Recall that if an attacker knows that a valid 0x01|0x01 padding was achieved, she can directly recover the last two bytes of P_4, since

$$0x01|0x01 = P_{4(14)}|P_{4(15)} \oplus \Delta_{(14)}|\Delta_{(15)}$$

Furthermore, she can keep on trying to recover the remaining bytes once she knows the first 2 bytes. The attacker needs to perform at most 2^{16} trials for detecting the last two bytes, and then up to 2^8 messages for each of the bytes that she wants to recover.

4. ANALYSIS OF LUCKY 13 PATCHES

The Lucky 13 attack triggered a series of patches for all major implementations of TLS [16]. In essence, all libraries were fixed to remove the timing side channel exploited by Lucky 13, i.e. implementations were updated to handle different CBC-paddings in constant time. However, different libraries used different approaches to achieve this:

- Some libraries implement *dummy functions* or processes,
- Others use *dummy data* to process the maximum allowed padding length in each MAC checking.

In the following, we discuss these different approaches for some of the most popular TLS libraries.

4.1 Patches Immune to Flush+Reload

In this section we will analyze those libraries that are secure against the flush and reload technique.

- **OpenSSL:** The Lucky 13 vulnerability was fixed in OpenSSL versions 1.0.1, 1.0.0k, and 0.9.8y by February 2013 without the use of a time consuming dummy function and by using dummy data. Basically, when a packet is received, the padding variation is considered and the maximum number of HMAC compression function evaluations needed to equalize the time is calculated. Then each compression function is computed directly, without calling any external function. For every message, the maximum number of compression functions are executed, so that no information is leaked through the time channel in case of the incorrect padding. Furthermore, the OpenSSL patch removed any data dependent branches ensuring a fixed data independent execution flow. This is a generic solution for microarchitectural leakage related attacks, i.e. cache timing or even branch prediction attacks.

- **Mozilla NSS:** This library is patched against the Lucky 13 attack in version 3.14.3 by using a constant time HMAC processing implementation. This implementation follows the approach of OpenSSL, calculating the number of maximum compression functions needed for a specific message and then computing the compression functions directly. This provides not only a countermeasure for both timing and cache access attacks, but also for branch prediction attacks.

- **MatrixSSL:** MatrixSSL is fixed against the Lucky 13 with the release of version 3.4.1 by adding timing countermeasures that reduce the effectiveness of the attack. In the fix, the library authors implemented a decoding scheme that does a sanity check on the largest possible block size. In this scheme, when the received message's padding length is incorrect, Matrix SSL runs a loop as if there was a full 256 bytes of padding. When there are no padding errors, the same operations are executed as in the case of an incorrect padding to sustain a constant time. Since there are no functions that are specifically called in the successful or unsuccessful padding cases, this library is not vulnerable to our *Flush+Reload* attack. In addition, Matrix SSL keeps track of all errors in the padding decoding and does the MAC checking regardless of valid or invalid padding rather than interrupting and finalizing the decoding process at the first error. However, since an if statement is used when the extra compression function is called, the library might be a suitable target for a branch prediction attack.

4.2 Patches Vulnerable to Flush+Reload

There are some patches that ensure constant time execution and therefore are immune to the original Lucky 13 attack [16] which are vulnerable to *Flush+Reload*. This implies a dummy function call or a different function call tree for valid and invalid paddings. Furthermore, if these calls are preceded by branch predictions, these patches might also be exploitable by branch prediction attacks. Some examples including code snippets are given below.

- **GnuTLS:** uses a `dummy_wait` function that performs an extra compression function whenever the padding is incorrect. This function makes the response time constant to fix the original Lucky 13 vulnerability. Since this function is only called in the case of incorrect padding, it can be detected by a co-located VM running a *Flush+Reload* attack.

```
if (memcmp (tag, &ciphertext->data[length],
tag_size) != 0 || pad_failed != 0)
 //HMAC was not the same.
{dummy_wait(params, compressed,pad_failed,
 pad,length+preamble_size);}
```

- **PolarSSL:** uses a dummy function called `md_process` to sustain constant time to fix the original Lucky 13 vulnerability. Basically the number of extra runs for a specific message is computed and added by `md_process`. Whenever this dummy function is called, a co-located adversary can learn that the last padding was incorrect and use this information to realize the Lucky 13 attack.

```
for( j = 0; j < extra_run; j++ )
 \\We need an extra run
md_process( &ssl->transform_in->
md_ctx_dec,ssl->in_msg );]*
```

- **CyaSSL:** was fixed against the Lucky 13 with the release of 2.5.0 on the same day the Lucky 13 vulnerability became public. In the fix, CyaSSL implements a timing resistant pad/verify check function called `TimingPadVerify` which uses the `Padcheck` function with dummy data for all padding length cases whether or not the padding length is correct. CyaSSL also does all the calculations such as the HMAC calculation for the *incorrect* padding cases which not only fixes the original Lucky 13 vulnerability but also prevents the detection of incorrect padding cases. This is due to the fact that the `Padcheck` function is called for both correctly and incorrectly padded messages which makes it impossible to detect with our *Flush+Reload* attack.

  However, for the correctly padded messages, CyaSSL calls the `CompressRounds` function which is detectable with *Flush+Reload* .Therefore, we monitor the correct padding instead of the incorrect padding cases.

  Correct padding case:

```
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);
ConstantCompare(verify, input +
(pLen − padLen − 1 − t), t) != 0)
```

  Incorrect padding case:

```
CYASSL_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);
```

5. REVIVING LUCKY 13 ON THE CLOUD

As the cross-network timing side channel has been closed (c.f. Section 4), the Lucky 13 attack as originally proposed no longer works on the recent releases of most cryptographic libraries. In this work we revive the Lucky 13 attack to target these (fixed) releases by gaining information through co-located VMs (a leakage channel not considered in the original paper) rather than the network timing exploited in the original attack.

5.1 Regaining the Timing Channel

Most cryptographic libraries and implementations have been largely fixed to yield an *almost* constant time when the MAC processing time is measured over the network. As discussed in Section 4, although there are some similarities in these patches, there are also subtle differences which—as we shall see—have significant implications on security. Some of the libraries not only closed the timing channel but also various cache access channels. In contrast, other libraries left an open door to implement access driven cache attacks on the protocol. In this section we analyze how an attacker can gain information about the number of compression functions performed during the HMAC operation by making use of leakages due to shared memory hierarchy in VMs located on the same machine. This is sufficient to re-implement the Lucky 13 attack.

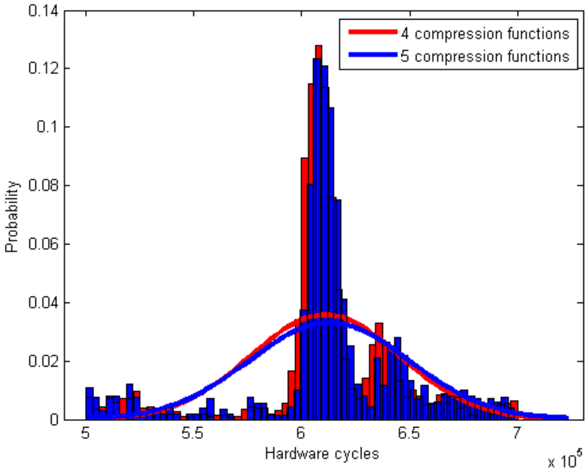

Figure 3: Histogram of network time measured for sent packages with valid (4 compression functions) and invalid (5 compression functions) paddings.

More precisely, during MAC processing depending on whether the actual MAC check terminates early or not, some libraries call a dummy function to equalize the processing time. Knowing if this dummy function is called or not reveals whether the received packet was processed as to either having a invalid padding, zero length padding or any other valid padding. In general, any difference in the execution flow between handling a well padded message, a zero padded message or an invalid padded message enables the Lucky 13 attack. This information is gained by the *Flush+Reload* technique if the cloud system enables deduplication features.

To validate this idea, we ran two experiments:

- In the first experiment we generated encrypted packets using PolarSSL client with valid and invalid paddings and measured the network time as shown in Figure 3. Note that, the network time in the two distributions obtained for valid and invalid paddings are essentially indistinguishable as intended by the patches.

- In the second experiment we see a completely different picture. Using PolarSSL we generated encrypted packets with valid and invalid paddings which were then sent to a PolarSSL server. Here instead, we measured the time it takes to load a specifically chosen PolarSSL library function running inside a co-located VM. Figure 4 shows the probability distributions for a function reloaded from L3 cache vs. a function reloaded from the main memory. The two distributions are clearly distinguishable and the misidentification rate (the area under the overlapping tails in the middle of the two distributions) is very small. Note that, this substitute timing channel provides much more precise timing that the network time. To see this more clearly, we refer the reader to Figure 2 in [16] where the network time

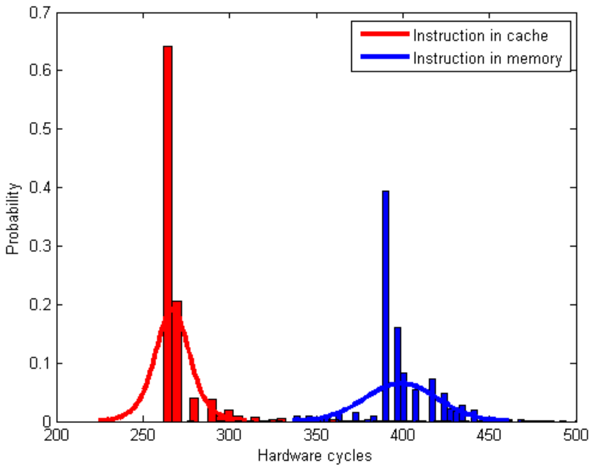

Figure 4: Histogram of access time measured for function calls from the L3 cache vs. a function called from the main memory.

is measured to obtain two overlapping Gaussians by measurements with OpenSSL encrypted traffic. This is not a surprise, since the network channel is significantly more noisy.

In conclusion, we regain a much more precise timing channel, by exploiting the discrepancy between L3 cache and memory accesses as measured by a co-located attacker. In what follows, we more concretely define the attack scenario, and then precisely define the steps of the new attack.

5.2 New Attack Scenario

In our attack scenario, the side channel information will be gained by monitoring the cache in a co-located VM. In the same way as in [16] we assume that the adversary captures, modifies, and replaces any message sent to the victim. However, TLS sessions work in such a way that when the protocol fails to decrypt a message, the session is closed. This is the reason why we focus in multi-session attacks where the same plaintext in the same place is being sent to the victim e.g. an encrypted password sent during user authentication.

The fact that we are working with a different method in a different scenario gives us some advantages and disadvantages over the previous Lucky 13 work:

Advantages:.

- Recent patches in cryptographic libraries mitigate the old Lucky 13 attack, but are still vulnerable in the new scenario.
- In the new scenario, no response from the server is needed. The old Lucky 13 attack needed a response to measure the time, which yielded a noisier environment in TLS than DTLS.
- The new attack does not suffer from the network channel noise. This source of noise was painful for the measurements as we can see in the original paper, where in case of TLS as many as 2^{14} trials were necessary to guess a single byte value.

Disadvantages:.

- Assumption of co-location: To target a specific victim, the attacker has to be co-located with that target. However the attacker could just reside in a physical machine and just wait for some potential random victim running a TLS operation.
- Other sources of noise: The attacker no longer has to deal with network channel noise, but still has to deal with other microarchitectural sources of noise, such as instruction prefetching. This new source of noise is translated in more traces needed, but as we will see, much less than in the original Lucky 13 attack. In Section 6 we explain how to deal with this new noise.

5.3 Attack Description

In this section we describe how an attacker uses *Flush+Reload* technique to gain access to information about the plaintext that is being sent to the victim.

- **Step 1 Function identification:** Identify different function calls in the TLS record decryption process to gain knowledge about suitable target functions for the spy process. The attacker can either calculate the offset of the function she is trying to monitor in the library, and then add the corresponding offset when the Address Space Layout Randomization (ASLR) moves her user address space. Another option is to disable the ASLR in the attackers VM, and use directly the virtual address corresponding to the function she is monitoring.
- **Step 2 Capture packet, mask and replace:** The attacker captures the packet that is being sent and masks it in those positions that are useful for the attack. Then she sends the modified packet to the victim.
- **Step 3 Flush targeted function from cache:** The flush and reload process starts after the attacker replaces the original version of the packet and sends it. The co-located VM flushes the function to ensure that no one but the victim ran the targeted function. Any subsequent execution of the targeted function will bear a faster reload time during the reload process.
- **Step 4 Reload target function & measure:** Reload the corresponding function memory line again and measure the reload time. According to a threshold that we set based on experimental measurements, we decide whether the dummy function was loaded from the cache (implying that the victim has executed the dummy function earlier) or was loaded from the main memory (implying the opposite).

Since the attacker has to deal with instruction prefetching, she will be constantly running *Flush+Reload* for a specified period of time. The attacker therefore distinguishes between functions preloaded and functions preloaded *and executed*, since the latter will stay for a longer period of time in the cache.

6. EXPERIMENT SETUP AND RESULTS

In this section we present our test environment together with our detection method in order to deal with different cache prefetch techniques that affect our measurements. Finally we present the results of our experiments for the PolarSSL, GnuTLS and CyaSSL libraries.

6.1 Experiment Setup

The experiments were run on an Intel i5-650 dual core at 3.2 GHz. Our physical server includes 256 KB per core L2 cache, and a 4 MB L3 cache shared between both cores. We used VMware ESXI 5.5.0 build number 162338 for virtualization. TPS is enabled with 4 KB pages. In this setting, our *Flush+Reload* technique can distinguish between L3 cache and main memory accesses.

For the TLS connection, we use an echo server which reads and re-sends the message that it receives, and a client communicating with it. Client and echo server are running in different virtual machines that use Ubuntu 12.04 guest OS. We modify the echo server functionality so that it adds a jitter in the encrypted reply message, modeling the Man in the Middle Attack. Once the message is sent, the echo server uses *Flush+Reload* to detect different function calls and concludes if the padding was correct or not. For the TLS connection, we use an echo server which reads and resends the message that it receives, and a client communicating with it. Client and echo server are running in different virtual machines that use Ubuntu 12.04 guest OS. We modify the echo server functionality so that it adds a jitter in the encrypted reply message, modeling the Man in the Middle Attack. Once the message is sent, the echo server uses *Flush+Reload* to detect different function calls and concludes if the padding was correct or not.

6.2 Dealing with Cache Prefetching

Modern CPUs implement cache prefetching in a number of ways. These techniques affect our experiments, since the monitored function can be prefetched to cache, even if it was not executed by the victim process. To avoid false positives, it is not sufficient to detect *if* the monitored functions were loaded to cache, but also for *how long* they have resided in the cache. This is achieved by counting the number of subsequent detections for the given function in one execution. Therefore, the attack process effectively distinguishes between *prefetched* functions and *prefetched and executed* functions.

We use experiments to determine a threshold (which differs across the libraries) to distinguish a prefetch and execute from a mere prefetch. For PolarSSL this threshold is based on observing three *Flush+Reload* accesses in a row. Assume that n is the number of subsequent accesses required to conclude that the function was executed. In the following we present the required hits for different libraries, i.e. the number of n-accesses required to decide whether the targeted function was executed or not.

6.3 Attack on PolarSSL1.3.6

Our first attack targets PolarSSL 1.3.6, with TLS 1.1. In the first scenario the attacker modifies the last two bytes of the encrypted message until she finds the Δ that leads to a 0x01|0x01 padding. Recall that 2^{16} different variations can be performed in the message. The first plot shows the success probability of guessing the right Δ versus L, where L refers to the number of 2^{16} traces needed. For example $L = 4$ means that $2^{16} * 4$ messages are needed to detect the right Δ. Based on experimental results, we set the access threshold such that we consider a hit whenever the targeted function gets two accesses in a row.

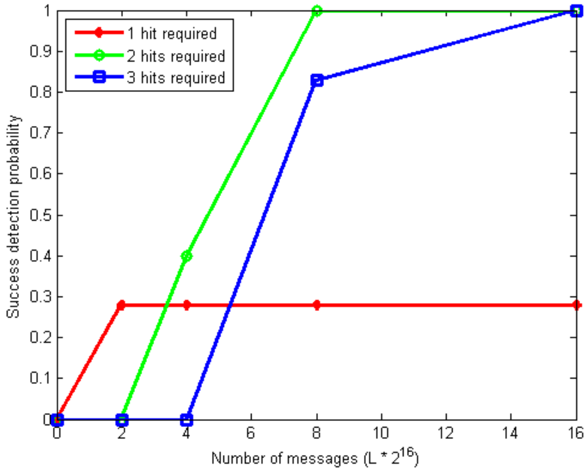

Figure 5: (PolarSSL 1.3.6) Success probability of recovering P_{14} and P_{15} vs. L, for different number of hits required. L refers to the number of 2^{16} traces needed, so the total number of messages is $2^{16} * L$.

The measurements were performed for different number of required hits. Figure 5 shows that requiring a single hit might not suffice since the attacker gets false positives, or for small number of messages she may miss the access at all. However when we require two hits, and if the attacker has a sufficient number of messages (in this case $L = 2^3$), the probability of guessing the right Δ is comfortably close to one. If the attacker increases the limit further to ensure an even lower number of false positives, she will need more messages to see the required number of hits. In the case of 3 hits, $L = 2^4$ is required to have a success probability close to one.

Figure 6 shows the success probability of correctly recovering P_{13}, once the attacker has recovered the last two bytes. Now the attacker is looking for the padding 0x02|0x02|0x02. We observed a similar behavior with respect to the previous case where with $L = 8$ and with a two hits requirement we will recover the correct byte with high probability. Again if the attacker increases the requirement to 3 hits, she will need more measurements; about $L = 16$ is sufficient in practice.

6.4 CyaSSL 3.0.0

Recall that the attack is much more effective if the attacker knows any of the preceding bytes of the plaintext, for example the last byte P_{15} of the plaintext. This would be the case in a javascript/web setting where adjusting the length of an initial HTTP request an attacker can ensure that there is only one unknown byte in the HTTP plaintext. In this case, the attacker would not need to try 2^{16} possible variations but only 2^8 variations for each byte that she wants to recover. This is the scenario that we analyzed in CyaSSL TLS 1.2, where we assumed that the attacker

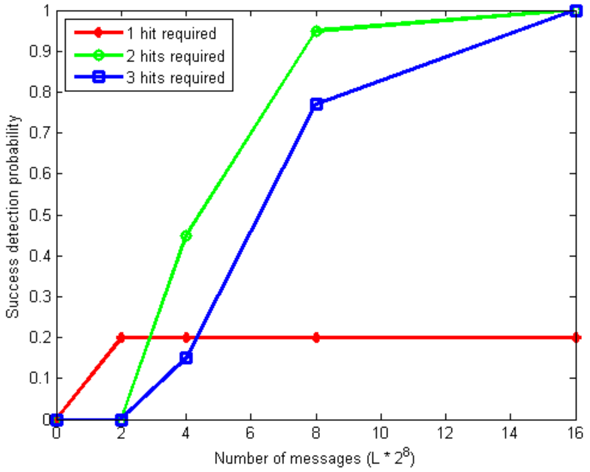

Figure 6: (PolarSSL 1.3.6) Success probability of recovering P_{13} assuming P_{14}, P_{15} known vs L, for different number of hits required. L refers to the number of 2^8 traces needed, so the total number of messages is $2^8 * L$.

knows P_{15} and she wants to recover P_{14}. Now the attacker is again trying to obtain a 0x01|0x01 padding, but unlike in the previous case, she knows the Δ to make the last byte equal to 0x01. The implementation of CyaSSL behaves very similarly to the one of PolarSSL, where due to the access threshold, a one hit might lead to false positives. However, requiring two hits with a sufficient number of measurements is enough to obtain a success probability very close to one. The threshold is set as in the previous cases, where a hit is considered whenever we observe two *Flush+Reload* accesses in a row.

6.5 GnuTLS 3.2.0

Finally we present the results confirming that GnuTLS3.2.0 TLS 1.2 is also vulnerable to this kind of attack. Again, the measurements were taken assuming that the attacker knows the last byte P_{15} and she wants to recover P_{14}, i.e., she wants to observe the case where she injects a 0x01|0x01 padding. However GnuTLS's behavior shows some differences with respect to the previous cases. For the case of GnuTLS we find that if we set an access threshold of three accesses in a row (which would yield our desired hit), the probability of getting false positives is very low. Based on experimental measurements we observed that only when the dummy function is executed we observe such a behavior. However the attacker needs more messages to be able to detect one of these hits. Observing one hit indicates with high probability that the function was called, but we also consider the two hit case in case the attacker wants the probability of having false positives to be even lower. Based on the measurements we conclude that the attacker recovers the plaintext with very high probability, so we did not find it necessary to consider the three hit case.

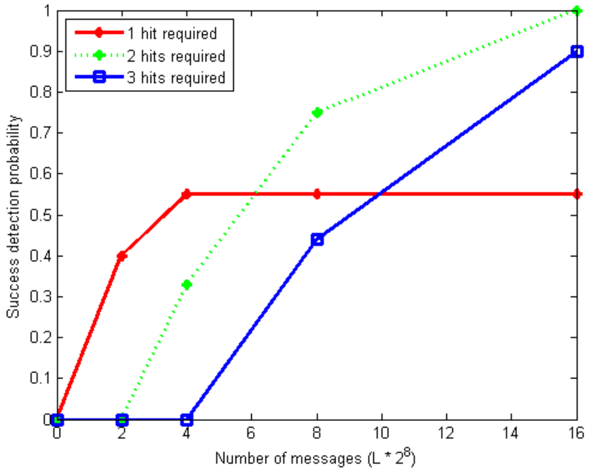

Figure 7: (CyaSSL3.0.0) Success Probability of recovering P_{14} assuming P_{15} known vs L, for different number of hits required. L refers to the number of 2^8 traces needed, so the total number of messages would be $2^8 * L$.

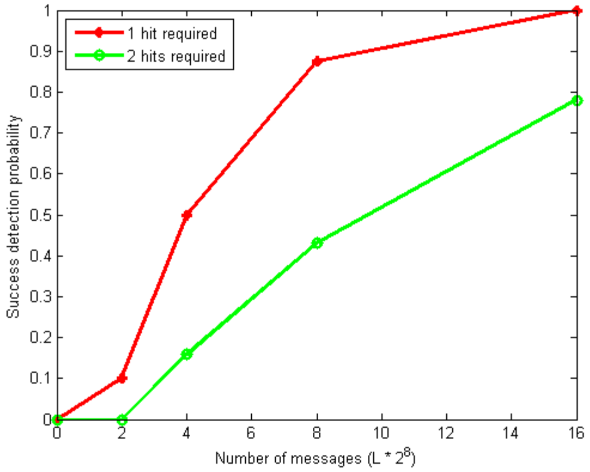

Figure 8: (GnuTLS3.2.0) Success Probability of recovering P_{14} assuming P_{15} known vs. L, for different number of hits required. L refers to the number of 2^8 traces needed, so the total number of messages would be $2^8 * L$.

7. COUNTERMEASURES

In this section we present various countermeasures that would prevent an attacker from implementing our modified Lucky 13 attack in a cloud environment. We first discuss software countermeasures, i.e, changes that can be made in the vulnerable cryptographic libraries to avoid the Lucky 13 attack. Then, we discuss more generic countermeasures to avoid the usage of *Flush+Reload* as a side channel technique to recover information. Note that library patches are less costly to implement than hardware based countermeasures.

On the downside, the software patches result in sub-optimal utilization of the memory hierarchy, thus, affecting the execution time performance.

Countermeasures in the cryptographic library: As our earlier survey of the library patches has revealed, there are two primary principles one needs to employ to securely patch cryptographic libraries against the cross-VM Lucky 13 attack:

- **Same function for valid/invalid padded cases:** The first pitfall that should be avoided takes place when a separate function call, e.g. a dummy function, is made to achieve a constant time implementation. This was part of the leakage exploited in this work where we monitor the dummy function calls made by another victim. In order to prevent it, a single function should be used during the entirety of the MAC operation of the message, as well as the additionally needed compression stages.
- **Same execution flow for valid/invalid padded cases:** This means that cryptographic library designers should avoid using message or key dependent branches that can leak information to an adversary monitoring the execution flow. Instead, logical operations like AND or XOR operations should be used to make the execution independent of vulnerable inputs. For instance, this solution has been adopted by OpenSSL, which calculates and always executes the maximum number of possible compression function calls.

An example algorithm that embodies these principles is presented in Algorithm 1. In the algorithm we are assuming that the maximum length of the processed message is 64 bytes, and that hash operations take 16 bytes of plaintext and that l is the length of the message once the padding is removed (for both correctly and incorrectly padded cases). The `md_process` function is used to perform the hash operations over all message blocks. This function puts the output in the hash variable. However, we use l to decide whether the output of the hash operation should be appended to the digest or not, depending on whether we are processing dummy data or the message. Note that the algorithm only uses a single function for both the valid message and the dummy data, thereby preventing execution flow distinguishing attacks. The code unifies the two separate execution flows.

Preventing *Flush+Reload* : Since our version of the Lucky 13 attack uses the *Flush+Reload* technique to extract timing information, any *Flush+Reload* countermeasure will also disable our attack. Here we note a few common *Flush+Reload* countermeasures.

- **Disabling deduplication features:** Our detection method is based on shared memory features that are offered by VMMs. Although these features have the advantage of significantly saving memory, they can also be used as a side channel to snoop sensitive information from a co-located user. Therefore, disabling deduplication closes the covert channel necessary to perform the attack presented in this work.
- **Cache Partitioning:** This countermeasure should be performed at the hardware level, and consists in splitting the cache into pieces so that each user uses only a *private* portion of the cache. In this scenario even when memory deduplication is enabled, an attacker could not interfere with the victim's data in the cache, and would no longer be able to distinguish whether the monitored function was used or not.

Algorithm 1: Data independent execution flow for md_process

```
//M=Message,l=length Message without padding
Input  : M,l
//Digest of M
Output: digest(M)
//Assume hash operates on a 16 byte message,
  and we have a maximum length of 64
for $i = 0$ to 4 do
    valid=(16*i/l);
    md_process(M[16*i]*$\overline{\text{valid}}$ + dummy_data*valid, hash);
    Append(digest[i],hash*$\overline{\text{valid}}$);
end
return digest;
```

- **Masking the cache loads:** This is a hardware-based countermeasure as well, where each user has a private masking value that is used when the data is loaded into cache and when the data being read from the cache. Since different users have different masking values, even when memory deduplication is enabled, attacker and victim would access the same data in memory through different cache addresses, preventing the attack in this work.

8. CONCLUSION

In this work we demonstrated that the Lucky 13 attack is still a threat in the cross-VM setting for a number of prominent cryptographic libraries already patched for the Lucky 13 attack. We discussed the different approaches taken by the major TLS libraries and showed that one class of timing side channel countermeasure, i.e, using dummy functions to achieve constant time execution, is vulnerable to cross-VM *Flush+Reload* attacks. With practical experiments we demonstrated that the side channel enabling Lucky 13 is still existent in PolarSSL, GnuTLS and CyaSSL if run in a deduplication enabled virtual machine. In fact, the new cache side channel is actually stronger, since it no longer suffers from network noise, making the attack succeed with significantly fewer observations than the original Lucky 13 attack in [16]. We also discussed how various crypto libraries fixed the Lucky 13 vulnerability in detail to better explain what makes a crypto library vulnerable to *Flush+Reload* based attacks.

In our test setting, we used the VMware ESXi with TPS enabled. This deduplication feature enabled us to detect dummy function calls that are implemented by the vulnerable libraries to equalize HMAC execution time in the case of incorrectly CBC-padded packets in TLS. Unlike in the case of vulnerable libraries, OpenSSL, Mozilla NSS, and MatrixSSL applied patches with a constant and padding-independent program flow to fix the Lucky 13 vulnerability. Libraries fixed this way are secure against the described attack.

With this study we showed that crypto library designers and authors should be careful about not implementing any data dependent execution paths and ensure true constant execution time. We conclude that, any function or process in a crypto library whose execution depends on the input data is exploitable by cache side-channel attacks and that libraries should be implemented accordingly.

9. ACKNOWLEDGMENTS

This work is supported by the National Science Foundation, under grant CNS-1318919 and CNS-1314770. We would like to thank the anonymous reviewers of AsiaCCS 2015 for their helpful comments. We would also like to thank Craig Shue for his help on understanding memory deduplication features.

10. REFERENCES

[1] CyaSSL: Embedded SSL library WolfSSL. http://www.wolfssl.com/yaSSL/Home.html, May 2014.

[2] Kernel samepage merging. http://kernelnewbies.org/Linux_2_6_32\#head-d3f32e41df508090810388a57efce73f52660ccb/, April 2014.

[3] MatrixSSL: Open source embedded SSL. May 2014.

[4] Aciİçmez, O. Yet another microarchitectural attack: Exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture* (New York, NY, USA, 2007), CSAW '07, ACM, pp. 11–18.

[5] AlFardan, N. J., Bernstein, D. J., Patterson, K. G., Poettering, B., and Schuldt, J. C. N. On the Security of RC4 in TLS. In *22nd USENIX Security Symposium* (2013).

[6] Arcangeli, A., Eidus, I., and Wright, C. Increasing memory density by using KSM. In *Proceedings of the Linux symposium* (2009), pp. 19–28.

[7] Bard, G. A challenging but feasible blockwise-adaptive chosen-plaintext attack on SSL. In *SECRYPT* (2006), pp. 99–109.

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

[9] Bardou, R., Focardi, R., Kawamoto, Y., Simionato, L., Steel, G., and Tsay, J.-K. Efficient padding oracle attacks on cryptographic hardware. In *CRYPTO* (2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*, Springer, pp. 608–625.

[10] Benger, N., van de Pol, J., Smart, N. P., and Yarom, Y. "ooh aah... just a little bit": A small amount of side channel can go a long way. In *CHES* (2014), pp. 75–92.

[11] Bernstein, D. J. Cache-timing attacks on AES, 2004. URL: http://cr.yp.to/papers.html#cachetiming.

[12] Bonneau, J. Robust Final-Round Cache-Trace Attacks against AES.

[13] Bonneau, J., and Mironov, I. Cache-Collision Timing Attacks against AES. In *Cryptographic Hardware and Embedded Systems—CHES 2006* (2006), vol. 4249 of *Springer LNCS*, Springer, pp. 201–215.

[14] Chen Cai-Sen, Wang Tao, C. X.-C., and Ping, Z. An improved trace driven instruction cache timing attack on RSA. Cryptology ePrint Archive, Report 2011/557, 2011. http://eprint.iacr.org/.

[15] Duong, T., and Rizzo, J. Here come the XOR ninjas.

[16] Fardan, N. J. A., and Paterson, K. G. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on* (May 2013), pp. 526–540.

[17] Goodin, D. Hackers break SSL encryption used by millions of sites. http://www.theregister.co.uk/2011/09/19/beast_exploits_paypal_ssl/, 2011.

[18] Gullasch, D., Bangerter, E., and Krenn, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. *IEEE Symposium on Security and Privacy 0* (2011), 490–505.

[19] Gullasch, D., Bangerter, E., and Krenn, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2011), SP '11, IEEE Computer Society, pp. 490–505.

[20] Hu, W.-M. Lattice scheduling and covert channels. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 1992), SP '92, IEEE Computer Society, pp. 52–.

[21] Irazoqui, G., İncİ, M. S., Eisenbarth, T., and Sunar, B. Fine grain Cross-VM Attacks on Xen and VMware are possible. preprint available at http://ecewp.ece.wpi.edu/wordpress/vernam/files/2014/04/main.pdf.

[22] Jones, M. T. Anatomy of Linux kernel shared memory. http://www.ibm.com/developerworks/linux/library/l-kernel-shared-memory/l-kernel-shared-memory-pdf.pdf/, April 2010.

[23] Kelsey, J., Schneier, B., Wagner, D., and Hall, C. Side channel cryptanalysis of product ciphers. In *Computer SecurityUESORICS 98*. Springer, 1998, pp. 97–110.

[24] Mavrogiannopoulos, N., and Josefsson, S. GnuTLS: The GnuTLS Transport Layer Security Library. May 2014.

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

[26] Mozilla. Mozilla NSS: Network security services. May 2014.

[27] Osvik, D. A., Shamir, A., and Tromer, E. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology* (Berlin, Heidelberg, 2006), CT-RSA'06, Springer-Verlag, pp. 1–20.

[28] Page, D. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel, 2002.

[29] Paterson, K. G., Ristenpart, T., and Shrimpton, T. Tag size does matter: Attacks and proofs for the TLS record protocol. In *Advances in Cryptology–ASIACRYPT 2011*. Springer Berlin Heidelberg, 2011, pp. 372–389.

[30] PolarSSL. PolarSSL: Straightforward,secure communication. `www.polarssl.org`.

[31] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 199–212.

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

[33] Suzaki, K., Iijima, K., Yagi, T., and Artho, C. Memory deduplication as a threat to the guest OS. In *Proceedings of the Fourth European Workshop on System Security* (2011), ACM, p. 1.

[34] Suzaki, K., Iijima, K., Yagi, T., and Artho, C. Software side channel attack on memory deduplication. *SOSP POSTER* (2011).

[35] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. `www.openssl.org`, April 2003.

[36] Tromer, E., Osvik, D., and Shamir, A. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology 23*, 1 (2010), 37–71.

[37] Tsunoo, Y., Saito, T., Suzaki, T., and Shigeri, M. Cryptanalysis of DES implemented on computers with cache. In *Proc. of CHES 2003, Springer LNCS* (2003), Springer-Verlag, pp. 62–76.

[38] Vaudenay, S. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS. In *Proceedings of In Advances in Cryptology - EUROCRYPT'02* (2002), Springer-Verlag, pp. 534–546.

[39] VMWare. Understanding Memory Resource Management in VMware vSphere 5.0. `http://www.vmware.com/files/pdf/mem_mgmt_perf_vsphere5.pdf`.

[40] Waldspurger, C. A. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 181–194.

[41] Yarom, Y., and Falkner, K. Flush+reload: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 719–732.

[42] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 305–316.