The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations

Eyal Ronen*, Robert Gillham†, Daniel Genkin‡, Adi Shamir*, David Wong§, and Yuval Yarom†**
*Weizmann Institute
†University of Adelaide
‡University of Michigan
§NCC Group
**Data61

***Abstract*—At CRYPTO'98, Bleichenbacher published his seminal paper which described a padding oracle attack against RSA implementations that follow the PKCS #1 v1.5 standard. Over the last twenty years researchers and implementors had spent a huge amount of effort in developing and deploying numerous mitigation techniques which were supposed to plug all the possible sources of Bleichenbacher-like leakages. However, as we show in this paper most implementations are still vulnerable to several novel types of attack based on leakage from various microarchitectural side channels: Out of nine popular implementations of TLS that we tested, we were able to break the security of seven implementations with practical proof-of-concept attacks. We demonstrate the feasibility of using those Cache-like ATacks (CATs) to perform a downgrade attack against any TLS connection to a vulnerable server, using a BEAST-like Man in the Browser attack. The main difficulty we face is how to perform the thousands of oracle queries required before the browser's imposed timeout (which is 30 seconds for almost all browsers, with the exception of Firefox which can be tricked into extending this period). The attack seems to be inherently sequential (due to its use of adaptive chosen ciphertext queries), but we describe a new way to parallelize Bleichenbacher-like padding attacks by exploiting any available number of TLS servers that share the same public key certificate. With this improvement, we could demonstrate the feasibility of a downgrade attack which could recover all the 2048 bits of the RSA plaintext (including the premaster secret value, which suffices to establish a secure connection) from five available TLS servers in under 30 seconds. This sequential-to-parallel transformation of such attacks can be of independent interest, speeding up and facilitating other side channel attacks on RSA implementations.**

I. Introduction

"Those who'll play with cats must expect to be scratched."
– Miguel de Cervantes, Don Quixote.

The Public Key Cryptography Standard #1 (PKCS #1) [60] is the main standard used for implementing the RSA public key algorithm [58] in a large variety of security protocols. Twenty years ago, Bleichenbacher [11] demonstrated that the padding scheme defined in PKCS #1 v1.5 (which is used to map shorter messages into full length RSA plaintexts) is vulnerable to a *padding oracle* attack. Specifically, given an indication whether the plaintext which corresponds to a given ciphertext is correctly formatted, an attacker can mount an adaptive chosen ciphertext attack which recovers the full plaintext from any given ciphertext.

Since its publication, multiple Bleichenbacher-like attacks have been demonstrated, exploiting a large variety of oracles, including error messages [12, 41], timing variations [39, 47] and memory access patterns [69]. After each attack, implementors had adopted some mitigation techniques to ensure that the use of PKCS #1 v1.5 does not leak information on the padding, but these mitigation techniques had become increasingly difficult to understand, to implement, and to maintain. Considering the number of demonstrated attacks and the ongoing mitigation efforts, we set out in this paper to answer the following basic question:

Are modern implementations of PKCS #1 v1.5 secure against padding oracle attacks?

A. Our Contribution.

Regrettably, our answer to this question is negative, as the vast majority of implementations we evaluated are still vulnerable to padding oracle attacks. Making the situation worse, we show that padding oracle attacks can be made extremely efficient, via more careful analysis and novel parallelization techniques. Finally, we show that while the use of RSA key exchange is declining, padding oracles can be used to mount downgrade attacks, posing them as a threat to the security of a much larger number of connections (including those done via protocols that do not even support the RSA key exchange).

More specifically, our contributions are as follows.

New Techniques for Microarchitectural Padding Oracle Attacks. We have tested nine fully patched implementations of various RSA-based security protocols (OpenSSL, Amazon s2n, MbedTLS, Apple CoreTLS, Mozilla NSS, WolfSSL, GnuTLS, BearSSL and BoringSSL). While all of these implementations attempt to protect against microarchitectural and timing side channel attacks, we describe new side channel attack techniques which overcome the padding oracle countermeasures. Notably, out of the nine evaluated implementations, only the last two (BearSSL and BoringSSL) could not be successfully attacked by our new techniques.

Downgrade Attacks. Next, we show the feasibility of performing downgrade attacks against all the deployed versions of TLS, including the latest TLS 1.3 standard (which does not even support RSA key exchange). More specifically, even though the use of RSA in secure connections is diminishing (only $\approx 6\%$ of TLS connections currently use RSA [1, 50]), this fraction is still too high to allow vendors to drop this mode. However, as we show in Section VI, supporting this small fraction of users puts everyone at risk, as it allows the attacker to perform a downgrade attack by specifying RSA as the only public key algorithm supported by the server.

Attack Efficiency. Rather then targeting premaster secrets of individual connections, we adopt a BEAST-like [23] approach, targeting instead the long term login tokens. As only a single broken connection is sufficient to recover the login token, in Section VI we show that the query complexity of padding oracle attacks can be substantially reduced (at the expense of the success probability of breaking a specific connection), while still preserving the attacker's ability to extract login tokens before the connection timeout enforced by almost all web browsers.

Attack Parallelization. As another major contribution, we show in this paper a novel connection between padding oracle attacks and the Closest Vector Problem (CVP). While there are some techniques to parallelize parts of padding oracle queries [41], those techniques could not overcome the fact that (perfect) padding oracles attacks use adaptively chosen ciphertexts, thereby requiring that some parts of the attack remain sequential. Using lattice reduction techniques we overcome this limitation, by combining the results from different (and partial) attack runs against different servers sharing the same RSA key. With those techniques in hand, we can show the feasibility of recovering a full 2048-bit RSA plaintext from five fully patched TLS servers in under the 30 second timeout enforced by almost all web browsers.

B. Software Versions and Responsible Disclosure

Our attacks were performed on the most updated versions of the cryptographic libraries evaluated, as published at the time of discovery. We compiled each library using its default compilation flags, leaving all side channel countermeasures in place. Following the practice of responsible disclosure, we disclosed our findings in August 2018 to all the vendors mentioned in this paper. We further participated in the design and empirical verification of the proposed countermeasures. Updated versions of the affected libraries are published concurrently with the publication of this preprint, in a coordinated public disclosure in November 2018. We note that, OpenSSL patched two of the vulnerabilities we discovered independently and in parallel to our disclosure process. (See Section IV-A and Appendix A-A.)

Issues identified in this work have been assigned the following CVE numbers: CVE-2018-12404, CVE-2018-19608, CVE-2018-16868, CVE-2018-16869, CVE-2018-16870.

II. Background

A. Padding Oracle Attacks on TLS

TLS has a long history of padding oracle attacks of different types. Those attacks led to the development and implementation of new mitigation techniques, and then new attacks.

The Lucky 13 attack by AlFardan and Paterson [5] showed how to use a padding oracle attack to break TLS CBC HMAC encryption. Irazoqui et al. [38] and Ronen et al. [59] have shown how to use cache attacks to attack code that has been patched against the original attack.

After the publication of the Bleichenbacher attack, the TLS specifications defined a new mitigation with the goal of removing the oracle [19, 20, 21]. However, it seems that completely removing the oracle is a very difficult task as was shown by multiple cycles of new attacks and new mitigations [12, 41, 47]. As we show in our paper, Bleichenbacher type attacks are still possible even on fully patched implementations.

B. RSA PKCS #1 v1.5 Padding

In this section we describe the PKCS #1 v1.5 padding standard, which dictates how a message should be padded before RSA encryption. Let (N, e) be an RSA public key, let (N, d) be the corresponding private key, and let ℓ be the length of N (in bytes). The encryption of a message m containing $k \leq \ell - 11$ bytes is performed as follows.

1) First, a random padding string PS of byte-length $\ell - 3 - k \geq 8$ is chosen such that PS does not contain any zero-valued bytes.
2) Set m^* to be $\mathsf{0x00}||\mathsf{0x02}||PS||\mathsf{0x00}||m$. Note that the length of m^* is exactly ℓ bytes.
3) Interpret m^* as an integer $0 < m^* < N$ and compute the ciphertext $c = {m^*}^e \bmod N$.

The decryption routine computes $m' = c^d \bmod N$ and parses m' as a bit string. It then checks whether m' is of the from $m' = \mathsf{0x00}||\mathsf{0x02}||PS''||\mathsf{0x00}||m''$ where PS'' is a string consisting of at least 8 bytes, all of them must be non-zero. In case this condition holds the decryption routine returns m''. Otherwise the decryption routine fails.

C. Bleichenbacher's Attack on PKCS #1 v1.5 Padding

In this section we provide a high level description of Bleichenbacher's "million message" attack [11] on the PKCS #1 v1.5 padding standard described above. At a high level, the attack allows an attacker to compute an RSA private key operation (e.g., $m^d \bmod N$) on a message m of his choice without knowing the secret exponent d.

Attack Prerequisites. Bleichenbacher's attack assumes the existence of an oracle Bl which given a ciphertext c as input answers whether c can be successfully decrypted using RSA PKCS #1 v1.5 padding as described above. More formally, let (N, d) be an RSA private key. The oracle Bl performs the following for every ciphertext c

$$\mathsf{Bl}(c) = \begin{cases} 1 \text{ if } c^d \bmod N \text{ has a valid PKCS \#1 v1.5 padding} \\ 0 \text{ otherwise} \end{cases}$$

As was previously shown, such an oracle can be obtained by several types of side channel leakage [12, 39, 41, 47, 69].

We now describe how an attacker can use the Bleichenbacher oracle Bl to perform an RSA secret key operation, such as decryption or signature, on c without knowing the secret exponent d. We refer the reader to [11] for a more complete description.

High Level Attack Description. Let c be an integer. To compute $m = c^d \bmod N$, the attack proceeds as follows.

- **Phase 1: Blinding.** The attacker repeatedly chooses random integers s_0 and computes $c^* \leftarrow c \cdot s_0^e \bmod N$. The attacker checks if c^* is a valid PKCS #1 v1.5 ciphertext by evaluating $\mathsf{Bl}(c^*)$. This phase terminates when an s_0 such that $\mathsf{Bl}(c^*) = 1$ is found. The phase and can be skipped completely if c is already a valid PKCS #1 v1.5 ciphertext in which case $s_0 = 1$.

We note that when the oracle succeeds ($\mathsf{Bl}(c^*) = 1$) the attacker knows that the corresponding message $m^* = m \cdot s_0 \bmod N$ starts with 0x0002. Thus, it holds that $m \cdot s_0 \bmod N \in [2B, 3B)$ where $B = 2^{8(\ell-2)}$ and ℓ is the length of N in bytes. Finally, the condition of $m \cdot s_0 \bmod N \in [2B, 3B)$ implies that there exists an integer r such that $2B \leq m \cdot s_0 - rN < 3B$, or equivalently:

$$\frac{2B + rN}{s_0} \leq m < \frac{3B + rN}{s_0}.$$

- **Phase 2: Range Reduction.** Having established that $\frac{2B+rN}{s_0} \leq m < \frac{3B+rn}{s_0}$, the attacker proceeds to choose a new random integer s, computes $c^* \leftarrow c \cdot s^e \bmod N$ and checks that $\mathsf{Bl}(c^*) = 1$. When a suitable s is found, the adversary can further reduce the possible ranges of m, see [11] for additional details. The attack terminates when the possible range of m is reduced to a single candidate.

Attack Efficiency. For N consisting of 1024-bits, Bleichenbacher's original analysis [11] requires about one million calls to the oracle Bl (e.g., requiring the attacker to observe one million decryptions). However, subsequent analysis has shown that the attack is possible with as little as 3800 oracle queries under realistic scenarios [7].

The Noisy Oracle Case. We note here that the Bleichenbacher attack does not require the oracle Bl to be perfect. Specifically, the attack can handle one sided errors where $\mathsf{Bl}(c) = 0$ for some valid PKCS #1 v1.5 ciphertexts (i.e. false negatives). All that the attack requires is that the attacker can correctly identify valid PKCS #1 v1.5 ciphertext sufficiently often.

D. Manger's Attack

Following Bleichenbacher's work, Manger [46] presented another padding oracle attack that allows an attacker to compute $c^d \bmod N$ without knowing the secret exponent d. Manger's attack, originally designed for attacking PKCS #1 v2.0, can be adapted to the PKCS #1 v1.5 case. The attack is more efficient than the Bleichenbacher attack, but it has different prerequisites.

Attack Prerequisites. In this case we assume the existence of an oracle Ma which given a ciphertext c answers whether the most significant byte of $c^d \bmod N$ is zero. More formally, let (N, d) be an RSA private key. The oracle Ma outputs the following for every ciphertext c

$$\mathsf{Ma}(c) = \begin{cases} 1 \text{ if } c^d \bmod N \text{ starts with } \texttt{0x00} \\ 0 \text{ otherwise} \end{cases}.$$

That is, the oracle outputs for a given ciphertext c whether its decryption $c^d \bmod N$ lies in the interval $[0, B-1]$ or not, where $B = 2^{8(\ell-1)}$ and ℓ is the length of N in bytes.

High Level Attack Description. Let $c = m^e \bmod N$ be a ciphertext. At a high level, Manger's attack is very similar to Bleichenbacher's attack, requiring the attacker to choose a value s, to compute $c^* \leftarrow c \cdot s^e \bmod N$ and to query Ma in an attempt to find a c^* such that $\mathsf{Ma}(c^*) = 1$.

Attack Efficiency. Manger's attack requires a little more than $\log_2(N)$ oracle calls to perform an RSA secret operation. This compares favorably with the approximate one million oracle calls required for the Bleichenbacher attack. However, in contrast to Bleichenbacher's attack, which can tolerate oracle false negatives, Manger's attack requires a "perfect" oracle which always answers correctly, without any errors.

E. The Interval Oracle Attack

Well before Bleichenbacher's work, Ben-Or et al. [8] proved the security of single RSA bits, by showing an algorithm for decrypting RSA ciphertexts given one bit of plaintext leakage. One of the oracles considered in that work is the interval oracle, that indicates if the plaintext is inside or outside a specific interval.

Attack Prerequisites. More specifically, for an RSA private key (N, d) assume we have an oracle that outputs the following for every ciphertext c

$$\mathsf{In}(c) = \begin{cases} 1 \text{ if } c^d \bmod N \text{ starts with bit } 1 \\ 0 \text{ otherwise} \end{cases}.$$

That is, the oracle outputs for a given ciphertext c whether its decryption $c^d \bmod N$ lies in the interval $[0, 2^{8\ell-1}]$ or not, where ℓ is the length of N in bytes.

High Level Attack Description. The main idea of the attack is to generate two random multiplications $c_1 = a \cdot c$ and $c_2 = b \cdot c$ of the ciphertext c, and then use an euclidean greatest common divisor (gcd) algorithm to compute $\gcd(c_1, c_2)$. When a pair of ciphertext c_1, c_2 is found such that $\gcd(c_1, c_2) = 1$, it is possible to efficiently recover $c^d \bmod N$. The gcd algorithm is calculated using the Interval oracle. See Ben-Or et al. [8] for a more complete description.

Attack Efficiency. The attack of Ben-Or et al. [8] is relatively efficient, requiring about $15 \log_2 N$ oracle queries to decrypt a ciphertext c. For a random choice of c_1 and c_2 the attack succeeds with a probability of $6/\pi^2$.

F. Notation and Additional Padding Oracle Attacks

Several works follow-up on the attacks of Ben-Or et al. [8], Bleichenbacher [11], and Manger [46], obtained similar results using other padding oracles commonly found in implementations of PKCS #1 v1.5, where some oracles provide more information than others [7, 41]. In this paper, we consider four different checks that an implementation can validate against the RSA-decrypted padded plaintext. All implementations start by checking that the padded plaintext starts with 0x0002, and then may proceed with further checks.

- The first check corresponds to the test for a zero byte somewhere after the first ten bytes of the plaintext.
- The second check verifies that there are no zero bytes in the padding string PS''.
- The third check verifies the plaintext length against some specific value (48 byte for a TLS premaster secret in our case).
- Finally, the fourth check is payload-aware and TLS-specific: it verifies the first two bytes of the payload; these bytes are set to the client's protocol version as defined in RFC-5246 [21].

Notation. We extend the notation of Bardou et al. [7] to refer to various oracles. Specifically, our notation is:

FFFF denotes an oracle that gets as input a ciphertext and returns true only if the corresponding plaintext passes all four checks. This is the same as the Bad-Version Oracle (BVO) of Klíma et al. [41].

FFFT denotes an oracle that returns true for ciphertexts corresponding to plaintexts that pass the first three checks, ignoring the fourth check.

FFTT is an oracle that only verifies first two checks. This is the Bleichenbacher oracle described in Section II-C

FTTT denotes an oracle that returns true if the decrypted plaintext passes the first check and disregards the last three checks.

TTTT is an oracle that disregards the four checks, returning true for ciphertexts whose corresponding plaintexts start with 0x0002.

M denotes a Manger oracle (Section II-D).

I denotes an Interval oracle (Section II-E).

G. The TLS Mitigation for the Bleichenbacher attack

The TLS specifications [19, 20, 21] define defences for the Bleichenbacher attack. The decrypted message m is used as a shared premaster secret between the client and the server. Crucially, the attacker does not know the plaintext of the messages sent as part of the attack, and cannot, therefore, distinguish random strings from correctly decrypted plaintexts. Thus, to mitigate the Bleichenbacher attack, the server regenerates a random premaster secret, and swaps it for the plaintext if the PKCS #1 v1.5 validation fails.

This choice of premaster secret depending on the validity of the padding must be done in constant-time as well. Unfortunately, correctly implementating this mitigation is a delicate task. Any differences in the server's behavior between the PKCS #1 v1.5 conforming and the non-conforming cases may be exploited to obtain a Bleichenbacher-type oracle [12, 47]. Although most implementations do attempt to implement constant-time code for this mitigation, we show that all but two are still vulnerable to microarchitectural side-channel attacks.

H. Microarchitectural Side Channels

To improve the performance of programs, modern processors try to predict the future program behavior based on its past behavior. Thus, processors typically cache some microarchitectural state that depends on past behavior and subsequently use that state to optimize future behavior. Unfortunately, when multiple programs share the use of the same microarchitectural components, the behavior of one program may affect the performance of another. Microarchitectural side channel attacks exploit this effect to leak otherwise unavailable information between programs [27].

Cache-Based Side Channel Attacks. Caching components, and in particular data and instruction caches, are often exploited for microarchitectural attacks. Cache-based attacks have been used to retrieve cryptographic keys [2, 10, 31, 37, 45, 55, 56, 63, 67], monitor keystrokes [32], perform website fingerprinting [54], and attack other algorithms [14, 65]. At a high level, cache attacks typically follow one of two patterns, which we now discuss.

FLUSH+RELOAD. In the FLUSH+RELOAD [67] attack and its variations [32, 33, 68], the attacker first evicts (flushes) a memory location from the cache. The attacker then waits a bit, before reloading the flushed location again, while measuring the time that this reload takes. If the victim accesses the same memory location between the flush and the reload phases, the memory will be cached, and access will be fast. Otherwise, the memory location will not be cached and the access will be slower. Thus, the attacker deduces information regarding the victim's access patterns to a given address.

PRIME+PROBE. Attacks employing the PRIME+PROBE technique [55, 56] or similar techniques [2, 22, 37, 45], first fill the cache with the attacker's data. The attacker then waits, allowing the victim to execute code before measuring the time to access the previously cached data. When the victim accesses its data, this data evicts some of the attacker's data from the cache. By measuring the access time to the previously cached data, the attacker can infer some information on the victim's memory access patterns.

Attack Limitations. Both attacks require that the victim and attacker share some CPU caching components, implying that both programs have to run on the same physical machine. At a high level, FLUSH+RELOAD tends to be more accurate and have fewer false positives than PRIME+PROBE [67]. However, FLUSH+RELOAD requires the attacker to share memory with the victim and thus is typically applied to monitoring victim code execution patterns, rather than data access patterns.

Branch-Prediction Attacks. The branch predictor of the processor has also been a target for microarchitectural attacks [3, 24, 25, 26, 43]. The branch predictor typically consists of two components, the Branch Target Buffer (BTB) which predicts branch destinations, and the Branch History Buffer (BHB), also known as the directional predictor, which predicts the outcome of conditional branches.

When a program executes a branch instruction, the processor observes the branch outcome and destination and modifies the state of the branch predictor. Attacks on the branch predictor exploit either the timing differences between correct or incorrect prediction or the performance monitoring information that the processor provides to recover the state of the predictor and detect the outcomes of prior branches executed by a victim program.

To mitigate Spectre attacks [42], Intel introduced mechanisms for controlling the branch predictor [36]. It is not clear whether these mechanisms completely eliminate branch prediction channels [28]. Furthermore, we have verified that by default Ubuntu Linux does not use the Indirect Branch Predictor Barrier mechanism to protect user processes from each other.

III. Attack Model and Methodology

In this paper, we target implementations of PKCS #1 v1.5 that leak information via microarchitectural side channels. We then exploit the leaked information to implement a padding oracle, which we use to decrypt or to sign a message. To mount our attacks the adversary needs three capabilities:

1. Side Channel Capability. The first capability an adversary needs is to mount a microarchitectural side channel attack against a vulnerable implementation. For that, the adversary needs the ability to execute code on the machine that runs the victim's implementation.

2. Privileged Network Position Capability. Our attacks exploit a padding oracle attack to perform a private key operation such as a signature or decryption of a message that has been sent to the victim. To decrypt a ciphertext and use its result, an adversary must first obtain a network man-in-the-middle position. To forge signatures, an adversary must first obtain the relevant data to sign and be in a privileged position to exploit it.

3. Decryption Capability. The third capability our adversary needs is the ability to trigger the victim server to decrypt ciphertexts chosen by the adversary.

A concrete attack scenario we consider in this work is attacking a TLS server running on the same physical hardware as an unprivileged attacker. For example, a TLS server running in a virtual machine on a public cloud server, where the physical server hardware is shared between the victim's TLS server and an attacker's virtual machine. Indeed, previous works have shown that attackers can achieve co-location [57], and leverage it for mounting side channel attacks [35]. Thus, the first capability is achievable for a determined adversary.

The second and third capabilities are achievable in this scenario by an attacker that controls any node along the path between the client and the server. Malicious network operators are one example of actors that have such control, but this is not the only case. In particular, attackers can exploit vulnerabilities in routers to assume control and mount our attack [18].

There are, however, some problems specific to this scenario. The recent version of the TLS protocol, TLS 1.3, no longer supports RSA key exchanges, and in TLS 1.2 (Elliptic Curve) Diffie-Hellman key exchanges are recommended over RSA key exchanges. Hence, the adversary needs to perform active protocol downgrade attacks to force the use of RSA in the communication. Furthermore, clients, such as browsers, impose time limits on the handshake, forcing the attacker to complete an attack that may require a large number of decryption within a short time. Section VI explains how we can perform such downgrade attacks, within the time limits.

IV. Vulnerability Classification

We now examine an outline of typical RSA PKCS #1 v1.5 implementations, explain where padding oracle vulnerabilities arise in these, and provide concrete examples from TLS implementations we investigated. Further examples are included in Appendix A.

Handling PKCS #1 v1.5 in TLS typically consists of three stages:

- **Data Conversion.** First, the RSA ciphertext is decrypted and the resulting plaintext is converted into a byte array.
- **PKCS #1 v1.5 Verification.** Next, the conformity of the array to the PKCS #1 v1.5 standard is checked.
- **Padding Oracle Mitigations.** Finally, if the array is not PKCS #1 v1.5 conforming, the server deploys the padding oracle countermeasures presented in Section II-G. As discussed, the risk of padding oracle attacks is only mitigated after the countermeasures are deployed.

Unfortunately, despite more than twenty years of research in both padding oracle attacks and side channel resistance, in this work we find that vulnerabilities still occur in all of these stages. We now provide a high level description of the various stages and their associated side channel vulnerabilities.

A. Data Conversion.

In RSA, the plaintext and the ciphertext are large numbers, e.g. 2048-bit long. These are typically represented as little-endian arrays of 32- or 64-bit words. PKCS #1 v1.5, however uses big-endian byte arrays, thus requiring a format conversation. For values of fixed length, this conversation is relatively straightforward. However, while the length of the RSA modulus provides an upper bound on the length of the RSA decryption result, the exact length of the RSA plaintext is not known until after RSA decryption of the corresponding ciphertext. Thus, if the RSA decryption result is too short, the little-to-big endian conversation code has to pad the ciphertext with a sufficient amount of zero bytes.

```
int RSA_padding_check_none(to, tlen, from, flen){
// to is the output buffer of maximum length tlen
    bytes
// from is the input buffer of length flen bytes
  memset(to, 0, tlen - flen);
  memcpy(to + tlen - flen, from, flen);
  return tlen;
}
```

Listing 1. Pseudocode of raw plaintext copy with no padding check function

As an example, consider the pseudo code of the implementation of the OpenSSL function RSA_padding_check_none in

Listing 1. The function is called as part of the implementation of the TLS protocol in OpenSSL, and its purpose is to copy the RSA decryption results to an output buffer, without perfoming any padding checks.

To handle the case that the plaintext from the RSA decryption is smaller than the output buffer, RSA_padding_check_none uses memset to pad the output buffer. The length of the padding is set to the difference between the lengths of the output array and the plaintext. In case of a full-length plaintext, the length of the padding is zero. Using a branch prediction attack we can detect this scenario, and learn whether the plaintext is full-length. This gives us the oracle required for a Manger attack.

Unfortunately, this example is by no means unique, and multiple implementation expose FTTT- or Manger-type padding oracles during the data conversion phase. See Appendix A for further examples.

B. PKCS #1 v1.5 Verification

Once the data is represented as a sequence of bytes, the implementation needs to check that it is PKCS #1 v1.5 conforming, that is, that the first byte is zero, the second is 0x02, the following eight bytes are non-zero, and that there is a zero byte at a position above 10. Yet, many implementations branch on the results of these checks, leaking the outcome to a side channel attacker via the implementation's control flow. The exact oracle obtained depends on the specific implementation and the type of leakage.

The OpenSSL RSA PKCS #1 v1.5 decryption API provides an example of such an issue. OpenSSL exports a function, RSA_public_decrypt, whose arguments are an input buffer containing the ciphertext, an output buffer for the plaintext, the RSA decryption key, and the padding mode to check the plaintext against. When using PKCS #1 v1.5 padding, RSA_public_decrypt invokes RSA_padding_check_PKCS1_type_2 to validate the padding after decryption. A pseudocode of the validation function is shown in Listing 2.

As the pseudocode shows, OpenSSL performs the checks outlined in Section II-B in constant-time (Lines 7–13), returning the length of the decrypted message if the decryption is successful, or -1 if there is a padding error. To set the return value, the function uses an explicit branch (Line 17). Furthermore, the memory copy in Line 21 is only executed in case of a successful decryption, whereas the error logging (Line 25) is invoked in the case of a padding error.

A comment in the code (Line 15) indicates that the authors are aware of the leakage, and the manual page for the function warns against its use [53]. Thus, OpenSSL does not use this PKCS #1 v1.5 verification code for its own implementation of the TLS protocol. Furthermore, both Xiao et al. [64] and Zhang et al. [69] exploit the leakage through the conditional error logging for mounting Bleichenbacher attacks.

Amazon's s2n. OpenSSL is the cryptographic engine underlying many applications, all of these are potentially vulnerable to our cache-based padding oracle attack. Specifically, Amazon's implementation of the TLS protocol, s2n [61], uses this API, and consequently leaks an FFTT-type oracle. For other vulnerabilities in s2n, see Appendix A-B.

C. Padding Oracle Mitigations.

As Section II-G describes, when a TLS implementation detects that a plaintext does not conform to the PKCS #1 v1.5 format, it cannot just terminate the handshake, because this creates a padding oracle. Instead it must replace the non-conforming plaintext with a random sequence of bytes and proceed with the TLS handshake. However, some implementations fail to protect this replacement, leaking the deployment of the countermeasure and allowing the creation of a padding oracle.

We can find examples of such leakage in CoreTLS, Apple's implementation of the TLS protocol that is sometimes used in both MacOS and iOS devices.

Listing 3 shows the code that handles the mitigation of Bleichenbacher's attack in CoreTLS (i.e., replacing the incorrectly-padded RSA plaintext with random data). Lines 7 and 8 perform the RSA decryption and the validation of the PKCS #1 v1.5 format. The code logs validation failure of the PKCS #1 v1.5 format in Line 11. It also checks that the output is of the expected length, issuing a log message on failure (Lines 13–17). For brevity we omit the code that handles the success case (Line 20). The main mitigation against Bleichenbacher attacks occurs in Line 24, where the code generates a random value to be used as the session key.

While the PKCS #1 v1.5 padding verification code in CoreTLS constant time, the code that handles the mitigations against padding oracle attacks is far from constant time. As seen in Listing 3, the code contains multiple sources of side channel leakage which we now describe.

First, all of the conditional if statements in the presented code can be exploited by branch prediction attacks to implement FFTT (Line 9), FFFT (Line 13), or FFFF (Lines 19 and 22) Bleichenbacher-type oracles.

Next, a cache attack can monitor either the code of the log message function or the code of the random number generator, which only runs if the PKCS #1 v1.5 validation fails. Another option is to monitor the bodies of the if statements in Lines 19 or 22. These attacks can be used to implement an FFFF-type padding oracle.

Finally, generating the random session key only on PKCS #1 v1.5 validation failure (Line 24) is a significant weakness in the implementation. Random number generation is a non-trivial operation that may take significant time and thus might expose a Bleichenbacher oracle via a timing attack. That is, by simply measuring the response time of a TLS server that uses the CoreTLS library, an attacker might get a FFFF-type Bleichenbacher oracle.

D. Summary of the Findings.

Table I summarizes our findings, showing the identified oracle types in each stage of the implementations we evaluated. As we can see, seven of the nine tested implementations expose padding oracles via microarchitectural attacks. Only BearSSL and Google's BoringSSL are not vulnerable to our attacks.

```
1  int RSA_padding_check_PKCS1_type_2(to, tlen, from, flen, num_bytes){
2    // to is the output buffer of maximum length tlen bytes
3    // from is the input buffer of length flen bytes
4    // num_bytes is the maximum number of bytes in an RSA plaintext
5    // returns the number of message bytes (not counting the padding) or -1 in case of a padding error
6
7    good = constant_time_is_zero(from[0]);
8    good &= constant_time_eq(from[1], 2);
9    zero_index = find_index_of_first_zero_byte_constant_time(from+2, flen);
10   good &= constant_time_greaterOrEqual(zero_index, 2 + 8); //first 10 plaintext bytes must be non-zero
11   msg_index = zero_index + 1; //compute location of first message byte
12   msg_len = num_bytes - msg_index; //compute message length
13   good &= constant_time_greaterOrEqual(tlen, msg_len); //check that to buffer is long enough
14
15   /* We can't continue in constant-time because we need to copy the result and we cannot fake its length.
        This unavoidably leaks timing information at the API boundary. */
16
17   if (!good) {
18     mlen = -1;
19     goto err;
20   }
21   memcpy(to, from+msg_index, mlen);
22
23   err:
24   if (mlen == -1)
25      RSAerr(RSA_F_RSA_PADDING_CHECK_PKCS1_TYPE_2, RSA_R_PKCS_DECODING_ERROR);
26   return mlen;
27 }
```

Listing 2. Pseudocode of RSA_padding_check_PKCS1_type_2

```
int SSLDecodeRSAKeyExchange(keyExchange, ctx){
  keyRef = ctx->signingPrivKeyRef;
  src  = keyExchange.data;
  localKeyModulusLen = keyExchange.length;
  ... // addtional inilization code omitted

  err = sslRsaDecrypt(keyRef, src,
    localKeyModulusLen,
    ctx->preMasterSecret.data,
    SSL_RSA_PREMASTER_SECRET_SIZE, &outputLen);
  if(err != errSSLSuccess) {
    /* possible Bleichenbacher attack */
    sslLogNegotiateDebug("SSLDecodeRSAKeyExchange:
                           RSA decrypt fail");
  } else if(outputLen !=
                  SSL_RSA_PREMASTER_SECRET_SIZE) {
    sslLogNegotiateDebug("SSLDecodeRSAKeyExchange:
    premaster secret size error");
    // not passed back to caller
    err = errSSLProtocol;
  }
  if(err == errSSLSuccess) {
      ... // (omitted for brevity)
  }
  if(err != errSSLSuccess) {
    ... // (omitted for brevity)
    sslRand(&tmpBuf);
  }
  /* in any case, save premaster secret (good or
    bogus) and proceed */
  return errSSLSuccess;
}
```

Listing 3. Apple's TLS mitigation function

TABLE I
SUMMARY OF IDENTIFIED PADDING ORACLES.

	Data Conv.	PKCS #1 v1.5 Verification	TLS Mitigation
OpenSSL	M	M	
OpenSSL API	M	FFTT	
Amazon s2n		FFFT	
MbedTLS	I	FFTT, FFFT*	
Apple CoreTLS			FFTT, FFFT, FFFF
Mozilla NSS	M	M, TTTT, FTTT*	FFFF
WolfSSL	M	M, FFTT	FFTT, FFFF
GnuTLS	M	M, TTTT, FFTT	FFTT, FFFT
BoringSSL		*Not Vulnerable*	
BearSSL		*Not Vulnerable*	

V. EXPERIMENTAL RESULTS

To validate that the vulnerabilities we identified can indeed be exploited, we mounted concrete side-channel attacks on some of the implementations. We now discuss some of the techniques we used for this validation.

A. Attacking the OpenSSL API

The vulnerability in the OpenSSL API (Section IV-B) has already been disclosed by both Xiao et al. [64] and Zhang et al. [69]. Our attack is similar to the attack of Zhang et al. [69], but achieves a significantly lower error rate, resulting in a lower number of required oracle invocations. Combined with our improved error handling (Section VI-B) we achieve a reduction by a factor of 6 in the number of oracle queries we require.

Our test machine uses a 4 core Intel Core i7-7500 processor, with a 4 MiB cache and 16 GiB memory, running Ubuntu 18.04.1. We use the Flush+Reload attack [67], as implemented in the Mastik toolkit [66].

To reduce the likelihood of errors, we monitor both the call-site to RSAerr (Line 25 of Listing 2) and the code of the function RSAerr. Monitoring each of these locations may generate false positives, i.e. indicate access when the plaintext is PKCS #1 v1.5 conforming. The former results in false positives because the call to RSAerr shares the cache line with the surrounding code, that is always invoked. The latter results in false positives when unrelated code logs an error. By only predicting a non-conforming plaintext if *both* locations are accessed within a short interval, we reduce the likelihood of false positives.

We note that this technique is very different to the approach of Genkin et al. [30] of monitoring two memory locations to reduce false negative errors due to a race between the victim and the attacker [6]. Unlike us, they assume access if *any* of the monitored locations is accessed.

Overall, our technique achieves a false positive rate of 4.3% and false negative rate of 1.1%.

B. Attacking the OpenSSL Data Conversion

We now turn our attention to the code OpenSSL uses for its own implementation of the TLS protocol. As discussed in Section IV-A, OpenSSL leaks a Manger oracle through the length argument in the call to memset in Line 4 of Listing 1. We now show how we detect that the length passed to memset is zero.

We implement a proof-of-concept attack on an Intel NUC computer, featuring an Intel Core i7-6770HQ CPU, with 32 GiB memory, running Centos 7.4.1708. The GNU C library provides multiple implementations for memset, each opimized for a different processor feature. During initialization, the library chooses the best implementation for the computer, and stores it in a function pointer. In run time, the program invokes the best implementation of memset by dereferencing the function pointer. On our system, the selected function is __memset_sse2. We show part of the (disassembled) code of this function in Listing 4.

```
1 <+209>: test   $0x1,%dl
2 <+212>: je     0x40e918 <__memset_sse2+216>
3 <+214>: mov    %cl,(%rdi)
4 <+216>: test   $0x2,%dl
5 <+219>: je     0x40e87a <__memset_sse2+58>
6 <+225>: mov    %cx,-0x2(%rax,%rdx,1)
7 <+230>: retq
```

Listing 4. A snippet of __memset_sse2

The presented code is only executed if the length argument for memset is less than 4. Line 1 of the code first tests the least significant bit of the length. If it is clear, i.e. if the length is 0 or 2, Line 2 branches over Line 3. In Line 4, the code tests if the second bit of the length, branching in Line 5 if the length is less than 2. Thus, if both branches at Lines 2 and 4 are taken, the length argument is 0.

Branch Prediction Attack. Our attack follows previous works in creating *shadow* branches, at addresses that match the least significant bits of monitored branches [25, 43]. Because the branch predictor ignores the high bits of the address, the outcome of the victim branch affects the prediction for the matching shadow branch. That is, when a monitored branch is taken, the BTB predicts that both the monitored branch and it shadow will branch to the same offset as the monitored branch.

Prior works either measure the time to execute the shadow branch [25] or check the performance counters [43] to detect mispredictions of the shadow branch, and from these infer the outcome of previous executions of the monitored branch. However, performance counters are not always available to user processes, and measurements of execution time of branches are noisy. Instead, we combine the branch prediction attack with FLUSH+RELOAD [67] to achieve high accuracy detection of mispredictions.

Specifically, for each monitored branch we create two shadows, the *trainer* and the *spy* branches. Each of these branches to a different offset, such that the offsets of the monitored branch and of the shadow branches each falls in a different cache line. The attack then follows a sequence of steps:

- Invoke the trainer shadow to train the branch predictor to predict the trainer offset for all three branches.
- Flush the cache line at the trainer offset from the spy branch from the cache.
- Execute the victim. If the victim branch is taken, it will update the BTB state to predict the victim offset for all three branches
- Invoke the spy branch. Because the branch predictor predicts either the victim or the trainer offset, the spy branch mispredicts. In the case that the victim branch has not been taken, the mispredicted branch will attempt to branch to the trainer offset from the branch, bringing the previously flushed line back into the cache.
- Measure the time to access the previously flushed line. If the victim branch has been taken, this line will not be cached, and access will be slow. If, however, the victim branch did not execute or was not taken, the line will be in the cache due to the misprediction in the previous step, and accesss will be fast.

We implemented this attack and we can predict the outcome of each of the monitored branches with a probability higher than 98%. We cannot, however, monitor both branches concurrently. Consequently, for the manger attack, we will have to send each message twice. Once for monitoring the outcome of the branch in Line 2 and the other for the branch in Line 5.

VI. MAN IN THE MIDDLE ATTACKS

The main scenario we investigate is an attacker mounting a padding oracle attack to recover the premaster secret used in TLS connections. When the TLS connection uses RSA key exchange, the attack is particularly powerful, because the attacker does not need to be able to decipher the encrypted communication on the fly. Instead, the attacker can record the encrypted communication, and perform the padding oracle attack at a later time to decipher the communication. The main limitation of the attack is that RSA key exchange is not a recommended or a popular choice for TLS connections. TLS 1.3 does not support RSA key exchange, and for TLS 1.2 RSA key exchange is not the recommended option. Consequently, at the time of writing, only about 6% of all TLS connections use RSA key exchange [1, 50]. To overcome this limitation, we mount an online downgrade attack to force RSA key exchange in a connection. We now discuss this downgrade attack.

Man-in-the-Middle Downgrade Attacks. Jager et al. [40] observe that padding oracle attacks can be combined with man-in-the-middle attacks to force protocol and cipher suite downgrade in the communication between a client and a server. In a nutshell, the attacker uses a man-in-the-middle (MitM) attack to change the protocol and cipher suite negotiation messages between the parties to only advertise TLS 1.2 RSA key exchange. It then uses the padding oracle attack to recover the premaster secret and uses it to complete the handshake between the parties.

As Jager et al. [40] observe, downgrade attacks can be applicable even when the client uses protocols, such as TLS 1.3 and QUIC, that do not support RSA key exchange. If the server uses the same certificate for both RSA key exchange and RSA signing, an attacker can leverage the RSA key exchange to fake server signatures, which are supported in the newer protocols. We note that at the time of writing, Amazon AWS servers use the same RSA certificate for signing and for key exchange.

To avoid losing clients, servers continue to support older protocols, and are likely to continue doing so for the foreseeable future. Consequently, padding oracle attacks are likely to remain a threat to *almost all* TLS connections.

Mounting Online Padding Oracle Attacks. To mount such an online padding oracle MitM attack, the attacker has to recover the premaster secret before the browser times out the TLS connection. The approach to achieve this depends on the browser that the victim uses. Adrian et al. [4] show a technique that forces Firefox to keep a TLS handshake alive indefinitely, thus allowing us to perform even very long attacks. Using a BEAST style technique [23] we can perform this attack in the background, without the user noticing any long delays. Other browsers, however, are not as easy to attack—they enforce stricter time limits on TLS handshakes. For example, Google Chrome and Microsoft Edge time TLS handshakes out after only about 30 seconds. Thus, when mounting padding oracle MitM attacks against these browsers, the attacker has to be extremely efficient and finish the attack before the timeout. At the same time, typical padding oracle attacks require a large number (several thousands) of TLS handshakes, which would take much longer to execute than the typical browser timeout.

Analysis and improvement of Padding Oracle Attacks. In this section, we analyze the complexity of padding oracle attacks for an online MitM scenario. Our contributions are as follows. First, we present a novel analysis of the query complexity required from a padding oracle attack (Section VI-A). Next, we handle the case of imperfect and noisy oracles (Section VI-B). Finally, in Section VII we address the question of parallelizing padding oracle attacks across any available number of servers, demonstrating a new application of lattice techniques to padding oracle attacks.

A. *Reducing the Query Complexity of Padding Oracles*

A key observation of our attack is that in many scenarios the attacker only has to succeed once. Consequently, instead of focusing on minimizing the expected number of oracle queries to break the secret, we aim at finding a strategy that would achieve some low probability of success. In this section we explore this strategy.

Overview. We start with a motivating scenario. We then look at some illustrative examples analyzing the number of oracle queries required to find a conforming ciphertext with a given probability. Finally, we perform simulations of padding oracle attacks and empirically determine the number of queries required for recovering the plaintext with several oracles.

Motivating Scenario. Assume we would like to break the security of a specific account in some popular online service (e.g., Gmail). As the connection is usually done via https (which uses TLS), one attack vector is to attempt to break the user's existing TLS connection with the online service. Using padding oracles to mount a MitM downgrade attack on a specific connection might be difficult given the 30 seconds browser-enforced timeout for completing the TLS handshake. In our new analysis, we assume that we perform a BEAST style attack [23]. In this scenario a malicious web site controlled by the attacker, causes the user's browser to repeatedly try to connect to the TLS server in the background without the user's knowledge. This attack only requires that the browser supports JavaScript, and does not need any special privileges. (In particular, the attacker does not have to compromise the normal operation of the target machine in any way.) A successful MitM attack on even a single TLS handshake will allow the attacker to decrypt the user's login token, thereby allowing a malicious login to the server.

Low Success Probability is Sufficient. The expected number of queries required for completing a Bleichenbacher style attack is large. With a short browser timeout, the likelihood of completing the attack before the timeout is very low. However, we can use the long tail distribution of the number of queries to devise a strategy that provides a high likelihood of success. Specifically, the probability of the attack completing before a browser timeout is low, but it is not negligible. Our strategy is to use the BEAST attack to amplify this low probability, by repeatedly attempting the connection to the server until a padding oracle attack succeeds or the connection times out. Because the success probability is not negligible, repeating the attack enough times is likely to eventually succeed.

Finding a Conforming Ciphertext. The complexity of the Bleichenbacher padding oracle attack is dominated by the number of oracle queries required for finding the first few conforming ciphertexts. That is, a ciphertexts c such that the plaintexts corresponding to them match the format detected by the oracle. Thus, we begin by analyzing the number of queries required for finding a conforming ciphertext under several oracle types.

Analyzing OpenSSL API FFTT Oracle. We first look at the FFTT padding oracle present in the OpenSSL decryption

TABLE II
NUMBER OF ORACLE QUERIES REQUIRED FOR 2048-BIT RSA MODULUS.

Oracle	Signature Forging with Success Probability **0.001**	**0.01**	**0.1**	**0.5**	Decryption with Success Probability **0.001**	**0.01**	**0.1**	**0.5**
FFTT Oracle (OpenSSL API)	16381	19899	40945	122377	14700	15147	16764	50766
FFFT Oracle (MbedTLS)	139426	192633	533840	1292250	116699	123359	237702	870664
Manger Oracle	≈ 2048	≈ 2048	≈ 2048	≈ 2048	≈ 2048	≈ 2048	≈ 2048	≈ 2048
FFTT Oracle With Errors	29989	33944	57130	147406	28170	28683	30494	70990
Manger Oracle With Errors	≈ 6144	≈ 6144	≈ 6144	≈ 6144	≈ 6144	≈ 6144	≈ 6144	≈ 6144

API (Section IV-B). Let (d, N) be an RSA private key. For a ciphertext c to be conforming, the following must hold:

1) First, the two most significant bytes of $c^d \bmod N$ (the RSA plaintext corresponding to c) must be 0x0002. For a random c, this happens with probability of 2^{-16}.
2) Next, the following eight padding bytes of the plaintext corresponding to c must be non-zero. For a random c, this event happens with probability of $(255/256)^8$.
3) The plaintext corresponding to c contains a zero byte. For a 2048-bit RSA modulus N, we have 246 remaining bytes. Thus, for a randomly selected c, this event holds with probability of $1-(255/256)^{246}$ (or $1-(255/256)^{502}$ for a 4096-bit modulus).

We obtain that for any 2048-bit RSA private key $\Pr_c[\mathsf{FFTT}(c) = 1]$, the probability that a random ciphertext c is conforming, is given by

$$\begin{aligned}\Pr_c[\mathsf{FFTT}(c) =& 1] \\ &= 2^{-16} \cdot \left(\frac{255}{256}\right)^8 \cdot \left(1-\left(\frac{255}{256}\right)^{246}\right) \\ &\approx 9.14 \cdot 10^{-6}.\end{aligned}$$

Similarly, for any 4096-bit RSA private key, we obtain that $\Pr_c[\mathsf{FFTT}(c) = 1] \approx 1.27 \cdot 10^{-5}$. Next, the expected number of oracle queries required to obtain a conforming ciphertext is $1/\Pr_c[\mathsf{FFTT}(c) = 1]$ which results in about 110000 queries for 2048-bit key and about 80000 queries for 4096-bit key.

Oracle queries are Bernoulli trials. Hence, the number of trials until success has a geometric distribution, and we can use the inverse of the cumulative distribution function (CDF) of the geometric distribution to find out the expected number of oracle queries for any desired probability of success. Using the inverse CDF, we find that only 110 queries are required to achieve a probability of $1/1000$ of finding a conforming ciphertext for a 2048-RSA key. For 4096-bit keys, only 80 queries are required. Hence, the complexity of the attack *decreases* as the key sizes grow.

Analyzing MbedTLS FFFT Oracle. We now proceed to analyze the FFFT padding oracle present in MbedTLS implementation of the PKCS #1 v1.5 verification code (Appendix A-C). Let (d, N) be an RSA private key. For a plaintext c to be conforming to an FFFT oracle, the following must hold.

1) The first two conditions of the FFTT oracle present in the OpenSSL decryption API hold. For a random ciphertext the probability that both conditions hold is $2^{-16} \cdot (255/256)^8$.
2) The size of the unpadded plaintext corresponding to c is between 0 and 48 bytes. For a 2048-bit RSA key, we have 256 bytes of padded plaintext. The first 10 bytes are checked in the first condition, leaving 246 bytes for the padding and the plaintext itself. As the padding string must consist of some number of non zero bytes and terminate with a zero byte, we obtain that for a random 2048-bit ciphertext c, this event holds with probability of $(255/256)^{246-48} \cdot (1-(255/256)^{48})$.
Similarly, for 4096-bit RSA key (containing 512 bytes), this event holds for a random ciphertext with probability of $(255/256)^{502-48} \cdot (1-(255/256)^{48})$.

Thus, for any 2048-bit RSA private key it holds that

$$\begin{aligned}&\Pr_c[\mathsf{FFFT}(c) = 1] \\ &\quad = 2^{-16} \left(\frac{255}{256}\right)^8 \left(\frac{255}{256}\right)^{198} \left(1-\left(\frac{255}{256}\right)^{48}\right) \\ &\quad \approx 1.16 \cdot 10^{-6}.\end{aligned}$$

For 4096-bit RSA private keys, we obtain that $\Pr_c[\mathsf{FFFT}(c) = 1] \approx 4.28 \cdot 10^{-7}$. Using the same formulas as above, we find that the expected number of trials to achieve a probability of $1/1000$ of finding a conforming ciphertext is 860 for 2048-bit keys and 2300 for 4096-bit keys.

Full Attack Simulation. While the query complexity of the entire padding oracle attack highly depends on the probability p that the padding oracle outputs 1 on a random ciphertext, for Bleichenbacher-type oracles the exact relation between p and the attacks' query complexity is rather difficult to analyze. Instead, we ran 500000 simulations of the full attack using the FFTT, FFFT and Manger type oracles, for a 2048-bit RSA modulus. The results of our simulation are presented in Table II, for both decryption and signature forging attacks. For each oracle type and attack type, we give the required number of oracle queries needed to complete the attack with the different success probabilities. As the table demonstrates, the number of queries required for achieving a success probability of $1/1000$ is an order of magnitude lower than that required for a probability of 50%. Yet, while the success probability of each individual attack attempt is low (1/1000), the attacker can always use BEAST-style techniques, repeatedly issuing

TLS connections to the target website. As soon as a single connection attempt is broken, the attacker can decrypt the user's login token, compromising the account. Finally, we note that because each attack attempt has a low oracle query complexity, it is possible to complete the attempt below the 30 seconds timeout enforced by Chrome and Edge.

Analyzing the Manger Oracle. The Manger attack complexity is much simpler, having the number of queries required be approximately the length of the RSA modulus in bits with very low variance (i.e., a little over 2048 queries for 2048-bit keys and 4096 queries for 4096-bit keys [46]).

B. Handling Oracle Errors

So far, the analysis assumed a "perfect" oracle that always produces the correct answer. However, oracles obtained via microarchitectural attacks are rarely perfect, and often have errors in the answers. Oracle errors can be *false positives* (FP), where the oracle considers a non-conforming ciphertext to be conforming, or *false negatives* (FN), where the oracle returns non-conforming for a conforming ciphertext. In this section we present strategies for error recovery. See Table II for a summary of the results.

Handling Errors in Manger Type Attack. As outlined in Section II-D the Manger attack is sensitive to errors, and any oracle query error would result in the attack failing to break the target TLS connection. Thus, to obtain an error-free result we propose to repeat each oracle query several times, taking a majority vote in the result. We now proceed to analyze the exact number of repetitions required by this approach.

Assume we want a padding oracle attack to succeed with a low probability of $p = 0.001$. For a 2048-bit RSA modulus, we will require about 2048 queries to break the target connection. This means that we require $(1-\Pr[\mathsf{error}])^{2048} > 0.001$ which yields $\Pr[\mathsf{error}] < 1 - \sqrt[2048]{0.001} \approx 0.00337$.

From the experimental results outlined in Section V-B, we have that our side-channel-based Manger oracle has an error rate of 0.02 for both false positive and false negative errors. With each oracle request having a probability of 0.02 of being erroneous, the error is higher than the 0.00337 we require for limiting the failure rate to 0.001. Assuming we make r oracle requests, for the majority to be incorrect, we need to have $\Pr[\mathsf{error}] < \sum_{i=r/2+1}^{r}(0.02)^i \approx (0.02)^{r/2+1} < 0.00337$, which yields $r = 3$. Thus, repeating each oracle request three times ensures that the overall error rate is small enough. Consequently, for the Manger oracle, we need a total of approximately $3 \cdot 2048 = 6144$ oracle requests.

Handling Errors in Bleichenbacher-type Oracles. Bleichenbacher-type oracles repeatedly test ciphertexts until finding one whose plaintext is PKCS #1 v1.5 conforming. Consequently, false negative errors are not fatal for the attack. When a false negative error is encountered, the attack continues until another conforming ciphertext is found. Conversely, when the attack relies on a false positive, it will fail.

To better understand the total query complexity required for a side-channel based Bleichenbacher-type oracle, we simulated the end-to-end attack using the false negative and false positive rates obtained in Section V-A (i.e., we set $\Pr[\mathsf{FP}] = 0.043$ and $\Pr[\mathsf{FN}] = 0.011$). Because the attack can tolerate false negative errors, we ignore the possibility of such errors and accept a non-conforming result as correct. However, when the oracle reports that a ciphertext is conforming, we need to be more careful. We issue a total of six queries with the same ciphertext, and require five or more of these queries to give a positive answer for us to accept the ciphertext as conforming. We note that this amount of repetitions was empirically chosen to minimize the attacks' total query complexity. Our simulation results (Table II) show that the presence of errors at most doubles the number of queries required for the attack.

VII. Parallelizing Padding Oracle Attacks

Large service providers often share the load of their web sites over multiple servers. To maintain the illusion of a single web site, all these servers share the same RSA key. In this section we exploit these servers to implement a parallel attack that further reduces the time to break the TLS secret.

Previous Works. Klíma et al. [41] are the first to suggest the possibility of parallelizing the Bleichenbacher attack. Their work suggests a trivial speedup of Phase 2 of the attack (Section II-C), in the case that the attacker finds multiple possible ranges. Böck et al. [12] also mention the possibility of using multiple servers to parallelize the attack. However, they do not present a concrete method of doing this. Nguyen [51] shows how to replace the search in the Bleichenbacher attack with a lattice technique that the author claims to be more intuitive than the search. However, this lattice technique is not used for paralleilzation and is much less efficient than the technique we present in this work.

Limitations of Trivial Parallelization. A trivial method for parallelizing the Bleichenbacher attack is to concurrently send multiple queries with different values for s_i in each phase of the attack. When one of the ciphertexts is found to be conforming, the attacker can reduce the range and proceed with the next step. Another approach is to parallelize the multiple identical queries for error correction in Bleichenbacher and Manger attacks mentioned in Section VI-B. The main drawback of these approaches is that both the Bleichenbacher and the Manger attacks require at least $\log_2 N$ sequential queries.

Our Approach. In this paper we present a new approach that reduces the number of sequential queries we need to perform. In a nutshell, we perform multiple padding oracle attacks in parallel, each starting from a different initial blinding value. We do some range reduction for each of the parallel attacks, but stop short of completing any of them. We then use a lattice technique to combine the information we learn in each of the attacks and recover the key.

A. Parallelization of the Manger Attack.

Recall the Manger attack. After completing Phase 1 of the attack, at each step, we know that $m \cdot s \bmod N$ is inside the interval $[a, b]$, where m is the unknown plaintext, s is the known blinding value, N is the RSA modulus, and the attack's goal is to decrease the size of the interval $[a, b]$. During each adaptive step of the attack, the size of the interval is reduced. When $|[a, b]| = 1$, we know that $a = m \cdot s \mod N$, and can recover the original plaintext by calculating $m = a \cdot s^{-1} \mod N$. If we can approximately halve the size of the interval in each step, we can complete the attack with $\approx \log_2 N$ adaptive queries.

We now look at the scenario where we run k attacks in parallel, but only have time for i adaptive steps for each attack. Thus, for attack after i queries the interval of attack j is $[a_j^i, b_j^i]$. We note that this reduces the search space for m to the interval $[a_j^i, b_j^i]$, which, abusing information theory terminology, represents learning $I_j^i = \log_2 N - \log_2 (b_j^i - a_j^i)$ bits of information on m. If after i adaptive queries $\sum_{j=1}^{k} I_j^i > \log_2 N$ we can recover the value of m.

At this stage we have a set of equations of the form

$$r_j^i = m \cdot s_j - a_j^i \mod N < 2^{\log_2 (b_j^i - a_j^i)} \tag{1}$$

We note that this set of equations is very similar to the hidden number problem [13], and similar to past solutions to the hidden number problem [9, 13, 15, 29, 34, 52], we use a lattice technique to solve our equations. Specifically, we form the lattice M^i

$$M^i = \begin{bmatrix} s_1 & s_2 & s_3 & \dots & s_k & 0 \\ N & 0 & 0 & \dots & 0 & 0 \\ 0 & N & 0 & \dots & 0 & 0 \\ 0 & 0 & N & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & N & 0 \\ a_1^i & a_2^i & a_3^i & \dots & a_k^i & N \cdot (k-1)/k \end{bmatrix}$$

We note that, from Equation 1, the vector

$$R^i = (r_1^i, r_2^i, \dots, r_k^i, -N \cdot (k1)/k)$$

is in the lattice, and assuming $r_j^i < N \cdot (k-1)/k$ we get that R^i is a short vector in the lattice M^i. We can now use LLL [44] to find a reduced basis for the lattice, and with high probability R^i is the second vector in the reduced basis.[1] We can now find m by calculating $m = (r_1^i + a_1^i) \cdot s_1^{-1} \mod N$

Analyzing the Parallel Attack. We would like to analyze the trade-off between the number of adaptive queries and the number of parallel oracles. In the Manger attack the blinding phase requires on average 128 parallel queries, and gives us 8 bits of information on the plaintext. The next two phases (called steps 1 and 2 in the original paper) are harder to analyze, but experiments show that they usually require 40–100 adaptive queries and give us 8–12 extra bits of information. After that, each adaptive query gives us approximately one bit of information. For an RSA modulus of 2048 bits the original Manger attack without blinding requires ≈ 2100 adaptive queries and just one oracle (which requires negligible computation). On the other extreme we can try a fully parallelized attack using only the blinding phase. This will require approximately $128 \cdot 256 = 32768$ parallel queries, that will result in 256 equations giving us 8 bits each. Recovering the plaintext will require us to reduce a relatively large lattice of dimension ≈ 256, which requires a considerable amount of computation. A more efficient trade-off is to run a moderate number of partial adaptive attacks in parallel.

Parallel Manger Attack Simulation. We ran a simulation to test the feasibility of performing a MitM on a TLS connection and a 2048 bits RSA with multiple parallel partial Manger attacks. We assume that we have 30 seconds before the TLS connection times out and that each TLS handshakes takes about 0.05 seconds (which is the actual time measured on a Core i7-7500U CPU @ 2.70GHz). We allow each of the parallel attacks to have 560 adaptive oracle queries, leaving two seconds for the lattice reduction and for finalizing the handshake. We simulate a parallel attack using five servers (The minimal number of servers required to fit at least 2048 queries in 30 seconds is four, but due to overheads we require at least five servers).

We start by running the blinding phase in parallel until we get five valid blinding values. We then use our remaining queries to continue the five attacks in parallel. As before, we perform 500000 simulations of the attack, each simulation running five attacks in parallel. With probability 0.001 we get at least 438 bits of information from each of the five attacks, or a total of more than 2190 bits. This is more than the required number of bits to recover the plaintext. We successfully implemented and tested a proof of concept of the lattice reduction and were able to perform the plaintext recovery using the LLL algorithm in Sage [62] with a negligible run time of less than 0.01 seconds (running on a Intel Corei7-4790 CPU @ 3.6GHz).

B. Parallelization of the Bleichenbacher Attack.

The Bleichenbacher attack can also be parallelized in the same way as we have shown for the Manger attack. We assume k parallel attacks. For each attack we start with a different blinding value, such that for attack number j we know that $2B < s_j^0 < 3B - 1$. After i adaptive queries we learn that $a_i < s_j^i < bi$.[2] Using this information we can recover the plaintext as we have done for the Manger attack.

Analyzing the Parallel Attack. As the Bleichenbacher attack has a much higher query complexity than the Manger attack, we will require a large number of servers to attack. However, if we have k servers, running k attacks in parallel is very inefficient, due to the high cost of the first blinding phase. Instead we use the fact that each adaptive step of the attack includes many queries that can be done in parallel. We start by using all servers for multiple parallel queries until we find a small number of blinded values (e.g. 5 as in the

[1]The first row is zero as M^i does not have full rank.

[2]With low probability we might have more than one possible domain, and in that case we can take the domain from one of the previous queries

Manger attack). We then split the k servers evenly between the blinded values to create multiple attacks. For each blinded value, multiple servers will be used to run the parallel queries required for each adaptive step.

VIII. Discussion and Conclusions

In this work we have answered negatively the question "Are modern implementations of PKCS #1 v1.5 secure against padding oracle attacks?". The systemic re-discovery of Bleichenbacher's attack on RSA PKCS #1 v1.5 encryption over the last 20 years has shown that the mitigations requirements are unrealistic towards developers. Among the nine popular implementations we surveyed, only two successfuly survived our analysis. The insistence that protocols preserve this broken padding standard still have consequences today, reaching even the latest version of TLS 1.3 released in August 2018.

A. Recommendations for Mitigation

As we have seen, implementing a completely secure and side channel free PKCS #1 v1.5-based RSA key exchange for TLS is not easy. We propose several approaches to reduce implementations' vulnerability to our attacks.

Deprecation of RSA Key Exchange. The safest countermeasure is to deprecated the RSA key exchange and switch to (Elliptic Curve) Diffie-Hellman key exchanges. This might be hard due to backward compatibility issues.

Certificate Separation. If RSA key exchange support is required, it should be done with a dedicated public key that does not allow signing. Similarly, to prevent downgrade attacks, support for multiple TLS versions should not reuse keys across versions. If multiple TLS servers are used, each server should use a different public key if possible to prevent parallelized attacks.

Constant-Time Code and Safe API. The decryption code should be constant-time, with no branching or memory accesses depending on the plaintext (e.g., as achieved in the BoringSSL and BearSSL code). A common problem across multiple implementations is that the expected plaintext size is not provided to the decryption function. Passing the expected plaintext size is safer because it facilitates constant-time implementations. Furthermore, we observe that side-channel leakage from code that uses the expected plaintext size results in weaker padding oracles that greatly increase the amount of time required for an attack.

Using Large RSA Keys. The minimal threshold for decryption using Bleichenbacher and Manger type attacks is $\approx \log N$ consecutive calls to the oracle. Larger keys (at least 2048 bits) take longer to attack and might make MitM attack less practical.

Handshake Timeouts. It is harder to do a MitM attack when the TLS handshake timeout is very short. Clients should use short TLS timeouts, and make sure they are resilient to any attack that can lengthen the timeout (such as the TLS warning alerts attack against Firefox [4]).

Speed Limitation. As RSA key exchanges are only a small fraction of today's TLS traffic [1, 50], limiting the speed of allowed RSA decryptions makes MitM attacks less practical.

Dedicated Hardware for Sensitive Cryptographic Code. Side channel attacks are extremely difficult to defend against. Critical and sensitive operations such as private key decryption should not be run on a hardware shared with other code if possible.

B. Future Work

Timeouts in TLS Client. As we have seen in this work and previous works [4], the possibility of doing some MitM attacks depends strongly on the amount of time the attacker has before the client gives up on the handshake. Clients that have long handshake timeouts (e.g. curl and git) or are vulnerable to a "timeout extension" attack (e.g. Firefox) put their users at risk. A systematic review of different client's timeouts configuration and their resilience to "timeout extension" attacks is required.

Keyless TLS Implementations. Many (often private) TLS implementations segregate private key operations from the protocol implementation by having a keyless server responding to signature and decryption requests from keyless clients. PKCS #1 v1.5 verification is not always done from the keyless server and decrypted ciphertexts of variable-length passed to the keyless clients can be passively observed from a privileged network position. A review of available implementations and standards (such as LURK [48]) is needed.

Acknowledgments

The authors would like to thank Orr Dunkelman for his contribution, in particular for his insights about the parallelization of the attack using LLL.

This research was supported in part by Intel Corporation.

References

[1] "The ICSI Notary," http://notary.icsi.berkeley.edu/#connection-cipher-details.

[2] O. Acıiçmez, "Yet another microarchitectural attack: Exploiting I-Cache," in *CSAW*, 2007.

[3] O. Acıiçmez, S. Gueron, and J. Seifert, "New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures," in *IMA Int. Conf.*, 2007.

[4] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Z. Béguelin, and P. Zimmermann, "Imperfect forward secrecy: How Diffie-Hellman fails in practice," in *CCS*, 2015.

[5] N. J. AlFardan and K. G. Paterson, "Lucky thirteen: Breaking the TLS and DTLS record protocols," in *IEEE SP*, 2013, pp. 526–540.

[6] T. Allan, B. B. Brumley, K. E. Falkner, J. van de Pol, and Y. Yarom, "Amplifying side channels through performance degradation," in *ACSAC*, 2016.

[7] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J. Tsay, "Efficient padding oracle attacks on cryptographic hardware," in *CRYPTO*, 2012.
[8] M. Ben-Or, B. Chor, and A. Shamir, "On the cryptographic security of single RSA bits," in *STOC*, 1983.
[9] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom, ""Ooh aah... just a little bit" : A small amount of side channel can go a long way," in *CHES*, 2014.
[10] D. J. Bernstein, "Cache-timing attacks on AES," 2005.
[11] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1," in *CRYPTO*, 1998.
[12] H. Böck, J. Somorovsky, and C. Young, "Return of Bleichenbacher's oracle threat (ROBOT)," in *USENIX Security*, 2018.
[13] D. Boneh and R. Venkatesan, "Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes," in *CRYPTO*, 1996.
[14] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *WOOT*, 2017.
[15] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *ESORICS*, 2011.
[16] J. V. Bulck, F. Piessens, and R. Strackx, "SGX-Step: A practical attack framework for precise enclave execution control," in *SysTEX@SOSP*, 2017.
[17] ——, "Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic," in *CCS*, 2018.
[18] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohney, M. Green, N. Heninger, R.-P. Weinmann, E. Rescorla, and H. Shacham, "A systematic analysis of the Juniper Dual EC incident," in *CCS*, 2016.
[19] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," RFC 2246, Jan. 1999.
[20] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," RFC 4346, Apr. 2006.
[21] ——, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, Aug. 2008.
[22] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. M. Tullsen, "Prime+Abort: A timer-free high-precision L3 cache attack using intel TSX," in *USENIX Security*, 2017.
[23] T. Duong and J. Rizzo, "Here come the $\oplus$ ninjas," 2011.
[24] D. Evtyushkin, D. Ponomarev, and N. B. Abu-Ghazaleh, "Understanding and mitigating covert channels through branch predictors," *TACO*, vol. 13, no. 1, 2016.
[25] D. Evtyushkin, D. V. Ponomarev, and N. B. Abu-Ghazaleh, "Jump over ASLR: attacking branch predictors to bypass ASLR," in *MICRO*, 2016.
[26] D. Evtyushkin, R. Riley, N. B. Abu-Ghazaleh, and D. Ponomarev, "BranchScope: A new side-channel attack on directional branch predictor," in *ASPLOS*, 2018.
[27] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *J. Cryptographic Engineering*, vol. 8, no. 1, 2018.
[28] Q. Ge, Y. Yarom, and G. Heiser, "No security without time protection: We need a new hardware-software contract," in *APSys*, Aug. 2018.
[29] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "ECDSA key extraction from mobile devices via nonintrusive physical side channels," in *CCS*, 2016.
[30] D. Genkin, L. Valenta, and Y. Yarom, "May the fourth be with you: A microarchitectural side channel attack on several real-world applications of Curve25519," in *CCS*, 2017.
[31] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," in *ACNS*, 2018.
[32] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security*, 2015.
[33] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A fast and stealthy cache attack," in *DIMVA*, 2016.
[34] N. Howgrave-Graham and N. P. Smart, "Lattice attacks on digital signature schemes," *Des. Codes Cryptography*, vol. 23, no. 3, 2001.
[35] M. S. Inci, B. Gülmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache attacks enable bulk key recovery on the cloud," in *CHES*, 2016.
[36] Intel, "Speculative execution side channel mitigations," May 2018.
[37] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES," in *IEEE SP*, 2015, pp. 591–604.
[38] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Lucky 13 strikes back," in *ASIA CCS*, 2015.
[39] T. Jager, S. Schinzel, and J. Somorovsky, "Bleichenbacher's attack strikes again: Breaking PKCS#1 v1.5 in XML encryption," in *ESORICS*, 2012.
[40] T. Jager, J. Schwenk, and J. Somorovsky, "On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption," in *CCS*, 2015.
[41] V. Klíma, O. Pokorný, and T. Rosa, "Attacking RSA-based sessions in SSL/TLS," in *CHES*, 2003.
[42] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Haburg, M. Lipp, S. Mangard, T. Prescher, M. Schwartz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE SP*, 2019.
[43] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *USENIX Security*, 2017.
[44] A. K. Lenstra, H. W. Lenstra, and L. Lovász, "Factoring polynomials with rational coefficients," *Mathematische Annalen*, vol. 261, no. 4, 1982.
[45] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE SP*, 2015.
[46] J. Manger, "A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized

in PKCS #1 v2.0," in *CRYPTO*, 2001.

[47] C. Meyer, J. Somorovsky, E. Weiss, J. Schwenk, S. Schinzel, and E. Tews, "Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks," in *USENIX Security*, 2014.

[48] D. Migault and I. Boureanu, "LURK extension version 1 for (D)TLS 1.2 authentication," IETF, Internet-Draft draft-mglt-lurk-tls12-01, 2018.

[49] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX amplifies the power of cache attacks," in *CHES*, 2017.

[50] Mozilla, "SSL handshake key exchange algorithm for full handshake," https://mzl.la/2BQjcMO.

[51] P. Q. Nguyen, *Public-Key Cryptanalysis*, ser. Contemporary Mathematics. AMS–RSME, 2009, vol. 477.

[52] P. Q. Nguyen and I. E. Shparlinski, "The insecurity of the Digital Signature Algorithm with partially known nonces," *J. Cryptology*, vol. 15, no. 3, 2002.

[53] OpenSSL, "RSA_public_encrypt," https://www.openssl.org/docs/man1.0.2/crypto/RSA_private_decrypt.html.

[54] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in JavaScript and their implications," in *CCS*, 2015.

[55] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *CT-RSA*, 2006.

[56] C. Percival, "Cache missing for fun and profit," in *Proceedings of BSDCan*, 2005.

[57] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *CCS*, 2009.

[58] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, 1978.

[59] E. Ronen, K. G. Paterson, and A. Shamir, "Pseudo constant time implementations of TLS are only pseudo secure," in *CCS*, 2018.

[60] *PKCS #1 v2.2: RSA Cryptography Standard*, RSA Laboratories, 2012.

[61] S. Schmidt, "Introducing s2n, a new open source tls implementation," 2015.

[62] The Sage Developers, *SageMath, the Sage Mathematics Software System (Version 8.3)*, www.sagemath.org, 2018.

[63] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, "Cryptanalysis of DES implemented on computers with cache," in *CHES*, 2003.

[64] Y. Xiao, M. Li, S. Chen, and Y. Zhang, "STACCO: differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves," in *CCS*, 2017.

[65] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn DNN architectures," *CoRR*, vol. abs/1808.04761, 2018.

[66] Y. Yarom, "Mastik: A micro-architectural side-channel toolkit," cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf, 2017.

[67] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014.

[68] X. Zhang, Y. Xiao, and Y. Zhang, "Return-oriented Flush-Reload side channels on ARM and their implications for Android devices," in *CCS*, 2016.

[69] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in PaaS clouds," in *CCS*, 2014.

```
1  int BN2binpad((bn, to)){
2  //bn is big number (storing the RSA plaintext)
3  //to is the output buffer
4  //BN_BYTES is the number of bytes in each bn word
5
6    i = BN_num_bytes(bn);
7    tolen=i
8    // Padding code removed for brevity
9    while (i--) {
10     l = bn[i / BN_BYTES];
11     *(to++) = (unsigned char)
12     ( l>> (8 * (i % BN_BYTES))) & 0xff;
13   }
14   return tolen;
15 }
```

Listing 5. Pseudocode of big number serialization functions

APPENDIX A
VULNERABILITIES DESCRIPTION

A. OpenSSL TLS Implementation

Perhaps aware of the side channel issues in its RSA decryption API, OpenSSL does not use the code described in Section IV-B for its own TLS implementation. Instead, OpenSSL reimplemented the RSA PKCS #1 v1.5 padding verification as part of its TLS protocol code. This constant time implementation does not appear to be vulnerable to a cache-based padding oracle attack. However, OpenSSL's code does contain two side channel vulnerabilities. One vulnerability has been described in Section IV-A and the other is presented here. We note that OpenSSL replaced the vulnerable code in both locations with constant-time implementations independently of our disclosure.

Leaky Data Conversation. As mentioned in Section IV, the big numbers representing the RSA ciphertext and plaintext are typically saved as an array of 32-bit words, while the result of the PKCS #1 v1.5 padding is an array of bytes. To convert the data from one representation to the other, OpenSSL uses a serialization function which takes as input a big number and serializes it into a byte array (where index 0 is the most significant byte). To avoid creating a padding oracle, it is important that the serialization function be written in a constant-time manner, and not leak the length of the RSA plaintext during the serialization process.

The pseudocode of OpenSSL's serialization function is presented in Listing 5. Notice the while loop in Line 9, which performs as many iterations as the number of non-zero bytes of the RSA plaintext, resulting in a Manger-type padding oracle. Traditionally, mounting such precise microarchitectural attacks is difficult, as a single loop iteration takes less time than the channel's temporal resolution. However, recent works [16, 17, 49] have shown that mounting high precision side channel attacks is possible in the case of trusted execution environments (e.g., Intel SGX), often with cycle-accurate resolution.

B. Amazon s2n

S2n is Amazon's implementation of the TLS protocol, used as part of Amazon Web Services. It simplifies the OpenSSL TLS implementation, removing uncommon and deprecated TLS configurations. The implementation of RSA decryption (Listing 6) invokes the OpenSSL RSA_private_decrypt API function to process and remove the PKCS #1 v1.5 padding (Line 6). We have already discussed the weakness due to the use of the OpenSSL function (Section IV-B). We now discuss another vulnerability in the s2n code.

Leaky PKCS #1 v1.5 Verification. In case the decryption and PKCS #1 v1.5 verification succeeds and the output is of the expected length, s2n copies the data to the output array (Line 7). Moreover, the decision of whether to copy and the copy itself is done in constant time to avoid leaking the result of the result of the PKCS #1 v1.5 unpadding.

However, the s2n API relies on the error status returned from OpenSSL to identify padding failures or mis-formatted output. Thus, s2n uses an if macro, which compiles to a conditional branch (see Line 8), which yields an FFFT oracle.

C. MbedTLS

MbedTLS aims at providing a portable, easy to use and to read implementation of the TLS protocol and is designed primerly to be used in low powered embedded devices. We have identified vulnerabilities in both the data conversion and the PKCS #1 v1.5 verification stages of the mbedTLS implementation which we now describe.

Leaky PKCS #1 v1.5 Verification. Listing 7 shows the relevant parts of the mbedTLS PKCS #1 v1.5 verification For brevity we omit the padding format and plaintext length validation, which execute in constant-time. The rest of the code, however uses conditional branches to handle padding validation failures (Lines 7–10) and incorrect plaintext length (Lines 12–15). Thus, despite the constant-time validation, the following form of oracles are still exposed.

- **Potentially Leaky Comparison.** First, the comparison in Line 6 may be implemented using conditional statements, which would leak via branch prediction. This does not happen in our test environment, where the comparison is implemented using a conditional set instruction, which to the best of our knowledge executes in constant-time. However without a guarantee that the compiler will use a constant-time implementation there is a potential for a leak in other environments.
- **Length Dependant Branches.** Both if statements in Lines 7 and 12 can be exploited for a branch prediction attack. The former allows a FFTT Bleichenbacher oracle and the latter allows an FFFT oracle variant. In fact, the oracle is slightly stronger than a standard FFFT oracle because the test is one sided, i.e. it only checks for maximum size instead of checking for exact size.
- **Length Dependant Early Termination.** Finally, due to early termination on bad inputs, the code that copies to the output (Line 18) is only executed if the plaintext is PKCS #1 v1.5 conforming. Thus we can implement an FFFT oracle via an instruction cache attack, monitoring either the call to memcpy or the code of memcpy itself.

Leaky Data Conversion. The last step in the implementation of RSA decryption in mbedTLS is to copy the plaintext to the output. As discussed in Section IV, there is no a-priory method for determining the plaintext's length, and applications can only determine the length after decryption. To determine the length, mbedTLS scans the words that represent the plaintext from the most significant to the least significant, looking for a non-zero word. In a padding oracle attack, this is very likely to be the first word of the plaintext. MbeTLS then scans the bits of the word to find the most significant non-zero bit. This scan, shown in Listing 8, loops over the bits, from the most significant to the least significant (Line 7), checking for a non-zero bit (Line 8). An adversary that can count the number of iterations executed can learn the leading number of zero bits, which can be used for a Manger type oracle. As in Appendix A-A, such attacks are unfesible for unprivileged adversaries, but can be performed by a root adversary attacking a code running in trusted execution environment (e.g., Intel SGX). Finally, we note that the adversary only needs to determine whether the loop body gets executed for implementing an Interval oracle (see Section II-E).

D. Mozilla NSS

Mozilla's Network Security Services (NSS) library is the cryptographic engine often used in applications developed by the Mozilla project. NSS implements countermeasures for padding oracle attacks, however, the TLS code ignores the possibility of leakage through microarchitectural channels Consequently, the TLS implementation exposes padding oracle in each of the three stages of handling PKCS #1 v1.5 padding.

Leaky Data Conversion. Listing 9 shows a leak in the data conversion stage. The code is the start of the function mp_to_fixlen_octets, which converts a large number into a fixed-length byte array. The function first determines the number of bytes required for storing the number (Line 8). Next, it zero-pads the output byte array, so that the final output is exatly length bytes (Lines 10–12). Finally, it converts the large number m from its little-endian representation to a big-endian byte array representation (omitted for brevity).

Unfortunately, mp_to_fixlen_octets does not perform the padding in constant time, thus leaking the number of leading zeros in the RSA decrypted plaintext to an adversary that can count (via the cache side channel) the number of iterations in the loop in Lines 10–12. Furthermore, a branch prediction

```
int s2n_rsa_decrypt(priv, in, out){
  unsigned char intermediate[4096];
  const s2n_rsa_private_key *key = &priv->key.rsa_key;
  S2N_ERROR_IF(s2n_rsa_private_encrypted_size(key) > sizeof(intermediate), S2N_ERR_NOMEM);
  S2N_ERROR_IF(out->size > sizeof(intermediate), S2N_ERR_NOMEM);
  int r = RSA_private_decrypt(in->size, in->data, intermediate, key->rsa, RSA_PKCS1_PADDING);
  GUARD(s2n_constant_time_copy_or_dont(out->data, intermediate, out->size, r != out->size));
  S2N_ERROR_IF(r != out->size, S2N_ERR_SIZE_MISMATCH);
  return 0;
}
```

Listing 6. Pseudocode of Amazon s2n's wrap for OpenSSL's API

```
int mbedtls_rsa_rsaes_pkcs1_v15_decrypt(
ilen, olen, input, output, output_max_len) {
    ...
    //Omited code checks for valid padding and length of decrypted plaintext

  bad |= ( pad_count < 8 );
  if( bad ){
    ret = MBEDTLS_ERR_RSA_INVALID_PADDING;
    goto cleanup;
  }

  if( ilen - ( p - buf ) > output_max_len ){
    ret = MBEDTLS_ERR_RSA_OUTPUT_TOO_LARGE;
    goto cleanup;
  }

  *olen = ilen - (p - buf);
  memcpy( output, p, *olen );
  ret = 0;

  cleanup:
  mbedtls_zeroize( buf, sizeof( buf ) );
  return( ret );
}
```

Listing 7. MbedTLS's unpadding function

```
size_t mbedtls_clz( x ){
// x is the RSA decrypted plaintext
// biL is the number of bits  in limb (typ. 64)
  size_t j;
  mask = 1 << (biL - 1);

  for( j = 0; j < biL; j++ ){
    if( x & mask ) break;
    mask >>= 1;
  }
  return j;
}
```

Listing 8. MbedTLS's bit length checking function function

attack can determine whether the body of the loop executed, allowing a Manger-type oracle.

Leaky PKCS #1 v1.5 Verification. We now describe the leaks from the PKCS #1 v1.5 verification code in NSS (Listing 10). The code performs a textbook verification of the PKCS #1 v1.5 format, e.g. Lines 10 and 11 check the values of the first two bytes in the message.

Unfortunately, the code in Listing 10 terminates early in case of verification failure. Thus, using a branch prediction attack to monitor any of the if statements in the code yields an TTTT-type padding oracle. Moreover, in case that the checks in Lines 10 and 11 are compiled into two differnet branches this can allow for a Manger type Oracle. Furthermore, as in Appendix A-C, monitoring the call to PORT_Memcpy (Line 24) using a cache side channel yields a stronger variant of FTTT-type padding oracle, as it only checks for zero anywhere after the first 2 bytes.

```
mp_to_fixlen_octets(mp, str, length)
{
  // mp is a number encoded in little endian
  // str is an array of  length bytes containing
  // a big endian encoding of mp
  int ix, pos = 0;
  unsigned int bytes;
  bytes = mp_unsigned_octet_size(mp);
  /* place any needed leading zeros */
  for (; length > bytes; --length) {
    *str++ = 0;
  }
  .../* code for convering a little-endian large
      * number mp into a big-endian fixed-length
      * byte array str (omitted for brevity) */
}
```

Listing 9. Data Conversion in NSS

```
1  RSA_DecryptBlock(key, output, outputLen,
2          maxOutputLen, input, inputLen)
3  {
4   ...
5   rv = RSA_PrivateKeyOp(key, buffer, input);
6   if (rv != SECSuccess)
7     goto loser;
8
9   /* XXX(rsleevi): Constant time */
10  if (buffer[0] != RSA_BLOCK_FIRST_OCTET ||
11      buffer[1] != RSA_BlockPublic) {
12    goto loser;
13  }
14  *outputLen = 0;
15  for (i = 2; i < modulusLen; i++) {
16    if (buffer[i] == RSA_BLOCK_AFTER_PAD_OCTET) {
17      *outputLen = modulusLen - i - 1;
18      break;
19    }
20  }
21  if (*outputLen == 0)
22    goto loser;
23  ...
24  PORT_Memcpy(output, buffer + modulusLen - *
       outputLen, *outputLen);
25  return SECSuccess;
26
27 loser:
28  PORT_Free(buffer);
29 failure:
30  return SECFailure;
31 }
```

Listing 10. NSS's PKCS #1 v1.5 Verification function

```
1  wc_RsaFunctionSync(in, inLen, out, outLen, key)
2  {
3    ... // code for perfoming RSA decryption of in
4        // result is stored in temp
5    if (ret == 0) {
6      len = mp_unsigned_bin_size(tmp);
7      while (len < keyLen) {
8        *out++ = 0x00;
9        len++;
10     }
11     ...
12   }
13   ...
14 }
```

Listing 11. WolfSSL's RSA decryption conversion

```
1  void nettle_mpz_to_octets(length, *s, x, sign){
2  // convert x in little endian big number to
3  // a big endian byte attay representation s
4  // of length bytes
5   uint8_t *dst = s + length - 1;
6   size_t size = mpz_size(x);
7   size_t i;
8
9   for (i = 0; i<size; i++) {
10   mp_limb_t limb = mpz_getlimbn(x, i);
11   size_t j;
12   for (j = 0; length && j<sizeof(mp_limb_t); j++){
13    *dst-- = sign ^ (limb & 0xff);
14    limb >>= 8;
15    length--;
16   }
17  }
18  if (length) memset(s, sign, length);
19 }
```

Listing 12. GnuTLS's Data Conversion function

Leaky Padding Oracle Mitigations. Finally, as in OpenSSL (Listing 2), the NSS code responsible for mitigating padding oracle attacks checks the results of the PKCS #1 v1.5 verification procedure using an if statement that translates to a conditional branch. Thus, monitoring this branch as done for Section IV-B results in a FFFF-type padding oracle.

E. WolfSSL

WolfSSL is a TLS library aimed at embedded devices. As in NSS, the WolfSSL code exposes oracles in all stages of PKCS #1 v1.5 handling.

Leaky RSA Decryption Routine. After performing RSA decryption, WolfSSL pads the plaintext to the length of the RSA modulus (Lines 7–10 in Listing 11) using a while loop. The number of iterations this loop performs leaks the number of leading zero bytes, exposing a Manger oracle.

Leaky PKCS #1 v1.5 Verification and Padding Oracle Mitigations. Additionally WolfSSL uses a naive, variable time code for PKCS #1 v1.5 verification, leaking Manger and FFTT-type padding oracles. Moreover, the padding oracle mitigation code leaks FFTT- and FFFF-type padding oracles through the microarchitectural channels.

F. GnuTLS

GnuTLS is another popular implementation of the TLS protocol. Like WolfSSL and NSS, GnuTLS does not use

```
1  int pkcs1_decrypt(key_size, m, length, message){
2    TMP_GMP_DECL(em, uint8_t);
3    uint8_t *terminator;
4    size_t padding;
5    size_t message_length;
6    int ret;
7    TMP_GMP_ALLOC(em, key_size);
8    nettle_mpz_get_str_256(key_size, em, m);
9    /* Check format */
10   if (em[0] || em[1] != 2){
11     ret = 0;
12     goto cleanup;
13   }
14   ...
15   memcpy(message, terminator+1, message_length);
16   *length = message_length;
17   ret = 1;
18   cleanup:
19   TMP_GMP_FREE(em);
20   return ret;
21 }
```

Listing 13. GnuTLS's PKCS #1 v1.5 verification

```
int proc_rsa_client_kx(session, data){
    ...
  // we do not need strong random numbers here.
  ret = gnutls_rnd(GNUTLS_RND_NONCE, rndkey.data,
    rndkey.size);
  ...
  ret = gnutls_privkey_decrypt_data(session->
    internals.selected_key, 0, &data, &plaintext);
  if (ret<0 || plaintext.size!=GNUTLS_MASTER_SIZE) {
    randomize_key = 1;
    ...
  }
  ...
  if (randomize_key != 0){
    session->key.key.data = rndkey.data;
    session->key.key.size = rndkey.size;
    rndkey.data = NULL;
  } else {
    session->key.key.data = plaintext.data;
    session->key.key.size = plaintext.size;
  }
  return ret;
}
```

Listing 14. Pseudocode of GnuTLS's padding oracle mitigation

constant time code for the PKCS #1 v1.5 verification, resulting is numerous side-channel-observable padding oracles.

Leaky Data Conversion. To convert RSA-decrypted plaintext from a little-endian big number format to big-endian byte array format, GnuTLS uses code from the Nettle cryptographic library[3]. Listing 12 shows the data conversion code in Nettle. Line 18 conditionally calls memset when there are leading zeros in the plaintext, exposing a Manger oracle.

Leaky PKCS #1 v1.5 Verification. GnuTLS also relies on leaky Nettle for PKCS #1 v1.5 verification (Listing 13). The branch in Line 10 allows for a Manger type oracle or a TTTT oracle. The conditional call to memcpy in Line 15 exposes an FFTT oracle.

Leaky Padding Oracle Mitigations. The GnuTLS padding oracle mitigation code is also not constant-time, see Listing 14

[3] https://www.lysator.liu.se/~nisse/nettle/

for a simplified version. In particular, the branches in Lines 7 and 12 yield a FFTT Bleichenbacher oracle. Another issue in the code present in Listing 14 is the misleading comment "we do not need strong random numbers here" (Line 3). We note that predicting the random session key used for padding oracle mitigation, renders the mitigation ineffective. The attacker can use this session key to generate the correct client finish message, thereby causing the server to complete the TLS handshake. This results in a remote Bleichenbacher FFTT oracle that does not require any side channel leakage. We believe that the random session key should be generated like other keys in the system (e.g., using the `GNUTLS_RND_KEY` RNG in GnuTLS).