Padding Oracle Attack on PKCS#1 v1.5: Can Non-standard Implementation Act as a Shelter?

Si Gao, Hua Chen, and Limin Fan

Trusted Computing and Information Assurance Laboratory,
Institute of Software, Chinese Academy of Sciences
{gaosi,chenhua,fanlimin}@tca.iscas.ac.cn

Abstract. In the past decade, Padding Oracle Attacks (POAs) have become a major threat to PKCS#1 v1.5. Although the updated scheme (OAEP) has solved this problem, PKCS#1 v1.5 is still widely deployed in various real-life applications. Among these applications, it is not hard to find that some implementations do not follow PKCS#1 v1.5 step-by-step. Some of these non-standard implementations provide different padding oracles, which causes standard POA to fail. In this paper, we show that although these implementations can avoid the threat of standard POA, they may still be vulnerable to POA in some way. Our study mainly focuses on two cases of non-standard implementations. The first one only performs the "0x00 separator" check in the decryption process; while the other one does not check for the second byte. Although standard POA cannot be directly applied, we can still build efficient padding oracle attacks on these implementations. Moreover, we give the mathematical analysis of the correctness and performance of our attacks. Experiments show that, one of our attacks only takes about 13 000 oracle calls to crack a valid ciphertext under a 1024-bit RSA key, which is even more efficient than attacks on standard PKCS#1 v1.5 implementation. We hope our work could serve as a warning for security engineers: secure implementation requires joint efforts from all participants, rather than simple implementation tricks.

1 Introduction

PKCS#1 is the standard for the implementation of public-key cryptography based on the RSA algorithm. The current version v2.2 [1], published by RSA in 2012, contains two encryption schemes: RSAES_PKCS1_v1.5 and RSAES_OAEP. For simplicity's sake, we denote them as PKCS#1 v1.5 and OAEP respectively. OAEP is required to be supported for new applications, while PKCS#1 v1.5 is included only for compatibility with existing applications.

Padding Oracle Attack. In the past decade, Padding Oracle Attacks (POAs) [2] have become a major threat to PKCS#1 v1.5. Padding Oracle Attack is a type of chosen ciphertext attack, which takes advantage of whether cryptographic operation is successfully executed. Usually, we assume the attacker can

M. Abdalla, C. Nita-Rotaru, and R. Dahab (Eds.): CANS 2013, LNCS 8257, pp. 39–56, 2013.

trick an honest user to decrypt the ciphertext he chose. In the decryption process, a format check is performed after decryption. Although the attacker does not have access to the decryption result, he can detect whether the ciphertext he chose passes the format check. We call such decryption process a "Padding Oracle" (PO) [3]. By collecting thousands of PO's responses, the original message can be extracted. In the past decade, POAs have drawn major attention from both symmetric and asymmetric cryptography research. In symmetric cryptography, CBC Padding Oracle has been used to build plaintext-recovery attacks on various network protocols [4–9]. In asymmetric cryptography, PKCS#1 v1.5 is the main target. The first POA on PKCS#1 v1.5, published by Bleichenbacher in 1998 [10], took about 1 million PO calls to recover a 1024-bit RSA plaintext. Bleichenbacher's attack has been extensively studied ever since, applied to SSL [11], PIN encryption in EMV [12], USB token [2] and XML encryption [13]. Recently, Bardou, Focardi, Kawamoto, Simionato, Steel and Tsay claim that using their improved version of Bleichenbacher's attack, a wrapped secret key can be recovered from RSA Securid 800 in only 13 minutes [2].

Other Attacks on PKCS#1 v1.5. Other non-POA attacks also exist for PKCS#1 v1.5: Coron, Joye, Naccache and Paillier proposed two brilliant attacks in 2000 [14], which can efficiently recover the plaintext, if the public exponent is small enough, or most message bits are zeros. Bauer, Coron, Naccache, Tibouchi and Vergnaud proposed a broadcast attack [15], which could reveal the identical plaintext when the public exponent is small. However, none of these attacks works for the commonly used public exponent 65537 with a random message. Bauer et al. also proposed a reliable distinguish attack [15]. Using one PO query, it predicts which of two chosen plaintexts corresponds to a challenge ciphertext. Although we only focus on full-plaintext-recovery attacks without requirements on exponent or plaintext, whether POAs can combine with these non-POA attacks may be an interesting topic for further study.

Does PKCS#1 v1.5 Still Matters in Today's Application? To avoid POA, RSA introduced OAEP as the new recommended encoding scheme in PKCS#1 v2.0. However, according to ECRYPT's "Yearly Report on Algorithms and Keysizes", PKCS#1 v1.5 is still widely deployed in today's application ((W)TLS, S/MME, XML, JSON, etc.) [16, 17]. Take USB tokens for instance: most tokens today support PKCS#1 v1.5, while only a few can support OAEP [2]. In software deployment, OAEP is widely supported today; while for backward compatibility reasons, PKCS#1 v1.5 is still mandatory. Jager and Paterson suggest that in such scenario [17], the attacker can trick the honest user to use legacy scheme (PKCS#1 v1.5) , and undermine the security of the up-to-date scheme (OAEP).

Motivation. Despite the fact that detailed implementation instructions are given in [1], implementations do not always follow them. For efficiency or other reasons, they tend to simplify the standard decryption process as long as valid ciphertexts can be decrypted correctly. For instance, in most of Microsoft's Cryptographic Service Providers (CSP), PKCS #1 v1.5 decryption does not check

padding string's length. Non-standard implementations also exist in many widely used cryptographic libraries (PGP, Boton, etc.) . Some may ignore padding string length check; others may not check the first byte of decryption result. For cryptographic devices, where the source code may not be available for public review, the situation is worse. In their full paper, Bardou et al. have already addressed this problem [2]. Bleichenbacher's attack can cover several non-standard implementations; while for the others, current POA on PKCS #1 v1.5 fails. Such implementations include Sata DKey (session key) [2] and perhaps many other devices that have not been studied yet. Since no POA works for them, non-standard implementations provide a "shelter" for cryptographic device vendors. In practice, such shelter can be pretty attractive: they only require minimal changes, which will not cause any compatibility trouble.

But can these non-standard implementations really prevent POA? Unfortunately, the answer remains unclear so far. All previous works mainly focus on standard implementation, leaving a lot of non-standard implementations in a "grey zone".

Our Contribution. In this paper, we focus on two types of non-standard implementations. The first one only checks if there is an byte with hexadecimal value `0x00` to separate padding string from the real message. The other one performs a thorough check, except for whether the second byte has hexadecimal value `0x02`. Recalls that Bleichenbacher's attack mainly takes advantage of the leading `0x0002` in the conforming check [10]. Clearly, Bleichenbacher's attack cannot apply to these cases.

Our attack on the second implementation is even more efficient than standard POA (it requires a mean of 13 000 oracle calls, while standard POA [2] needs 49 001 oracle calls). Our attack on the first implementation requires about 0.1 million oracle calls, which is still practical in application [13]. We give detailed correctness proof and complexity analysis of our attacks, as well as experiments to show their practical validity. Moreover, each attack can cover several non-standard implementations: together with Bleichenbacher's attack, we can see that secure "shelter" is indeed hard to find.

The rest of this paper is organized as follows. We first recall PKCS#1 v1.5 standard and Bleichenbacher's attack in Section 2. In Section 3, we present two case studies on certain implementation, and show that most of non-standard implementations are vulnerable to POA in some way. Experimental results are given in Section 4.

2 Padding Oracle Attacks on Standard PKCS#1 v1.5

2.1 PKCS#1 v1.5

We briefly recall PKCS#1 v1.5 standard in this section. For clarity, we use the same notations as [10]. PKCS#1 v1.5 encryption block format is given in Fig.1.

Encryption process takes the original message, pads it with pseudo-randomly generated non-zero bytes, then uses RSA to encrypt. Decryption process simply uses RSA to decrypt, then checks whether the padded message is valid. For valid padded message, the message in the data block is returned as decryption result. Otherwise, decryption fails with a "decryption error".

Fig. 1. PKCS#1 v1.5 block format for encryption

A valid "padded message" must satisfy all following conditions [1]:

a) The first byte has hexadecimal value `0x00`
b) The second byte has hexadecimal value `0x02`
c) The length of non-zero padding string (PS) is at least 8 bytes
d) There is an byte with hexadecimal value `0x00` to separate PS from data block

In the standard decryption process, all above conditions must be checked. However, some implementations simplify this process, only check part of these conditions. We call such implementations "non-standard implementations".

2.2 Bleichenbacher's Attack

Bleichenbacher shows that if the attacker can decide whether a chosen ciphertext is valid, the whole padded message can be recovered [10]. Following the notations of [10], we have (n, e) as the RSA public key; (p, q, d) as the corresponding secret key; k as the byte length of n. Let $B = 2^{8(k-2)}$, according to condition a) and b), for any valid ciphertext c, the corresponding padded plaintext m must satisfy

$$2B \leq m \bmod n < 3B$$

Bleichenbacher's attack takes full advantage of this interval, while ignoring other conditions. For more details on Bleichenbacher's attack, we refer to [10].

3 Padding Oracle Attacks on Non-standard PKCS#1 v1.5 Implementations

Through this section, attacks take a valid ciphertext c as input, and try to find the corresponding padded plaintext m through padding oracle calls. We use s as the multiplier of m, which is used in $c' = cs^e \bmod n$ to build the ciphertext for sm. The integer i represents the round counter, and $[a,b]$ represents the current m's interval.

3.1 Case 1: Implementation only Checks "the 0x00 Separator"

In this case, after RSA decryption, implementation searches for a 0x00 byte, from the most significant byte to the least significant byte. Whenever a 0x00 byte is found, PO returns 'True'; otherwise, PO returns 'False'. Take the need of real-life application into consideration, this PO may also have some other features:

- A 0x00 in the first or second byte does not count. For any valid padded message, the first two bytes are set to 0x0002. The most reasonable check is to ignore these bytes, start from the third byte and search for 0x00.
- If the only 0x00 exists in the last byte, decryption will fail. No message can be extracted from this ciphertext.

These additional conditions cause some trouble in our analysis, as we shall see soon.

Basic Idea. The "0x00 separator check" is rather complicated. Unlike condition a) and b) ("0x0002 ") , it does not have a clear mathematical expression. Luckily, certain multipliers can bridge this gap. For example, $256m$'s binary representation is the binary representation of m shifted left for 1 byte. On $\mathbb{Z}_n^*$, if $256m < n$, m passes the "0x00 separator check", $256m$ should pass the check as well; while if $256m > n$, since n is a random modular, $256m \bmod n$ could fail. Similar property holds for the messages that fail the check. Notice here we consider the second additional condition, otherwise PO will always returns 'True' for the ciphertext of $256m$ when $256m < n$. Suppose we have a randomly-generated padded message m_1 on $\mathbb{Z}_n^*$, with corresponding ciphertext $c_1 = {m_1}^e \bmod n$. Let $\mathrm{PO}(c_1)$ represent padding oracle's reply when input c_1. Let $m_2 = 256m_1 \bmod n$, $c_2 = {m_2}^e \bmod n$. In general, if we get different replies from $\mathrm{PO}(c_1)$ and $\mathrm{PO}(c_2)$, we can conclude that $256m_1 > n$.

However, considering the "additional conditions", a few abnormal points appear: for instance, when $256m_1 < n$, if m_1 has only one 0x00 byte in the third byte, $\mathrm{PO}(c_1)$ returns 'True' while $\mathrm{PO}(c_2)$ returns 'False' .

Formally, $PO(c_1) = F$ means m_1 contains no 0x00 from the third byte to the second to last byte. Since m_1 is picked uniformly on $\mathbb{Z}_n^*$, $Pr(PO(c_1) = F) = \left(\frac{255}{256}\right)^{k-3}$. Thus, the following proposition holds.

Proposition 1. *Assume m_1 is picked uniformly on $\mathbb{Z}_n^*$, $m_2 = 256m_1 \bmod n$, $c_1 = {m_1}^e \bmod n$, $c_2 = {m_2}^e \bmod n$. For commonly used RSA modulus length, $Pr(PO(c_1) \neq PO(c_2)|256m_1 < n)$ and $Pr(PO(c_1) \neq PO(c_2)|256m_1 > n)$ are distinguishable.*

Proof. When $256m_1 > n$, $PO(c_1)$ and $PO(c_2)$ are nearly independent. Notice that $256m_1 > n$ merely adds constraint condition on the first two bytes of m_1, $Pr(PO(c_1) = F) = Pr(PO(c_1) = F|256m_1 > n)$. Thus, we have

$$Pr(PO(c_1) \neq PO(c_2)|256m_1 > n) = 2\left(1 - \left(\frac{255}{256}\right)^{k-3}\right)\left(\frac{255}{256}\right)^{k-3}$$

When $256m_1 < n$, $Pr(PO(c_1) = T, PO(c_2) = F)$ means the only `0x00` appears in the third byte of m_1. Thus, we have $Pr(PO(c_1) = T, PO(c_2) = F|256m_1 < n) = \frac{1}{256}\left(\frac{255}{256}\right)^{k-4}$. Similarly, we also have $Pr(PO(c_1) = F, PO(c_2) = T|256m_1 < n) = \frac{1}{256}\left(\frac{255}{256}\right)^{k-3}$. Thus,

$$Pr(PO(c_1) \neq PO(c_2)|256m_1 < n) = \frac{1}{256}\left(\frac{255}{256}\right)^{k-4} + \frac{1}{256}\left(\frac{255}{256}\right)^{k-3}$$

$$Pr(PO(c_1) \neq PO(c_2)|256m_1 > n) = 2\left(1 - \left(\frac{255}{256}\right)^{k-3}\right)\left(\frac{255}{256}\right)^{k-3}$$

For $k = 128$,

$$Pr(PO(c_1) \neq PO(c_2)|256m_1 < n) \approx 0.0048$$

$$Pr(PO(c_1) \neq PO(c_2)|256m_1 > n) \approx 0.4744$$

Larger k leads to smaller $Pr(PO(c_1) \neq PO(c_2))$, although the probabilities above are still distinguishable. □

Since the probability difference here is quite significant, we can simply calculate $Pr(PO(c_1) \neq PO(c_2))$ in a small block near m_1, and use a threshold to decide whether $256m_1 > n$. With m's current interval, we can easily find a pair of (s_{max}, s_{min}) which can guarantee $s_{max}m \bmod n > \frac{n}{256}$ and $s_{min}m \bmod n < \frac{n}{256}$. Use $sm \bmod n$ as the m_1 in the procedure above, a tighter bound for s can be found, which leads to a smaller interval for m. This procedure can be extended to $[rn, (r+1)n)$, as in Algorithm 1.

It is worth mentioning that `256` is not the only multiplier that causes probability difference. Similar proposition also holds for the multiplier `2`.

Proposition 2. *Assume m_1 is picked uniformly on $\mathbb{Z}_n^*$, $m_2 = 2m_1 \bmod n$, $c_1 = {m_1}^e \bmod n$, $c_2 = {m_2}^e \bmod n$. For commonly used RSA modulus length, $Pr(PO(c_1) \neq PO(c_2)|2m_1 < n)$ and $Pr(PO(c_1) \neq PO(c_2)|2m_1 > n)$ are distinguishable.*

Proof. When $2m_1 > n$, the probability will be exactly the same as before. For $2m_1 < n$, if $PO(c_2) = T$ and $PO(c_1) = F$, there must exist a position $p \in [2, k-2]$ which satisfies $m_1[p] = 128$ and $0 < m_1[p+1] < 128$. Therefore,

$$Pr(PO(c_2) = T, PO(c_1) = F|2m_1 < n) = \left(1 - \left(1 - \frac{127}{255^2}\right)^{k-3}\right)\left(\frac{255}{256}\right)^{k-3}$$

Similarly, we also have

$$Pr(PO(c_2) = F, PO(c_1) = T|2m_1 < n) = \frac{127}{128}\left(1 - \left(\frac{509}{510}\right)^{k-3}\right)\left(\frac{255}{256}\right)^{k-3}$$

Algorithm 1. Multiply 256 method with additional condition, on $[rn, (r+1)n)$

Require: Padding Oracle PO, ciphertext c, m's current interval $[a, b]$, r
  compute the possible interval of s $[s_{min}, s_{max}]$
  $s_{min} = \left\lfloor \frac{(r+\frac{1}{256})n}{b} \right\rfloor$, $s_{max} = \left\lceil \frac{(r+\frac{1}{256})n}{a} \right\rceil$
  while $s < s_{max}$ **do**
    collect some sample points near current s, $c_1 = s^e c \bmod n$, $c_2 = (256s)^e c \bmod n$
    call PO to calculate $Pr(PO(c_1) \neq PO(c_2))$
    if $Pr(PO(c_1) \neq PO(c_2)) < threadhold$ **then**
      break
    else
      $s{+}{+}$
    end if
  end while
  $a' = \left\lceil \frac{r+\frac{1}{256}n}{s+block+1} \right\rceil$, $b' = \left\lfloor \frac{r+\frac{1}{256}n}{s-block} \right\rfloor$
  return $[a', b']$

For k=128,

$$Pr(PO(c_1) \neq PO(c_2) | 2m_1 < n) \approx 0.2653$$

$$Pr(PO(c_1) \neq PO(c_2) | 2m_1 > n) \approx 0.4744$$

Larger k leads to smaller $Pr(PO(c_1) \neq PO(c_2))$, although the probabilities above are still distinguishable. □

The probability difference here is not as significant as before. In order to decide whether $2m_1 > n$, better statistical tools should be applied. The multiply 2 version of Algorithm 1 is denoted as Algorithm 2. The initial interval of s in Algorithm 2 is quite large, a binary search can be applied here.

Attack Algorithm. Both Algorithm 1 and 2 can narrow down m's interval with different r, thus, we can keep running them until there's only one possible m. In some cases, keep using Algorithm 1 leads to a "stuck problem" (no available r can be found for Algorithm 1). For this reason, we introduce Algorithm 2. Since the probability difference in Algorithm 2 isn't as significant as Algorithm 1, for efficiency, Algorithm 2 is only used when Algorithm 1 is "stuck" . The full-plaintext-recovery attack is presented in Algorithm 3.

Analysis

Correctness Proof. First, we prove the correct padded message m always stays in interval $[a, b]$. In this section, assume that Algorithm 1 and 2 can always succeed. In round 0, we have $m \in [2B, 3B-1]$. If in round $i-1$,we have $m \in [a, b]$, in round i, the process can go to either Step 2.b or Step 2.c. Define s_{exact} as the

Algorithm 2. Multiply 2 method with additional condition,on $[rn, (r+1)n)$

Require: Padding Oracle PO, ciphertext c, m's current interval $[a, b]$, r
  compute the possible interval of s $[s_{min}, s_{max}]$
  $s_{min} = \left\lfloor \frac{(r+\frac{1}{2})n}{b} \right\rfloor$, $s_{max} = \left\lceil \frac{(r+\frac{1}{2})n}{a} \right\rceil$
  $sl = s_{min}$,$su = s_{max}$
  while $su > sl$ **do**
    $s = \lfloor \frac{su+sl}{2} \rfloor$
    collect some sample points near current s, call PO to get $Pr(PO(c_1) \neq PO(c_2))$
    if $Pr(PO(c_1) \neq PO(c_2))$ suggest $2m_1 < n$ **then**
      $sl = s$
    else
      $su = s$
    end if
  end while
  $a' = \left\lceil \frac{r+\frac{1}{2}n}{s+block+1} \right\rceil, b' = \left\lfloor \frac{r+\frac{1}{2}n}{s-block} \right\rfloor$
  return $[a', b']$

s satisfying both $256r_in < 256s_{exact}m \leq (256r_i + 1)\, n$ and $256\,(s_{exact} + 1)\, m > (256r_i + 1)\, n$. For Step 2.b, the initial interval of s we chose in Algorithm 1 always contains s_{exact}. After a round of Algorithm 1, $s_{exact} \in [s - block, s + block]$, thus:

$$a' = \left\lceil \frac{r_i + \frac{1}{256}n}{s + block + 1} \right\rceil \leq \left\lceil \frac{r_i + \frac{1}{256}n}{s_{exact} + 1} \right\rceil < m$$

$$b' = \left\lfloor \frac{r_i + \frac{1}{256}n}{s - block} \right\rfloor \geq \left\lfloor \frac{r_i + \frac{1}{256}n}{s_{exact}} \right\rfloor \geq m$$

Hence, if Step 2.b is chosen,we always have $m \in [a', b']$ when round i ends. Same analysis works for Step 2.c, as long as we change multiplier `256` above to `2`. Therefore, in this attack, the interval $[a, b]$ in each round always contains m.

Next, we prove our attack always narrows down m's interval in each round. Obviously, for m in $[2B, 3B-1]$, round 0 runs Step 2.b with $r_0 = 0$ successfully. Suppose after round $i-1$, m's interval is $[a, b]$, denote $t = \frac{b}{a}$. In round i, we need to prove m's new interval $[a', b']$ is smaller than $[a, b]$. Roughly speaking, this means in Algorithm 1 and 2, the length of $[s_{min}, s_{max}]$ should be larger than $2block + 1$. We use $block_{256}$, $block_2$ to denote the parameter $block$ in Algorithm 1 and 2.

If in round i, Step 2.b runs, we have

$$\begin{aligned} s_{max} - s_{min} &\geq (r_i + \frac{1}{256})n \times \frac{b-a}{ab} \\ &> \frac{n}{256} \times \left(1 + \frac{1}{t-1}\right) \times \frac{t-1}{at} - n \times \frac{b-a}{ab} \\ &= \frac{n}{256a} - n \times \frac{b-a}{ab} \end{aligned} \tag{1}$$

Algorithm 3. Full plaintext recovery attack for case 1

Require: ciphertext c, RSA modulus n and public exponent e
Step 1: Attack Start
Set $i = 1$,$r = -1$,$[a,b]$=$[2B,3B-1]$
Step 2: Using Padding Oracle to narrow down m's interval
while $b - a >$"Ending Parameter" **do**
Step 2.a: Check if multiply 256 method can be used. Compute $r_i = \left\lfloor \frac{1}{256\left(\frac{b}{a}-1\right)} \right\rfloor$
if $r_i > r_{i-1}$ **then**
Step 2.b: Multiply 256 method. Using Algorithm 1 with r_i
else
$r_i = r_{i-1}$
Step 2.c: Multiply 2 method. Using Algorithm 2 with $r = \left\lfloor \frac{1}{2\left(\frac{b}{a}-1\right)} \right\rfloor$,
end if
end while
Step 3: Exhaustive search to find m
return m

In Step 2.b we have $r_i > r_{i-1}$, thus

$$\begin{aligned} s_{max} - s_{min} &\geq (r_i + \frac{1}{256})n \times \frac{b-a}{ab} \\ &\geq (r_{i-1} + \frac{1}{256})n \times \frac{b-a}{ab} + n \times \frac{b-a}{ab} \end{aligned} \tag{2}$$

In experiment, $block_{256} = 8$ can ensure Algorithm 1 will success with high probability. It is easy to see $\frac{n}{256a} > 4block_{256} + 2$, thus from (1)(2) we can always prove $s_{max} - s_{min} > 2block_{256} + 1$.

Otherwise, if in round i, Step 2.c runs

$$\begin{aligned} s_{max} - s_{min} &\geq (r + \frac{1}{2})n \times \frac{b-a}{ab} \\ &> \frac{n}{2} \times \left(1 + \frac{1}{t-1}\right) \times \frac{t-1}{at} - n \times \frac{b-a}{ab} \\ &= \frac{n}{a} \times \left(\frac{1}{t} - \frac{1}{2}\right) \end{aligned} \tag{3}$$

In experiment, we find $block_2 = 200$ can ensure Algorithm 2 will success with high probability. Since $1 < t < 1.5$, $\frac{1}{6} < \frac{1}{t} - \frac{1}{2} < \frac{1}{2}$, we can always prove $s_{max} - s_{min} > 2block_2 + 1$ from (3). Thus, our attack always narrows down m's interval in each round. □

Complexity Analysis. In round i, Step 2.b will always be chosen when $r_i - r_{i-1} \geq 1$. Suppose in the beginning of round $i-1$, m exists in $[a_1, b_1]$, denote $t_1 = \frac{b_1}{a_1}$. Let $len = b - a$, $len_1 = b_1 - a_1$, $d = \frac{len}{len_1}$. No matter which method runs in round $i-1$, we always have $r_{i-1} \leq \frac{1}{256(t_1-1)}$. Therefore,

$$r_i - r_{i-1} > \frac{1}{256} \times \left(\frac{1}{t-1} - \frac{1}{t_1 - 1}\right) - 1 = \frac{1}{256} \times \left(\frac{a}{len} - \frac{a_1}{len_1}\right) - 1$$
$$\geq \frac{a}{256} \times \left(\frac{1}{len} - \frac{d}{len}\right) - 1 = \frac{1}{256} \times \frac{1}{t-1} \times (1-d) - 1 \quad (4)$$

From above, it is not hard to see for Step 2.b $1 - \mathrm{d} \geq \frac{1}{2block_{256}+2}$, while for Step 2.c,$1 - \mathrm{d} \geq 1 - \frac{2block_2+1}{\frac{n}{6a}}$. Notice $\frac{1}{t-1}$ keeps growing in our attack. Therefore, after some rounds, we will reach a situation where $\frac{1}{t-1}$ is large enough to ensure $r_i - r_{i-1} \geq 1$. From this point, Step 2.b will always be chosen until we find m. For $k = 128$, after 3 rounds of Step 2.c, $\frac{1}{t-1}$ will be large enough to ensure this. Roughly speaking, if round $i-1$ and i both runs Step 2.b, $d \approx \frac{2block_{256}+1}{\frac{n}{256a}} \approx 0.24$, while in Step 2.c d is close to 0.04. In practice, binary search is used for Step 2.c, which leads to complexity close to $2\mathrm{block}_2 \times \log_2 |s_{max} - s_{min}| = 2\mathrm{block}_2 \times \log_2 \frac{n}{2a} \approx 5300$. In Step 2.b, the interval of s is rather small (most times less than 100) ; we have to search for all of s between s_{max} and s_{min}, which leads to nearly 170 times oracle calls. We give an overall complexity approximation as $5300 \times 3 + 170 \times \left(\log_{0.24}\left(\frac{EndParam}{B} \div 0.04^3\right) + 1\right)$. For EndParam=100000, the overall complexity is close to 0.1 million oracle calls.

3.2 Case 2: PO Checks All Conditions Expect for the Second Byte

In this case, PO checks condition a) (the `0x00` prefix) . This seems to be a pleasant condition, since it has a perfect mathematical expression. Let $T = 256B = 2^{8(k-1)}$, condition a) equals to $sm \bmod n < T$. This condition alone results in a very efficient attack [18], although condition c) and d) (PS length and the `0x00` separator) have complicated the case.

Basic Idea. With condition c) and d), oracle will not return 'True' for every $sm \bmod n < T$. Thus, highly efficient binary search from [18] cannot be applied. However, the framework of our attack in Section 3.1 still works. In fact, now we have a better situation: one side of the interval is deterministic. Whenever get a 'True' from oracle, we know for sure $sm \bmod n < T$.

Attack Algorithm. With the same notation, the "narrow down process" for this case is as follow. Search from the lower bound, whenever get a 'True' from PO, set the new lower bound of s to current s. If after a while no more $sm \bmod n$ causes a 'True' reply from PO, we can conclude that we have passed s_{exact}, and set the new upper bound of s to current s. As in the previous section, a parameter *block* is used to describe this interval. This process is presented in Algorithm 4 in Appendix C. Note that we can also search from the upper side, like Algorithm 5 in Appendix C.

Full-plaintext-recovery attack can be built from Algorithm 4 and 5, like we did in the previous section. Step 1 and 3 are exactly the same as before, here we only presents Step 2 in detail.

Step 2: Using Padding Oracle to narrow down m's interval
while $b - a >$ "Ending Parameter" **do**
  Step 2.a: Check if Algorithm 4 can be used. Compute $r_i = \left\lfloor \frac{1}{\frac{b}{a}-1} \times \frac{T}{n} \right\rfloor$
  if $r_i > r_{i-1}$ **then**
    Step 2.b: use Algorithm 4 with r_i
  else
    $r_i = r_{i-1}$
    Step 2.c: use Algorithm 5 with $r = \left\lfloor \frac{1}{\frac{b}{a}-1} \left(1 - \frac{T}{n}\right) - \frac{T}{n} \right\rfloor$
  end if
end while

Analysis. Since attack for this case has the same skeleton as before, correctness proof and complexity analysis is quite similar. Due to the length limitation, we present this part in Appendix B.

3.3 Other Non-standard Implementations

According to Section 2.1, standard PO returns 'True' if the RSA decryption result satisfies all four conditions. Therefore, we describe the type of implementations as a four bits integer. The most significant bit represents whether condition a) is checked in PKCS#1 v1.5 decryption. If condition a) is checked, this bit is 1; otherwise, it is 0. The other three bits (from the second most significant to the least significant) , represents whether condition b), c), d) is checked, respectively. Thus, type 15 stands for standard implementation, while type 1 represents the case in Section 3.1. Based on the two most significant bits, implementations can be further divided into four groups:

Group I (Both the `0x00` prefix and the `0x02` are ignored) : We ignore type 0, because it can not separate padding string from the data block. Our attack in Section 3.1 is designed for type 1. Unlike Bleichenbacher's attack, our attack is insensitive to subtle condition changes. Since condition c) (PS length check) has limited influence on the overall probability distribution, our attack also works on type 3. Type 2 does not seem to be a reasonable implementation in practice: there is no separator between data and padding string. Without such separator, the implementation should have a fixed length for the data block. In that case, checking the length of padding string seems unnecessary.

Group II (The `0x02` is checked while the `0x00` prefix is ignored) : In this group, implementations check condition b) (`0x02`) while ignore condition a) (the `0x00` prefix) . Unlike other groups, the distribution of m that causes a 'True' reply from PO is "partly-discrete" . It is not as discrete as Group I, neither does it have a neat mathematical expression like Group III or IV. We suspect

this group could be vulnerable to a simple variant[1] of Bleichenbacher's attack, although some adjustment is needed for Step 3.

Group III (The `0x00` prefix is checked while the `0x02` is ignored) : Our attack in Section 3.2 is designed for type 11 (with PS length and `0x00` separator check). For the same reason, it also works for type 9 (without PS length check). Type 8 forms a perfect PO for Manger's attack [18], which is the most efficient attack in all non-standard implementations. For type 10, the situation is similar to type 2.

Group IV (Both the `0x00` prefix and the `0x02` are checked) : Bleichenbacher's attack can be applied to all implementations in this group. Type 15 is the standard implementation. Type 12, 13, 14 corresponds to the "TTT" , "FTT" and "TFT" oracle in [2], respectively.

4 Experimental Results

With discussion in Section 3.3, implementation types with corresponding attacks is given in Table 1, along with the complexity in terms of oracle calls. In this section, all experiments use 1024-bit RSA modulus n, e=65537, with PKCS#1 v1.5 encryption. Each experiment runs 1000 times, with 16 bytes of randomly generated messages. For Bleichenbacher's attack, we use Bardou el al.'s improved version[2] [2].

Table 1. Implementation types with corresponding attack algorithm

Type	Conditions				Group	Available Attack Algorithm	Performance	
	a)	b)	c)	d)			Median	Mean
1				✓	Group I	Group I Attack	113 520	115 978
2			✓			Unnecessary	—	—
3			✓	✓		Group I Attack	111 890	114 331
4		✓			Group II	Variant of Bleichenbacher's attack	—	—
5		✓		✓				
6		✓	✓					
7		✓	✓	✓				
8	✓				Group III	Manger's attack	1 168	1 174
9	✓			✓		Group III Attack	12 843	12 878
10	✓		✓			Unnecessary	—	—
11	✓		✓	✓		Group III Attack	13 047	13 058
12	✓	✓			Group IV	Bleichenbacher's attack	4 762	14 532
13	✓	✓		✓		Bleichenbacher's attack	15 315	92 820
14	✓	✓	✓			Unnecessary	—	—
15	✓	✓	✓	✓		Bleichenbacher's attack	17 473	104 839

[1] In this paper, "variant" means simply adjust some of the parameters in the original algorithm, while the basic skeleton and the core unit remains the same.

[2] Notice our results here are worse than [2]. This is probably caused by the *Parallel thread method*. However, even if we use the performance in [2], the performance order remains the same.

Condition c) (PS length check) alone has rather limited influence: type 1 and 3 share the same attack algorithm, and have similar complexity. Same property holds for type 9 and 11, type 13 and 15. Compare with others, Group IV Attack's (Bleichenbacher's attack) performance varies a lot. Their density distribution curves have longer "tails", which is given in Fig.2 (Appendix A) . All attacks above run in different situations, it is not fair to compare their performance with each other. Nonetheless, from Table 1, we can easily conclude that most of the non-standard implementations are insecure against POA.

5 Conclusion

In the last decade, Padding Oracle Attacks (POAs) have become a major threat to PKCS#1 v1.5 [2, 10]. To our knowledge, all previous works mainly focus on the standard implementation. However, in today's application, some implementations do not always follow the standard step-by-step. Since the padding oracles in these non-standard implementations are quite different from the standard padding oracle, Bleichenbacher's attack cannot cover all of them. Using the similar idea as Bleichenbacher's attack, we propose two attacks for certain non-standard implementations: one requires about 0.1 million oracle calls, while the other requires only 13 000 oracle calls. Together with Bleichenbacher's attack, we can see that most of the "non-standard implementations" are vulnerable to POA in some way. We hope our work could convince industry engineers that the threat of POA can not be prevented by some simple implementation tricks.

Acknowledgements. We would like to thank the anonymous reviewers for providing valuable comments. We would also like to thank Damien Vergnaud for his generous help and valuable advice. This work is supported by the National Natural Science Foundation of China (No.91118006) and the National Basic Research Program of China (No.2013CB338002).

References

1. RSA Laboratories: PKCS #1 v2.2: RSA Cryptography Standard (October 27, 2012)
2. Bardou, R., Focardi, R., Kawamoto, Y., Simionato, L., Steel, G., Tsay, J.-K.: Efficient Padding Oracle Attacks on Cryptographic Hardware. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 608–625. Springer, Heidelberg (2012)
3. Vaudenay, S.: Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS... In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 534–545. Springer, Heidelberg (2002)
4. Canvel, B., Hiltgen, A.P., Vaudenay, S., Vuagnoux, M.: Password Interception in a SSL/TLS Channel. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 583–599. Springer, Heidelberg (2003)
5. Paterson, K.G., Yau, A.K.L.: Cryptography in Theory and Practice: The Case of Encryption in IPsec. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 12–29. Springer, Heidelberg (2006)

6. Degabriele, J., Paterson, K.: Attacking the IPsec Standards in Encryption-only Configurations. In: IEEE Symposium on Security and Privacy, SP 2007, pp. 335–349 (2007)
7. Albrecht, M., Paterson, K., Watson, G.: Plaintext Recovery Attacks against SSH. In: 2009 30th IEEE Symposium on Security and Privacy, pp. 16–26 (2009)
8. Degabriele, J.P., Paterson, K.G.: On the (in)security of IPsec in MAC-then-encrypt configurations. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, pp. 493–504. ACM, New York (2010)
9. Rizzo, J., Duong, T.: Practical Padding Oracle Attacks. In: WOOT 2010: 4th USENIX Workshop on Offensive Technologies. USENIX Association (2010)
10. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 1–12. Springer, Heidelberg (1998)
11. Klíma, V., Rosa, T.: Further Results and Considerations on Side Channel Attacks on RSA. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 244–259. Springer, Heidelberg (2003)
12. Smart, N.P.: Errors Matter: Breaking RSA-Based PIN Encryption with Thirty Ciphertext Validity Queries. In: Pieprzyk, J. (ed.) CT-RSA 2010. LNCS, vol. 5985, pp. 15–25. Springer, Heidelberg (2010)
13. Jager, T., Schinzel, S., Somorovsky, J.: Bleichenbacher's Attack Strikes again: Breaking PKCS#1 v1.5 in XML Encryption. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 752–769. Springer, Heidelberg (2012)
14. Coron, J.-S., Joye, M., Naccache, D., Paillier, P.: New Attacks on PKCS#1 v1.5 Encryption. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 369–381. Springer, Heidelberg (2000)
15. Bauer, A., Coron, J.-S., Naccache, D., Tibouchi, M., Vergnaud, D.: On the Broadcast and Validity-Checking Security of PKCS#1 v1.5 Encryption. In: Zhou, J., Yung, M. (eds.) ACNS 2010. LNCS, vol. 6123, pp. 1–18. Springer, Heidelberg (2010)
16. European Network of Excellence in Cryptology II: ECRYPT II Yearly Report on Algorithms and Keysizes (2009-2010). Technical report, European Network of Excellence in Cryptology II (March 30, 2010)
17. Jager, T., Paterson, K.G., Somorovsky, J.: One Bad Apple: Backwards Compatibility Attacks on State-of-the-Art Cryptography. In: NDSS Symposium 2013 (2013)
18. Manger, J.: A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 230–238. Springer, Heidelberg (2001)

A Density Distribution of Oracle Calls of the Attacks in Table 1

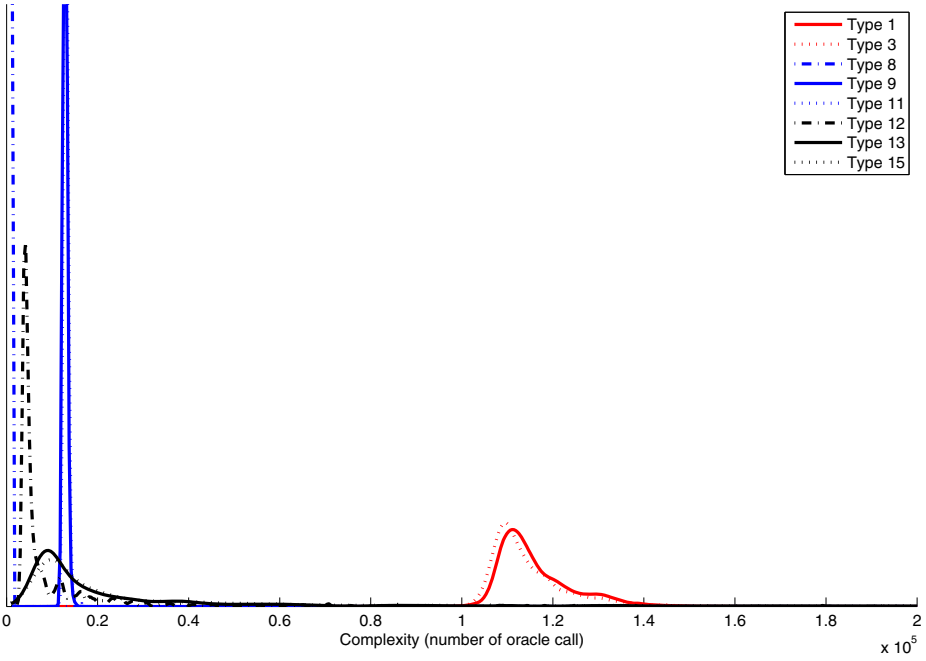

Fig. 2. Distribution of oracle calls of attacks in Table 1, each attack runs 1000 times

B Analysis of the Attack for Case 2

Correctness Proof. Assume *block* is chosen to ensure Algorithm 4 and 5 always succeed. Since Algorithm 5 has a more clear form, we start with Algorithm 5.

First, we prove m always stays in the interval $[a, b]$. Initially, we have $m \in [2B, 3B-1]$, which means the proposition holds for round 0. Suppose in round i-1, we have $m \in [a, b]$; according to Algorithm 5, after Step 2, we have $s_{\text{exact}} \in [s, s + block]$. s_{exact} is defined as the s satisfies , $s_{exact}m \leq r_i n + T$ and $(s_{exact} + 1)m > r_i n + T$. Obviously such s_{exact} exist in $[s_{min}, s_{max}]$. Thus, in Algorithm 5 we have:

$$a' = \left\lceil \frac{r_i n + T}{s + block + 1} \right\rceil \leq \left\lceil \frac{r_i n + T}{s_{exact} + 1} \right\rceil < m$$

$$b' = \left\lfloor \frac{r_i n + T}{s} \right\rfloor \geq \left\lfloor \frac{r_i n + T}{s_{exact}} \right\rfloor \geq m$$

Proof for Algorithm 4 works exactly the same way.

Then we show in each round of our attack, m's interval is narrowed down as expected. In experiment, we find $block = 17$ provide a success rate larger than 90%. Obviously, for m in $[2B, 3B-1]$, round 0 runs Step 2.b with $r_0 = 0$ successfully. Suppose after round i-1, m's interval is $[a, b]$, denote $t = \frac{b}{a}$. In round i, we need to prove s's initial interval $[s_{min}, s_{max}]$ is always larger than $block+1$.

In round i, if Step 2.b is executed,

$$\begin{aligned} s_{max} - s_{min} &\geq (r_i n + T) \times \frac{b-a}{ab} \\ &> \left(\left(\frac{1}{t-1} \times \frac{T}{n} - 1\right) n + T\right) \times \frac{b-a}{ab} \\ &= \frac{T}{a} - n \times \frac{b-a}{ab} \end{aligned} \tag{5}$$

Since $r_i > r_{i-1}$, we also have

$$\begin{aligned} s_{max} - s_{min} &\geq (r_i n + T) \times \frac{b-a}{ab} \\ &> (r_{i-1} n + T) \times \frac{b-a}{ab} + n \times \frac{b-a}{ab} \end{aligned} \tag{6}$$

Since we have $\frac{T}{a} > 2\,(block+1)$, from (5)(6), we can always prove $s_{max} - s_{min} > block + 1$.

Otherwise, if Step 2.c is executed

$$\begin{aligned} s_{max} - s_{min} &\geq (r_i n + T) \times \frac{b-a}{ab} \\ &> \left(\left(\frac{1}{t-1} \times \left(1 - \frac{T}{n}\right) - \frac{T}{n} - 1\right) n + T\right) \times \frac{b-a}{ab} \\ &= \frac{n-T}{b} - n \times \frac{t-1}{b} = \frac{(2-t) \times n - T}{b} \end{aligned} \tag{7}$$

Since $t < 1.5$, $(2-t)n > 0.5n$,$s_{max} - s_{min} > \frac{63T}{3B} > block + 1$.This completes our correctness proof. □

Complexity Analysis Complexity approximation is similar to our analysis in Section 3.3. To make sure in round i Step 2.b is executed, we need $r_i - r_{i-1} > 1$.

$$\begin{aligned} r_i - r_{i-1} &> \frac{T}{n} \times \left(\frac{1}{t-1} - \frac{1}{t_1 - 1}\right) - 1 = \frac{T}{n} \times \left(\frac{a}{len} - \frac{a_1}{len_1}\right) - 1 \\ &> \frac{aT}{n} \times \left(\frac{1}{len} - \frac{d}{len}\right) - 1 = \frac{T}{n} \times \frac{1}{t-1} \times (1-d) - 1 \end{aligned} \tag{8}$$

Since $\frac{T}{n} \in \left(\frac{1}{256}, \frac{1}{128}\right)$, in Step 2.b we have $1 - d \geq \frac{1}{block+2}$, while for Step 2.c,d $< \frac{block+1}{\frac{0.5n-T}{b}} < 0.003$, $1 - d > 0.997$. $\frac{1}{t-1}$ keeps increasing in our attack. Thus, we have the same "turning point" as before, after which only Step 2.b will be executed. Normally, only 1 or 2 round of Algorithm 5 is need.

Now let's consider how many oracle calls is needed for each round. Noted in Section 3.1, both multiply 2 method and multiply 256 method have a "balanced" interval. Roughly speaking, in narrow down process, s_{exact} is most likely to appear in the middle part of $[s_{min}, s_{max}]$. Duo to the facts that Algorithm 4 and 5 is "one-side-deterministic", now s_{exact} is most likely to appear in somewhere near s_{min}. Suppose for round i-1, we got new bounds for s as $[s_{newmin}, s_{newmax}]$. For a random m, condition c) and d) can be satisfied with probability $Pr\,(c\&d) = \left(\frac{255}{256}\right)^8 \left(1 - \left(\frac{255}{256}\right)^{k-10}\right)$. For $k = 128$, this probability is close to 0.36. Thus, the distribution sequence of s_{exact}'s position should follow Table 2. Clearly this is a

Table 2. The distribution sequence of s_{exact}'s position

$s_{exact} - s_{newmin}$	0	1	2	...	q	...
Probability	0.36	$0.36 \cdot 0.64$	$0.36 \cdot 0.64^2$	...	$0.36 \cdot 0.64^q$	...

geometric distribution, with expectation of $\frac{1-0.36}{0.36} = 1.78$. We use *slen* denoted the length of $[s_{min}, s_{max}]$. If in round i, *slen* is exactly the same as the length of $[s_{newmin}, s_{newmax}]$(a.k.a, *block*) in round i-1, at the most times, we only need to search for less than $1.78 + 1 + block$ points for Algorithm 4 to find s_{exact}'s new lower bound. For Algorithm 4, roughly speaking, we have

$$slen = s_{max} - s_{min} \approx (r_i n + T) \times \frac{b-a}{ab} = \left(\frac{T}{t-1} + T\right) \times \frac{t-1}{at} = \frac{T}{a}$$

which means *slen* is almost stable. In that case, $d = \frac{block}{\frac{T}{a}}$ is also stable. Choosing the r_i in Algorithm 4 enlarge $[s_{newmin}, s_{newmax}]$ of round i-1 to $[s_{min}, s_{max}]$ in round i. This means the number of oracle calls in round i is roughly $1.78 \times \frac{1}{d} + d \times \frac{T}{a} + 1$. In fact, this is why we favor Algorithm 4 over Algorithm 5. Although Algorithm 4 has a neat form and larger r, we can see above that it always starts searching for s_{exact} from the "unlikely" upper side, which causes complexity waste. For complexity approximation, we simply ignore the several rounds of Algorithm 5 in the beginning. Thus, the overall complexity should be:

$$C\,(d) = \left(1.78 \times \frac{1}{d} + d \times \frac{T}{a} + 1\right) \times \log_d \frac{1}{B}$$

$C(d)$ reaches its minimal at $d = 0.08$, increases for $d > 0.08$. In Algorithm 3, for $block = 17$, even if we choose the largest r possible (like we did in our attack),we cannot get a $d = 0.08$. Thus, the minimal of d lead to the minimal of the complexity of our attack. With $d = 0.175$, we have $C(d)$ around 11 thousands.

C Pseudo-code for Algorithms in the Text

Algorithm 4. Narrow down process for PO that does not check the second byte

Require: Padding Oracle PO, ciphertext c, m's current interval $[a, b]$, r
  compute the possible interval of s $[s_{min}, s_{max}]$
  $s_{\min} = \lfloor \frac{rn+T}{b} \rfloor$, $s_{\max} = \lceil \frac{rn+T}{a} \rceil$
  $fcount = 0$,$s = s_{min}$
  while $s \leq s_{max}$ **do**
    $c_1 = s^e \bmod n$;
    if PO(c_1)==T **then**
      $fcount \leftarrow 0$;
    else
      $fcount$++;
    end if
    if $fcount > block$ **then**
      break;
    end if
    s++;
  end while
  $a' = \lceil \frac{rn+T}{s} \rceil, b' = \left\lfloor \frac{rn+T}{s-block-1} \right\rfloor$
  return $[a', b']$

Algorithm 5. Another narrow down process for PO that does not check the second byte

Require: Padding Oracle PO, ciphertext c, m's current interval $[a, b]$, r
  compute the possible interval of s $[s_{min}, s_{max}]$
  $s_{\min} = \lfloor \frac{rn+T}{b} \rfloor$, $s_{\max} = \lceil \frac{rn+T}{a} \rceil$
  $s = s_{max}$
  while PO(c_1)==T **do**
    $s = s - 1$;
    $c_1 = s^e \bmod n$;
  end while
  $a' = \left\lceil \frac{rn+T}{s+block+1} \right\rceil, b' = \lfloor \frac{rn+T}{s} \rfloor$
  return $[a', b']$