Attacking GlobalPlatform SCP02-compliant Smart Cards Using a Padding Oracle Attack

Gildas Avoine[1,2] and Loïc Ferreira[3,1]

[1] Univ Rennes, INSA Rennes, CNRS, IRISA, France
[2] Institut Universitaire de France
[3] Orange Labs, Applied Cryptography Group, Caen, France

Abstract. We describe in this paper how to perform a padding oracle attack against the GlobalPlatform SCP02 protocol. SCP02 is implemented in smart cards and used by transport companies, in the banking world and by mobile network operators (UICC/SIM cards). The attack allows an adversary to efficiently retrieve plaintext bytes from an encrypted data field. We provide results of our experiments done with 10 smart cards from six different card manufacturers, and show that, in our experimental setting, the attack is fully practical. Given that billions SIM cards are produced every year, the number of affected cards, although difficult to estimate, is potentially high. To the best of our knowledge, this is the first successful attack against SCP02.

Keywords: Security protocol · Padding oracle attack · GlobalPlatform · Smart cards · Timing side-channel

1 Introduction

1.1 Context

GlobalPlatform is an organisation that aims at defining technical mechanisms related to the chip technology (e.g., smart cards, application processors, SD cards, USB tokens, secure elements), used to securely add and remove applications, and related parameters, into the chips. This initiative aims also at facilitating the interoperable deployment and management of applications on these types of device, regardless of the manufacturer, as well as "*promot*[ing] *a global infrastructure for smart card implementation across multiple industries*" [Glo18].

Several Secure Channel Protocols (SCP) are specified by GlobalPlatform. Most of them are based on symmetric-key cryptography (e.g., SCP01, SCP02, SCP03, SCP81). Regarding the protocols status, SCP01 (based on the DES algorithm) is now deprecated. SCP02 (based on 3DES) is currently the most deployed symmetric-key based SCP protocol[1], while the use of SCP03 (based on AES) seems to be less widespread.[2]

SCP02 is a protocol aiming at establishing a secure channel between a card and an "*off-card entity*". When used over-the-air, the main purpose of the protocol (yet not the only one) is the management of a (remote) card. Through the secure channel established with SCP02, applets, files, secret data (e.g., encryption keys, PIN codes), etc., may be transmitted and stored into the card.

According to GlobalPlatform, SCP02 is used by transport companies, in the banking world and by mobile network operators (UICC/SIM cards). The number of SIM cards used

[1] Last release: March 2018 [Glo18].
[2] Last release: July 2014 [Glo14].

IACR Transactions on Cryptographic Hardware and Embedded Systems ISSN 2569-2925, Vol. 2018, No. 2, pp. 149–170
DOI:10.13154/tches.v2018.i2.149-170

worldwide in 2015 is estimated to 5.3 billion by the SIMalliance [SIM16], and 7.6 billion[3] by the GSMA [GSM16], although it is difficult to estimate what proportion implements SCP02. A typical usage of the SCP02 protocol is the management of a specific application storing user credentials (e.g., for payment or transport transactions) located in the UICC card that is plugged into the smartphone. Depending on the context, an additional security protocol (e.g., TLS [DR08] or SCP80 [ETS17]) may be used together with SCP02. For instance, in order for a remote server to send SCP02 commands to the UICC, a SCP02 channel is established between the remote server and the UICC through the intermediary of an application on the smartphone. In addition a second secure channel is established between the remote server and the application on the smartphone (e.g., with TLS or SCP80). This second security layer embeds the SCP02 commands sent by the server. The data exchanged between the server and the application are protected with the two security layers, whereas the data exchanged between the application on the smartphone and the UICC are protected with SCP02 only.

1.2 Contribution

We describe how to perform a padding oracle attack against the SCP02 protocol. This attack allows an attacker to retrieve the plaintext data sent to a smart card through the secure SCP02 channel. We present the results of our practical experiments made with 10 smart cards from six card manufacturers. We explain how to exploit a timing channel provided by these cards in order to successfully achieve the attack against them.

1.3 Paper outline

Section 2 summarises previous work related to padding oracle attacks. Section 3 describes the SCP02 protocol. Based on the regular padding oracle attack described in Section 4, we explain in Section 5 how to exploit a weakness of several smart cards implementing SCP02 in order to apply the attack. We present our setting and our experimental results. In Section 6, we list possible countermeasures. Section 7 deals with our responsible disclosure of this work to various card manufacturers. Finally, we conclude in Section 8.

1.4 Notations

A byte value is written as `7E`. The value $\texttt{00}^i$ corresponds to the byte string made of i bytes equal to `00`. $b_i|b_{i+1}$ refers to the concatenation of the bytes b_i and b_{i+1}. $C\|B$ refers to the concatenation of the two DES blocks C and B. ENC indicates a symmetric encryption function, while ENC^{-1} indicates the corresponding decryption function.

2 Related work

In 2002, Vaudenay [Vau02] describes the theoretical principles of an attack against a symmetric-key encryption algorithm in CBC mode, leaning on the padding data, that enables the decryption of encrypted data. He presents applications against several security protocols (e.g., TLS 1.0 [DA99], IPsec [KS05] and ESP [Ken05], WTLS [Wir01]). The latter is adapted by Canvel, Hiltgen, Vaudenay, and Vuagnoux [CHVV03] who succeed in bypassing two limitations of Vaudenay's attack: the lack of differentiated error messages when the padding is invalid and when the MAC tag is invalid (a timing side-channel is used instead), and the break of the secure channel when a cryptographic error occurs (the underlying application they target repeatedly sets up a TLS session and transmits the same secret). Thereby they provide a fully practical attack against TLS 1.0.

[3] Including M2M connections.

AlFardan and Paterson [PA12] describe a padding oracle attack against DTLS [RM06] and targeting actual implementations even though TLS implementations have been patched in order to thwart the padding oracle attacks.[4] Their attack is made easier because a DTLS session is not stopped when a cryptographic error occurs. On the other hand, in such a case, the invalid packet is silently discarded and no error message is sent. In order to circumvent this drawback, they perform a timing attack (using a kind of keep-alive command) and amplify the response time difference thanks to a batch of numerous packets which all either have valid or invalid padding and hence all contribute to an accumulated timing difference in the same way.

Afterwards, the padding oracle attack has been applied to various network and application protocols, and schemes such as IPsec [PY04, YPM05, DP07, DP10], SSL 3.0 [MDK14], EMV [DLP+12], ASP.NET [DR11], XML [JS11, JSS12, KMSS15], JavaServer Faces CAPTCHA, Ruby on Rails framework, and OWASP Enterprise Security API Toolkits [RD10]. In addition, Rizzo and Duong [RD10] describe how to turn a padding oracle into an encryption oracle[5], under the condition that the encryption key is invariant (at least it does not change throughout several messages decryption, in particular it is not renewed in case of a cryptographic error).

Using the same kind of technique as [PA12], AlFardan and Paterson [AP13] use another side-channel to perform attacks targeting DTLS (practical) and TLS (almost practical). Their timing attack, called "Lucky 13", relies on the number of inner compression function iterations made during the HMAC computation when verifying a message authentication tag. The attack remains possible despite the fact that the targeted cryptographic libraries implement the recommended countermeasure. Indeed the recommended mitigation consists, when the padding data is invalid, in performing a MAC verification on the decrypted data as if there were *no padding data*. This leaves a small timing channel that can be exploited (based on the discrepancy when the padding data is valid and when one assumes there is no padding data).

Using a variant of Lucky 13, Albrecht and Paterson [AP16] succeed in attacking the Amazon' s2n implementation of SSL/TLS [Sch15, AWS], overcoming the countermeasures implemented in the cryptographic library.

Irazoqui, İnci, Eisenbarth, and Sunar [AIES15] show that it is still possible to apply the Lucky 13 attack in a cloud setting even if the recommended mitigations against the attack are implemented. They consider co-located virtual machines which are able to know if a dummy function (used to equalize the processing time) is called or not.

Paterson and Watson [PW08] provide a formal security treatment of the CBC mode encryption with padding in the chosen plaintext setting. They show that a padding method that has no invalid padded message achieves immunity against padding oracle attacks when the underlying block cipher is modeled as a pseudo-random permutation family. Paterson and Watson [PW12] also extend existing security models for authenticated encryption of Bellare and Namprempre [BN08] to incorporate padding oracle attacks in the chosen ciphertext setting.

Moreover the attack presented in 1998 by Bleichenbacher [Ble98] against the PKCS #1 v1.5 RSA encryption scheme in SSL can retrospectively be seen as a padding oracle or, more generally, a "format oracle" attack. This attack aims at retrieving the "premaster secret" negotiated by a client and a server, and used to compute the session keys. This adaptive ciphertext attack uses the specific plaintext format in order to gradually narrow the interval the plaintext belongs to until one possible value remains. In order to know if his guess is correct, Bleichenbacher exploits an error code sent by the server when the decrypted message does not correspond to a valid format. Bleichenbacher attack has been followed by subse-

[4] As noted in [PA12], the explanation could lie in the fact that DTLS provides no error messages in case of a cryptographic error, neither does the secure tunnel end. Thereby a remaining timing channel could have been seen as not usable in order to build a padding oracle.

[5] The underlying algorithm provides encryption only, not *authenticated* encryption.

quent improvements and developments (e.g., [Man01, KPR03, BFK+12, JSS15, ASS+16]).

To the best of our knowledge, there is no security analysis of SCP02 besides a theoretical attack by Sabt and Traoré [ST16]. They describe a chosen plaintext attack based on the fact that the encryption scheme is deterministic (data is encrypted in CBC mode with a static IV). In addition, they provide the first security proof of SCP03 [Glo14].

Regarding the padding method, Black and Urtubia [BU02] describe several of them that may provide a suitable oracle in order to perform an attack, among which a padding scheme also used in SCP02. In addition, they present methods supposed to resist padding oracle attacks.

3 The SCP02 protocol

SCP02 aims at establishing a secure link between an "*off-card entity*" and a card [Glo18].[6] SCP02 is based on symmetric-key algorithms. The card and the server share one or several sets of symmetric keys. A set is made of three keys (which value may be equal). From that set, session keys are computed each time a new channel is established. The card manages a sequence counter related to a given keyset. Its initial value is 0 and incremented after each successful session (established with that keyset). Once the sequence counter has reached its maximum value, the card must not start anymore a session with the corresponding keyset.

Depending on the security level negotiated during the key exchange[7], the commands sent by the server and the responses sent by the card are encrypted and protected with a MAC tag. Regarding the commands, the lowest security level is data integrity. Moreover data encryption solely is not allowed.[8]

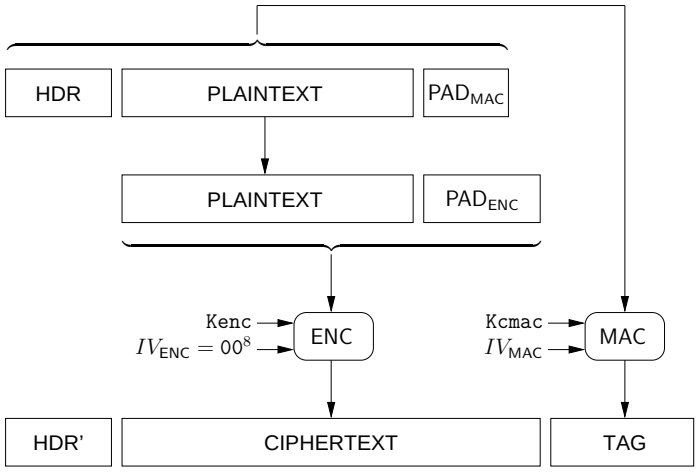


Figure 1: Encryption and MAC computation of a command data with SCP02

Data encryption is done with 3DES in CBC mode with a null IV [ISO17]. Prior to encryption the plaintext is (always) padded with a fixed string of bytes according to the method described in [ISO11]: a byte equal to 80 is appended to the plaintext, then as

[6] For simplicity, we will use the term "server" to refer to the party involved in the communication with the card.

[7] Since the operations done during the key exchange phase are not significant for the remainder of this paper, we skip the corresponding description. The interested reader is referred to the SCP02 specification [Glo18].

[8] In the remainder of the paper we consider that the commands are encrypted and MAC-protected.

many null bytes as necessary (possibly none) are added in order to get a string which length is a multiple of a DES block [ISO17].

The 8-byte MAC tag is computed on the command header, the plaintext, and a padding data.[9] The ISO 9797-1 MAC algorithm 3, also known as "retail MAC", is used [ISO11]. In addition, an IV is involved in the MAC computation. The value of the first IV is usually 00^8. The IV used for the next command is equal to the MAC tag computed on the previous command. The ciphertext and the MAC tag become then the data field of the server's command.

Figure 1 depicts the data encryption and MAC computation of a command.

4 Description of the regular padding oracle attack

The padding oracle attack is based on the fact that a device behaves differently depending on the correctness of the (encryption) padding data. From that differentiated behaviour, the attacker tries to get some information (e.g., some bits or bytes of plaintext). That difference may be based on the nature of the response (presence or absence of the response, type: e.g., "regular" or error message, value, etc.) or on the duration of the operations performed (or not) by the device. Regarding the symmetric-key encryption case, the whole decryption procedure usually includes (among other possible operations) the MAC verification, and the padding extraction and verification. It is commonly recommended to provide data authenticity and confidentiality by applying the so-called Encrypt-then-MAC (EtM) paradigm [Kra01, BN08].[10] Nonetheless some cryptographic mechanisms apply other methods (e.g., MAC-then-Encrypt in TLS 1.2 [DR08], Encrypt-and-MAC (E&M) in SSH [YL06c, YL06a, YL06d, YL06b]).

Therefore, if a padding data is used during the encryption process, and the padding data must be, during the decryption procedure, verified *prior* to the MAC computation, then it may be possible to perform an attack aiming at retrieving sensitive data. If one follows the MtE or the E&M method (as in SCP02), the whole decryption procedure involves (usually) three main steps:

1. the evaluation of the decryption function on the encrypted data,
2. the extraction and verification of the padding data,
3. the computation of the MAC tag on the remaining decrypted data.

Once the ciphertext is decrypted, either the padding data is valid and can be removed, and the MAC computation can be done, or the padding data is invalid and the MAC tag cannot be computed (at least on the genuine data).

Let us illustrate the attack with an example. For the sake of clarity, we use the specifics of SCP02, namely the encryption function (and the corresponding block size), and the padding scheme. Let C be the last encrypted block carried in a protected command, and let V be the block used as IV during the encryption operation that yields C (V denotes either the null IV if the command carries one encrypted block only, or the previous encrypted block if the command carries two or more encrypted blocks). Let $b_0|\cdots|b_5$ be the plaintext data corresponding to C. In SCP02, the encryption is done with 3DES in CBC mode. Since the plaintext length is less than 8 bytes a padding data is appended, and this yields $B = b_0|\cdots|b_5|80|00$. The encryption process outputs

$$\begin{aligned} C &= \mathsf{ENC}(V \oplus B) \\ &= \mathsf{ENC}((v_0 \oplus b_0)|\cdots|(v_5 \oplus b_5)|(v_6 \oplus 80)|(v_7 \oplus 00)) \end{aligned}$$

[9] The genuine header HDR can be retrieved from the header HDR' of the encrypted command.

[10] Note that some encryption modes (e.g., [Dwo04]), coupled with a security proof [Jon03], correspond to the MAC-then-Encrypt (MtE) paradigm.

Conversely the decryption operation outputs

$$\begin{aligned}\mathsf{ENC}^{-1}(C) \oplus V &= (v_0 \oplus b_0)|\cdots|(v_5 \oplus b_5)|(v_6 \oplus \mathtt{80})|v_7 \\ &\quad \oplus \\ &\quad v_0|\cdots|v_7 \\ &= b_0|\cdots|b_5|\mathtt{80}|\mathtt{00} \\ &= B\end{aligned}$$

Let us consider an adversary who replaces V with $\tilde{V}$. If the block $\tilde{V}$ is randomly generated, likely the decryption operation does not yield a valid padding data. If the block $\tilde{V}$ is carefully chosen by replacing the last two bytes of V $v_5|v_6$ with $(v_5 \oplus g \oplus \mathtt{80})|(v_6 \oplus \mathtt{80})$, where g is some byte, then the decryption yields the following result

$$\begin{aligned}\mathsf{ENC}^{-1}(C) \oplus \tilde{V} &= (v_0 \oplus b_0)|\cdots|(v_5 \oplus b_5)|(v_6 \oplus \mathtt{80})|v_7 \\ &\quad \oplus \\ &\quad v_0|\cdots|v_4|(v_5 \oplus g \oplus \mathtt{80})|(v_6 \oplus \mathtt{80})|v_7 \\ &= b_0|\cdots|b_4|(b_5 \oplus g \oplus \mathtt{80})|\mathtt{00}|\mathtt{00}\end{aligned}$$

The decryption operation is correct if and only if the padding data is valid. In our example, this depends on the value $b_5 \oplus g \oplus \mathtt{80}$:

- If $b_5 \oplus g \oplus \mathtt{80} = \mathtt{80}$, then the padding data is valid.
- If $b_5 \oplus g \oplus \mathtt{80} \neq \mathtt{80}$, then the padding data is (likely) invalid.[11]

Obviously $b_5 \oplus g \oplus \mathtt{80} = \mathtt{80}$ if and only if $b_5 = g$. In other words, the result of the decryption operation (namely, the padding verification) reveals the value of the plaintext byte b_5: knowing if the padding data is valid allows getting b_5.

Iterating that process with different blocks $\tilde{V}$ produced by replacing successively $v_i|v_{i+1}$ with $(v_i \oplus g \oplus \mathtt{80})|(v_{i+1} \oplus b_{i+1})$, where b_{i+1} is the plaintext byte found during the previous step, allows retrieving the plaintext byte b_i. For each byte b_i the attacker wants to retrieve, a new ciphertext $\tilde{V}\|C$ is built with different g values, and each resulting ciphertext is sent to an oracle $\mathcal{O}$. The oracle returns 1 if the padding data is valid, 0 otherwise. The response from the oracle eventually reveals if the attacker's guess g is correct or not. This procedure eventually provides all the $n = 6$ plaintext bytes $b_0, \ldots, b_5$. As we can see, the attack leans on the malleability of the CBC mode, and uses the block V as a pivot in order to get bytes encrypted within C.

The overall attack is described by Algorithm 1 (we assume without loss of generality that the targeted block C includes at least one byte of padding, i.e., 80).

If the plaintext to retrieve corresponds to more than two blocks B_0, ..., B_{k-1}, $k > 2$ (which encryption yields the blocks C_0, ..., C_{k-1}), the attacker applies Algorithm 1 by using one after the other each pair of encrypted blocks (C_{k-2}, C_{k-1}), (C_{k-3}, C_{k-2}), ..., (C_0, C_1), (C_{-1}, C_0) where C_{-1} is the CBC IV (equal to $\mathtt{00}^8$ in SCP02). In the remainder of this paper we assume that the ciphertext is made of two blocks V, C.

Note that the length of the padding is helpful since knowing this value shortens the duration of the attack. Yet it is not necessary since the attack can retrieve the padding data as any other unknown plaintext byte. If unknown, the padding length can be found using the dichotomous algorithm proposed by Black and Urtubia [BU02] (which complexity is $\log_2(d)$ where d is the block size), or a linear search by testing all bytes starting from the rightmost one until the decryption of the modified block yields a valid padding (this method finds the length in $\min(d, h+1)$ steps where h is the padding length).

[11] If $b_5 \oplus g \oplus \mathtt{80} = \mathtt{00}$ this may also yield a valid padding (if the preceding decrypted bytes are equal to $\mathtt{80}\ \mathtt{00}^j$). But this (rare) case is easy to manage.

Algorithm 1: Regular padding oracle attack

FindBlock(V, C)
  for $i = n - 1$ down to 0
    $b_i \leftarrow$ **FindByte**(b_{i+1}) // $b_n = \texttt{80}$
  end for
  return $b_0 \cdots b_{n-1}$
FindByte(b)
  for $g = \texttt{00}$ to $\texttt{FF}$
    $\tilde{V} \leftarrow$ in V replace $v_i|v_{i+1}$ with $(v_i \oplus g \oplus \texttt{80})|(v_{i+1} \oplus b)$
    send $\tilde{V}\|C$ to the oracle $\mathcal{O}$
    $r \leftarrow \mathcal{O}(\tilde{V}\|C)$
    if $r = 1$
      return g
    end if
  end for

5 Padding oracle attack applied to SCP02

5.1 Timing channel

As soon as the valid ciphertext $V\|C$ is changed into $\tilde{V}\|C$, the MAC verification yields an error, the padding data being valid or not, because the ciphertext is modified. In SCP02 this ends with the smart card outputting an error code. Furthermore, among the smart cards we have tested, an error code is *always* sent, and the *same* error code is provided whatever the validity of the padding data. Therefore we have used the time spent by the smart card during the whole decryption procedure to build the oracle $\mathcal{O}$.

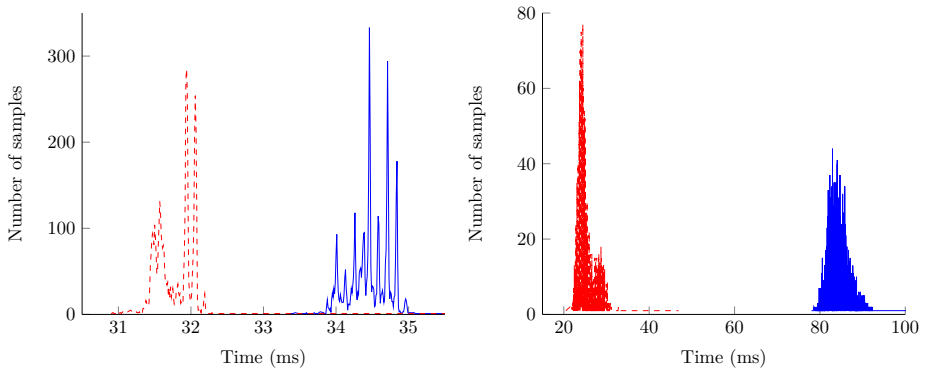
The experiments we have done show that the card's response time (which reflects the card's computation time) when the padding data is valid is higher compared to the case when the padding data is invalid. This suggests that the MAC tag is not verified in the latter case. Depending on the smart card, the timing difference ranges from quite low to unexpectedly high. For instance, for Card B, the timing difference when the padding data is valid (15.60 ms) and invalid (14.83 ms) is less than 1 ms, while for Card C, the timing difference between both cases (84.34 ms vs. 25.17 ms) is higher than 59 ms.[12]

The experiments we have made with each card show that the two distributions corresponding to the response time when the padding is valid (D_R) and when it is invalid (D_W) are clearly distinguishable and almost disjointed. Figure 2 illustrates the results corresponding to four of the attacked smart cards.[13]

Let t_{min} be a lower bound of the values corresponding to D_R. If a response time is lower than t_{min} it is highly likely a wrong guess (since we never observed that a correct guess corresponds to a response time lower than t_{min}). Hence, in such a case, one attempt only allows discarding an incorrect value. On the other hand, if a response time is higher than this threshold t_{min}, it is rather likely a correct guess (since we observed only a very few number of values corresponding to D_W that are higher than this threshold). Yet it is also possible that it corresponds to a wrong guess with an unexpected long response time. Therefore, in order to detect a right guess, several trials may be necessary. As we will see, this heuristic allows having a success probability close to 1. Moreover, in order to increase the discrepancy of the response time when the padding is valid and when it is invalid, we prepend extra blocks $R_0, \ldots, R_{m-1}$ to the ciphertext, following the same

[12] The figures provided correspond to the expected values regarding both padding possibilities, and using a command that carries two encrypted blocks and the 8-byte MAC tag.

[13] For each configuration (valid and invalid padding), 5000 encrypted commands have been sent to the smart card.

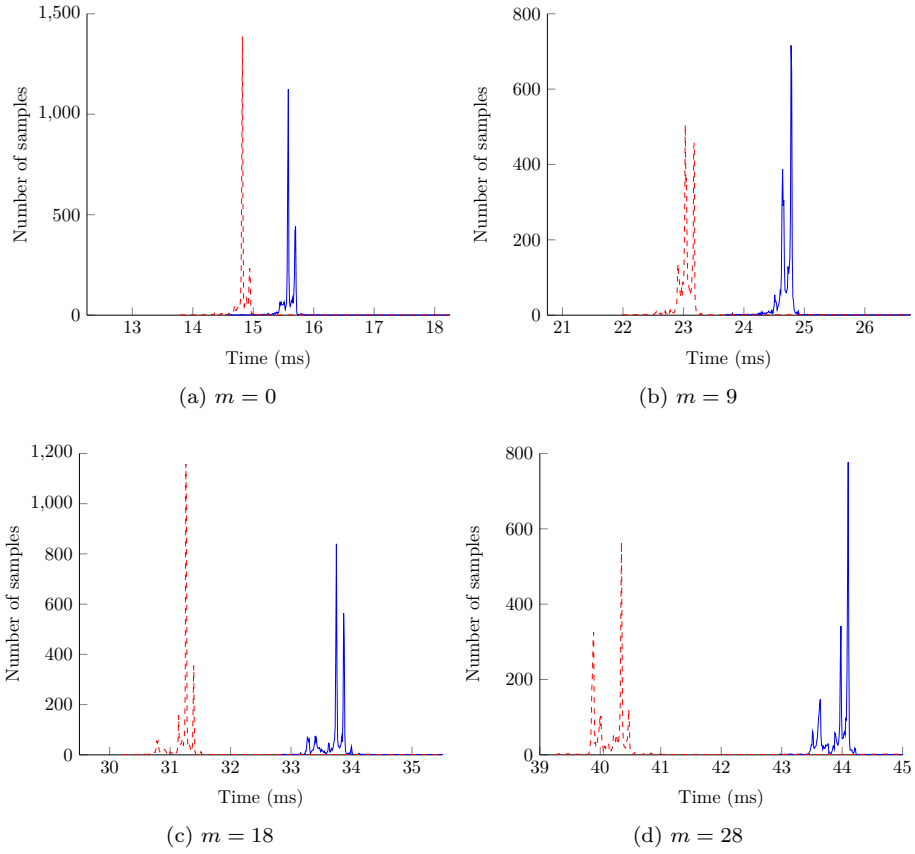


(a) Card F, $m = 28$ (b) Card C, $m = 0$

(c) Card E, $m = 0$ (d) Card J, $m = 28$

Figure 2: Distribution of the number of cases when the padding data is valid (D_R, continuous blue line) and when it is invalid (D_W, dashed red line) with respect to the response time. The command's data field carries $m+2$ encrypted blocks (including padding data), and the 8-byte MAC tag.

technique as [CHVV03], so that the MAC verification (when it is actually done) involves also these additional blocks. That is, instead of $V\|C$, the smart card receives as the encrypted data the string $R_0\|\cdots\|R_{m-1}\|\tilde{V}\|C$. Indeed, during the decryption operation, the DES function (or its inverse) is called $3(q+1)$ times, where q is the number of plaintext blocks. During the MAC verification, the same function is called $q+3$ times.[14] Therefore the timing difference separating a valid and an invalid padding is $O(q)$. As an illustration, Figure 3 depicts the distributions D_W and D_R corresponding to Card B for several values m.[15] The difference between the mean values corresponding to D_R and D_W is 4.6 times higher when $m = 28$ (3.7 ms) compared to $m = 0$ (0.8 ms).



(a) $m = 0$ (b) $m = 9$

(c) $m = 18$ (d) $m = 28$

Figure 3: D_R (continuous blue line) and D_W (dashed red line) corresponding to Card B with different values m.

5.2 Leveraging the timing channel

Let K_R be the number of attempts necessary to detect a right guess, and K_W the number of attempts necessary to discard a wrong guess. Let ϵ_+ be the probability of a bad decision when the distribution is actually D_W, and ϵ_- the probability of a bad decision when the distribution is D_R. Let p_i be the probability to find the byte b_i, $0 \leq i < n$. If we assume that all bytes are independent, then the probability to find the n bytes is $p = p_0 \times \ldots \times p_{n-1}$. Moreover if the bytes are uniformly drawn at random from a set of size ℓ, each byte has

[14] Optionally the IV used during the MAC computation is encrypted. In such a case the DES function is called $q+4$ times during that operation.

[15] Note that for each graph, the abscissa axis has the same 6-ms amplitude.

the same probability to be correctly found. Therefore $p = p_0^n$. Each byte can take any of the ℓ possible values. A guess g is equal to a byte b_i if and only if

- the choice for g is correct (which probability is $\frac{1}{\ell}$),
- incorrect values for g are discarded (which probability is $(1-\epsilon_+)^j$ where j is the number of incorrect values tried before the correct one),
- and the correct value for g is detected (which probability is $1-\epsilon_-$).

Hence

$$\begin{aligned} p_0 &= \sum_{j=0}^{\ell-1} \frac{1}{\ell}(1-\epsilon_+)^j(1-\epsilon_-) \\ &= \frac{1-\epsilon_-}{\ell} \times \frac{1-(1-\epsilon_+)^\ell}{\epsilon_+} \\ &\simeq (1-\epsilon_-)(1-\epsilon_+)^{\frac{\ell-1}{2}} \end{aligned}$$

if $\epsilon_+ \ll \frac{1}{\ell}$. Therefore

$$p \simeq (1-\epsilon_-)^n(1-\epsilon_+)^{n\frac{\ell-1}{2}}$$

Let t be the response time corresponding to a trial with some value g. Let us assume that all trials are independent. Let τ_+ and τ_- be respectively $\Pr[t > t_{min}$, when the distribution is $D_W]$ and $\Pr[t \leq t_{min}$, when the distribution is $D_R]$. Following our heuristic, we discard a guess g as soon as the corresponding response time t is lower than t_{min} (i.e., $K_W = 1$). Therefore the event corresponding to ϵ_+ occurs when the distribution is D_W, if $t > t_{min}$ K_R successive times. Hence

$$\epsilon_+ = \tau_+^{K_R}$$

The event corresponding to ϵ_- occurs when the distribution is D_R, if $t > t_{min}$ at most $K_R - 1$ times, and $t \leq t_{min}$ the subsequent trial. Therefore

$$\epsilon_- = \sum_{j=0}^{K_R-1} (1-\tau_-)^j\tau_- = 1-(1-\tau_-)^{K_R}$$

Hence, we have that

$$p \simeq (1-\tau_-)^{n\cdot K_R}\left(1-\tau_+^{K_R}\right)^{n\frac{\ell-1}{2}}$$

Moreover t_{min} is defined such as $\tau_- = 0$. Therefore, this simplifies into

$$p \simeq \left(1-\tau_+^{K_R}\right)^{n\frac{\ell-1}{2}}$$

If each byte b_i is uniformly drawn at random among ℓ possible values, the average number of values g to be tested before the right one is found is $\frac{\ell+1}{2}$.

The average number of trials to find one byte is $\frac{\ell-1}{2}K_W + K_R$. The overall complexity is then $Z = n \times \left(\frac{\ell-1}{2}K_W + K_R\right)$ to retrieve a n-byte string. Since the burden in the complexity lies on the number of wrong attempts, slightly increasing K_R allows increasing the overall probability of success p while it marginally increases the complexity Z. Indeed, if $\tau_+ < 1$, p is an increasing function of K_R.

The attack is then the following. When looking for a byte b_i, for each possible value g, a modified ciphertext $\tilde{V}\|C$ is sent to the targeted smart card. The different response times are collected until a decision can be taken (which means in practice $K_W \simeq 1$ attempt to

discard a wrong guess and $K_R \in \{1, 2, 3\}$ attempts to detect a right guess). If the decision is that the guess is correct, the trials are stopped and the attack continues with the byte b_{i-1}. Otherwise, another guess value g is tested.

Algorithm 2 describes this timing attack (we assume without loss of generality that C includes at least one byte of padding, i.e., `80`). For a given value g, the **Stop** procedure returns `true` as soon as a response time is lower than t_{min} or if the number of successive response times higher than t_{min} reaches K_R. The **Correct** procedure returns `true` if the number of successive response times higher than t_{min} is equal to K_R.

As soon as a cryptographic error occurs on the smart card side (e.g., the smart card

Algorithm 2: Padding oracle attack based on the card response time

```
FindBlock
  for i = n − 1 down to 0
    b_i ← FindByte(b_{i+1}, ..., b_n)      // b_n = 80
  end for
  return b_0 ··· b_{n−1}
FindByte(b_{i+1}, ..., b_n)
  for g = 00 to FF
    j = 0
    do
      get a new ciphertext V‖C
      Ṽ ← in V replace v_i|v_{i+1}|···|v_n
            with (v_i ⊕ g ⊕ 80)|(v_{i+1} ⊕ b_{i+1})|···|(v_n ⊕ b_n)
      send R_0‖···‖R_{m−1}‖Ṽ‖C as encrypted data to the smart card
      t_j ← response time
      j = j + 1
    until Stop(t_0, ..., t_{k−1}) = true
    if Correct(t_0, ..., t_{k−1}) = true
      return g
    end if
  end for
```

receives a message with an invalid MAC tag, which happens when the attacker changes $V\|C$ into $\tilde{V}\|C$), the session is stopped. Therefore a new session has to be started in order to perform each trial. Hence a new ciphertext $V\|C$ (corresponding to the same plaintext that the attack aims at retrieving) must be obtained. Since a new ciphertext is used for each attempt, we have to take into account the bytes b_{i+1}, ..., b_n already found, and change V accordingly.

5.3 Discussion

The attack we have described is fully successful in our experimental setting. The necessary conditions in order to succeed are the following ones:

1. The attacker sits between the remote server and the card at a point where she can directly eavesdrop on SCP02 encrypted commands and send modified commands to the card.
2. The attacker is able to discriminate response times corresponding to a valid and an invalid padding.
3. The remote server repeatedly sets up a (new) secure channel with the card.
4. The same secret information is sent through each such secure channel.

5. The secret information is sent at a predictable position.

The attack can be tried in the following real use-case of SCP02: the upload of an applet into an UICC that is plugged into a smartphone. In order to send the applet to the UICC, a SCP02 channel is established between a remote server and the UICC, through the intermediary of an application on the smartphone. In addition, another secure channel is established between the remote server and the application on the smartphone. That is, the SCP02 commands that carry the applet are embedded into this second secure channel (e.g., TLS or SCP80). Once the data are received by the application on the smartphone, they are decrypted (w.r.t. TLS or SCP80), and the output (i.e., the encrypted SCP02 commands) is sent to the UICC. Therefore the applet is protected only by the means of SCP02 between the application and the UICC. A command such as `STORE DATA` can be used to upload such an applet, which may carry secret data (e.g., a symmetric key used to encrypt data, or to authenticate the user w.r.t. the service provided by the applet). A malicious application or a Trojan on the smartphone can apply the attack in order to break the SCP02 channel, and to retrieve these sensitive data. It behaves as a local man-in-the-middle attacker between the application on the smartphone and the smart card (see Figure 4).

The attacker can proceed as follows. First it succeeds in getting the victim download a malicious application (embedded in an apparently inoffensive application deployed on a popular store) into his Android smartphone. Then the malicious application can use a vulnerability lying in the legitimate application in order to escalate privileges, and to get access to the application memory space as described by Davi, Dmitrienko, Sadeghi, and Winandy [DDSW11]. The latter assumes that a flaw in the legitimate application is exploitable. Alternatively the attacker can use a Trojan embedded in a popular application, or in a game (such as Trojans Dvmap or Tordow in Pokemon Go, Telegram). Once into the Android smartphone, the Trojan may escalate privileges to gain root access [Kiv], or inject a malicious code into system libraries, which will then be executed with system rights [Unu]. To that point, the Trojan is able to get access to the memory space of the legitimate application. Hence, it can read the genuine SCP02 commands (carrying the symmetric key), and modify it. The crafted command is then sent to the UICC, completing the process initiated by the legitimate application.

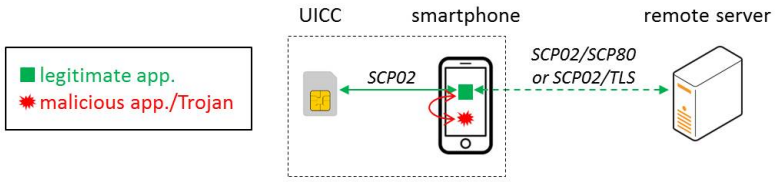


Figure 4: Padding oracle attack targeting an UICC.

Let us assume that encrypting the applet with SCP02 yields k blocks $C_0, \ldots, C_{k-1}$ (possibly transmitted to the card through several successive `STORE DATA` commands), and that a 16-byte symmetric key lies in the blocks C_i, C_{i+1}, $i + 1 \leq k - 1$. The number of SCP02 sessions a card can establish is bounded (to 2^{15}) as per specification. Hence the number of trials the attacker can make is also limited. Therefore *if* (and only if) the attacker needs to decrypt the whole encrypted applet, it may happen that the key be out of reach. However, if the attacker knows the position of the key within the encrypted data (condition 5), she can directly target the corresponding encrypted blocks (whatever the size of the encrypted applet). In such a case, the attacker uses first the blocks C_i,

C_{i+1} as described in Section 4 and by Algorithm 2. The block C_i is changed into $\tilde{C}_i$ in order to incorporate the attacker's guess g, and the string $\tilde{C}_i \| C_{i+1}$ is placed into a SCP02 command data field (with a trailing 8-byte arbitrary MAC tag, and an optional leading string $R_0 \| \cdots \| R_{m-1}$). This fake SCP02 command is then sent to the UICC by the malicious application. Based on the time elapsed until the UICC sends a response, the attacker knows if her guess is correct or not. The attacker uses the blocks C_i, C_{i+1} to retrieve the 8 bytes of the symmetric key encrypted as C_{i+1}, and the blocks C_{i-1}, C_i in order to get the 8 bytes of the symmetric key encrypted as C_i.

The attacker expects a clean and stable response time (condition 2). Yet it may happen that this time value be influenced by surrounding activities. In such a case more sophisticated statistical tools may be used by the attacker (e.g., see [CHVV03, AP16]).

Another necessary condition is that the same secret be repeatedly sent through the secure channel (condition 4). The nominal behaviour of the underlying application protocol could not be as expected by the attacker. Yet this condition could be fulfilled if, upon reception of the cryptographically invalid command (modified by the attacker), the underlying application within the card triggers a message asking the server for a new delivery of the same message (which contains the secret). Also, upon reception of an error code, the server could decide to send again the command carrying the secret.[16] This error code may be sent either by the card itself (because the command it received has been modified by the attacker), or by the attacker.[17] Then the server may keep sending the same command[18] until it is acknowledged by the card ($\texttt{SW} = \texttt{9000}$).

The complexity per byte is $\frac{\ell-1}{2} K_W + K_R$. The figures provided in Section 5.4 assume that each byte is uniformly distributed over $\{\texttt{00}, \ldots, \texttt{FF}\}$. However the alphabet size the secret bytes belong to can be lower than $\ell = 256$ (e.g., the secret is some sort of PIN code or password). That would decrease the overall complexity. Moreover it may also be possible for the attacker to retrieve only a substring of the secret and then complete the attack through an exhaustive search or a dictionary attack, if the attacker is able to test the remaining values.

Furthermore, the attack is made easier if the targeted card does not close the secure channel when it receives an invalid command (the one modified by the attacker). In such a case the attacker needs to eavesdrop on the encrypted command (that carries the secret data) once only. Then she can reuse the same encrypted version of the secret to perform the successive changes (with the different values g), and send the modified commands to the card. This would remove the conditions 3 and 4 listed above, necessary to the attack. Yet a card behaving so would diverge from the SCP02 specification. We stress that none of the smart cards we have tested behave so.

5.4 Experimental results of the complete attack

We have developed the attack with the Java OPAL library [SSD] which implements several protocols for smart cards (including SCP02). The communication with the smart card and the reader is managed with the `javax.smartcardio` package. The experiments have been done on two laptops HP EliteBook and Dell Latitude E6430 running on Windows 7. Four card readers have been used: the inner laptop readers (Broadcom Corp Contacted SmartCard, Alcor Micro USB Smart Card Reader), a contact reader (Omnikey 5321 Smart Card Reader USB), and a contactless reader (SpringCard Prox'N'Roll).

Our program simulates both the legitimate server and the attacker. The kinematic is the following. First an n-byte string is pseudo-randomly generated (the secret value

[16] Note that, if the remote server sends again this command only or the whole applet, it is very likely that the symmetric key lie at the same position within the command and the applet. Hence condition 5 remains fulfilled.

[17] By default, in SCP02, the responses sent by the card are not encrypted nor protected with a MAC tag.

[18] If not everlastingly, at least many times, which allows the attacker to retrieve a substring of the secret.

the attacker has to retrieve). Then the server procedure sets up an SCP02 channel with the smart card and computes shared session keys. Through the channel, an encrypted command carrying the n-byte secret is sent to the card. A copy of that encrypted command is given to the attacker procedure, which modifies it, sends it to the card and measures the response time. The measured time is equal to the elapsed time between the moment the command is sent and a response is received from the smart card (a single Java procedure, `transmit`, handles the command to send and the response to receive). Since the channel is closed by the smart card (because the modified command sent by the attacker procedure is cryptographically invalid), the server procedure sets up a new secure channel and sends again the same n-byte secret. This new encrypted version of the secret is given to the attacker procedure, and so forth.

We have made experiments with 10 smart cards produced by six different card manufacturers (see Table 1). Prior to the attack, we have made trials in order to get the distributions D_W and D_R for each smart card. This is straightforward since we own the card keys. D_R corresponds to a valid padding (and a wrong MAC tag), and D_W corresponds to an invalid padding (and a wrong MAC as well). We were able to use an invalid padding during the encryption procedure. Yet in a real context, the adversary that has access to an SCP02 encryption oracle is able to get a distribution very close to D_W by changing the trailing bytes of the encrypted data. With high probability, changing these encrypted bytes yields an invalid padding during the decryption procedure. Alternatively, the attacker can perform offline tests with a specimen of the targeted card.

As the smart card was the target, we have used commands and not responses to launch the attack. We have used different types of command (`PUT KEY`, `STORE DATA`, `GET DATA`, `GET STATUS`), including commands that are not supposed to carry an application payload.[19] Yet our purpose is to show that it is indeed possible to retrieve plaintext bytes whatever the command used, be it a command not supposed to carry data besides possible flags (e.g., `GET STATUS`, `GET DATA`), or commands which are designed to transmit sensitive data to the smart card (e.g., `PUT KEY`, `STORE DATA`). We observe that the payload carried in a `PUT KEY` command is encrypted with 3DES in CBC mode, as any command payload in secure mode. However, prior to be transmitted through the secure channel, the "plaintext data" is first encrypted (with 3DES and another session key than the one used to provide data confidentiality throughout the secure channel). We stress that we do not break this inner encryption layer. Yet, the padding oracle attack allows retrieving all the (encrypted) bytes sent through the secure channel even when this "sensitive" command is used.

The maximum data size for a command is 255 bytes. Without the MAC tag, there remain 247 bytes = 30 DES blocks + 7 bytes. The attack involves two useful blocks V and C. This leaves at most 28 extra blocks R_i in order to amplify the timing discrepancy. In practice we have used either 0 or 28 random blocks depending on the targeted smart card.

In the best case (i.e., $K_W = K_R = 1$), the average complexity to retrieve n bytes of plaintext is $Z = n \times \frac{\ell+1}{2}$. The number of available SCP02 sessions is at most 2^{15} (i.e., the maximum value of a keyset' sequence counter). Since a new session has to be established for each trial, the maximum number of plaintext bytes that can be retrieved in the best case is bounded by $2 \times \frac{2^{15}}{\ell+1} < 2^8$, if $\ell = 256$.

Table 1 summarises our results. The figures provided are the mean corresponding to the 300 tests roughly that we have performed with each smart card. For each card, we provide the average complexity Z to retrieve $n = 16$ bytes, the average complexity per byte (Z/n), the number of trials to discard a wrong attempt (K_W), the number of trials to detect a right attempt (K_R), the expected response time in case of an invalid padding (μ_W) and valid padding (μ_R), the threshold t_{min} used to detect a correct guess, the probability τ_+ that an invalid padding yields a high response time, and the number m

[19] Note however that such commands, when protected, carry the encrypted padding and the MAC tag.

Table 1: Experimental results with $n = 16$ (pseudo-random) bytes of plaintext. (M: manufacturer, C: card)[20]

M	C	μ_W (ms)	μ_R (ms)	t_{min} (ms)	m	τ_+ (%)	K_W	K_R	Z	Z/n
1	A	39.60	42.59	41.00	28	0.16	1	3	2055.71	128.48
	B	40.19	43.94	42.00	28	0.44	1	3	2077.78	129.86
2	C	25.17	84.34	75.00	0	0.00	1	2	2043.95	127.75
	D	26.64	34.36	32.00	0	0.00	1	2	2066.54	129.16
3	E	15.61	25.65	23.00	0	0.00	1	2	2134.03	133.38
4	F	31.81	34.48	33.00	28	0.48	1	3	2109.71	131.86
	G	15.64	18.53	17.00	0	0.28	1	3	2103.62	131.48
5	H	25.18	84.86	72.00	0	0.00	1	2	2048.34	128.02
6	I	25.90	35.85	32.00	0	0.06	1	3	2108.60	131.79
	J	14.32	19.92	17.50	0	0.10	1	2	2094.85	130.93

of additional blocks R_i used to increase the computation time discrepancy. Note that the minimum for K_R is 2 in order to be able to correctly find a plaintext byte equal to `80` (in such a case, at least one additional attempt with the same value g is made). But in fact, for some cards (Card C, Card D, Card H), $K_R = 1$ is enough to detect a right guess.

As we can see, the complexity per byte Z/n is almost optimal (close to 128.5). Moreover the heuristic we use is actually valid since the probability τ_+ that a wrong guess yields a high response time (i.e., above t_{min}) is low.

The duration of the attack to retrieve $n = 16$ bytes ranges roughly from 160 s (Card A, Card B) up to 680 s (Card C).

Estimating the number of smart cards affected by this vulnerability is not easy. However cards likely implementing SCP02 are produced in their billions every year [SIM16, GSM16]. Therefore, the number of impacted smart cards is potentially high.

6 Overview of existing countermeasures

Several countermeasures aiming at mitigating a padding oracle attack have been proposed formerly. A simple fix proposed by Vaudenay [Vau02] is to pad the plaintext first, and then to compute the MAC on the padded data. During the whole decryption procedure, the MAC must be verified first, and the padding data removed after.

As observed by Black and Urtubia [BU02], and proved by Paterson and Watson [PW08], slightly changing a *byte*-oriented padding of the form `80` $\texttt{00}^i$ into a *bit*-oriented padding of the form $\texttt{10}\cdots\texttt{0}$ makes the padding oracle (almost) vanish. Indeed, any decrypted block is correctly padded with respect to a padding of the latter form unless the decrypted block contains no bit equal to 1 (i.e., the block is equal to $\texttt{00}^d$, where d is the block byte length).[21] Therefore the padding oracle is not usable any more if each byte of the plaintext block is uniformly drawn at random, because the attacker cannot look for each byte independently but has to enumerate all possible block values. Yet, as noticed by Black and Urtubia, a dictionary attack may still be conceivable if the attacker looks for a secret value belonging

[20] For confidentiality purpose, the manufacturer names and the card identifiers are not provided in this paper.

[21] This padding scheme is recommended for use with CBC mode by ISO [ISO17], with a slight difference: the number of optional 0 bits must be as few as possible. Therefore the padding data is carried in one block at most. This means that any plaintext which *last block* is all-zero is invalid with respect to this padding scheme.

to a reduced size set (if the attacker guesses correctly, the decryption yields $\mathtt{00}^d$, which is the only invalid plaintext).

Another scheme resisting padding oracle attacks and proposed by Black and Urtubia is the following. An arbitrary byte x distinct from the last plaintext byte is picked, and the data is padded with x. The receiver removes all matching trailing bytes until either a distinct byte is found or the empty string is reached. With respect to this padding scheme, any decrypted block is valid. Therefore it seems that the oracle is removed. We observe however that even this scheme may still provide an oracle, depending on the behaviour of the receiver when the decryption outputs an empty string. If this yields a distinct error or a discrepancy in the calculation duration, that could be exploited in order to perform a (dictionary) attack. Let $V\|C$ be the encryption of some secret value B. The attacker tries all possible values B' and changes V into $V' = V \oplus B'$. Then the decryption of $V'\|C$ yields $\mathsf{ENC}^{-1}(C) \oplus V' = (V \oplus B) \oplus (V \oplus B') = B \oplus B'$. If $B \oplus B'$ contains at least two distinct bytes, then there is at least one remaining byte after the padding extraction. If all bytes in $B \oplus B'$ are equal, then no byte remains after the padding removal. If the attacker is able to detect such a case, she knows that $B = B' \oplus (y| \cdots |y)$ where each byte y can take at most 256 values. Therefore this leaves at most 256 candidates for B.

Canvel *et al.* [CHVV03] suggest to make error responses time-invariant by simulating a MAC verification even when there is a padding error. This implies to carefully implement the decryption procedure in order to eliminate all timing channels. Indeed AlFardan *et al.* [AP13] have shown that even a tight channel can be exploited. In addition, the latter authors warn that adding random delays during the decryption procedure may not be sufficient if the delays follow a uniform distribution. This change would merely increase the complexity of the attack.

In turn, Askarov, Zhang, and Myers [AZM10] propose a mitigation scheme that applies to a broad class of computations. With respect to SCP02, this means sending the responses at scheduled times (that is somehow padding the response time to an upper bound). These authors observe that this does not prevent timing leaks, yet it bounds the amount of information provided through this side channel as a function of the elapsed time.

Another fix is to replace the CBC mode with an authenticated encryption algorithm as suggested by Kupser, Mainka, Schwenk, and Somorovsky [KMSS15].

The latter proposal or another construction than E&M may be a suitable choice since Paterson and Watson [PW12] extend the results of Bellare and Namprempre [BN08] to prove that the E&M construction using CBC mode with a padding method as its encryption component is not *generically* secure in the chosen ciphertext setting.[22] They also prove that the EtM construction (known to be secure in general when padding is not considered) is also secure when a padding method is used. In addition, their security models and proofs can be used to select padding schemes provably secure with respect to the strong notion security of indistinguishability of encryptions.

In addition to these countermeasures, in the SCP02 case, one may use the `PUT KEY` command in order to send secret values to the smart card (e.g., symmetric keys). The data field of such a command corresponds to the output of a double encryption process: first with a session key different than the one used to encrypt and MAC the other commands, then with these same secure channel session keys. Therefore, applied to a `PUT KEY` command, the attack breaks the upper encryption layer but not the inner one, and yields the data encrypted under this additional session key but not the genuine plaintext data.

Furthermore, the attack can be mitigated by limiting, on the server side, the number of times the same secret value is sent to the smart card.

[22] Note that Bellare, Kohno, and Namprempre [BKN04] prove that a particular Encode-then-Encrypt-and-MAC construction is secure when a padding method is used. But the encoding function is assumed to be collision resistant (the computation of the authentication tag involves a sequence number unique per message that aims at precluding collisions between two encoded messages).

7 Responsible disclosure

Between October 2017 and April 2018, we have informed GlobalPlatform and the manufacturers of the tested smart cards. While this paper was under submission, GlobalPlatform has deprecated SCP02.

As this paper results from a scientific work, we think that our duty is to describe accurately the experiments we have done, and to provide all the technical specifics so that any researcher can redo the experiments in order to validate or to refute our findings. We also think that providing the explicit card models would have fit in with this process.

8 Conclusion

We have shown that the SCP02 protocol (likely the most used GlobalPlatform symmetric-key based protocol, and implemented in a multitude of smart cards) is subject to a padding oracle attack. This attack allows retrieving plaintext bytes from an encrypted data field. We have provided results obtained in an experimental setting made on 10 smart cards from six different card manufacturers. The attack is fully successful and allows retrieving many bytes of plaintext within minutes. To the best of our knowledge, this is the first successful attack against SCP02.

We have also recap several countermeasures formerly proposed in order to mitigate a padding oracle attack, and suggest additional methods that can be applied in the SCP02 case.

The reasons why the attack is possible are twofolds. Firstly it is due to the specifics of the SCP02 protocol (namely the E&M procedure used to protect a command, and the padding scheme). Secondly the implementation itself provides the timing channel that allows applying the attack. It is unclear to us if the smart card's differentiated behaviour can be explained by the source code or by the constraints of the chip the protocol is implemented in.

These results, based on a technique described in 2002, show that a security scheme (and any product that implements it) has an expiry date, and should periodically be analysed anew in light of the last cryptographic findings. Despite existing countermeasures, and due to other undesirable properties weakening SCP02, we advocate the deprecation of the protocol and its replacement by SCP03.

Acknowledgments

We thank the anonymous reviewers and especially Kerstin Lemke-Rust for their valuable comments, and Jean-Louis Lanet for telling us about the OPAL library. We also thank Hosni Majdoub and several unnamed people for their technical help and insightful comments, and Sébastien Canard for his technical and non-technical support.

References

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

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and*

Privacy, pages 526–540, Berkeley, CA, USA, May 19–22, 2013. IEEE Computer Society Press.

[AP16] Martin R. Albrecht and Kenneth G. Paterson. Lucky microseconds: A timing attack on amazon's s2n implementation of TLS. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 622–643, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany.

[ASS+16] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS Using SSLv2. In *Proceedings of the 25th USENIX Security Symposium*, USENIX Security 2016, pages 689–706, Austin, TX, 2016. USENIX Association.

[AWS] AWS. s2n : an implementation of the TLS/SSL protocols. Available via https://github.com/awslabs/s2n.

[AZM10] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 10: 17th Conference on Computer and Communications Security*, pages 297–307, Chicago, Illinois, USA, October 4–8, 2010. ACM Press.

[BFK+12] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient padding oracle attacks on cryptographic hardware. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 608–625, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[BKN04] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Breaking and Provably Repairing the SSH Authenticated Encryption Scheme: A Case Study of the Encode-then-Encrypt-and-MAC Paradigm. *ACM Trans. Inf. Syst. Secur.*, 7(2):206–241, May 2004.

[Ble98] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Heidelberg, Germany.

[BN08] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21(4):469–491, October 2008.

[BU02] John Black and Hector Urtubia. Side-Channel Attacks on Symmetric Encryption Schemes: The Case for Authenticated Encryption. In *Proceedings of the 11th USENIX Security Symposium*, USENIX Security 2002, pages 327–338, Berkeley, CA, USA, 2002. USENIX Association.

[CHVV03] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password interception in a SSL/TLS channel. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 583–599, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Heidelberg, Germany.

[DA99] T. Dierks and C. Allen. The TLS Protocol – Version 1.0. RFC 2246, January 1999. Available via https://tools.ietf.org/html/rfc2246.

[DDSW11] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In Mike Burmester, Gene Tsudik, Spyros S. Magliveras, and Ivana Ilic, editors, *ISC 2010: 13th International Conference on Information Security*, volume 6531 of *Lecture Notes in Computer Science*, pages 346–360, Boca Raton, FL, USA, October 25–28, 2011. Springer, Heidelberg, Germany.

[DLP+12] Jean Paul Degabriele, Anja Lehmann, Kenneth G. Paterson, Nigel P. Smart, and Mario Strefler. On the joint security of encryption and signature in EMV. In Orr Dunkelman, editor, *Topics in Cryptology – CT-RSA 2012*, volume 7178 of *Lecture Notes in Computer Science*, pages 116–135, San Francisco, CA, USA, February 27 – March 2, 2012. Springer, Heidelberg, Germany.

[DP07] Jean Paul Degabriele and Kenneth G. Paterson. Attacking the IPsec standards in encryption-only configurations. In *2007 IEEE Symposium on Security and Privacy*, pages 335–349, Oakland, CA, USA, May 20–23, 2007. IEEE Computer Society Press.

[DP10] Jean Paul Degabriele and Kenneth G. Paterson. On the (in)security of IPsec in MAC-then-encrypt configurations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 10: 17th Conference on Computer and Communications Security*, pages 493–504, Chicago, Illinois, USA, October 4–8, 2010. ACM Press.

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol – Version 1.2. RFC 5246, August 2008. Available via https://tools.ietf.org/html/rfc5246.

[DR11] Thai Duong and Juliano Rizzo. Cryptography in the web: The case of cryptographic design flaws in asp.net. In *2011 IEEE Symposium on Security and Privacy*, pages 481–489, Berkeley, CA, USA, May 22–25, 2011. IEEE Computer Society Press.

[Dwo04] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, May 2004. NIST Special Publication 800-38C. Available via http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38c.pdf.

[ETS17] ETSI. Smart Cards; Secure packet structure for UICC based applications (release 12), 2017. TS 102.225, v12.1.0 (2014-10).

[Glo14] GlobalPlatform. GlobalPlatform Technology – Secure Channel Protocol '03' – Card Specification v2.2 – Amendment D, July 2014. version 1.1.1, GPC_SPE_014. Available via http://www.globalplatform.org/specificationscard.asp.

[Glo18] GlobalPlatform. GlobalPlatform – Card Specification – Version 2.3.1, March 2018. Reference GPC_SPE_034. Available via https://www.globalplatform.org/specificationscard.asp.

[GSM16] GSMA. The Mobile Economy, 2016. Available via https://www.gsma.com/mobileeconomy/archive/GSMA_ME_2016.pdf.

[ISO11] ISO/IEC JTC 1/SC 27. ISO/IEC 9797-1:2011 – Information technology – Security techniques – Message Authentication Codes (MACs) – Part 1: Mechanisms using a block cipher, 2011.

[ISO17] ISO/IEC JTC 1/SC 27. ISO/IEC 10116:2017 – Information technology – Security techniques – Modes of operation for an n-bit block cipher, 2017.

[Jon03] Jakob Jonsson. On the security of CTR + CBC-MAC. In Kaisa Nyberg and Howard M. Heys, editors, *SAC 2002: 9th Annual International Workshop on Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 76–93, St. John's, Newfoundland, Canada, August 15–16, 2003. Springer, Heidelberg, Germany.

[JS11] Tibor Jager and Juraj Somorovsky. How to break XML encryption. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 11: 18th Conference on Computer and Communications Security*, pages 413–422, Chicago, Illinois, USA, October 17–21, 2011. ACM Press.

[JSS12] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. Bleichenbacher's attack strikes again: Breaking PKCS#1 v1.5 in XML encryption. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *ESORICS 2012: 17th European Symposium on Research in Computer Security*, volume 7459 of *Lecture Notes in Computer Science*, pages 752–769, Pisa, Italy, September 10–12, 2012. Springer, Heidelberg, Germany.

[JSS15] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15: 22nd Conference on Computer and Communications Security*, pages 1185–1196, Denver, CO, USA, October 12–16, 2015. ACM Press.

[Ken05] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303, December 2005. Available via https://tools.ietf.org/html/rfc4303.

[Kiv] Anton Kivva. The banker that can steal anything. Available via https://securelist.com/the-banker-that-can-steal-anything/76101/. 20/09/2016.

[KMSS15] Dennis Kupser, Christian Mainka, Jörg Schwenk, and Juraj Somorovsky. How to Break XML Encryption – Automatically. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, WOOT'15, pages 11–11, Berkeley, CA, USA, 2015. USENIX Association.

[KPR03] Vlastimil Klíma, Ondrej Pokorný, and Tomás Rosa. Attacking RSA-based sessions in SSL/TLS. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 426–440, Cologne, Germany, September 8–10, 2003. Springer, Heidelberg, Germany.

[Kra01] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany.

[KS05] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, December 205. Available via https://tools.ietf.org/html/rfc4301.

[Man01] James Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 230–238, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany.

[MDK14] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE Bites: Exploiting The SSL 3.0 Fallback, September 2014. Available via https://www.openssl.org/~bodo/ssl-poodle.pdf.

[PA12] Kenneth G. Paterson and Nadhem J. AlFardan. Plaintext-recovery attacks against datagram TLS. In *ISOC Network and Distributed System Security Symposium – NDSS 2012*, San Diego, CA, USA, February 5–8, 2012. The Internet Society.

[PW08] Kenneth G. Paterson and Gaven J. Watson. Immunising CBC mode against padding oracle attacks: A formal security treatment. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN 08: 6th International Conference on Security in Communication Networks*, volume 5229 of *Lecture Notes in Computer Science*, pages 340–357, Amalfi, Italy, September 10–12, 2008. Springer, Heidelberg, Germany.

[PW12] Kenneth G. Paterson and Gaven J. Watson. *Authenticated-Encryption with Padding: A Formal Security Treatment*, pages 83–107. Springer, Berlin, Heidelberg, 2012.

[PY04] Kenneth G. Paterson and Arnold Yau. Padding oracle attacks on the ISO CBC mode encryption standard. In Tatsuaki Okamoto, editor, *Topics in Cryptology – CT-RSA 2004*, volume 2964 of *Lecture Notes in Computer Science*, pages 305–323, San Francisco, CA, USA, February 23–27, 2004. Springer, Heidelberg, Germany.

[RD10] Juliano Rizzo and Thai Duong. Practical Padding Oracle Attacks. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[RM06] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347, April 2006. Available via https://tools.ietf.org/html/rfc4347.

[Sch15] Stephen Schmidt. Introducing s2n, a new open source TLS implementation, June 2015. Available via https://aws.amazon.com/fr/blogs/security/introducing-s2n-a-new-open-source-tls-implementation.

[SIM16] SIMalliance. SIMalliance Reports 4.7 Billion Global SIM Shipments in 2015. http://simalliance.org/media/press-releases/simalliance-reports-4-7-billion-global-sim-shipments-in-2015/, April 2016.

[SSD] SSD Research Team. OPAL library. XLIM Labs, University of Limoges, France. Available via https://bitbucket.org/ssd/opal.

[ST16] M. Sabt and J. Traoré. Cryptanalysis of GlobalPlatform Secure Channel Protocols. In L. Chen, D. McGrew, and C. Mitchell, editors, *Security Standardisation Research*, volume 10074 of *LNCS*, pages 62–91, Cham, 2016. Springer International Publishing.

[Unu] Roman Unuchek. Dvmap: the first Android malware with code injection. Available via https://securelist.com/dvmap-the-first-android-malware-with-code-injection/78648/. 08/06/2017.

[Vau02] Serge Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS... In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Heidelberg, Germany.

[Wir01] Wireless Application Protocol Forum. Wireless Transport Layer Security – Wireless Application Protocol, 2001. WAP-261-WTLS-20010406-a, version 06-Apr-2001. Available via http://www.openmobilealliance.org/tech/affiliates/wap/wap-261-wtls-20010406-a.pdf.

[YL06a] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Authentication Protocol. RFC 4252, January 2006. Available via https://tools.ietf.org/html/rfc4252.

[YL06b] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Connection Protocol. RFC 4254, January 2006. Available via https://tools.ietf.org/html/rfc4254.

[YL06c] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251, January 2006. Available via https://tools.ietf.org/html/rfc4251.

[YL06d] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, January 2006. Available via https://tools.ietf.org/html/rfc4253.

[YPM05] Arnold K. L. Yau, Kenneth G. Paterson, and Chris J. Mitchell. Padding oracle attacks on CBC-mode encryption with secret and random IVs. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption – FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 299–319, Paris, France, February 21–23, 2005. Springer, Heidelberg, Germany.