An Adaptive-Ciphertext Attack against "I ⊕ C" Block Cipher Modes With an Oracle

Jon Passki
jon@passki.us

Tom Ritter
tritter@isecpartners.com

June 3, 2012

Abstract

Certain block cipher confidentiality modes are susceptible to an adaptive chosen-ciphertext attack against the underlying format of the plaintext. When the application decrypts altered ciphertext and attempts to process the manipulated plaintext, it may disclose information about intermediate values resulting in an oracle. In this paper we describe how to recognize and exploit such an oracle to decrypt ciphertext and control the decryption to result in arbitrary plaintext. We also discuss ways to mitigate and remedy the issue.

1 Introduction

Quoting from [1]:

> Adaptive chosen-ciphertext attacks on cryptographic protocols allow an attacker to decrypt a ciphertext C, getting the plaintext M, by submitting a series of chosen-ciphertexts $C\prime \neq C$ to an oracle which returns information on the decryption. The ciphertexts can be adaptively chosen so that information on previous decryptions is available before the next chosen ciphertext is submitted.

Adaptive chosen-ciphertext attacks against different confidentiality modes are not novel. The CBC confidentiality mode can suffer from a side channel attack against padding verification [2], popularized by [3]. A variant of the Cipher Feedback (CFB) confidentiality mode has been attacked in different encryption mail protocols by [4, 5, 1], and the padding schemes of asymmetric ciphers are another source of such attacks [6, 7]. In some form, they rely on the use of an *oracle* that leaks or communicates information back to attackers. We examine four different confidentiality modes encrypting plaintext that is separated by a delimiter, and the absence or inclusion of that delimiter generates an *oracle* by the application. We call this oracle a *Separator Oracle*.

2 "I ⊕ C" Confidentiality Modes of Block Ciphers

A common form of block cipher construction is to implement decryption such that an XOR is performed with the ciphertext *after* the cipher operation[1] to produce the decrypted plaintext block. We refer to this property as "I ⊕ C", where C represents the ciphertext and I an Intermediate Value. It is featured in the Cipher Block Chaining (CBC), Counter (CTR), Cipher Feedback (CFB), and Output Feedback (OFB) modes of operation and may be present in other less well known modes.

[1] In some of these modes, the cipher (e.g. AES) is run in the forward or encryption direction even during decryption. In others, it is run in the decryption direction. The direction is irrelevant to the attack.

CBC mode encrypts a value derived from the plaintext block XOR-ed with the previous ciphertext block - or the Initialization Vector if it is the first block. This value, after it is encrypted, becomes the ciphertext block. To decrypt, CBC mode decrypts the ciphertext block, resulting in an Intermediate Value. This Intermediate Value is then XOR-ed with either the Initialization Vector or the prior ciphertext block. In many scenarios, the ciphertext is attacker-controlled; in CBC mode, the attacker controlled data is processed in the decryption mode.

Figure 1 shows CBC mode encryption and decryption for blocks of data numbered *j*. In this figure, and the three following, the $I \oplus C$ construction is visible in the decryption equation. Targeting the $I \oplus C$ construction has been mentioned in [8], with specific detail for CBC demonstrated in both academic and practical contexts [8, 9, 10], including recent work on breaking XML encryption in [10] - but it is made difficult because modifications to a ciphertext block propagate to other blocks[2].

Figure 1: CBC Mode

$$C_j = \begin{cases} \text{if } j = 0 & E_k(P_j \oplus IV) \\ \text{else} & E_k(P_j \oplus C_{j-1}) \end{cases}$$

CBC Encryption

$$P_j = \begin{cases} \text{if } j = 0 & E_k(C_j) \oplus IV \\ \text{else} & E_k(C_j) \oplus C_{j-1} \end{cases}$$

CBC Decryption

CTR mode encrypts an incrementing counter to produce an Intermediate Value, which is XOR-ed with the plaintext to produce the ciphertext. The independent count prevents modification propagation. The process is repeated for decryption: the counter is encrypted and the result XOR-ed with the ciphertext to produce the plaintext. CTR mode is often recommended because attacker-controlled data is never processed through the encryption function, reducing the risk of side channel attacks [11]. Figure 2 shows CTR mode encryption and decryption for blocks of data numbered *j*.

Figure 2: CTR Mode

$$C_j = E_k(nonce_j) \oplus P_j$$

CTR Encryption

$$P_j = E_k(nonce_j) \oplus C_j$$

CTR Decryption

OFB Mode repeatedly encrypts an Intermediate Value, which is XOR-ed with the plaintext (or ciphertext) block to produce the ciphertext (or plaintext) block. Like CTR mode, attacker controlled data is never processed through the encryption function and a modification in a ciphertext block is not propagated. Figure 3 shows OFB mode encryption and decryption for blocks of data numbered *j*.

Figure 3: OFB Mode

$$I_j = \begin{cases} \text{if } j = 0 & E_k(IV) \\ \text{else} & E_k(I_{j-1}) \end{cases}$$

$$C_j = I_j \oplus P_j$$

OFB Encryption

$$I_j = \begin{cases} \text{if } j = 0 & E_k(IV) \\ \text{else} & E_k(I_{j-1}) \end{cases}$$

$$P_j = I_j \oplus C_j$$

OFB Decryption

CFB mode encrypts an Initialization Vector to produce an Intermediate Value, which is XOR-ed with the first block of plaintext to produce the first block of ciphertext. Each subsequent block of intermediate values is produced by encrypting the prior block of ciphertext, and the ciphertext is produced by XOR-ing the intermediate block with the plaintext block. In CFB mode, attacker controlled data is processed in the decryption mode. Figure 4 shows CFB mode encryption and decryption for blocks of data numbered *j*. Like CBC mode, a modification to a ciphertext block is propagated. The details and difficulties of performing an attack on CFB mode in whole are presented in Appendix A.

[2]In other literature, this modification propagation is often referred to as 'error propagation' - in our application the modifications are intentional

Figure 4: CFB Mode

$$C_j = \begin{cases} \text{if } j = 0 & E_k(IV) \oplus P_j \\ \text{else} & E_k(C_{j-1}) \oplus P_j \end{cases}$$

CFB Encryption

$$P_j = \begin{cases} \text{if } j = 0 & E_k(IV) \oplus C_j \\ \text{else} & E_k(C_{j-1}) \oplus C_j \end{cases}$$

CFB Decryption

During the decryption phase of each of these block cipher modes, attacker controlled data is XOR-ed against an Intermediate Value to produce the plaintext block. It is possible to produce any desired plaintext block by modifying the ciphertext block correctly. This requires knowing the Intermediate Value produced by the encryption function - but in some cases cryptographic oracles can leak enough information to make it possible to learn the Intermediate Value.

3 Separator Oracle

Assume an application uses a block cipher mode with a construction as described above, for example CTR mode. It encrypts plaintext, sends the ciphertext to users, and does not include a MAC or other integrity check. Users send the ciphertext back to the application, perhaps for session management, which the application decrypts. If a user modifies the ciphertext, the application will decrypt it to an unexpected value. If the application produces error messages based on the structure of the mangled plaintext, information about the plaintext can be derived. Depending on the type of information derived, it may be possible to decrypt the ciphertext to learn the original plaintext or alter the ciphertext to decrypt to a chosen plaintext.

As a practical example, we will consider the case when the plaintext is composed of delimited values, such as `username|timestamp|access-level`. The separator character, the pipe | in this example, will be denoted by P_S. If the application cannot split the delimited plaintext into the appropriate number of values by performing a split operation using P_S, the application responds with a state we will call *SeparatorException*. If the split operation succeeds, the application continues - if the split values are considered invalid for another reason the application may respond with another state we will call *OtherException*. If the application does not raise any exception, we will consider it normal application state.

SeparatorException is a discernible state different from the normal state or *OtherException*. This difference is called a *Separator Oracle*. Applications that disclose an oracle during encryption or decryption have been studied and attacked previously. Although labeled an "Exception", the oracle may manifest as a verbose error message, an HTTP Status Code, or a measurable difference in response time. The Separator Oracle occurs as a result of application code following the cryptographic routine - no changes to a cryptography library can be made to silence this oracle, the change must occur in the application code.

4 Separator Oracle in CTR Mode

We will now present an algorithm for decrypting ciphertext encrypted with CTR mode using a Separator Oracle. This algorithm assumes the underlying plaintext uses a one-byte encoding format, such as ASCII, with a single encoding of each glyph. Algorithms targeting other plaintext encoding formats like ASN.1 or UTF-8, or encodings with multiple representations of the same glyph, need to adjust accordingly.

4.1 Phase 1: Locating The Separator(s)

We are given ciphertext C and plaintext P. We denote the least significant byte of C as $C_{[0]}$ and P as $P_{[0]}$.

The application expects m number of P_S characters to appear in the plaintext. Starting at $C_{[0]}$, we negate the least significant bit of $C_{[0]}$ by XOR-ing with 0x01 ($C_{[0]} \oplus 0x01$), and submit the ciphertext. The application will

respond with either *SeparatorException*, *OtherException*, or normal application state. For the purposes of the attack, *OtherException* and normal application state are functionally identical; we will concern ourselves only with *SeparatorException* or not.

By negating the least significant bit, we have negated the least significant bit of $P_{[0]}$, altering it. If a *SeparatorException* was generated, this indicates that either a P_S was removed from the plaintext and the values could not be split correctly, or a P_S was added to the plaintext, resulting in too many separators. From a single bit negation, we are unsure which is the case - in fact at this point we are unsure if the application does produce a *SeparatorException* if too many P_S characters are present! However, we can learn if a separator was added or subtracted by performing a second bit negation. We negate the second-least significant bit of $C_{[0]}$ by XOR-ing with 0x02 ($C_{[0]} \oplus 0x02$) and submit the ciphertext. Because P_S cannot be produced by two different negations within the same byte, if we receive a *SeparatorException* we can confirm that we are *removing* a separator by modifying $P_{[0]}$ - and thus the first byte of the plaintext contains a separator. We denote the first separator found as P_{S_0}, the second P_{S_1}, and the total number of separators as P_{S_m}.

If no *SeparatorException* is received after modifying $C_{[0]}$, we restore it to its original value, and perform the same operation on a bit of the next byte, $C_{[1]}$. By performing the test on each byte individually, we can determine which bytes of the ciphertext are separators. For a ciphertext of length n, this algorithm determines the location and number of separators in a maximum of $(2 \times n)$ queries to the oracle.

```
#Input: String: ciphertext
#Output: List: separatorPositions
foreach(i in [0..len(ciphertext)]): # Iterate over all bytes
  ciphertext' = ciphertext
  ciphertext'[i] = ciphertext[i] ^ 1  # Negate the Least Significant Bit of the byte
  result = queryOracle(ciphertext')

  if result == SeparatorException:
    # Confirm this is a separator
    ciphertext' = ciphertext
    ciphertext'[i] = ciphertext[i] ^ 2  # Negate Second Least Significant Bit of the byte
    result = queryOracle(ciphertext')

    if result == SeparatorException: # If both results match
      Byte at position i is a separator
    else:
      # Byte at position i not a separator, but is 1 bit removed from it
      #  such that negating the least significant bit made it a separator
      #  and the application cannot handle extra separators
  else: # OtherException or Normal Operation
    # Byte at position i is not a separator
```

Listing 1: Algorithm for Determining Separators in Plaintext

4.2 Phase 2: Approximating the Message

At this stage, we know the positions of all the separators in the plaintext. We take the original ciphertext, and negate a bit in the byte at the first separator position, P_{S_0}. When this modified ciphertext, call it ciphertext′, is submitted, a *SeparatorException* will be returned because the application expects one more separator than is present in the modified plaintext.

However, by modifying a byte known *not* to be a separator, we can correct the *SeparatorException* and induce an *OtherException* or normal application state. Set $C_{[0]}$ [3] to each possible byte value B successively and query the oracle. If a *SeparatorException* is returned, modifying $C_{[0]}$ to B did not produce a new P_S in the plaintext. Before modifying $C_{[0]}$ we had $m - 1$ separator characters. As long as we receive a *SeparatorException*, we know that modifying $C_{[0]}$ to B did not turn $P_{[0]}$ into a separator.

[3]We will assume P_{S_0} != 0, and thus modify $C_{[0]}$ - if this is not true, modify $C_{[1]}$ instead.

But some value of B *will* result in an *OtherException* or normal application state. That means that value of B, when XOR-ed with the Intermediate Value from the encryption function, produces a separator ($B \oplus I_{[0]} = P_S$), and the plaintext now has the expected number of separators. We solve this formula for $I_{[0]}$, producing $I_{[0]} = B \oplus P_S$.

We recall the original decryption relation in Figure 2 and summarize it as $P_j = I_j \oplus C_j$. We solve this equation for the Intermediate Value, using byte [0], creating the following equation: $I_{[0]} = P_{[0]} \oplus C_{[0]}$. We substitute $I_{[0]}$ in this formula with the results of the aforementioned formula, $B \oplus P_S$, and solve for $P_{[0]}$. This produces $P_{[0]} = C_{[0]} \oplus P_S \oplus B$. We store B as an approximate value $A_{[0]}$ for index 0, restore $C_{[0]}$ to the original value, and repeat the process for all bytes that are not separators. At the end, we have a series of relations for each byte [i]: $P_{[i]} = C_{[i]} \oplus P_S \oplus A_{[i]}$, where $C_{[i]}$ and $A_{[i]}$ are known. For a ciphertext of length n, we determine these relations in a maximum of $(255 \times n)$ queries to the oracle.

```
#Input: String: ciphertext, List: separatorPositions
#Output: Array: approximateValues

Initialize approximateValues as an empty array

# Set ciphertext' to a string known to produce a SeparatorException
ciphertext' = ciphertext
ciphertext'[separatorPositions[0]] = ciphertext'[separatorPositions[0]] ^ 1

foreach(i in [0..len(ciphertext)]): # Iterate over all bytes
  if byte b at position i is a separator:
    # Append 0 to approximateValues
  else:
    foreach(B in [1..255]): # Byte Loop, Iterate over all possible byte values
      ciphertext'' = ciphertext'
      ciphertext''[i] = B

      queryOracle(ciphertext'')
      if result == SeparatorException:
        continue #b XOR Intermediate Value does not produce a Separator
      else:
        # Append b to approximateValues
        # break Byte Loop
```

Listing 2: Algorithm for Determining Approximate Values

4.3 Phase 3: Determining the Separator Value and Decrypting the Ciphertext

At this point we are a single value away from decrypting the entire message. Recall we have an array containing an approximate values $A_{[i]}$ for all byte positions [i]. We can use this array with the relation $P_{[i]} = C_{[i]} \oplus P_S \oplus A_{[i]}$ to find the plaintext $P_{[i]}$ for each byte, *if* we know the separator character P_S.

```
#Input: Array: approximateValues, String: ciphertext, Character: separator, List: separatorPositions
#Output: String: plaintext

Initialize plaintext as an empty string

foreach(i in [0..len(ciphertext)]): # Iterate over the ciphertext
  if position i is a separator:
    byte plaintextChar = separator
  else:
    byte plaintextChar = ciphertext[i] ^ separator ^ approximateValues[i]
  append plaintextChar to plaintext
```

Listing 3: Decrypting the ciphertext using the separator character.

If the separator character is not known, run the algorithm for each likely separator character, and see if any of the plaintext are recognizable. No further queries to the oracle are made, allowing ciphertext decryption in a maximum of $(257 \times n)$ queries for a ciphertext of length n.

5 Plaintext Modification

Recall the construction of $I \oplus C$ confidentiality modes. The final step XORs attacker controlled data against the Intermediate Value to produce the plaintext. Therefore, if the Intermediate Value is known, it is possible to calculate a ciphertext that results in an arbitrary plaintext value equal to or less than the original length. For the chosen plaintext to be accepted by the application, we must choose a plaintext that contains the same number of P_S characters as the original plaintext. To create plaintext values of greater lengths than recovered Intermediate Values, insert the ciphertext and repeat the attack outlined to recover these values. Once recovered, additional plaintext can be added.

After decrypting the ciphertext C, we know the plaintext P, and can calculate the Intermediate Value I, $I = C \oplus P$. For a chosen plaintext P', we calculate a new ciphertext C' via Figure 5, for each byte [i].

Figure 5: Equation for Performing Plaintext Modification

$$C'_{[i]} = C_{[i]} \oplus P_{[i]} \oplus P'_{[i]}$$

6 Remedies

6.1 Short Term: Remove the Oracle

As a short term solution, an application developer could modify existing code to silence the Separator Oracle. One potential work-around is to utilize a common exception handler that returns the same values for *SeparatorException* and *OtherException*. Another fix may be to use exception catching routines around the separator function (e.g. `try/catch/finally`).

While these work-arounds may decrease the likelihood of exploiting the Separator Oracle, they do not guarantee the application is invulnerable to other types of oracles. They should only be seen as temporary work-arounds until a more robust solution described in Section 6.2 is implemented.

6.2 Long Term: Prevent Adaptive Chosen-Ciphertext Attacks

Adaptive chosen-ciphertext attacks can be prevented when the authenticity of the message is verified by the crypto-system. Two approaches can be used to provide authenticity: using an encryption mode that includes authentication or utilizing message authentication codes in addition to an encryption mode.

6.2.1 Authenticated Encryption Modes

NIST's 800-38 series defines two encryption modes that authenticate the ciphertext: Counter with Cipher Block Chaining-Message Authentication Code (CCM) [12] and Galois/Counter Mode (GCM) [13]. An authenticated cipher mode builds integrity into the block cipher mode. Any alteration to the ciphertext is detectable as part of the decryption routine, an error will be raised, and no plaintext will be produced.

6.2.2 Message Authentication Codes

A crypto-system can calculate a message authentication code (MAC) over the ciphertext, and transport the MAC next to it. Once received, an application verifies the MAC and should not decrypt a ciphertext whose MAC is incorrect. If the MAC is calculated over the plaintext, the decryption function must run; however, the result should be discarded before use if the calculated MAC does not match the provided MAC. Care needs to be taken to ensure the MAC verification is performed in constant time [14], or a separate timing attack may be exposed. The authors recommend the former Encrypt-then-MAC pattern [15], versus the latter Encrypt-and-MAC pattern.

7 Conclusion

We have demonstrated an attack against block ciphers that use an "I $\oplus$ C" confidentiality mode when the implementation discloses a *Separator Oracle*. An attack was shown against CTR mode that allowed plaintext recovery in a low number of queries to the oracle, and how to manipulate the ciphertext to decrypt to arbitrary plaintext. Finally, short-term and long-term solutions to the vulnerability were presented.

8 Acknowledgments

We wholeheartedly thank Nate Lawson, Marsh Ray, and Matthew Green for reviewing various versions this paper. Their valuable feedback is very much appreciated. We would also like to thank several of our inspirations: Julian Rizzo and Thai Duong for demonstrating padding oracles, Adam Langley, Moxie Marlinspike, Dino Dai Zovi, Brandon Edwards, Dan Guido, and Peter Oehlert. We would also like to thank our employers, Aspect Security and iSEC Partners; and our partners, whose constant encouragement we could not do without.

We thank and apologize to Juraj Somorovsky for leading us to more work on the topic.

9 References

[1] S. Mister and R. Zuccherato, "An attack on cfb mode encryption as used by openpgp," 2005. 1

[2] S. Vaudenay, "Security flaws induced by cbc padding - applications to ssl, ipsec, wtls," in *Proceedings of In Advances in Cryptology - EUROCRYPT'02*, pp. 534–546, Springer-Verlag, 2002. 1

[3] T. Duong and J. Rizzo, "Padding oracles everywhere." http://www.ekoparty.org/archive/2010/ekoparty_2010-Duong_Rizzo-Padding_Oracles_Every_Where.pdf, 2010. 1

[4] J. Katz and B. Schneier, "A chosen ciphertext attack against several e-mail encryption protocols," in *9th USENIX Security Symposium*, pp. 241–246, 2000. 1

[5] K. Jallad, J. Katz, J. J. Lee, and B. Schneier, "Implementation of chosen-ciphertext attacks against pgp and gnupg," in *Information Security, 5th International Conference, volume 2433 of Lecture Notes in Computer Science*, pp. 90–101, Springer-Verlag, 2002. 1

[6] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs #1," in *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*, pp. 1–12, 1998. 1

[7] J. Manger, "A chosen ciphertext attack on rsa optimal asymmetric encryption padding (oaep) as standardized in pkcs #1 v2.0," in *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pp. 230–238, 2001. 1

[8] C. J. Mitchell and R. Holloway, "Error oracle attacks on cbc mode: Is there a future for cbc mode encryption?," 2005. 2

[9] J. Black and H. Urtubia, "Side-channel attacks on symmetric encryption schemes: The case for authenticated encryption," in *In Proceedings of the 11th USENIX Security Symposium*, pp. 327–338, 2002. 2

[10] T. Jager and J. Somorovsky, "How to break xml encryption," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011. 2

[11] C. Percival, "Cryptographic right answers." http://www.daemonology.net/blog/2009-06-11-cryptographic-right-answers.html, 2009. 2

[12] M. Dworkin, "Recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality," SP 800-38c, U.S. DoC/National Institute of Standards and Technology, 2004. See http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C_updated-July20_2007.pdf. 6

[13] M. Dworkin, "Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac," SP 800-38d, U.S. DoC/National Institute of Standards and Technology, 2007. See http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf. 6

[14] N. Lawson, "Timing attack in google keyczar library." http://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/, 2009. 6

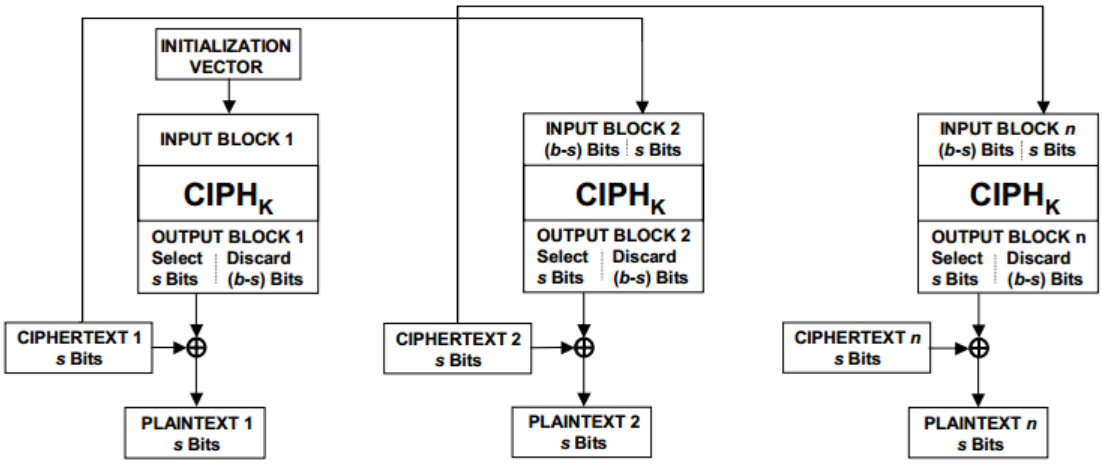
[15] C. Percival, "Encrypt-then-mac." http://www.daemonology.net/blog/2009-06-24-encrypt-then-mac.html, 2009. 6

[16] M. Dworkin, "Recommendation for block cipher modes of operation," SP 800-38a, U.S. DoC/National Institute of Standards and Technology, 2001. See http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf. 9

A CFB Mode

Cipher Feedback Mode is defined in [16] as a confidentiality mode of block ciphers that requires an integer parameter denoted *s* that specifies the *segment size* of the algorithm. The segment size is a value greater than or equal to 1 and less than or equal to the block size of the cipher algorithm that refers to how many bits of the Intermediate Value are used, and correspondingly, how many bits are encrypted or decrypted for each invocation of the cipher operation. For a segment size of 1, an entire cipher operation is performed, an Intermediate Value the size of the algorithm's block size produced, and a single bit selected from the most significant side of the Intermediate Value. The single bit is then XOR-ed with the first bit of the plaintext (or ciphertext) to produce the first bit of ciphertext (or plaintext). Figure A, taken from [16], depicts CFB Decryption using the segment size parameter.

Figure 6: CFB Mode Decryption depicting segment parameter



A.1 The Effect of Segment Size on the Separator Oracle Attack

CFB Mode is starkly different from CTR and OFB modes of operation with regards to the Separator Oracle attack. In CTR and OFB modes, Intermediate Values are derived from application-controlled values: an increasing counter and a repeatedly encrypted Initialization Vector respectively. In CFB mode, the ciphertext itself feeds into the encryption routine to produce Intermediate Values. The Separator Oracle attack relies on detecting the effect of changing a single bit or byte of the plaintext at a time. However, segment j's Intermediate Value is created by performing the cipher operation on segment $j-1$ - therefore altering a bit or byte in segment $j-1$ changes the Intermediate Value and plaintext for segment j. This breaks the underlying assumption that a single bit or byte change in the ciphertext produces a corresponding single bit or byte change in the resulting plaintext.

A.2 Challenges of Multi-Segment Ciphertext

If the ciphertext spans multiple segments, manipulating segment j will entirely corrupt the plaintext of segment $j+1$. Additionally, implementation behavior begins to take a much greater role in possible outcomes. If an application allows extra P_S characters to appear in the plaintext, or if altering some plaintext values produces no discernible change in application state, it can be difficult to determine how the plaintext was affected by the changed ciphertext. Some techniques are presented, but these are not exhaustive, as ultimately individual implementation will determine what information about the plaintext can be derived.

A.2.1 Randomly Introduced Separators

If a byte $C_{[B]}$ in segment j is altered, segment $j+1$ will be corrupted. Corrupting segment $j+1$ may introduce the separator character P_S to the segment. If segment $j+1$ did not contain a separator, the corruption may cause a false positive for the byte $C_{[B]}$ changed in segment j. A false positive occurs when a byte $C_{[B]}$ in segment j, altered by XOR-ing with some value X, introduces one or more separators into the plaintext of the corrupted segment $j+1$. But the false positive can be detected by performing the same query on byte $C_{[B]}$ XOR-ed with different values: *W*, *Y*, *Z*, etc. Separator characters may appear in the corrupted segment $j+1$ for some values, including X, but are unlikely to appear for all values. There are 255 ways $C_{[B]}$ can be altered, and the majority of them should not introduce separators into the subsequent corrupted segment. Discarding the outlier result(s) can be used to eliminate the random effect of a separator being introduced into a corrupted segment.

The same problem presents itself during the second approximation phase. Byte $C_{[B]}$ in segment j is modified to some value X and we look for the *absence* of a *SeparatorException* to indicate that X is the Approximate Value for $C_{[B]}$ we are searching for. However, it is also possible that altering byte $C_{[B]}$ corrupted segment $j+1$ and introduced a P_S, giving another type of false positive. This type of false positive can also be detected, although the segment size *s* comes into play, and s may be unknown. Byte $C_{[B]}$ is held at value X; X either produces a false positive or is the Approximate Value we are searching for. Assume byte $C_{[B]}$ is not the last byte in a segment - modifying byte $C_{[B+1]}$ would then corrupt segment $j+1$ in a new way. If we are dealing with a false positive - the *SeparatorException* will reappear; if X is the legitimate Approximate Value, the *SeparatorException* will *probably* not reappear[4]. Now assume byte $C_{[B]}$ is the last byte in a segment, and byte $C_{[B+3]}$ in the corrupted segment $j+1$ is producing a P_S. Modifying byte $C_{[B+1]}$ will not remove that P_S, all bytes up to byte $C_{[B+(s/8)]}$[5] must be changed to detect the false positive. A segment size of 8 (one byte), is the exception - although all bytes are on a segment boundary, modifying byte $C_{[B+1]}$ will result in a new cascade of alternations.

A.2.2 Corrupted Blocks

If the plaintext of segment $j+1$ does contain a P_S, corrupting it by altering a byte $C_{[B]}$ in segment j will remove that P_S. Thus, all bytes in segment j will consistently return a *SeparatorException*[6]. When segment $j+1$ is processed, the algorithm will work as normal, as segment j is unaltered and segment $j+1$ uncorrupted. By skipping segment j, the algorithm can still proceed on segment $j+1$. However if segment $j+2$, or segment $j+x$, also contains a P_S, the pattern repeats. In this case, it is advisable to work backwards to find the final (or only) P_S in some segment k. When the final P_S, P_{S_m} is found in segment k, all segments from k onward may be processed.

However, there are mitigating factors to processing segment k onwards also. The plaintext contains separator-delimited values $v_1, v_2, \ldots, v_n$. P_{S_m} delimits value v_n. Consider the case where v_n spans two segments: segment k and $k+1$. If the application does not give discernible behavior when an extra separator appears in v_n – either through a *SeparatorException* or other behavior – then segment $k+1$ cannot be decrypted. In order to decrypt segment $k+1$, segment k would need to be altered to remove P_{S_m} – but by altering segment k, segment $k+1$ is corrupted and cannot be processed.

A.2.3 Summary of Challenges

It is clear that these situations are complicated and depend heavily on implementation specific details. This Appendix does not seek to present algorithms that correctly handle all cases, but rather discuss some techniques to handle certain situations when some knowledge is available *a priori*.

[4] It is conceivable that even though we have found the correct Approximate Value, the corrupted segment $j+1$ now contains a separator character P_S and the application issues a *SeparatorException* if too many P_S are present. Modifying byte $C_{[B+1]}$ to alternate values and discarding outliers can reduce this possibility.

[5] The segment size s is given in bits, thus $C_{[B+(s/8)]}$ indicates one segment past position B.

[6] Unless the unlucky situation happens where a separator is introduced by random into the corrupted segment as covered in Section A.2.1 Randomly Introduced Separators.

A.3 Segment Size Greater Than 1 Byte

If the segment size used in CFB mode is greater than one byte, it is always possible to conclusively locate one separator – but fully decrypting some or all of the ciphertext depends on the plaintext being attacked and the implementation of the application.

First the segment containing the final separator P_{S_m} must be located. This segment will be called k. When segment $k-1$ is altered, segment k will be corrupted – therefore altering any byte in a segment prior to k will cause a *SeparatorException*[7]. If we work backwards altering one byte at a time, the first occurrence of two *SeparatorExceptions* in a row is likely to indicate we have reached segment $k-1$ and corrupted segment k. Although implementation specific, it is unlikely for two separators to be found in a row – therefore, once two sequential *SeparatorExceptions* are found, we consider the first of the two to be the final byte in segment $k-1$. As we worked backwards, we encountered at least one *SeparatorException* occurring on its own; indicating a separator character P_S.

```
#Phase 1: Finding the separators
#Input: String: ciphertext
#Output: Array: byteIndicators

const Type.TemporaryMarker = 3
const Type.PreKSegment = 2
const Type.Separator = 1
const Type.CiphertextByte = 0
separatorExceptionsInARow = 0

foreach(i in [len(ciphertext)-1..0]): # Iterate over all the bytes in reverse
  ciphertext' = ciphertext
  ciphertext'[i] = ciphertext[i] ^ 1  # Flip the Least Significant Bit of the byte
  result = queryOracle(ciphertext')

  if result == SeparatorException: # Confirm this is a separator
    ciphertext' = ciphertext
    ciphertext'[i] = ciphertext[i] ^ 2 # Flip Second Least Significant Bit of the byte
    result = queryOracle(ciphertext')

    if separatorExceptionsInARow >= 1 and result == SeparatorException:
      if Type.Separator not in byteIndicators:
        # First SeparatorException was actually a Separator, then we crossed into the prior block.
        #  May miss scenario where two separators are both in segment k
        byteIndicators[i+1] = Type.Separator
      else:
        byteIndicators[i+1] = Type.PreKSegment
      foreach(j in [i..0]: # Now mark all remaining bytes as off limits
        byteIndicators[j] = Type.PreKSegment
      break
    elif separatorExceptionsInARow == 0 and result == SeparatorException:
      # Byte at position i is a separator (maybe)
      # Set to Type.TemporaryMarker so we don't accidentally overwrite when looking for Type.Separator
      byteIndicators[i] = Type.TemporaryMarker
      separatorExceptionsInARow = 1
    else:
      if Type.TemporaryMarker in byteIndicators: # Plaintext byte, confirm any outstanding separator
        byteIndicators[byteIndicators.index(Type.TemporaryMarker)] = Type.Separator
      byteIndicators[i] = Type.CiphertextByte # Byte at position i is not a separator
      separatorExceptionsInARow = 0
  else: #OtherException or Normal Operation
    if Type.TemporaryMarker in byteIndicators: # Plaintext byte, confirm any outstanding separator
      byteIndicators[byteIndicators.index(Type.TemporaryMarker)] = Type.Separator
    byteIndicators[i] = Type.CiphertextByte # Byte at position i is not a separator
    separatorExceptionsInARow = 0
```

Listing 4: Algorithm for Determining Separators when Segment Size > 1 Byte

The output of this algorithm will be an array containing indicators for each position of the ciphertext. A 2 (or

[7]Barring a separator being introduced into the corrupted segment as covered in Section A.2.1 Randomly Introduced Separators.

Type.PreKSegment) indicates a byte on a segment we cannot process. Altering any of those bytes would corrupt subsequent segments. A 0 (or Type.Separator) indicates a ciphertext byte we can attempt to decrypt, and a 1 (or Type.Separator) indicates a separator. We will alter the first separator so it is no longer a separator, as we did in the single-segment case. Then we will alter each 0-indicated byte to produce a separator to find the Intermediate Value.

```
# Phase 2: Approximating the final plaintext value
# Input: String: ciphertext, Array: byteIndicators
# Output: Array: approximateValues

const Type.PreKSegment = 2
const Type.Separator = 1
const Type.CiphertextByte = 0

Initialize approximateValues as an empty array

separatorPosition = index of first 1 in byteIndicators
ciphertext' = ciphertext
ciphertext'[separatorPosition] = ciphertext'[separatorPosition] ^ 1

foreach(i in [0...len(ciphertext')]): #Ciphertext Loop
  if byteIndicators[i] == Type.Separator:
    append 0 to approximateValues
  elif byteIndicators[i] == Type.PreKSegment:
    append 0 to approximateValues
  else:
    foreach(b in [1..255]): # Byte Loop
      ciphertext'' = ciphertext'
      ciphertext''[i] = b
      queryOracle(ciphertext'')
      if result != SeparatorException:
        append b to approximateValues
        break Byte Loop
      else:
        continue # b is not an Approximate value
```

Listing 5: Algorithm for Determining Approximate Values when Segment Size > 1 Byte

Finally, we can perform the decryption phase, omitting any bytes in segments prior to k.

```
# Phase 3: Decrypting the final plaintext value
# Input: Array: approximateValues, String: ciphertext, Character: separator, Array: byteIndicators
# Output: String: plaintext

Initialize plaintext as an empty string

foreach(i in [0..len(ciphertext)]):
  if byteIndicators[i] == 1:
    byte plaintextChar = separator
  else if byteIndicators[i] == 2:
    byte plaintextChar = '?' # This value cannot be decrypted
  else:
    byte plaintextChar = ciphertext[i] ^ separator ^ approximateValues[i]
  append plaintextChar to plaintext
```

Listing 6: Decrypting the ciphertext when Segment Size > 1 Byte

The success of the above algorithm does depend on the structure of the (unknown) plaintext. It will only work when the final separator appears in the final segment. In this scenario, the final segment will be decrypted entirely. If the final separator does not appear in the final segment, and the final value v_n spans two segments as described in Section A.2.2 Corrupted Blocks, the algorithm will *not* work correctly.

If the final separator does appear in 2^{nd} to last segment, it is only possible to partially decrypt that segment. Call the number of segments n, and the segment the final separator appears in segment $n - r$, where $r >= 1$. Altering segment $n - r$ during the approximation phase 2 will corrupt segment $n - r + 1$, making it possible that a separator appears randomly in the corrupted segment. In this event it is impossible to determine if the modification made

to byte $C_{[B]}$ in segment $n - r$ was a valid Approximate Value for $C_{[B]}$ or a corruption of a subsequent segment. (In practice, the corrupted segment regularly produces the random separator.) The only recourse is to store all the values that produce a separator[8], and try each value for the position and see if the plaintext is recognizable. These results are deemed partial, as it may be possible to assemble multiple reasonable values of segment $n - r$ from the choices of stored values.

Application-specific behavior determines whether partial results of segments $n - r + 1 \ldots n - 1$ and decryption of segment n is possible. In some cases, it may not be possible to decrypt segments $n - r + 1 \ldots n$, even partially. If the application does not throw a *SeparatorException* if an extra separator is present in the plaintext, then segments $n - r + 1 \ldots n$ cannot be decrypted. If it does, we can continue. The byte in segment $n - r$ containing the final separator must remain unaltered so segment $n - r + 1$ is not corrupted. Keeping segment $n - r$ equal to the original value, each byte $C_{[B]}$ in segment $n - r + 1$ is altered until the application throws a *SeparatorException* – indicating the Approximate Value for $C_{[B]}$. However, the same problem with randomly introduced separators applies to segment $n - r + 1$ also: a list of potential values must be stored. When the final segment n is reached, the values can be determined conclusively as no further segments exist to be corrupted.

A.4 Segment Size of 1 Byte

If the segment size used in CFB mode is a single byte, each byte functions as its own segment. This configuration is relatively common, and is the default for both PyCrypto in python and mcrypt in PHP. In 8-bit CFB mode, modifying byte $C_{[B]}$ corrupts all subsequent bytes. However, it is always possible to work backwards to find the final separator in some byte position i.

```
#Phase 1: Finding the final separator
#Input: String: ciphertext
#Output: Int: separatorPosition
foreach(i in [len(ciphertext)-1..0]): # Iterate over all the bytes in reverse
  ciphertext' = ciphertext
  ciphertext'[i] = ciphertext[i] ^ 1  # Flip the Least Significant Bit of the byte
  result = queryOracle(ciphertext')

  if result == SeparatorException:
    # Confirm this is a separator
    ciphertext' = ciphertext
    ciphertext'[i] = ciphertext[i] ^ 2 # Flip Second Least Significant Bit of the byte
    result = queryOracle(ciphertext')

    if result == SeparatorException:
      # Byte at position i is the final separator
      break
    else:
      # Byte at position i is not a separator
  else: #OtherException or Normal Operation
    # Byte at position i is not a separator
```

Listing 7: Algorithm for Finding the Final Separator when Segment Size equals 1 Byte

After the separator has been found at position i, it may be possible to decrypt bytes after i. Altering byte i to remove the separator would corrupt all bytes subsequent to i. Again, implementation-specific behavior must be relied on. First we must confirm that the application throws a *SeparatorException* if an extra separator is present in the plaintext. Alter byte $i + 2$ through all possible values. If a *SeparatorException* occurs once or a few times, then the application does throw a *SeparatorException* when an extra separator is present in the plaintext. When this is the case, all subsequent bytes from i can be decrypted. For each subsequent byte from i, alter it in all ways until a *SeparatorException* is generated - this is the Approximate Value. Move to the next byte and repeat.

[8] It may be possible to use the technique for the Approximation Phase presented in Section A.2.1 Randomly Introduced Separators to narrow down the possible values for $C_{[B]}$.

```
#Phase 2: Approximating the final plaintext value
#Input: String: ciphertext, Int: separatorPosition
#Output: Array: approximateValues

Initialize approximateValues as an empty array

foreach(i in [0...len(ciphertext')]): #Ciphertext Loop
  if i == separatorPosition:
    append 0 to approximateValues
  else if i < separatorPosition:
    append 0 to approximateValues
  else:
    foreach(b in [1..255]): # Byte Loop
      ciphertext' = ciphertext
      ciphertext'[i] = b
      queryOracle(ciphertext')
      if result == SeparatorException:
        append b to approximateValues
        break Byte Loop
      else:
        continue # b is not an Approximate value
```

Listing 8: Algorithm for Determining Approximate Values when Segment Size equals 1 Byte

Again, we can perform the decryption phase, omitting any bytes prior to position i.

```
#Phase 3: Decrypting the final plaintext value
#Input: Array: approximateValues, String: ciphertext, Character: separator, Int: separatorPosition
#Output: String: plaintext

Initialize plaintext as an empty string

foreach(i in [0..len(ciphertext)]):
  if i == separatorIndex:
    byte plaintextChar = separator
  else if position i < separatorIndex:
    byte plaintextChar = '?' # This value cannot be decrypted
  else:
    byte plaintextChar = ciphertext[i] ^ separator ^ approximateValues[i]
  append plaintextChar to plaintext
```

Listing 9: Decrypting the ciphertext when Segment Size equals 1 Byte

If no *SeparatorException* is thrown after altering bytes $i + 2$ onward, the application does not seem to throw a *SeparatorException* if extra separators appear in the plaintext. In this case, it may not be possible to decrypt from bytes $i + 1$ onward.

A.5 Segment Size Less than 1 Byte, or Not a Multiple of a Byte

The Segment size used in CFB mode is a size in bits from 1 up to the block size of the cipher. In practice, performing an entire encipherment for a single bit is a waste of resources. While segment sizes of a single byte are seen in practice, it is uncommon to find a segment size that does not align on a byte boundary. If the segment size does not align on a byte boundary, it is unknown any algorithm relying on determining ASCII values of the plaintext would operate.