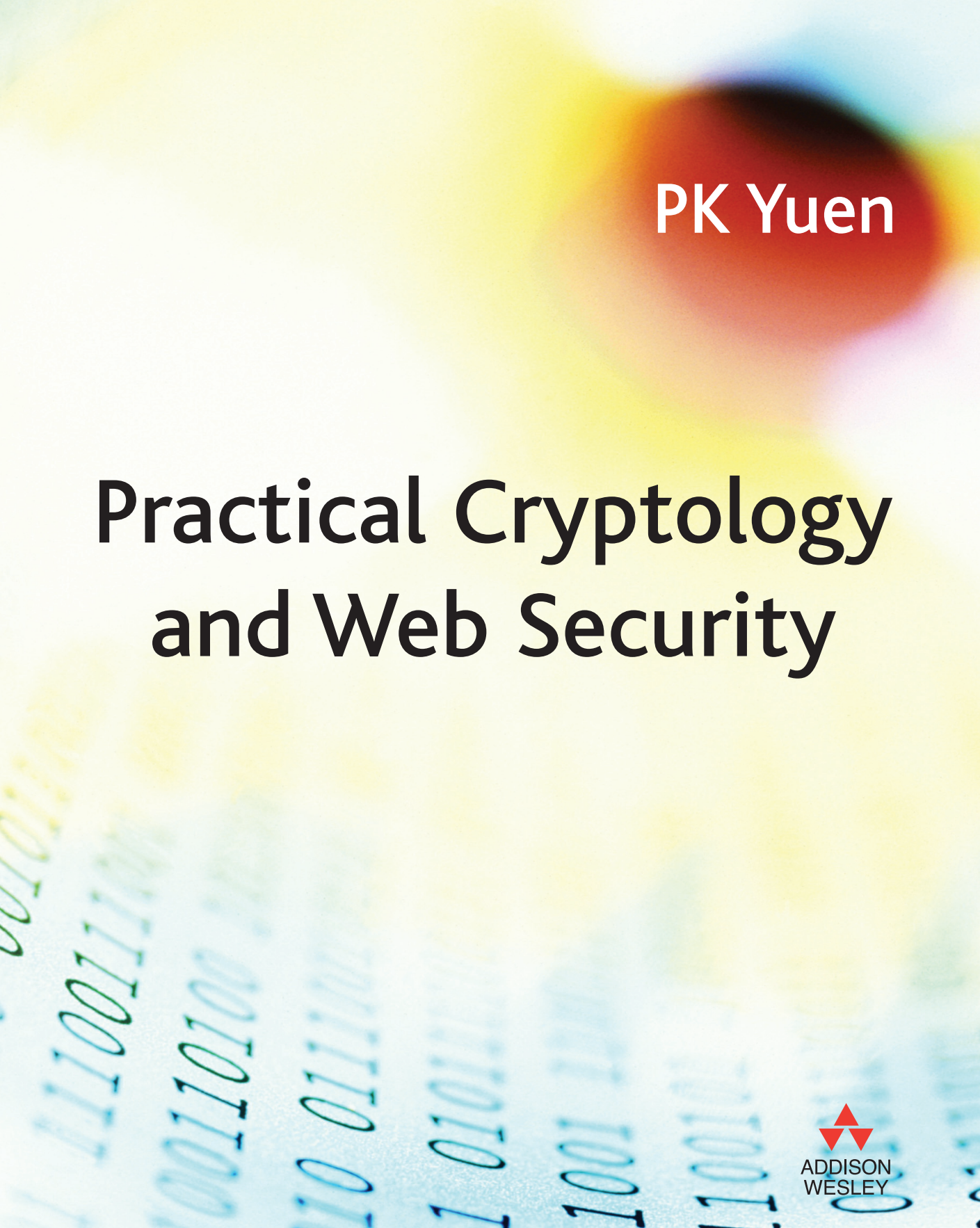
PK Yuen

Practical Cryptology and Web Security



ADDISON WESLEY

Practical Cryptology and Web Security

Visit the *Practical Cryptology and Web Security* Companion Website at **www.pearsoned.co.uk/yuen** to find valuable **student** learning material including:

- Sample contents
- Source code for program examples from each chapter

The second parameter is called `key`. It is an array `key[]` of eight elements containing the user input 64-bit password or raw key. Each element is one byte or an 8-bit integer. Given an array of eight elements `key[]`, the for-loop in lines 36–39 is used to perform `PC1` table lookup. Line 37 gets the value in table `PC1` and stores it as variable `k`. This `k` (or the k^{th} bit) represents the bit position within the password. Consider the statement in line 38,

```
pc1m[j]=((key[(k >>> 3)] & BIT_MASK_FOR_BYTE[k & 0x07]) != 0);
```

The expression (`k>>>3`) will find out which element (byte) in array `key[]` contains the k^{th} bit. The `key[(k>>>3)]` will single out that element. The expression `BIT_MASK_FOR_BYTE[k & 0x07]` will find out which bit inside element `key[(k>>>3)]` is the k^{th} bit. If this bit is not zero, the variable `pc1m[j]` stores the bit. This process is equivalent to performing table `PC1` permutation on array `key[]`.

The `i` for-loop in lines 40–61 generates all 16 subkeys. If the variable `encrypting` is `true`, the statement in line 41 makes sure that the subkeys are in ascending order for encryption. Otherwise the statement in line 42 will be executed and the subkeys are in reverse order for decryption. Consider when `i = 0`: the first subkey is generated and will be stored in variables `rKey[0]` and `rKey[1]`. Each variable is a 32-bit integer storing half (i.e. 24 bits) of the subkey. Putting it together, the 48-bit subkey is stored. The two for-loops in lines 46–55 are used to perform the shift table. By calculating the total rotation, i.e. `TOTAL_ROTATION[i]`, in line 47, the first 28 elements in `pc1m[]` can be shifted by the for-loop in lines 46–50. Another 28 elements in `pc1m[]` are shifted by the for-loop in lines 51–55. The result is stored in array `pcr[]`. With `pcr[]`, table `PC2` permutation can be done by the for-loop in lines 57–60. Consider the statement in line 58:

```
if (pcr[PC2[j]])    rKey[m] |= BIT_MASK_FOR_24BIT[j];
```

This statement is equivalent to checking through all the bits of `pcr[]` via the variable `j` under table `PC2` permutation. If the bit is not zero, it is cast into the return key `rKey[]`. This is the trick we have used from time to time to perform table lookup and should be easy to read by now. When `i = 0`, the first subkey is stored in variables `rKey[0]` and `rKey[1]`. When `i = 1`, with the total rotation, the second subkey is generated and stored in `rKey[2]` and `rKey[3]`. At the end, all 16 subkeys are stored in `rKey[0],...,rKey[31]`.

Now, let's see how to schedule these subkeys into the correct format suitable for DES implementation. Consider part III of script `ex04-04.js`.

Consider the first subkey situation (`i = 0`). Each variable `rKey[0]` and `rKey[1]` is 32-bit storing 24 bits of the subkey. If we consider each 6-bit of the subkey as a chunk, the bit streams of `rKey[0]` and `rKey[1]` are

```
i1 = rKey[0] = 0000 0000 p1 p2 p3 p4
i2 = rKey[1] = 0000 0000 p5 p6 p7 p8
```

The symbol `p1` is the first 6-bit of the subkey, `p2` is the second 6-bit and so on. To schedule this subkey for our optimized algorithm, the four statements in

Example: Continuation of ex04-04.js **(Part III)**

```
63:
64:   for (i = 0; i < 32; i += 2)
65:   {
66:     i1 = rKey[i]; i2 = rKey[i + 1];
67:     rKey[i]     = ((i1 & 0x00fc0000) <<   6)
68:                 | ((i1 & 0x00000fc0) <<  10)
69:                 | ((i2 & 0x00fc0000) >>> 10)
70:                 | ((i2 & 0x00000fc0) >>>  6);
71:
72:     rKey[i + 1] = ((i1 & 0x0003f000) << 12)
73:                 | ((i1 & 0x0000003f) << 16)
74:                 | ((i2 & 0x0003f000) >>> 4)
75:                 | (i2 & 0x0000003f);
76:   }
77:   return rKey;
78: }
```

lines 67–70 will extract `p1`, `p3`, `p5` and `p7` and combine them in variable `rKey[0]` with the format

```
rKey[0] = 00p1 00p3 00p5 00p7
```

Each 6-bit p_i is an 8-bit boundary and can be easily extracted to perform an SP-box operation. This is exactly what we want in the algorithm. Similarly, the statements in lines 72–75 compose the subkey `rKey[1]=00p2 00p4 00p6 00p8`. All 32 array elements `rKey[]` containing all 16 subkeys are returned to the caller by the statement in line 77.

You may have already worked out that this script, `ex04-04.js`, together with `ex04-03.js` will form the core program and codes for a completely optimized DES implementation. In fact, we will use it to develop the Tri DES and other DES encryption tools.

4.3 Optimized DES, triple DES and some encryption tools

4.3.1 A functional optimized DES page

To construct a fully functional DES encryption/decryption tool with the optimized codes is straightforward. One simple way is to make a copy of `ex04-03.js` and `ex04-04.js` and merge them into a single script called `ex04-05.js`. At the end of this new script, add the following codes.

Example: ex04-05.js - DES encryption/decryption

```
297: function keyStToSubkeys(encrypt,inKeySt)
298: {
299:   var i, rKeyArr = new Array(8);
300:   for (i=0; i<8; i++) rKeyArr[i] = inKeySt.charCodeAt(i);
301:   return (createKeys(encrypt,rKeyArr));
302: }
303:
304: function des(keySt, msgSt, encrypt)
305: {
306:   var m=0, chunk=0, len = msgSt.length;
307:   var i, tResult="", result="";
308:   var iKeyArr=new Array(8), oKeyArr=new Array(), lrArr=new Array(2);
309:
310:   keySt += "\0\0\0\0\0\0\0\0"; keySt = keySt.substring(0,8);
311:   if (encrypt) oKeyArr = keyStToSubkeys(true,keySt);
312:   else oKeyArr = keyStToSubkeys(false,keySt);
313:
314:   msgSt += "\0\0\0\0\0\0\0\0";
315:   while (m < len) {
316:    lrArr[0]= (msgSt.charCodeAt(m++) << 24)
                 |(msgSt.charCodeAt(m++) << 16)|
317:               (msgSt.charCodeAt(m++) << 8)
                 | msgSt.charCodeAt(m++);
318:    lrArr[1]= (msgSt.charCodeAt(m++) << 24)
                 |(msgSt.charCodeAt(m++) << 16)|
319:               (msgSt.charCodeAt(m++) << 8)
                 | msgSt.charCodeAt(m++);
320:
321:    lrArr= des_64bits(oKeyArr,lrArr[0],lrArr[1]);
322:
323:    tempSt= String.fromCharCode(
324:      ((lrArr[0] >>> 24) & 0xff), ((lrArr[0] >>> 16) & 0xff),
```

```
325:       ((lrArr[0] >>>  8) & 0xff), ( lrArr[0]          & 0xff),
326:       ((lrArr[1] >>> 24) & 0xff), ((lrArr[1] >>> 16) & 0xff),
327:       ((lrArr[1] >>>  8) & 0xff), ( lrArr[1]          & 0xff));
328:
329:     tResult += tempSt; chunk += 8;
330:     if (chunk    512) {
331:        result += tResult; tResult = ""; chunk = 0;
332:      }
333:   }
334:   return result + tResult;
335: }
336:
```

The total number of lines from files `ex04-03.js` and `ex04-04.js` is 296. As a continuation, the starting line of this script is 297. This script contains two functions. The first function `keyStToSubkeys(encrypt,inKeySt)` in lines 297–302 takes two parameters. If the first parameter `encrypt` is `true`, the function will process the key string `inKeySt` and return all 16 subkeys for encryption. If the `encrypt` value is `false`, it will return subkeys for decryption.

The second function `des(keySt, msgSt, encrypt)` has three parameters: the user raw key (`keySt`), message string (`msgSt`), and a boolean type variable `encrypt` to switch between encryption and decryption. First, the key string `keyST` is expanded and then cut to eight characters so that the key is always 64-bit. If encryption is `true`, the function `keyStToSubkeys()` in line 311 will generate a set of subkeys in array `oKeyArr[]` for encryption. Otherwise, the statement in line 312 will create a set of subkeys for decryption.

When the subkeys are generated, the input message string `msgSt` is expanded in line 314 so that message padding is available if necessary. The while-loop in lines 315–335 runs through the entire message for processing. The statements in lines 316–319 capture eight characters (64 bits) of `msgSt` and convert it into two 32-bit integers called `lrArr[0]` (i.e. 'left') and `lrArr[1]` (i.e. 'right'). Together with the subkeys `oKeyArr[]`, variables `lrArr[0]` and `lrArr[1]` represent one chunk of data (64-bit) to be processed by the function `des_64bits()` in line 321. The result is returned and stored in array `lrArr[]`.

The statements in lines 323–327 convert `lrArr[]` into a string of eight characters in `tempSt`. This `tempSt` is one chunk of data and is stored in variable `tResult` in line 329 temporarily. In order to do some administrative work and reduce some string manipulations, the string in `tResult` is copied to another variable `result` whenever 512 characters are reached. Therefore, the statement in line 329 always works with small strings with little overhead. When the entire message is processed, the combined result (`result + tResult`) is returned to the function caller. To test this script, consider part I of `ex04-05.htm`.

Example: ex04-05.htm – DES encryption/decryption page (Part I)

```
 1: <style>
 2:  .butSt{font-size:16pt;width:250px;height:40px;
 3:           font-weight:bold;background:#dddddd;color:#ff0000}
 4:  .radSt{font-size:16pt;width:35px;height:30px;
 5:           font-weight:bold;background:#88ff88;color:#ff0000}
 6: </style>
 7: <body style="font-family:arial;font-size:26pt;text-align:center;
 8:       background:#000088;color:#ffff00">
 9:  DES Encryption/Decryption On The Web<br />
10:  <img alt="pic" src="line1.gif" height="7" width="650" /><br />
11:
12: <table style="font-size:18pt" align="center">
13:  <tr><td><br />Enter The Input Message: </td></tr>
14:  <tr><td><textarea style="font-size:16pt;width:570px;height:100px;
15:           font-weight:bold;background:#dddddd" rows="5" cols="40"
16:           id="in_mesg">Meet Me At 2pm Tomorrow</textarea></td></tr>
17: </table>
18: <form action="">
19: <table style="font-size:18pt" cellspacing="10" align="center">
20:  <tr><td>Enter The Key: </td>
21:      <td style="text-align:center" colspan=2>
22:        <input type="text" id="key_v" size="8" maxlength="8"
23:         style="font-size:16pt;width:370px;height:40px;
24:         font-weight:bold;background:#dddddd;color:#ff0000"
            value="agent001" /></td></tr>
25:
26:  <tr><td style="width:180px;text-align:left"><input type="radio"
             checked
27:          id="b_rad" name="b_rad" class="radSt" /> Encryption</td>
28:    <td style="width:180px;text-align:left"><input type="radio"
29:        id="b_rad" name="b_rad" class="radSt" /> Decryption</td>
30:    <td style="width:180px;text-align:left"><input size="20"
31:        type="button" class="butSt" style="width:180px"
           value="OK" onclick="des_fun()" />
32:    </td></tr>
33: </table>
34: </form>
35: <table style="font-size:18pt" align="center">
36:  <tr><td>The Output Message is: </td></tr>
37:  <tr><td><textarea rows="5" cols="40" id="out_mesg" readonly
38:             style="font-size:16pt;font-weight:bold;width:570px;
39:             height:100px;background:#aaffaa"></textarea></td></tr>
40: </table>
```

This XHTML code is a simple interface page to get user input. Basically it contains two text areas, one text box, two radio buttons and one OK button. The first text area defined in lines 14–16 allows a user to enter a message for processing. The default message is 'Meet Me At 2pm Tomorrow' as illustrated in line 16. The text box in lines 22–24 is to get the user raw key (or password). The default password is 'agent001'. The two radio buttons allow a user to switch between encryption and decryption. When one of the radio buttons is checked and the `OK` button is clicked, the function `des_fun()` is activated. The message is processed by DES encryption or decryption and the result is displayed in the second text area defined in lines 37–39. The function `des_fun()` is defined in part II of `ex04-05.htm`.

Example: Continuation of ex04-05.htm **(Part II)**

```
41: <script src="ex04-05.js"></script>
42: <script src="hexlib.js"></script>
43: <script>
44:  function des_fun()
45:  {
46:    var key="",message="",llst=""
47:    llV = document.getElementsByName("b_rad")
48:    if (llV.item(0).checked) {
49:      key = document.getElementById("key_v").value
50:      message = document.getElementById("in_mesg").value
51:
52:      llst = byteStToHex(des(key, message, 1))
53:      document.getElementById("out_mesg").value = llst
54:    }
55:    if (llV.item(1).checked) {
56:      key = document.getElementById("key_v").value
57:      message = hexStToByteSt(document.getElementById("in_mesg").value)
58:
59:      llst = des(key, message, 0)
60:      document.getElementById("out_mesg").value = llst
61:    }
62:  }
63: </script>
64: </body
```

The function `des_fun()` is defined in the script block in lines 44–62. The statement in line 47 gets the status of all the radio buttons. If the first radio button (i.e. encryption) is checked, lines 49–53 are executed. In this case, the user input raw key and message are captured by lines 49–50 and stored in variables `key` and `message`. Consider the function call in line 52

```
llst = byteStToHex(des(key, message, 1))
```