1.0 Introduction

This report covers my implementation of the Skipjack algorithm and Key Exchange Algorithms in C using some assembly code routines, if necessary, to optimize the encryption and decryption speeds. Then this implementation is compared to the speed of encryption and decryption for optimized DES implementations available. The Key Exchange Algorithm (KEA) is implemented to be compatible with those used by the FORTEZZA card.

The development was conducted on a Pentium II using GNU C under Red Hat Linux 5.2. The optimized implementation is written in C and is portable across many platforms. The standard C libraries are sufficient for implementing the Skipjack encryption and decryption algorithms. However, in order to implement the KEA they require a library to generate large prime numbers as well as large random numbers. For operations on these large numbers I chose to use the SSLeay libraries. They contain header files and a binary library that can be linked with the final object file to provide functions to perform the above operations.

2.0 Overview

This section covers an overview of the SKIPJACK algorithm as well as the Key Exchange Algorithm. Each of the stages of the SKIPJACK encryption and decryption algorithms will be given in detail. Followed by a detailed description of the store and forward KEA for exchanging files or email with another user. Implementations on the FORTEZZA card are also covered.

2.1 SKIPJACK Algorithm

SKIPJACK operates on a single 64-bit input, as four 16-bit words, producing a 64-bit output using an 80-bit cryptovariable (key). The algorithm works by alternating between two stepping rules (A and B) described below. Each rule is executed eight times before alternating, and there are two alternations for a total of 32 rounds. Each of these stepping rules uses a counter that is initialized with a value of 1 for encryption and 32 for decryption. Both of the stepping rules are reversible functions, allowing for decryption.

2.1.1 Stepping Rule A

The basic structure of the stepping rule A for encryption is given below (the G permutation function is covered later).

a. \( G \) permutes \( w_1 \),
b. the new \( w_1 \) is the exclusive or of the \( G \) output, the counter, and \( w_4 \),
c. words \( w_2 \) and \( w_3 \) shift one register to the right; i.e. become \( w_3 \) and \( w_4 \) respectively,
d. the new \( w_2 \) is the \( G \) output
e. the counter is incremented by one
The equations for this rule are shown below. The superscript is the counter value:

<table>
<thead>
<tr>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1^{k+1} = G^k(w_1^k) \oplus w_2^k \oplus counter^k$</td>
<td>$w_1^{k-1} = [G^{k-1}]^{-1}(w_2^k)$</td>
</tr>
<tr>
<td>$w_2^{k+1} = G^k(w_1^k)$</td>
<td>$w_2^{k-1} = w_3^k$</td>
</tr>
<tr>
<td>$w_3^{k+1} = w_2^k$</td>
<td>$w_3^{k-1} = w_4^k$</td>
</tr>
<tr>
<td>$w_4^{k+1} = w_3^k$</td>
<td>$w_4^{k-1} = w_4^k \oplus counter^{k-1}$</td>
</tr>
<tr>
<td>$counter^{k+1} = counter^k + 1$</td>
<td>$counter^{k-1} = counter^k - 1$</td>
</tr>
</tbody>
</table>

### 2.1.2 Stepping Rule B

The basic structure of the stepping rule B for encryption is given below (the G permutation function is covered later).

1. $G$ permutes $w_1$,
2. words $w_3$ and $w_4$ shift one register to the right; i.e. become $w_4$ and $w_1$ respectively,
3. the new $w_3$ is the exclusive or of $w_1$, the counter, and $w_2$,
4. the new $w_2$ is the $G$ output
5. the counter is incremented by one

The equations for this rule are shown below. The superscript is the counter value:

<table>
<thead>
<tr>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1^{k+1} = w_2^k$</td>
<td>$w_1^{k-1} = [G^{k-1}]^{-1}(w_2^k)$</td>
</tr>
<tr>
<td>$w_2^{k+1} = G^k(w_1^k)$</td>
<td>$w_2^{k-1} = w_3^k \oplus w_2^k \oplus counter^{k-1}$</td>
</tr>
<tr>
<td>$w_3^{k+1} = w_1^k \oplus w_2^k \oplus counter^k$</td>
<td>$w_3^{k-1} = w_4^k$</td>
</tr>
<tr>
<td>$w_4^{k+1} = w_3^k$</td>
<td>$w_4^{k-1} = w_4^k$</td>
</tr>
<tr>
<td>$counter^{k+1} = counter^k + 1$</td>
<td>$counter^{k-1} = counter^k - 1$</td>
</tr>
</tbody>
</table>
2.1.3 Stepping Sequence

To encrypt data the counter value is initialized to 1, and the input is divided into four 16-bit words labeled $w_1^0, w_2^0, w_3^0, w_4^0$. Next, 8 steps according to Rule A are performed, followed by 8 steps according to Rule B. After this another 8 steps of Rule A are performed and then completed with another 8 steps of Rule B, for a total of 32 steps. The output is then the results $w_1^{32}, w_2^{32}, w_3^{32}, w_4^{32}$.

To decrypt data the counter value is initialized to 32, and the input is divided into four 16-bit words labeled $w_1^{32}, w_2^{32}, w_3^{32}, w_4^{32}$. Next, 8 steps according to Rule B$^{-1}$ are performed, followed by 8 steps according to Rule A$^{-1}$. After this another 8 steps of Rule B$^{-1}$ are performed and then completed with another 8 steps of Rule A$^{-1}$, for a total of 32 steps. The output is then the results $w_1^0, w_2^0, w_3^0, w_4^0$.

2.1.4 G Permutation

The permutation function G is key dependent four round Feistel structure. The round function uses a fixed byte substitution table (which operates on 8-bit bytes) called the F-table. For encryption the 16-bit input is divided into its upper 8 bits and lower 8 bits as $g_1$ and $g_2$, respectively. The final output is then the concatenation of two 8-bit values $g_5$ and $g_6$. The general formula for G, recursively, is given by:

$$g_i = F(g_{i-1} \oplus cv_{4k+i-3}) \oplus g_{i-2}$$

Where k is the value of the counter (step number), F is the substitution, and $cv_{4k+i-3}$, is the $(4k+i-3)^{th}$ byte in the cryptovariable (key) schedule. Since the key is 10 bytes long, the value of $4k+i-3$ is taken modular 10, with the key bytes labeled 0 though 9.

For decryption the 16-bit input is divided into its upper 8 bits and lower 8 bits as $g_5$ and $g_6$, respectively. The final output is then the concatenation of two 8-bit values $g_1$ and $g_2$. The general formula for G$^{-1}$, recursively, is given by:

$$g_{i-2} = F(g_{i-1} \oplus cv_{4k+i-3}) \oplus g_i$$

The expansion of both recursive functions produces:

<table>
<thead>
<tr>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_1 = F(g_2 \oplus cv_{4k}) \oplus g_1$</td>
<td>$g_4 = F(g_5 \oplus cv_{4k+3}) \oplus g_6$</td>
</tr>
<tr>
<td>$g_4 = F(g_3 \oplus cv_{4k+1}) \oplus g_2$</td>
<td>$g_3 = F(g_4 \oplus cv_{4k+2}) \oplus g_5$</td>
</tr>
<tr>
<td>$g_5 = F(g_4 \oplus cv_{4k+2}) \oplus g_3$</td>
<td>$g_2 = F(g_3 \oplus cv_{4k+1}) \oplus g_4$</td>
</tr>
<tr>
<td>$g_6 = F(g_5 \oplus cv_{4k+3}) \oplus g_4$</td>
<td>$g_1 = F(g_2 \oplus cv_{4k}) \oplus g_3$</td>
</tr>
</tbody>
</table>

Schematically G and G$^{-1}$ are shown in the figure below:
2.1.5 F Table

The SKIPJACK F-table is shown below in hexadecimal notation. The upper 4 bits of the input index the row and the lower 4 bits index the column. For example, F(6D) = 98.

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
<th>x6</th>
<th>x7</th>
<th>x8</th>
<th>x9</th>
<th>xA</th>
<th>xB</th>
<th>xC</th>
<th>xD</th>
<th>xE</th>
<th>xF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>a3</td>
<td>d7</td>
<td>09</td>
<td>83</td>
<td>f8</td>
<td>48</td>
<td>f6</td>
<td>f4</td>
<td>b3</td>
<td>21</td>
<td>15</td>
<td>78</td>
<td>99</td>
<td>b1</td>
<td>af</td>
</tr>
<tr>
<td>1x</td>
<td>e7</td>
<td>2d</td>
<td>4d</td>
<td>8a</td>
<td>ce</td>
<td>4c</td>
<td>ca</td>
<td>2e</td>
<td>52</td>
<td>95</td>
<td>d9</td>
<td>1e</td>
<td>4e</td>
<td>3e</td>
<td>44</td>
</tr>
<tr>
<td>2x</td>
<td>0a</td>
<td>df</td>
<td>02</td>
<td>a0</td>
<td>17</td>
<td>f1</td>
<td>60</td>
<td>68</td>
<td>12</td>
<td>b7</td>
<td>7a</td>
<td>c7</td>
<td>e9</td>
<td>fa</td>
<td>6a</td>
</tr>
<tr>
<td>3x</td>
<td>96</td>
<td>84</td>
<td>6b</td>
<td>ba</td>
<td>39</td>
<td>f2</td>
<td>63</td>
<td>9a</td>
<td>19</td>
<td>7c</td>
<td>ae</td>
<td>e5</td>
<td>f5</td>
<td>f7</td>
<td>16</td>
</tr>
<tr>
<td>4x</td>
<td>39</td>
<td>b6</td>
<td>7b</td>
<td>0f</td>
<td>c1</td>
<td>93</td>
<td>81</td>
<td>1b</td>
<td>ee</td>
<td>b4</td>
<td>la</td>
<td>ea</td>
<td>d0</td>
<td>91</td>
<td>2f</td>
</tr>
<tr>
<td>5x</td>
<td>55</td>
<td>b9</td>
<td>da</td>
<td>85</td>
<td>3f</td>
<td>41</td>
<td>bf</td>
<td>e0</td>
<td>5a</td>
<td>58</td>
<td>80</td>
<td>5f</td>
<td>66</td>
<td>0b</td>
<td>d8</td>
</tr>
<tr>
<td>6x</td>
<td>35</td>
<td>d5</td>
<td>c0</td>
<td>a7</td>
<td>33</td>
<td>06</td>
<td>65</td>
<td>69</td>
<td>45</td>
<td>00</td>
<td>94</td>
<td>56</td>
<td>6d</td>
<td>98</td>
<td>9b</td>
</tr>
<tr>
<td>7x</td>
<td>97</td>
<td>fc</td>
<td>b2</td>
<td>c2</td>
<td>b0</td>
<td>fe</td>
<td>db</td>
<td>20</td>
<td>e1</td>
<td>eb</td>
<td>d6</td>
<td>e4</td>
<td>dd</td>
<td>47</td>
<td>4a</td>
</tr>
<tr>
<td>8x</td>
<td>42</td>
<td>ed</td>
<td>9e</td>
<td>6e</td>
<td>49</td>
<td>3c</td>
<td>cd</td>
<td>43</td>
<td>27</td>
<td>d2</td>
<td>07</td>
<td>d4</td>
<td>de</td>
<td>c7</td>
<td>67</td>
</tr>
<tr>
<td>9x</td>
<td>89</td>
<td>cb</td>
<td>30</td>
<td>1f</td>
<td>8d</td>
<td>c6</td>
<td>8f</td>
<td>aa</td>
<td>c8</td>
<td>74</td>
<td>dc</td>
<td>c9</td>
<td>5d</td>
<td>5c</td>
<td>31</td>
</tr>
<tr>
<td>Ax</td>
<td>70</td>
<td>88</td>
<td>61</td>
<td>2c</td>
<td>9f</td>
<td>0d</td>
<td>2b</td>
<td>87</td>
<td>50</td>
<td>82</td>
<td>54</td>
<td>64</td>
<td>26</td>
<td>7d</td>
<td>03</td>
</tr>
<tr>
<td>Bx</td>
<td>34</td>
<td>4b</td>
<td>1c</td>
<td>73</td>
<td>d1</td>
<td>c4</td>
<td>fd</td>
<td>3b</td>
<td>cc</td>
<td>fb</td>
<td>7f</td>
<td>ab</td>
<td>e6</td>
<td>3e</td>
<td>5b</td>
</tr>
<tr>
<td>Cx</td>
<td>ad</td>
<td>04</td>
<td>23</td>
<td>9c</td>
<td>14</td>
<td>51</td>
<td>22</td>
<td>f0</td>
<td>29</td>
<td>79</td>
<td>71</td>
<td>7e</td>
<td>ff</td>
<td>8c</td>
<td>0e</td>
</tr>
<tr>
<td>Dx</td>
<td>0c</td>
<td>ef</td>
<td>bc</td>
<td>72</td>
<td>75</td>
<td>6f</td>
<td>37</td>
<td>a1</td>
<td>ec</td>
<td>d3</td>
<td>8e</td>
<td>62</td>
<td>8b</td>
<td>86</td>
<td>10</td>
</tr>
<tr>
<td>Ex</td>
<td>08</td>
<td>77</td>
<td>11</td>
<td>be</td>
<td>92</td>
<td>4f</td>
<td>24</td>
<td>c5</td>
<td>32</td>
<td>36</td>
<td>9d</td>
<td>cf</td>
<td>f3</td>
<td>a6</td>
<td>bb</td>
</tr>
<tr>
<td>Fx</td>
<td>5e</td>
<td>6c</td>
<td>a9</td>
<td>13</td>
<td>57</td>
<td>25</td>
<td>b5</td>
<td>e3</td>
<td>bd</td>
<td>a8</td>
<td>3a</td>
<td>01</td>
<td>05</td>
<td>59</td>
<td>2a</td>
</tr>
</tbody>
</table>

2.2 SKIPJACK Modes of Operation

SKIPJACK can be used in four modes of operation, which is a subset of the FIPS-81 description of modes of operation for DES. These include Output Feedback (OFB) modes, Cipher Feedback (CFB) modes, Codebook, and Cipher-block Chaining (CBC). The FORTEZZA card implements the Codebook and CBC modes of operation for SKIPJACK.
2.3 Key Exchange Algorithm

The KEA requires that each user have a private and public key, where the public key is available to both sender and receiver. There is no mention of how an end user is to validate the public keys of other users, however typical implementations use X.509 certificates. The private key of a user consists of 4 parameters:

- \( p \) 1024-bit prime modulus
- \( q \) 160-bit prime divisor of \( p-1 \) for public component checking
- \( g \) 1024-bit base for the exponentiation, element of order \( q \) in the multiplicative group \( \text{mod } p \).
- \( x \) 160-bit user secret number \((0 < x < q)\)
The modulus and other parameters are generated as per the Digital Signature Standard, FIPS-186 specification. The 1024-bit public key is then generated using the equation:

\[ Y = g^x \mod p \]

The KEA also requires the generation of 160-bit random number, which is then converted into a public number to be sent to the other end using the same equation as that for the public key:

\[ R = g^r \mod p \]

Users of a network share the same, p, q, and g values. For the KEA, this must be true, therefore only users of the same network can use the KEA.

2.3.1 Store and Forward (E-Mail) Applications

For store and forward applications there is no active connection between the two ends of communication. The sender generates all the data necessary for the KEA and then sends public versions of this information to the other end. The only data generated for store and forward applications is a single 160-bit private random number. This number is then converted into a public random number and sent to the receiver. For consistency in notation subscripts of A refer to the sender and subscripts of B refer to the receiver.

2.3.1.1 Sender Algorithm

The first step is to obtain the public key of the receiver and validate its authenticity. The next step is to verify the public key is from a user on the network, by checking that the following equations are satisfied:

\[ 1 < Y_B < p \text{ and } (Y_B)^y \equiv 1 \mod p \]

The sender generates a random 160-bit number and converts it to a public 1024-bit number using:

\[ R_A = g^r \mod p \]

Next the following values are computed:

\[ t_{AB} = (Y_B)^{r_A} \mod p = g^{r_As} \mod p \]

\[ u_{AB} = (Y_B)^{r_A} \mod p = g^{r_As} \mod p = g^{r_As} \mod p \]

\[ w = (t_{AB} + u_{AB}) \mod p \]

If, \( w \neq 0 \), then \( w \) is split into two different sections:

\[ v_1 = \left( \frac{w}{2^{1024-80}} \right) \mod 2^{80} \text{ and } v_2 = \left( \frac{w}{2^{1024-160}} \right) \mod 2^{80} \]

Which is equivalent to saying that \( v_1 \) is the upper most 80 bits of \( w \) and \( v_2 \) is the penultimate upper 80 bits of \( w \).

They key is then calculated using:

\[ Key = 2^{16} \left[ E_{\text{v1}} \oplus \left( E_{\text{v1}} \oplus \left( \frac{v_2}{2^{16}} \mod 2^{64} \right) \right) \right] \oplus \left( \frac{E_{\text{v1}} \oplus \left( \frac{v_2}{2^{16}} \mod 2^{64} \right)}{2^{48}} \right) \oplus (v_2 \mod 2^{16}) \]

The pad used above is defined as 72f1a87e92824198ab0b hex. Schematically, this is represented by:
2.3.1.2 Receiver Algorithm

The first step is to obtain the public key of the sender and validate its authenticity as well as the public random number generated by the sender. The next step is to verify the public key and random number is from a user on the network, by checking that the following equations are satisfied:

\[ 1 < Y_A, R_A < p \text{ and } (Y_A)^n \equiv 1 \mod p \text{ and } (R_A)^n \equiv 1 \mod p \]

Next the following values are computed:

\[ t_{BA} = (R_A)^x \mod p = g^{r_x s} \mod p \]
\[ u_{AB} = (Y_A)^x \mod p = g^{x x_s} \mod p \]
\[ w = (t_{AB} + u_{AB}) \mod p \]

If, \( w \neq 0 \), then \( w \) is split into two different sections:

\[ v_1 = \left( \frac{w}{2^{[1024-80]}} \right) \mod 2^{80} \text{ and } v_2 = \left( \frac{w}{2^{[1024-160]}} \right) \mod 2^{80} \]

Which is equivalent to saying that \( v_1 \) is the upper most 80 bits of \( w \) and \( v_2 \) is the penultimate upper 80 bits of \( w \).

They key is then calculated using:
The pad used above is defined as 72f1a87e92824198ab0b hex. Schematically, this is represented by:

\[
Key = 2^{16} \left[ E_{v_1 \oplus pad} \left( E_{v_1 \oplus pad} \left( \frac{v_2}{2^{16}} \mod 2^{64} \right) \right) \right] \oplus \left( \frac{E_{v_1 \oplus pad} \left( \frac{v_2}{2^{16}} \mod 2^{64} \right)}{2^{48}} \right) \oplus (v_2 \mod 2^{16})
\]

2.3.2 FORTEZZA KEA

The key generated by the KEA is not used directly to encrypt the information being sent the receiver. Rather a temporary message encryption key (MEK) is generated instead and used to encrypt the message. Then the key generated by the KEA, called a temporary encryption key (TEK) is used to encrypt the MEK. This encryption process is known as wrapping the MEK. There are no published details on the algorithm used to wrap the MEK. The implementation used by me is to pad the MEK with six bytes of 06 (hexadecimal) to produce a 128-bit input. Which is then encrypted with SKIPJACK using the TEK for a key as two separate 64-bit blocks in the Codebook mode. What is important to notice is that to send a large file to multiple users it is only necessary to encrypt the large file once.

The FORTEZZA card uses the CBC mode for encryption, which requires a 64-bit initialization vector. Therefore, the information that needs to be sent to the receiver, along with the encrypted file, includes the wrapped MEK, the public random number, and the initialization vector. This is assuming that both parties have the public key of the
3.0 SKIPJACK Optimization

This section discusses the optimizations made to the SKIPJACK implementation to increase the speed at which decryption and encryptions are performed. The optimizations were originally going to be made with assembly language routines. However, it was found that it was possible to optimize the code using the c programming language and the gcc compiler under Red Hat Linux 5.2.

3.1 Key Scheduling
The most noticeable optimization possible was to precompute combinations of each byte of the key with possible inputs to the F-table function. This is because the input to the F-table function is the exclusive or of any possible byte with a single byte of the key. Therefore, a table can be created which contains the 256 possible outputs of the F-table based on a single byte of the key. This will save four XOR instructions per step or 128 XOR instructions over a single encryption of a 64-bit block.

3.2 Memory Management

The stepping rules involve moving each 16-bit word to one place to the right each step. This is extremely slow and wasteful. It is possible to leave each 16-bit word in place and just change the operations around so that it is not necessary to perform 4 MOV instructions per step.

Also, it was noticed that the Intel instruction set has only 4 general purpose registers EAX, EBX, ECX, and EDX, however penalties are incurred for using the ECX and EDX registers for mathematical operations. This does not leave enough registers to load all four 16-bit words into registers to optimize the speed at which functions can be performed upon the data. Another, subtle optimization, is one in which pointers are used to manipulate the data directly in memory and not to copy them into local variables.

3.3 Macros vs. Subroutines

The second biggest optimization comes from inlining the $G$ and $G^{-1}$ functions rather than subroutines. Subroutines require a lot of overhead to pass parameters and push the new instruction pointer on the stack. Therefore, by explicitly defining each step saves an enormous amount of time.

```c
#define G(k0,k1,k2,k3,g1,g2,g5,g6) g5 = KeySchedTable[k0][g2]^g1; g6 = KeySchedTable[k1][g5]^g2; g5 = KeySchedTable[k2][g6]^g5; g6 = KeySchedTable[k3][g5]^g6;
#define INV_G(k0,k1,k2,k3,g5,g6,g1,g2) g2 = KeySchedTable[k3][g5]^g6; g1 = KeySchedTable[k2][g2]^g5; g2 = KeySchedTable[k1][g1]^g2; g1 = KeySchedTable[k0][g2]^g1;
```

3.4 Compiler Optimizations

This is the most machine dependent part of the optimization process. The compiler used was gcc on Red Hat Linux 5.2. The processor used is a Pentium II 233 MHz machine. The command line to compile the optimized SKIPJACK routines is:

```
egcs -s -static -Wall -O3 -fomit-frame-pointer -funroll-loops -malign-loops=2 -malign-functions=2 -malign-jumps=2 -mpentiumpro -o kea kea.c optsjlib.c kealib.c -lcrypto
```

It is due to these compiler optimizations that the code generated is optimized greatly. Each round produces approximately 15 lines of assembly, which uses commands that do not suffer from penalties of using registers outside of the EAX and EBX.

3.5 Comparison with Optimized DES

For the optimized version of DES I used the SSLeay libraries implementation of DES. The SSLeay libraries use an extremely fast byte lookup implementation. To compare the two implementations a 64-bit test vector was used and two different keys, since DES requires a 56-bit key and SKIPJACK requires an 80-bit key. Both implementations
precompute key-scheduling values, therefore the timing is only for the encryption and decryption routines. In order to be able to get comparable values 100,000 encryptions and decryptions are performed and timed. The results are shown in the following table (average time to perform 100,000 iterations):

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKIPJACK</td>
<td>34 ms</td>
<td>29 ms</td>
</tr>
<tr>
<td>DES</td>
<td>16 ms</td>
<td>16 ms</td>
</tr>
</tbody>
</table>

From a simple comparison standpoint the DES algorithm consists of 16 rounds, whereas SKIPJACK is 32 rounds. It is conceivable then, that an optimized version of SKIPJACK would be twice as slow as that for DES. Using this simple comparison method, then the optimized version of SKIPJACK that I implemented is on target.

4.0 KEA Implementation

This section covers the KEA implemented to transfer a file (or E-Mail) to another user on a network. It was desired to create a KEA that is compatible with the FORTEZZA PCMCIA security cards. However, since access to these cards was not available all the data that would normally be contained on the card is stored in text files. Also for some of the internal routines to the card, for which algorithms were not available from the FORTEZZA Application Developer’s Guide, a best guess implementation was used.

The KEA operates on numbers that are on the order of 1024 bits and therefore require a large number library to perform these functions. I used the SSLeeay big number libraries for this purpose.

4.1 Public Key File

This is a simple text file that contains the 1024-bit public keys of users on the network. The format of the file is a unique identifier followed by an equal sign and then the users public key in hexadecimal format. Each record is kept on a separate line. This file is named public.key.

ALICE=2D29EC0D2E3497A67222D8DEBC286131D149F4581B3586D0151024C02E88B23DA09A430E2CA5ED1A4B2D772562316E4D2804D226788284ED655CF54610D38F6E6FAB1A0A2E2D3C6614401901D9758D566722AFF734B2ABDB2B67F1300CE455F0968CA791A876787363D7D49EE74A28DC349D9FDB96B01F0CF10F69OC9E6BOB=773D4BBF3A2EFDD218E7043E8612014CEC06C205F5419293B65C69A971E5455EB79A0DB90AB214C5240EDE6CFD58C719C52695D7F660B61CDB2FF64864EE51987F270034BC390AD7316B2095E2608C3D7987F649FB716887633EB574B39CC73DF89951FC1BD6D3889D48FE2244B829AFD40506AB9221BA562C07

4.2 Private Key File

This is a text file containing the 4 components of a user’s private key. The user’s secret number (x), p, q, and g. Each component is on a separate line labeled by either X, P, Q, or G, an equal sign and the value in hexadecimal. This is stored in the file private.key.

X=62319AC47DE145180ABD322C59E2B6002781E494P=9D4C66D42EA91C826D76D499AF01B8E5B8573D0FAEE7BD569DD1914E3AD4759C85331EDA1459FB56E8ADD4736652A82B276E82AC63F5B78DB75A03EB34D397DBE7B374D8F7216AA CB0879E61C718A37F5154B507BA7649FB3D4FB4C481E01062C5241F229FA580423363D51090DBF25351F0C58000E05BF2A6A9Q=97AD85FD2B371ED069B18AB3C6ED8773D9B029D G=595D7443EC897C8251E5FA9D22AB875C0FC57B0969F880DA366A10001912A0196BC81C41AC8485031AC598B5481EAE2726B719D8993A6105937432760C0A6A2C723CD6700D341F54BF28
4.3 Key Exchange Data File

When a file is encrypted using the KEA program it is necessary to give the receiver the initialization vector, the public random number, and the wrapped MEK. Therefore a text file is created which contains these values in the same format as the public and private key file. The KEA data file has the same name as the output (encrypted) file with the extension .KEA-userid attached. Where userid is the unique identifier for the public key used in the KEA.

RA=4A3222E2767B90C49CF6EAD9AA634D6D2C74B2B513E14CE94B92E2259284E86E665B3FDCC445A79244ADE831987CC0AE38312F335171F06462C6D49FDE48AC3FB7F395D8B752B40017E73016480A06182F8444DF05BF412C92AD05603E1C0ECB86CC49366FFDE3717AB50BF575B6395CC1F84643EEE83F349C8F63CBF7DDBFC2
MEK=8F8787D92D6EA189EF1E2B28F04E83AB
IV=9B6FD1355C5DC91B

5.0 Conclusions

Creating an optimized SKIPJACK encryption and decryption implementation was possible without using assembly code routines. The optimized versions are about twice as slow as those are for DES. For 100,000 encryptions the optimized SKIPJACK averaged 34 ms and 29 ms for decryption.

The KEA is mostly compatible with the FORTEZZA security cards. Where possible routines were implemented to mimic library calls that are available with the FORTEZZA Developer’s Libraries. Only the wrapping of the MEK is not necessarily completely compatible since no information has been made available as to the algorithm for performing this function.