An abridged version of this paper appears in the Proceedings of the 17th Annual Computer Security Applications Conference. (December 11 - 14, 2001, New Orleans, Louisiana, USA)

The Performance Measurement of Cryptographic Primitives on Palm Devices *

Duncan S. Wong, Hector Ho Fuentes and Agnes H. Chan
College of Computer Science
Northeastern University
Boston, MA 02115, USA
{swong, hhofuent, ahchan}@ccs.neu.edu

Abstract

We developed and evaluated several cryptographic system libraries for Palm OS® which include stream and block ciphers, hash functions and multiple-precision integer arithmetic operations. We noted that the encryption speed of SSC2 outperforms both ARC4 (Alleged RC4) and SEAL 3.0 if the plaintext is small. On the other hand, SEAL 3.0 almost doubles the speed of SSC2 when the plaintext is considerably large. We also observed that the optimized Rijndael with 8KB of lookup tables is 4 *times faster than DES. In addition, our results show that implementing the cryptographic algorithms as system libraries does not degrade their performance significantly. Instead, they provide great flexibility and code management to the algorithms. Furthermore, the test results presented in this paper provide a basis for performance estimation of cryptosystems implemented on PalmPilot™.*

1. Introduction

With the continuous growth of the Internet and the advancement of wireless communications technology, handheld devices such as the Palm devices are also experiencing booming demands for accessing information and getting connected with the Internet anytime, anywhere. At the same time, users are expecting secure data transmission and storage on these devices, which in turn require handheld devices to provide efficient cryptographic algorithms.

Many cryptographic algorithms, which are simple and efficient to implement on high-performance microprocessors such as those found in current desktop computers, may not be implementable efficiently on smaller and less powerful microprocessors found in low-power handheld devices. We will see shortly that some cryptographic operations, which take only a few milliseconds or less and are widely used in securing data, performing authentication and integrity check on desktop machines, may spend seconds or even minutes to carry out on a PalmPilot. Furthermore, the memory space on low-power handheld devices are usually limited which may also introduce new challenges on the implementation of cryptosystems. Hence it is critical for users to choose appropriate algorithms for the implementation of cryptosystems on low-power devices.

We developed several cryptographic system libraries for Palm OS which include the following algorithms:

Stream Ciphers We tested the encryption speed of three stream ciphers: SSC2 [12], ARC4 (Alleged RC4)[1] and SEAL 3.0 [8].

Block Ciphers Various modes of some block ciphers have been tested. The algorithms surveyed in this paper are Rijndael [1], DES and its variants such as DESX and Triple-DES.

Hash Functions Several widely used hash functions are evaluated such as MD2 [3], MD4 [6], MD5 [7] and SHA-1 [5].

Multiple-precision Integer Arithmetic Operations These operations are the core in most public-key cryptographic implementations which involve integers hundreds of digits long. We wrote a system library called MPLib[2] and tested its performance on some commonly used operations.

In the following sections, we give speed measurement results of these algorithms. These results help determine if a cryptographic system is feasible for the PalmPilot or if it is too complex. They can also be used to estimate the performance of a system or a protocol which is constructed based on these algorithms. Our purpose here is to investigate the viability of using these cryptographic primitives on low-power handheld devices.

*This work was sponsored by the U. S. Air Force under contracts F30602-00-2-0536 and F30602-00-2-0518

[1] http://www.achtung.com/crypto/arcfour.txt
[2] http://www.ccs.neu.edu/home/swong/MPLib

In this paper, we describe the whole gamut of the tests from design to analysis. First we provide information on the hardware and software of the test platforms. Then we explain the test methodologies, followed by the test details of each algorithm we conducted. In the last section, we summarize the results and point out the viability of employing each of them on a low-power handheld device.

2. Test Platforms and System Libraries

Tests were conducted on a 2MB Palm V and a 8MB Palm IIIc running on a 16MHz and a 20MHz Motorola DragonBall-EZ (MC68EZ328) microprocessor respectively. The processor has a 68K core which implements the standard Motorola 68K instruction set architecture and is in big Endian architecture. There are 16 general purpose 32-bit registers. Details of the processors can be found at Motorola's web site[3]. The RAM of a Palm device is divided into two logical areas: storage and dynamic. The storage area keeps all the databases, and is analogous to disk storage on a typical desktop system. The dynamic area holds the kernel's globals, dynamic allocations as well as application programs' globals, stacks and dynamic allocations. The size of the dynamic area on a particular device varies according to the running OS version, the amount of physical RAM available, and the requirements of pre-installed software such as the TCP/IP stack or IrDA stack. For Palm V and IIIc, running Palm OS v3.3 and v3.5 respectively, the dynamic heap sizes are 128KB and 256KB. However not all the dynamic heap space can be used by an application program. For example there are only 56KB of the dynamic heap that can be used by applications in a Palm V. The remaining 72KB are reserved for the system and the TCP/IP stack. More information can be found at Palm, Inc.'s Hardware Comparison Matrix webpage[4] and Palm OS Memory Architecture (Take Two: 3.0 and Beyond)[5].

We used Metrowerks CodeWarrior for Palm OS Release 6 as the Integrated Development Environment (IDE) and the compiler. The version of Palm SDK was 3.5. Also, Palm, Inc.'s Constructor for Palm OS version 1.2b7 was used for creating the user interface. During compilation, the following settings were selected:

- 68K Processor Code Model: Small
- Global Optimizations
  - Optimize For: Faster Execution Speed
  - Optimization Level: 4

[3]http://www.motorola.com/SPS/WIRELESS/pda/index.html
[4]http://www.palmos.com/dev/tech/hardware/compare.html
[5]http://oasis.palm.com/dev/kb/papers/1145.cfm

2.1. System Libraries

All the algorithms we discussed here were implemented as system libraries. System libraries are supported by Palm OS `SysLib*` API calls and are *officially* described in Palm OS FAQ *Shared Libraries and Other Advanced Project Types*[6]. Further details can be found in Ian Goldberg's article, *Shared libraries on the Palm Pilot*[7]. A system library is a runtime shared library which allows multiple applications using the common library functions dynamically without having to have a copy of the code in each application's code resource. Shared libraries in general can also help to overcome two memory constraints of Palm OS, namely the 32KB jump limit (of the CodeWarrior default model) and the code resource size limit. From software engineering point of view, system libraries also provide better code management and help developers to write efficient and robust code because the functions in a system library can be tested, modified and upgraded independently.

On the other hand, if the cryptographic algorithms are implemented inside each application program, then they usually achieve better performance. We call this type of implementation as the *bundled* version of an algorithm. For system library based implementation, all library functions are invoked as system calls by the applications. Hence a system trap instruction is executed when a library function is called. Now if an algorithm is implemented in the application code resource, it is also in user mode as the rest of the application code does. Therefore it saves time from executing the system trap related instructions. Furthermore, for the bundled version of algorithms, the compiler can now parse the code of the algorithms in conjunction with the application code to achieve further optimization which is not possible for system libraries because the algorithms are treated as system calls in the applications. In our tests, we found a slight but not significant improvement on the performance of the algorithms when they were in bundled version.

Another type of shared libraries is called GLib[8], which is more user-friendly (or programmer-friendly) than the system library mentioned above and is also faster because no system traps are used when calling the GLib functions. However, as of this writing, neither CodeWarrior nor the latest version of PRC-Tools supports GLibs. Only PRC-Tools 0.5.0 and Michael Sokolov's 0.6.0 beta[9] do so. We choose to implement the algorithms as system libraries for ease of adaptation to later versions of Palm OS and to developing tools.

[6]http://oasis.palm.com/dev/kb/papers/1143.cfm
[7]http://www.isaac.cs.berkeley.edu/pilot/shlib.html
[8]http://www.isaac.cs.berkeley.edu/pilot/GLib/GLib.html
[9]http://www.escribe.com/computing/pcpqa/m9618.html

3. Methodology

Performance measurements were conducted by determining *the amount of time* required to perform cryptographic operations of an algorithm. We measured how many bytes of data could be encrypted in one second for ciphers and how many bytes of data could be digested per second for hash functions. For multiple-precision integer arithmetic operations, we measured the time taken to perform a particular operation. We used the `TimGetTicks` function provided by the Palm OS time-manager API to calculate the processor time consumed in the execution of the algorithms. `TimGetTicks` is a *System Tick* function. A Palm device maintains a *tick count* which increments 100 times per second when the Palm device is in doze or in the running mode. This 0.01-second timer is initialized to zero every time when the device is reset. Since the rate of the tick count is not so high, several iterations of the same operation are required to be carried out in order to achieve a finer resolution on the speed of that operation. The details are given as follows.

For each algorithm, a number of tests were conducted where the time taken for each test was recorded as a *sample*. For each sample, a specific cryptographic primitive operation was executed for a number of times. The number of times the operation executed is called the number of *calls per sample*. Upon completion of the execution, the samples were sorted and the median time value was determined. Then the standard deviation was calculated. Samples that fell outside a pre-specified range of standard deviation from the median were discarded. The remaining sample values were used to compute the average speed of the algorithm. Pseudo code for generating the timing information of an encryption algorithm is shown below.

```
(t = 0; t < samples; t++);
   (Start Timer)
   (c = 0; c < calls; c++);
      makeKey();
      cipherInit();
      Encrypt(data of a pre-specified size);
   (Stop Timer)
```

4. Stream Ciphers

For stream ciphers and block ciphers (Section 5), we measured their performance according to the time taken for each cipher to encrypt a block of data. We used three different sizes of data blocks to do the encryption tests:

2KB A small data block such as a single webpage downloaded via a secure channel.

50KB This block size is comparable to the size of a database such as a secure application database (e.g. a phonebook) or an application program itself.

4MB This size is even bigger than the total RAM available on some Palm devices. It refers to the downloading of some data stream to a Palm device in realtime.

Each test program generated 30 samples, each of which made 512 calls. All accepted samples were within 3 standard deviation range.

4.1. SSC2

SSC2 [12] is a software-efficient stream cipher designed for low-power wireless handsets. It supports various key sizes from 32 bits to 128 bits. All operations in SSC2 are word-oriented (32-bit word) and therefore the keystream generated by SSC2 can also be considered as a word sequence. The word sequence is then added modulo-2 to the words of a plaintext in the manner of a Vernam cipher to get the ciphertext. The algorithm, which consists of three main operations, is shown in the following pseudo code.

```
(Master key generation)
(i = 0; i < message_size_in_words; i++);
  (keystream generation -- one word at a time)
  (bitwise XOR keystream word and plaintext word)
```

Table 1 shows the results of running SSC2 system library on Palm V and Palm IIIc. These results also include the time for master key generation.

Message Size	Throughput (bytes/sec)	
	Palm V	Palm IIIc
2KB	32,604	44,582
50KB	35,804	49,829
4MB	35,501	49,434

Table 1. Performance of SSC2 System Library

For keystream generation alone, we recorded the average throughput of 189,046 bytes/sec and 136,533 bytes/sec on Palm IIIc and Palm V respectively. In addition, the bundled version of SSC2 was about 0.6% to 3% faster than the system library implementation.

Since the SSC2 algorithm operates on words, an appropriate byte ordering conversion may be needed when converting a byte-stream to a 32-bit word-stream and vice versa for interoperability among different system platforms. The Big Endian architecture of the Motorola DragonBall microprocessor favors the conversion. An architecture independent implementation of SSC2 is only about 82% of the speed of a Big Endian architecture optimized code and is slower than ARC4.

On the memory requirement, it takes only 21 words to store the four stages of the LFSR and the 17 stages of the lagged-Fibonacci generator. Hence it is suitable for applications under severely limited memory conditions.

4.2. ARC4

RC4[10] is another software-optimized variable-key-size stream cipher whose detailed algorithm is proprietary. In September 1994, the source code of ARC4 (Alleged RC4) was made available to ftp sites around the world. It is generally accepted that ARC4 is comparable to RC4. We ported ARC4[11] to the Palm system library and measured its encryption speeds with respect to different data block sizes. The results are shown in Table 2.

Message Size	Throughput (bytes/sec)	
	Palm V	Palm IIIc
2KB	30,768	42,281
50KB	32,100	45,110
4MB	31,699	44,501

Table 2. Performance of ARC4 System Library

Results show that ARC4 is very efficient and can be a good candidate for data encryption on Palm devices as well. It requires 256 bytes to store a state array which is bigger than the memory requirement of SSC2. A bundled version of ARC4 achieved 3% to 5% improvement in speed.

4.3. SEAL 3.0

SEAL 3.0 [8] is a stream cipher which is optimized for 32-bit processors. The cipher is a pseudorandom function family which is under control of a key. The cipher stretches a 32-bit position index into a long, pseudorandom string which can then be used as the keystream of a Vernam cipher. In order to make the cipher run well, we need to allocate a memory chunk which is slightly over 3KB for a set of tables. These tables are preprocessed from the key and are used to speed up the keystream generation process.

We first ported the SEAL 1.0 code obtained from Bruce Schneier's book Applied Cryptography [9], then we modified the code so that it conformed with SEAL 3.0. In our tests, the parameters of SEAL(a, n, L) were set to: $a = 160$ bits; $n = 32$ bits and $L = 4096$ bytes. For simplicity and speed, each call to the SEAL keystream generator prepares one keystream block of 4KB long. Hence the program also needs another 4KB of memory to store the keystream. As a result, over 7KB of memory needs to be allocated dynamically when running the SEAL algorithm. Table 3 shows the results when running the SEAL system library on Palm V and Palm IIIc. The results include the time for initializing the tables.

[10]http://www.rsasecurity.com/rsalabs/faq/3-6-3.html
[11]http://www.achtung.com/crypto/arcfour.txt

Message Size	Throughput (bytes/sec)	
	Palm V	Palm IIIc
2KB	2,469	3,427
50KB	28,723	39,121
4MB	51,396	71,980

Table 3. Performance of SEAL 3.0 System Library

We first notice that the encryption speed for the 2KB case is very slow, even slower than that of DES in ECB mode. The reason is that time taken to initialize the tables dominates the entire encryption process. In addition, only 2KB of the 4KB keystream generated has been used to encrypt the message. These can be seen more clearly in the cases of 50KB and 4MB where the overhead for initializing the tables was averaged out. Also all the generated keystream would be fully utilized when the message size is a multiple of 4KB which also induces the optimal case in performance. In addition, SEAL 3.0 is also the fastest cipher we measured in the case of 4MB. From our tests, we noticed that the table initialization in the key-setup phase of SEAL is costly on low-power devices. It is not as efficient as other stream ciphers such as SSC2 or ARC4 when encrypting short messages such as a web page or a single database resource unless precomputation of the tables is allowed in the target application. As the authors mentioned in their paper [8], SEAL is an inappropriate choice for applications which require rapid key-setup. A comparison of the performance of these three stream ciphers is exhibited in Figure 1.

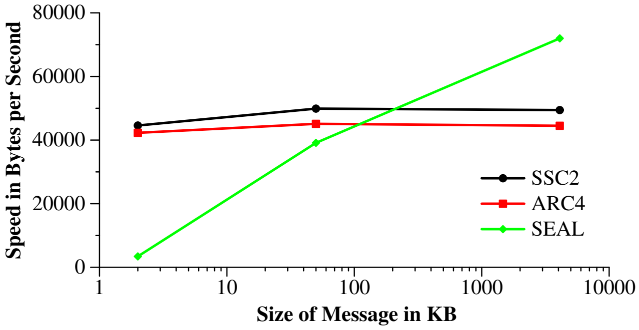

Figure 1. Performance of SSC2, ARC4 and SEAL on the Palm IIIc

5. Block Ciphers

In this section, we present the test results of several block ciphers. They are Rijndael, DES and its variants DESX and Triple-DES. In order to facilitate a comparison with the stream ciphers, we adopt the same data block sizes in our tests.

5.1. Rijndael

Rijndael [1], selected as the AES[12] algorithm, is an iterated block cipher supporting variable block lengths as well as variable key lengths of multiples of 32 bits. The number of rounds employed is a function of the block and key lengths. Each round transformation is composed of four steps, `ByteSub`, `ShiftRow`, `MixColumn` and `AddRoundKey`. Following the description in [1, Section 4.1], we call the output of each round transformation a State which can be represented as a $4 \times \mathtt{Nb}$ matrix $[a_{i,j}]$, where $a_{i,j}$ represents a byte and $4\mathtt{Nb}$ is the block length in bytes. Similarly each round key can be represented as a matrix with four rows. In [1], it is shown that by combining the four steps into a set of table lookups, we can attain very efficient implementations on 32-bit processors. In this section, we will evaluate the performance of several optimization options of this approach and suggest the optimal one in terms of code size and encryption/decryption throughput rates.

For the sake of comparison, we ported both Joan Daemen's ANSI C reference code version 2.1 (using the NIST API) and Brian Gladman's code[13] to Palm OS system libraries. The ANSI C reference code is written for clarity instead of efficiency. It uses four static arrays, namely $S[256]$, $Si[256]$, Logtable[256] and Alogtable[256], where each element of the arrays is one byte long. S is the S-box used in `ByteSub` and Si is its corresponding inverse. Logtable and Alogtable are used for multiplication in $\mathrm{GF}(2^8)$ in the `MixColumn` transformation. The Palm OS system libraries are not allowed to have their own global or static variables (see Jeff Ishaq's paper, Mastering Shared Libraries[14]). In our tests, we have tried two methods to tackle this limitation. The first one, which we adopted, is to put the arrays into routines as local variables with initialization. Another method is to allocate a memory chunk to hold these arrays and store a handle of this chunk in the `globalsP` field of the library's *library table entry* structure, `SysLibTblEntryType`. The library can then call `SysLibTblEntry` to get a pointer to the library table entry and hence obtain the handle of the memory chunk. This method makes the code easier to maintain, but is slower than the first method. According to our test results, it is only half of the speed of the first method. The reason is that whenever the library accesses an array, it needs to lock the memory chunk of the array and gets a pointer to it. After finishing accessing the array, the library has to unlock the memory chunk in order to minimize the dynamic heap fragmentation. The cost of doing these is more expensive than having local variables initialized every time when a library function is called.

On the Palm V, the throughput of this ANSI C reference code porting is only 853 bytes/sec when encrypting a 2KB message under a 128-bit key with 128-bit block length in ECB mode. As we will see in the next section, this is only comparable to the speed of Triple-DES. One main reason of this poor performance is due to the software design approach. The code was written for clarity, in which each of the four steps of the round transformation was written as a routine. This requires significant amount of time to do stack push and pop for argument passing and housekeeping of local variables.

Optimized Implementations To study the optimized implementations, we need to describe Rijndael in greater details. Recall the four steps in each round transformation: `ByteSub` can be represented as $b_{i,j} = S[a_{i,j}]$ where S is a 256-byte substitution table (the S-box); `ShiftRow` can be represented as $b_{i,j} = a_{i,j-C_i}$ where C_i is the shift offset of row i; `MixColumn` can be represented as a matrix multiplication over $\mathrm{GF}(2^8)$ given by

$$\begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{bmatrix} = M \begin{bmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{bmatrix} \text{ where } M = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix};$$

finally, `AddRoundKey` can be represented as $b_{i,j} = a_{i,j} \oplus k_{i,j}$ where $[k_{i,j}]$ is the round key. Combining these four steps, we can represent each column of a round transformation output (e_j) as

$$e_j = T_0[a_{0,j}] \oplus T_1[a_{1,j-C_1}] \oplus T_2[a_{2,j-C_2}] \oplus T_3[a_{3,j-C_3}] \oplus k_j,$$

where $\{T_i\}_{0 \le i \le 3}$ is a set of tables with

$$T_0[a] = \begin{bmatrix} S[a] \cdot 02 \\ S[a] \\ S[a] \\ S[a] \cdot 03 \end{bmatrix}$$

and $T_i[a] = \mathtt{RotByte}(T_{i-1}[a])$ for $1 \le i \le 3$. The function `RotByte` is a cyclic column shift shifting down by one entry. Hence by considering each column as a 4-byte word, with the top entry as the most significant byte, each table T_i contains 256 4-byte word entries. By having these four tables, we can see that only 4 table lookups and 4 bitwise XOR operations are required per column per round.

We ported Brian Gladman's code to a Palm OS system library because the code gives us flexibility to adjust the extent of optimization in the code. The code uses preprocessors extensively to perform macro substitution and conditional compilation. The following sets of defines can be used optionally and they give significant impacts on both speed and code size. In the remaining part of this section, we will discuss the optimal combination of these defines.

[12] http://www.nist.gov/aes
[13] http://fp.gladman.plus.com/cryptography_technology/rijndael/
[14] http://oasis.palm.com/dev/kb/papers/1670.cfm

ONE_TABLE / FOUR_TABLES

These two defines control the use of tables $\{T_i\}_{0\le i\le 3}$ in the main encryption and decryption round transformations. In the case of `FOUR_TABLES`, all the 4 tables are present during encryption. For decryption, 4 additional tables are required to do the inverse. Since each table takes 1KB of memory, the memory requirement for both encryption and decryption is 8KB. In `ONE_TABLE` case, the code only has T_0 available for encryption (another 1KB table for decryption) while the other three tables are derived using the `RotByte` function. Although the `ONE_TABLE` case requires 3 additional rotations per round per column, it only takes 2KB of table space for both encryption and decryption. If neither of them is defined, tables are not used but we will see that the throughput would be severely degraded.

ONE_LR_TABLE / FOUR_LR_TABLES

Since the final round transformation does not have the `MixColumn` step, the tables in `ONE_TABLE` or `FOUR_TABLES` case cannot be used directly. To use the table-lookup approach for the final round, the code has to precompile some additional tables. Similar to the above, `ONE_LR_TABLE` requires 2KB of memory while `FOUR_LR_TABLES` requires 8KB of memory for tables.

ONE_IM_TABLE / FOUR_IM_TABLES

As described in [1, Section 5.3], table lookups can also be applied to obtain inverse round keys during decryption key scheduling.

UNROLL / PARTIAL_UNROLL

These two defines control the extent of which the for-loops of round transformations are unrolled in the main encryption and decryption routines. `UNROLL` completely unrolls all the rounds while `PARTIAL_UNROLL` only unrolls every two rounds. The trade-off in these two cases is the code size and the throughput.

Since each set of defines has three possible choices and there are four sets of defines, the total number of possible combinations is 81. However most of the combinations are poor in the sense that they generate large code sizes but do not give much improvement on throughput. For example, `ONE_TABLE` combining with `FOUR_LR_TABLES` would not give as much improvement on the throughput as `FOUR_TABLES` does even the former one requires more memory space. Base on this conjecture, we can skip most of the possible combinations and focus ourselves to the following few cases.

We first tested the least optimized case, namely there is no unrolling of the encryption and decryption rounds and there is no precompiled tables except the S-box and the inverse of it. For 128-bit key and 128-bit block length, the encryption throughput is $4,085$ bytes/sec in ECB mode on a Palm V. This is 4.8 times faster than the ANSI C reference code porting. The main difference between this code and the ANSI C reference code is that this code extensively uses macro substitution instead of routine calls which require extra stack push and pop operations and have expensive overheads in maintaining local variables.

Next, we tested the case with `ONE_TABLE` defined. The encryption throughput reaches $5,809$ bytes/sec under the same system setup as above. With the additional 2KB of memory space for storing the tables, it gives 42% improvement on throughput. In the `FOUR_TABLES` case, it gives the encryption throughput of $9,689$ bytes/sec which is a further 67% improvement on speed with 6KB additional table space. This is due to the elimination of 3 byte-wise rotations and leaving only 4 table lookups and 4 XOR operations per round per column.

When both `FOUR_TABLES` and `FOUR_LR_TABLES` are defined, the average encryption throughput we found is only $8,977$ bytes/sec under the same system setup as above while it takes 16KB of memory space for tables. One reason for having a slower result than the case of `FOUR_TABLES` is that the code spends more time to lock and unlock additional table records of the database resource for encryption while the efficiency gained by having direct table lookups for the final round may not be significant enough to offset the overhead. In this case, four more pairs of locks and unlocks are taken for encryption while the 4 additional tables are only used at the last round of every block of encryption. The lock and unlock system calls are so costly that it overrides the benefits of having four direct table-lookups for the final round of each column. The evidence comes from the slower speed measured in the 2KB data encryption of this case as compared with the `FOUR_TABLES` case while the speed difference is reduced in the 50KB data encryption because the cost of locking and unlocking the 4 additional tables during encryption (or decryption) is averaged out over the 50KB message.

In fact, there is *no* benefit in creating additional tables for the final round as explained in the following. Since there is no `MixColumn` transformation in the last round, we can replace the matrix M by the unit matrix I and denote the last round output (e_j) as

$$e_j = T'_0[a_{0,j}] \oplus T'_1[a_{1,j-C_1}] \oplus T'_2[a_{2,j-C_2}] \oplus T'_3[a_{3,j-C_3}] \oplus k_j$$

where

$$T'_0[a] = \begin{bmatrix} S[a] & 00 & 00 & 00 \end{bmatrix}^{\mathsf{T}}$$

and $T'_i[a] = \mathtt{RotByte}(T'_{i-1}[a])$ for $1 \le i \le 3$. In the `FOUR_LR_TABLES` case, all the four tables $\{T'_i\}_{0\le i\le 3}$ are present. Thus the final round transformation requires 4 table lookups and 4 bitwise XOR operations for each column of the State. As we can see, the final round output can also be written as

$$e_j = \texttt{Bytes2Word}(S[a_{0,j}], S[a_{1,j-C1}], S[a_{2,j-C2}], S[a_{3,j-C3}]) \oplus k_j.$$

The function `Bytes2Word` concatenates the four input bytes to a word and is commonly defined as follows under the Big Endian architecture.

```
#define Bytes2Word(B0, B1, B2, B3) \
  ((UInt32)(B0) << 24 | (UInt32)(B1) << 16 | \
   (UInt32)(B2) <<  8 | (B3))
```

Therefore by using S-box only, the code takes 4 table lookups, 1 XOR operation, 3 bitwise shift and 3 OR operations. Since the bitwise shift, XOR and OR operations are extremely fast on a Palm device when compared with table lookups which involve indirect memory access, we can ignore their cost. Thus it is clear that the efficiencies of these two implementations are essentially the same.

By considering the additional cost of locking and unlocking the memory chunks in the `FOUR_LR_TABLES` case, it is less efficient than the one without these four tables. Similarly, we found that it worsens the decryption key scheduling efficiency by defining `ONE_IM_TABLE` or `FOUR_IM_TABLES`.

The complete or partial unrolling of the for-loops usually help the compiler to apply optimization algorithms more effectively. In our implementation, the `PARTIAL_UNROLL` option gives the most speed-and-code-size performance improvement though the `UNROLL` option always gives the best throughput result. Table 4 shows the encryption throughput of the optimized code in ECB mode with `UNROLL` and `FOUR_TABLES` defined. It attains 12.5-times speedup over the ANSI C reference code porting and is also 4 times faster than DES in ECB mode. The decryption is about 0.1% slower than the encryption in the 2KB case due to the additional operations in the inverse round key scheduling. The CBC mode is about 8% slower than the ECB mode.

Key Size (bits)	Message: 2KB Throughput (Bps)		Message: 50 KB Throughput (Bps)		Message: 4 MB Throughput (Bps)	
	V	IIIc	V	IIIc	V	IIIc
128	10,673	14,340	10,711	14,776	10,522	14,528
192	8,989	12,091	8,947	12,337	8,769	12,114
256	7,764	10,452	7,673	10,616	7,516	10,387

Table 4. Performance of Optimized Rijndael System Library

5.2. DES and Its Variants

In this section, we give the performance measurement results of DES and its two variants namely DESX and Triple-DES. DESX is an encryption algorithm that extends the DES algorithm to a keysize of 120 bits but still encrypts 64-bit data block at a time. It does this by simply XORing the input block with a bit pattern (pre-Whitening), encrypting with standard DES, and then XORing the result with another bit pattern (post-Whitening). The main motivation for DESX is to provide a computationally simple way to dramatically improve on the resistance of DES to exhaustive key search attacks.

We implemented four different operation modes namely Electronic Codebook (ECB) mode, Cipher Block Chaining (CBC) mode, Cipher Feedback (CFB) mode and Output Feedback (OFB) mode. In the last mode, we implemented two different modalities: the first one is compliant with Federal Information Processing Standards Publication 81 and the second one is compliant with ISO 10116. Table 5, Table 6, and Table 7 show the results.

Mode	Message: 2KB Throughput (Bps)		Message: 50 KB Throughput (Bps)		Message: 4 MB Throughput (Bps)	
	V	IIIc	V	IIIc	V	IIIc
ECB	2,633	3,710	2,612	3,663	2,580	3,594
CBC	2,614	3,690	2,593	3,637	2,558	3,573
FB n = 64	2,438	3,449	2,421	3,395	2,404	3,334
FB n = 32	1,264	1,782	1,262	1,766	1,263	1,750
FB n = 16	637	896	639	894	648	888

Table 5. Performance of DES System Library (n = bits shifted and size of input block, FB for OFB and CFB modes due to similar results.)

Mode	Message: 2KB Throughput (Bps)		Message: 50 KB Throughput (Bps)		Message: 4 MB Throughput (Bps)	
	V	IIIc	V	IIIc	V	IIIc
ECB	2,610	3,683	2,590	3,635	2,558	3,558
CBC	2,604	3,675	2,585	3,624	2,566	3,555
FB n = 64	2,425	3,436	2,408	3,383	2,391	3,231
FB n = 32	1,258	1,774	1,256	1,758	1,248	1,742
FB n = 16	634	892	637	890	634	884

Table 6. Performance of DESX System Library (n = bits shifted and size of input block, FB for OFB and CFB modes due to similar results.)

The speeds of DES and its variants are relatively slow, compared to the stream ciphers presented in this paper. The results are roughly 9% of the speed of the stream ciphers. When comparing with optimized implementation of Rijndael, the throughput of DES is only 25% of that of Rijndael in ECB mode.

The memory requirements for DES and its variants are not as high as that for SEAL and optimized Rijndael. We found that in the limited environment of the Palm OS, with its 3.25 KB stack size (by default), special treatment on some local tables have to be done in order to prevent using

Mode	Message: 2KB		Message: 50 KB		Message: 4 MB	
	Throughput (Bps)		Throughput (Bps)		Throughput (Bps)	
	V	IIIc	V	IIIc	V	IIIc
ECB	878	1,234	891	1,246	890	1,234
CBC	877	1,231	890	1,243	885	1,231
FB n = 64	856	1,201	868	1,215	879	1,205
FB n = 32	436	610	470	645	544	617
FB n = 16	237	307	220	308	220	308

Table 7. Performance of Triple DES System Library (n = bits shifted and size of input block, FB for OFB and CFB modes due to similar results.)

up all the spaces for stack and local variables. In our implementation, the memory needed for tables is about 2KB.

Figure 2 gives a comparison of the encryption speeds of these block ciphers.

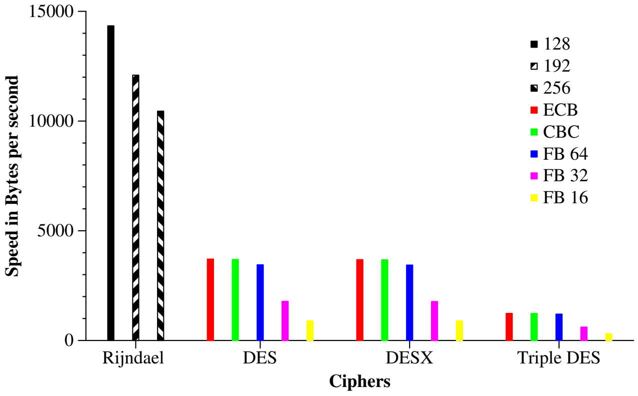

Figure 2. Performance of Rijndael, DES, DESX and Triple-DES on a Palm IIIc (2KB case)

6. Hash Functions

In this section, we give the performance measurement results of MD2, MD4, MD5 and SHA-1. All of these algorithms were ported from open source coded in C and implemented in the system library format. The test results are shown in Table 8.

If the hash functions are bundled on the application code resource directly, we found a slight improvement on the hashing speed for all these hash functions due to the reasons explained in Section 2.1. The improvement ranges from 1% to 4%. The relative speeds of these hash functions running on Palm devices are shown in Figure 3. The speed of SHA-1 is about 47% of that of MD5.

	Message size: 2KB		Message size: 50KB		Message size: 4MB	
	Throughput (Bps)		Throughput (Bps)		Throughput (Bps)	
	V	IIIc	V	IIIc	V	IIIc
MD2	1,738	2,461	1,747	2,489	1,739	2,463
MD4	46,152	61,826	45,714	62,534	45,853	62,873
MD5	38,102	51,200	37,647	52,178	37,608	51,492
SHA-1	17,429	23,239	17,770	24,497	17,664	24,118

Table 8. Performance of Some Hash Functions Implemented as System Libraries

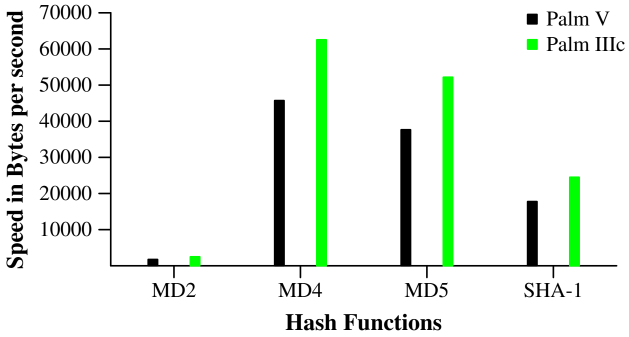

Figure 3. Performance of MD2, MD4, MD5 and SHA-1

7. Multiple Precision Integer Arithmetic Operations

Many public-key cryptosystems require computations in $\mathbb{Z}_m$, the integers modulo m where m is a large positive integer of hundreds of digits long. Besides $\mathbb{Z}_m$, other algebraic structures such as polynomial rings, finite fields and finite cyclic groups are also extensively used. It is generally noticed that efficient implementations of these operations are becoming of utmost importance in modern applied cryptography.

In this section, we describe a multiple-precision integer arithmetic system library we developed for Palm OS and evaluated its performance. The library is called Multiple-precision Integer Arithmetic Library for Palm OS (MPLib)[15]. It contains functions to perform multiple-precision integer arithmetic operations and other related algorithms. In general, there is no limit applied to the precision of the integers except the available dynamic memory implied by the memory manager of the Palm OS. MPLib has a rich set of functions, and the functions have a regular interface. A complete list of the functions currently supported by MPLib can be found in the User's Manual[16].

[15] http://www.ccs.neu.edu/home/swong/MPLib
[16] http://www.ccs.neu.edu/home/swong/MPLib/MPLibmanual.txt

To test the speeds of the MPLib algorithms, we used a similar approach to that of testing the speeds of ciphers and hash functions. Below is the pseudo-code of testing the speed of 512-bit modular multiplication ($m = ab \bmod d$ where $|a| = |b| = |d|$).

```
(t = 0; t < samples; t++);
   a = randGen(512);
   b = randGen(512);
   d = randGen(512);
   (Start Timer)
   (c = 0; c < calls; c++);
      MPLib_mod_mul(m, a, b, d);
   (Stop Timer)
```

For each sample, the testing program randomly generates three 512-bit integers a, b and d. Then the modular multiplication function of the MPLib is called for `calls` times and the time taken for doing these computations is recorded.

Table 9 shows the average speed of computing a 512-bit modular reduction for `calls`= 1024.

	Speed (msec)	Average Tick Count
Palm V	3.13	320
Palm IIIc	1.88	192

Table 9. Speed of 512-bit Modular Reduction

This is a simple multiple-precision division. Table 10 shows the speed of computing a 512-bit modular addition.

	Speed (msec)	Average Tick Count
Palm V	3.28	336
Palm IIIc	2.21	226

Table 10. Speed of 512-bit Modular Addition

We first note that the modular reduction and modular addition are relatively fast due to the simplicity of these operations. The cost of doing these operations can be ignored for most of the applications. Table 11 shows the performance of computing a 512-bit and a 1024-bit modular multiplication. From now on, the value of `calls` has been set to 32.

	512-bit Mod Mult		1024-bit Mod Mult	
	Speed (msec)	Tick Count	Speed (msec)	Tick Count
Palm V	107	341	410	1311
Palm IIIc	79	253	299	956

Table 11. Speed of Modular Multiplication

This operation has a higher cost than the modular reduction or the modular addition. About 100 msec is taken to perform one 512-bit modular multiplication which is still doable if a cryptosystem [11, 10] requires only one or two such operations. Table 12 shows the performance of computing a 512-bit modular inverse.

	Speed (msec)	Average Tick Count
Palm V	1,381	4420
Palm IIIc	998	3194

Table 12. Speed of 512-bit Modular Inverse

This is an implementation of the extended Euclidean algorithm. It has a much higher cost than any of the three operations mentioned above. On Palm V, the average time of performing one 512-bit modular inverse is over one second. When designing a cryptosystem for PalmPilot, we suggest to minimize or even eliminate any calls to the modular inverse operation for large integers. Table 13 shows the performance of computing 512-bit modular exponentiations with different magnitude of exponents.

	512-bit Exponent		Small Exponent ($\leq$ 8 bits)	
	Speed	Tick Count	Speed	Tick Count
Palm V	96.91 sec	310112	1,831 msec	5858
Palm IIIc	70.24 sec	224761	1,174 msec	3758
	Special Case 1 (exp = 3)		**Special Case 2 (exp = 65537)**	
Palm V	365 msec	2273	2,627 msec	8407
Palm IIIc	262 msec	1645	1,899 msec	6077

Table 13. Speed of 512-bit Modular Exponentiation

They were computed using the Montgomery exponentiation algorithm [2] except when the exponent is 3 where the classical repeated square-and-multiply algorithm [4] was used. Montgomery exponentiation algorithm usually computes faster than the repeated square-and-mutliply algorithm. For instance, the average speed of computing a 512-bit modular exponentiation with a 512-bit exponent is 96.91 seconds on a Palm V if Montgomery algorithm is used. However it takes 157 seconds for the same operation if the repeated square-and-multiply algorithm is used. On the other hand, for a special case when the exponent is 3, the repeated square-and-multiply algorithm only carries out one modular multiplication and one modular squaring. We found that the overhead of performing the Montgomery algorithm outweighs its subsequent simple operations when compared with the repeated square-and-multiply algorithm. The Montgomery algorithm takes an average of 710 msec in this special case while the repeated square-and-multiply algorithm only takes about half of the time. When compared with other number theoretic operations, modular exponentiation with large exponent is very expensive. While for small exponents (e.g. less than 8 bits or in a special case like exp = 3), the time taken for pure computation is significantly shorter. The results suggest that the Palm devices are only suitable for limited computation of modular exponentiation for most of the applications. For example, it may be acceptable to conduct one RSA encryption or signature verification but not the RSA decryption or signature generation. Figure 4 summarizes the performance of modular multiplication, modular inversion and modular exponentiation.

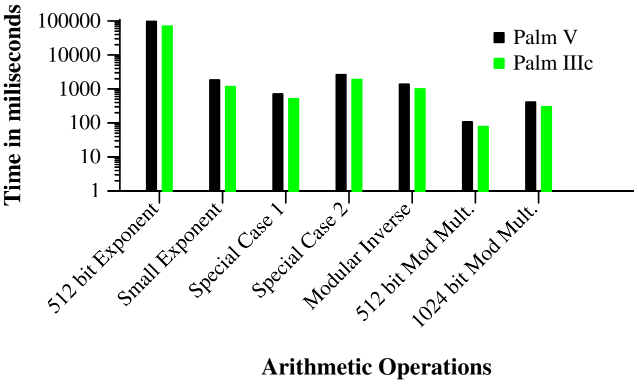

Figure 4. Performance of Modular Multiplication, Modular Inverse and Modular Exponentiation

The implementation also revealed that there is a common set of instructions such as rotational shifts, exclusive-or, and data moves, which are widely used among ciphers, hash functions and multiple-precision arithmetic operations we developed. Improvements should be made at an instructional or hardware level in order to further improve their efficiency. Furthermore, due to the limited number of registers in the microprocessor and there is no cache, most of the data requests induce direct memory access requests, each of which takes several clock cycles to complete. We believe that by reducing the number of memory access or by providing faster memory accesses, one can have a significant effect on the overall efficiency of the algorithms.

8. Suggestions and Conclusions

SSC2 gives the most efficient keystream generation and encryption speed in most of the cases. It outperforms ARC4 by 10% in most of the cases. In addition, it requires less memory space for storing the internal states than the ARC4 does. On the other hand, SEAL 3.0 is very efficient in encrypting very large data. It can be very useful to encrypt real-time data-stream if precomputation of the key setup tables is allowed or the overhead of key setup is not an issue for a target application. The drawback is that large memory chunk is needed (over 3KB) for storing T, S and R tables. This may cause problems on some other low-power devices with very limited memory space.

Block ciphers are much slower than stream ciphers. Although they are still relatively fast for small data blocks when compared with public-key encryption, our results show that stream ciphers such as SSC2 and ARC4 should be used whenever possible.

Finally, for applications which desire eminently efficient algorithms, the code of the algorithms should be put inside the applications in order to eliminate any system trap instructions. Also hand-coded optimization is always possible through a closer inspection of the assembly code derived from a compiled program.

References

[1] J. Daemen and V. Rijmen. AES Proposal: Rijndael. *AES Algorithm Submission*, Sep 1999. http://www.nist.gov/aes.

[2] Stephen R. Dussé and Burton S. Kaliski Jr. A cryptographic library for the Motorola DSP56000. In I.B. Damgård, editor, *Advances in Cryptology — Eurocrypt '90*, pages 230–244, New York, 1990. Springer-Verlag.

[3] B.S. Kaliski, Jr. *RFC 1319: The MD2 Message-Digest Algorithm*. IETF RFC 1319, Apr 1992.

[4] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press LLC, 1997.

[5] NIST FIPS PUB 180-1. *Secure Hash Standard*, Apr 1995.

[6] Ronald Rivest. *RFC 1320: The MD4 Message-Digest Algorithm*. IETF RFC 1320, Apr 1992.

[7] Ronald Rivest. *RFC 1321: The MD5 Message-Digest Algorithm*. IETF RFC 1321, Apr 1992.

[8] Phillip Rogaway and Don Coppersmith. A software-optimized encryption algorithm. *Journal of Cryptology*, 11(4), First Quarter 1998.

[9] Bruce Schneier. *Applied Cryptography : protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., second edition, 1996.

[10] Duncan S. Wong and Agnes H. Chan. Efficient and mutually authenticated key exchange for low power computing devices. to appear in Proc. of ASIACRYPT 2001, Dec 2001.

[11] Duncan S. Wong and Agnes H. Chan. Mutual authentication and key exchange for low power wireless communications. to appear in IEEE MILCOM 2001 Conference Proceedings, Oct 2001.

[12] Muxiang Zhang, Christopher Carroll, and Agnes H. Chan. The software-oriented stream cipher SSC2. *Fast Software Encryption Workshop 2000*, 2000.