Fast Software Encryption: Designing Encryption Algorithms for Optimal Software Speed on the Intel Pentium Processor

Bruce Schneier
Counterpane Systems
101 E Minnehaha Parkway
Minneapolis, MN 55419
schneier@counterpane.com

Doug Whiting
Stac Electronics
12636 High Bluff Drive
San Diego, CA 92130
dwhiting@stac.com

Abstract. Most encryption algorithms are designed without regard to their performance on top-of-the-line microprocessors. This paper discusses general optimization principles algorithms designers should keep in mind when designing algorithms, and analyzes the performance of RC4, SEAL, RC5, Blowfish, and Khufu/Khafre on the Intel Pentium with respect to those principles. Finally, we suggest directions for algorithm design, and give example algorithms, that take performance into account.

1 Overview

The principal goal guiding the design of any encryption algorithm must be security. In the real world, however, performance and implementation cost are always of concern. The increasing need for secure digital communication and the incredible processing power of desktop computers make performing software bulk encryption both more desirable and more feasible than ever.

The purpose of this paper is to discuss low-level software optimization techniques and how they should be applied in the design of encryption algorithms. General design principles are presented that apply to almost all modern CPUs, but specific attention is also given to relevant characteristics of the ubiquitous Intel X86 CPU family (e.g., 486, Pentium, Pentium Pro). Several well-known algorithms are examined to show where these principles are violated, leading to sub-optimal performance. This paper concerns itself with number of clock cycles per byte encrypted—given a basic encryption algorithm "style." Factors of two, three, four, or more in speed can be easily obtained by careful design and implementation, and such speedups are very significant in the real world.

In the past, cryptographers have often designed encryption algorithms without a full appreciation of low-level software optimization principles associated with high-performance CPUs. Thus, many algorithms have unknowingly incurred significant performance losses which apparently could have been avoided at the design stage without impairing security. The goal of this paper to encourage greater awareness of these performance issues, and to encourage dialogue

between cryptographers and programmers at the algorithmic design stage.

2 General Optimization Principles

By its nature, almost every known encryption algorithm includes a small inner loop where the vast majority of all processing time is spent. This inner loop is a natural candidate for assembly language implementation to achieve ultimate speed. With today's advanced CPU architectures, such optimization has become as much art as science. It is true that optimizing C compilers can produce very fast code in many cases. However, without understanding the architecture of the underlying CPU, it is easy to write C code which is apparently efficient but which cannot be compiled to utilize the CPU completely, particularly for such tight inner loops and for "register poor" CPUs (e.g., Intel Pentium). Lest cryptographers in particular feel slighted, note that these observations apply equally well to software optimization outside of cryptography; i.e., there are many high-level language programmers who do not understand how to take full advantage of a CPU.

The ensuing discussion needs to be prefaced with an important caveat: the fact that a particular operation is expensive in terms of CPU clock cycles does not mean a priori that the operation should not be used in an encryption algorithm. The associated slowdown may be deemed acceptable because of additional security afforded thereby, but the algorithm designer should be aware that such a tradeoff is being made.

Ideally, it is desirable to express an algorithm in a high-level language (e.g., C) in such a way that it can be compiled optimally to all target CPU platforms. Due to dramatic variances in CPU architectures and in the quality of compiler code generation, this ideal can rarely be achieved in practice; however, an understanding of how a CPU executes assembly language instructions helps the programmer comprehend how a given set of lines of code will run. There are purists who believe that optimizing compilers totally obviate the need for programming in assembly language. This is largely true, except in the exact case of interest here: designing an algorithm to wring every last ounce of performance out of a CPU. The compiler's job is in fact to translate high-level statements into assembly language, so the algorithm designer must understand (at least roughly) what the resulting compiled code will look like and how it will execute in order to achieve optimal speed. Alternately, the designer (or an associate) may code the algorithm directly in assembly language to see where the performance roadblocks are encountered.

Many modern CPUs have at their core a Reduced Instruction Set Computer (RISC) architecture. Basically, this means that most simple operations (memory load/store, register-to-register add/subtract/XOR/or/and, etc.) execute in one clock cycle. Even high-end CPUs with a Complex Instruction Set Computer (CISC) heritage, such as the Intel 486/Pentium and the Motorola 68K family, have migrated to a modified RISC approach, with only the more complex (and

less frequently used) operations requiring multiple clocks. In other words, the RISC vs. CISC war is over, and RISC has won by assimilation. Even many embedded microcontrollers today use a RISC architecture, and this discussion applies equally well to them. To maximize performance, it is generally recommended that most instructions be selected from the RISC "subset." In order to execute assembly language instructions in a single clock, multiple levels of pipelining are used inside the CPU; that is, each instruction actually executes in several stages, requiring multiple clocks, but on each clock one instruction is completed. The more complex instructions (e.g., integer multiply or divide) can "stall" the pipeline, backing up the execution of subsequent instructions. Thus, it behooves the algorithm designer to know exactly which instructions are in the RISC subset. For example, on the Pentium processor, 32-bit integer multiplication (as in IDEA) requires 10 clocks[1], and from 13–42 clocks on the Intel 486.

Another important architectural improvement in high-end CPUs is superscalar execution; more than one instruction can be executed in a single clock cycle. For example, the Pentium processor can execute up to two RISC instructions per clock, in two separate pipelines, so careful design to allow complete instruction "pairing" on the Pentium can result in a doubling of speed. Such additional pipelines generally require even further lookahead in the instruction stream and can significantly increase the relative cost of any pipeline stalls. In addition, only certain types of instructions can be executed in parallel, which can introduce an extra (often non-intuitive) level of dependency between consecutive instructions or lines of C code in order to utilize fully the superscalar processing. Given two instructions, if the second one depends on the result of the first, the two cannot be executed in parallel. Thus, while superscalar architectures offer the possibility of substantial speed increases, they also dramatically increase the penalty for improper design.

As future CPUs achieve higher levels of superscalar execution, the rules for optimization will certainly become more complex. For example, the Pentium Pro can execute up to three instructions per clock. Almost all modern RISC CPU families (e.g., PowerPC, DEC Alpha, SUN Sparc) include processors of varying cost/performance levels, some of which are superscalar and some of which are not. Some of the principles discussed below do not apply to the non-superscalar CPUs, but it is assumed here that the algorithms of interest are intended to run well on a broad class of processor types. Fortunately, it has been empirically observed that, in most cases, an algorithm optimized for a high-end superscalar processor (e.g., Pentium) is also very close to optimal on a lower-end CPU (e.g., 486),[2] although unfortunately the converse is not usually true. The techniques discussed here should continue to apply in general, and high degrees of independence and parallelism should be included in a design to attempt to take full advantage of such future architectures

[1] It is actually one clock cycle slower to do a 16-bit multiply on the Pentium.
[2] Later we will see that RC4 is a counterexample.

Following are some general issues to be considered in designing an algorithm:

- Avoid conditional jumps in the inner loop. Any unpredictable change of the flow of control in the algorithm will normally flush the pipeline and cost extra clock cycles. In particular, any if/then/else or the C operator "?" will cause assembly language jump instructions to be executed. Sometimes table lookup can be used to remove the need for a jump without causing a pipeline flush. Jumps also increase vulnerability to timing attacks [Koc96].
- Unroll loops. Consider performing several iterations of the inner loop "inline" without looping, thus removing a jump (and a test for completion) at each iteration. This issue actually deals with implementation, not algorithmic design. However, at the design stage, it is useful to realize that the programmer implementing the algorithm will probably unroll the loop at least to a certain extent, so it is critical to understand any dependencies between the last instructions of one iteration and the first instructions of the next that may cause pipeline stalls.
- Avoid intrinsically expensive instructions. Among this category are multiplies, divides, and any other instructions which are slow on the processor(s) of interest. For example, on a Pentium, a variable rotate/shift instruction (i.e., where the rotation/shift amount is not known a priori) requires four clocks and cannot be paired with any other instruction; thus, the variable rotate alone costs as much as eight RISC subset instructions.
- Limit the number of variables. Many RISC processors have a fairly large general purpose register set (at least sixteen registers), but the Pentium has only seven general purpose registers. If too many variables are used extensively in the inner loop, they cannot fit in registers, so the performance will suffer greatly due to excessive memory accesses. Any memory access other than load/store (e.g., memory-to-register or register-to-memory add) will require multiple clocks.
- Limit table size. Although larger tables are better from a cryptographic standpoint, smaller is definitely better from a software speed standpoint. It is tempting to use large tables for S-boxes, but if those tables do not fit into the CPU's on-chip data cache, a substantial performance degradation may be incurred. In general, tables should be limited to no more than 4K bytes for today's CPUs.
- Allow parallelism. The general idea is to allow as many independent operations to be executed as possible. Unfortunately, this principle is often in direct opposition to the generally desirable goal of maximizing the "cascading" of bits, so there is usually a design tradeoff involved. From a high-level language, this guideline can be checked by breaking the code down into RISC subset operations and then seeing how the operations could be rearranged to perform as many as possible in parallel at each point. As an example, assuming that the variables a, b, and c all fit in registers, the C statement "`a = (a + b) ^ c;`" would be broken down into the two RISC statements "`a += b; a ^= c;`" In this case, notice that the second operation cannot be performed in parallel with the first. After breaking the statements down

to this level, the rearranged code often looks very unlike the original code, but this analysis is key to achieving high speed on superscalar architectures. On the Pentium and many of today's other superscalar CPUs, the goal is to have exactly two operations paired together at all times.

– Allow setup time for table indexing. This guideline is somewhat subtle and seems to be violated frequently in existing algorithms. When accessing tables (e.g., S-boxes) based on a function of the algorithm's variables, the table index must be computed as far ahead as possible. For example, on the 486, if an address is computed in a register on one cycle, and that register is used on the ensuing cycle to access memory, a one cycle penalty is incurred. On the Pentium, things get worse due to superscalar operation: the address should be computed at least two instructions ahead of when it is to be used; otherwise a one clock penalty is incurred. For example, the deceptively simple C statement "`y = S[x & 0xFF];`" on a Pentium would require at least two other instructions to be inserted between the "&" operation and the table access to utilize the CPU fully. Note that this address setup time is required regardless of whether the data is in the on-board CPU data cache.

These guidelines are fairly simple, but the performance gains resulting from following them can be quite impressive.

3 Analysis of Known Algorithms

In this section, several well-known encryption algorithms are analyzed in terms of the guidelines presented above. This scrutiny is not intended to be critical of the algorithms in question, but rather to illustrate the general principles with concrete examples. Only key portions of the inner loops of the algorithms are examined here.[3] The same principles could be applied to other portions of the algorithms (e.g., initialization), but the performance improvement is minimal. For each algorithm, some brief comments are also included about suitability of the algorithm for direct hardware implementation.

3.1 RC4

The inner loop of RSA's stream cipher RC4 [Sch96] is shown in C below:

```
unsigned char S[256];     /* initialized based on keys */

void RC4encrypt(unsigned char *p,int cnt)
    {
```

[3] It should be noted that most of the timings discussed below are approximate and are intended only to point out general performance issues; the only guaranteed way to obtain exact performance numbers is to implement and time the algorithm on the given platform.

```
int i,j,t;
unsigned char tmpI,tmpJ;

for (i=j=0;cnt;cnt--,p++)
    {
    i=(i+1) & 0xFF;                /* update i */
    tmpI=S[i];
    j=(j+tmpI) & 0xFF;             /* update j */
    tmpJ=S[j];
    S[j]=tmpI;              /* swap S[i], S[j] */
    S[i]=tmpJ;
    t=(tmpI+tmpJ) & 0xFF;
                   /* compute ‘‘random’’ index */
    *p ^= S[t];     /* XOR keystream into data */
    }
}
```

Entries in the S-box array $S[\,]$ each consist of 8 bits and are initialized as a permutation of 0..255 based on the encryption key. In most assembly language implementations of RC4, the variables i and j are kept in 8-bit registers, so there is no need to mask with `0xFF` as shown in the C code. This code compiles fairly efficiently, but there are some dependencies which limit performance on high-performance CPUs.

For example, j cannot be updated without first fetching $S[i]$, but only after i has been updated. The update of i, if performed literally as indicated, would cause an pipeline stall because of the setup time required for the table indexing. In a fully optimized (unrolled) version of this code, the increment of i would probably be deferred, only being computed once every 4 or 8 iterations; e.g., on every eighth iteration, i would be incremented by 8. In this case, the `tmpI` value would be loaded in a single instruction from $S[i + N]$, where N is a constant dependent on the "unroll" position. This method works around the address setup time, although a special "wrap" case has to be handled when i approaches 255. In any case, the fact that $S[j]$ is accessed immediately after updating j almost certainly results in a clock penalty in any implementation. A similar penalty probably applies for t and $S[t]$, although, with appropriate overlapping of the next iteration, this penalty may be avoided on some CPUs.

As an aside, a very surprising empirical result illustrates the difficulty in projecting optimization techniques onto future processors. A fully unrolled and optimized assembly-language version of RC4 has been measured as encrypting one byte roughly every seven clocks (averaged over a large block) on a Pentium, using byte registers for all variables to avoid the need for masking with `0xFF`. For example, a 150 MHz Pentium can encrypt with RC4 at over 20 Mbytes/sec. However, the Pentium Pro processor contains an obscure internal pipeline stall which occurs when a register is accessed as a 32-bit quantity (e.g., `EAX`) soon after modifying an 8-bit subset (e.g., `AL`) of that register. The penalty in this

case is a whopping six clocks (or more); as a result, the optimized Pentium code runs on a Pentium Pro at less than half the speed of a Pentium of the same frequency. One simple workaround in this case is to use 32-bit entries in $S[\,]$ and for all variables, necessitating the mask with `0xFF` and quadrupling the table size. Such a change is fairly simple, but the resulting code is no longer completely optimal on a Pentium.

The general performance problem with RC4 as designed is that almost every statement depends immediately on the statement before it, including the table index computation and the associated table accesses, limiting the amount of parallelism achievable. Fortunately, most (although not all) of these dependencies can be worked around without penalty in a fully unrolled optimized loop. It is extremely unlikely that an optimizing compiler would come close to such optimizations unless the C code was considerably unrolled, since the optimizations include carefully overlapping the start of the next iteration with the end of the previous one.

From a design standpoint, it would have been better to update i at the end of the loop to simplify unrolling. Even better would have been to use the previous value of j for the swap while updating the current value of j, thus allowing a full iteration for lookahead computation. It would probably also help if t could be computed at the beginning of the loop, or a previous value of t used, in order to allow maximum address setup time. Such changes typically cost nothing with appropriate register assignments, but they do allow much greater parallelism. It is possible that some of these proposed design changes might raise security issues, particularly with the initial few keystream bytes. If so, the inner loop should be run a few times as part of initialization, discarding the keystream value $S[t]$, to get rid of any such startup problems, still resulting in a significant speedup for the bulk of the encryption. Such changes probably would not have affected the overall security of the algorithm, although no cryptanalysis has been performed to verify this supposition.

Note that an RC4 implementation in hardware requires only 256 bytes of RAM, which is quite reasonable. However, each iteration also requires at least five RAM accesses (three reads, two writes), limiting the speed to no fewer than five clocks per byte, unless a dual-ported RAM is used, which is costly in terms of chip area. By contrast, a DES chip can easily run at two clocks per byte, even with each round implemented sequentially.

3.2 SEAL

SEAL [RC94] is a fast software stream cipher designed by Phil Rogaway and Don Coppersmith, and patented by IBM [CR95]. The inner loop uses four 32-bit registers, as well as 32-bit tables T with 512 entries and S with 256 entries. The table contents (more than 3K bytes) are initialized based on the encryption key in a fairly computationally intensive operation, so the algorithm does not seem to be appropriate if the key needs to change frequently. The relevant portion of the inner loop is given below:

```
long        T[512],S[256],N[4]; /* initialize based on keys */

void SEALencrypt(void)
    {
    int j;
    long a,b,c,d,p,q;

    /* some setup here */

    for (j=0;j<64;j++)
        {
        p =   a  & 0x7FC; b += T[p/4]; a=ROT9(a); b ^= a;
        q =   b  & 0x7FC; c ^= T[q/4]; b=ROT9(b); c += b;
        p =(p+c) & 0x7FC; d += T[p/4]; c=ROT9(c); d ^= c;
        q =(q+d) & 0x7FC; a ^= T[q/4]; d=ROT9(d); a += d;

        p =(p+a) & 0x7FC; b ^= T[p/4]; a=ROT9(a);
        q =(q+b) & 0x7FC; c += T[q/4]; b=ROT9(b);
        p =(p+c) & 0x7FC; d ^= T[p/4]; c=ROT9(c);
        q =(q+d) & 0x7FC; a += T[q/4]; d=ROT9(d);

        /* at this point, output:
        b+S[4*j], c^S[4*j+1], d+S[4*j+2], a^S[4*j+3]  */

        a+=N[2*(j&1)]; /* easily handled by unrolling */
        c+=N[2*(j&1) + 1];
        }
    }
```

The indexing with $p/4$ or $q/4$ is actually simple, since the index is multiplied by 4 (i.e., sizeof(long)). The major problem with this algorithm is the strong dependence between consecutive operations, allowing only minimal parallelism; in other words, on the Pentium, it is rare that both pipelines are utilized here. In addition, the table access (e.g., $T[p/4]$) follows immediately after the computation of the table index (p or q), thus incurring a clock penalty on a Pentium (and probably on most other RISC CPUs). For example, each of the first four lines of the inner loop involve a register-to-register move/add, a mask operation, a memory-to-register load from the table, a register-to-register add/XOR, a fixed rotate, and a register to register XOR/add. On a Pentium, with perfect pipelining and no address setup penalties, these four lines could ideally execute in sixteen clocks. It should be noted that the official Intel Pentium documentation claims that the opcode for rotation by a constant amount (e.g., 9 bits) can pair with another instruction, suggesting an ideal time of twelve clocks. However, it has been determined empirically [4] that only rotations by one bit can pair, thus cost-

[4] See http://www.geocities.com/SiliconValley/9498/p5opt.html.

ing extra clocks. This anomaly underscores the importance of always checking the actual performance against the theoretical performance.

Instead, because of the problems mentioned above, these four lines actually require 24 clocks. By contrast, on a 486, the "ideal" speed of 28 clocks is actually achieved. The second set of four lines can be executed in 20 Pentium clocks, as opposed to the ideal of 10, while the 486 incurs no penalties and runs in 24 clocks. Obviously, this algorithm was designed with the 486 in mind, not the Pentium. In fact, at the same clock frequency, a Pentium is only slightly faster than a 486 at running SEAL. A fully optimized Pentium version should be able to encrypt at about 4 clocks per byte, which is quite a bit faster than RC4.

To maximize Pentium throughput, SEAL could be redesigned to stagger the usage of variables so that consecutive instructions can run in parallel. For example, q should generally be used as an index immediately after computing p (and vice versa), and the final statement of each of the first four lines (e.g., `b ^= a`) should modify a different variable than that being modified by the table entry (e.g., `d ^= a`, etc.). Such changes cannot be made without considerable thought and analysis, but it appears that minor modifications could produce an algorithm very similar in spirit (and hopefully in security) at nearly twice the speed on a Pentium, without slowing down the algorithm on a 486. It should also be noted that preserving the values of p and q across iterations of j should not incur any cost and would likely increase security somewhat.

SEAL does not appear very attractive for hardware implementation, principally because of the large size of the tables. A hardware implementation could fairly easily run at about one byte output every clock, which is faster than a simple DES implementation, but there is no obvious way to use pipelining to obtain much higher speed for SEAL, as is possible with DES.

3.3 RC5

RC5 [Riv95] is a Feistel-network block cipher (patent pending by RSA Data Security, Inc.). RC5 gets its strength from data-dependent rotations, which were first discussed in [Mad84]. The inner loop is shown in C below, where "<<<" indicates circular rotation:

```
long        S[2*ROUNDS+2];  /* initialized based on keys */
long        A,B;               /* values to be encrypted */

void RC5encrypt(void)
    {
    int j;

    A+=S[0];
    B+=S[1];
    for (j=1;j<ROUNDS;j++)
```

```
        {
        A = ((A^B) <<< B) + S[2*j];
        B = ((B^A) <<< A) + S[2*j+1];
        }
    }
```

The algorithm looks simple and elegant. The recommended value for ROUNDS is 16. Observe that, since all table indices depend only on j, no table access penalties should occur in an unrolled version. At first glance, this algorithm should run very fast.

Unfortunately, on a Pentium processor, the cost of a variable rotation is very high (four clocks). Even worse, variable rotations cannot pair with any other instructions on the Pentium. Each round involves two memory loads, two adds, two XORs, and two rotations (which on the Pentium include an instruction to move the rotation amount in the `CL`). Thus, on a Pentium, assuming perfect pairing other than for the rotations, each round will require at least twelve clocks, whereas on an ideal superscalar RISC CPU the number would be only six clocks per round, given the order dependencies of the operations. At sixteen rounds, RC5 should encode at about 24 clocks per byte on a Pentium.[5] While this is fast compared to most block ciphers, it is disappointingly slow compared to what might be initially expected, again pointing out the danger in estimating the speed of an algorithm from a high-level language without understanding the underlying processor limitations. On a Pentium Pro, these pipeline problems appear to have been removed, so RC5 should run at or near its impressive theoretical software speed limit of six clocks per round, or 12 clocks per byte at sixteen rounds.

In hardware, RC5 should be able to execute one round in two (or perhaps four) clocks, thus giving a speed which is comparable to DES. However, it should also be noted that a full 32-bit variable rotation engine is expensive in hardware.

3.4 Blowfish

Blowfish [Sch94] is also a Feistel-network block cipher. It uses four tables, each consisting of 256 entries of 32-bits each, plus a separate table of eighteen 32-bit entries, for a total of more than 4K bytes. The table entries are initialized based on the keys, with a fairly large setup time, severely limiting the algorithm's usefulness if the key needs to be changed frequently. Here is the code for the inner loop of Blowfish:

```
long  P[18], S[4][256];   /* initialized based on keys */
long  A,B;                   /* values to be encrypted */

void BlowfishEncrypt(void)
    {
```

[5] RSADSI, Inc. has reference code that encrypts at 23 clock cycles per byte.

```
int j;

for (j=0;j<16;j++)                    /* 16 rounds */
    {
    A ^= P[j];
    B ^= ((S[0][ A & 0xFF] + S[1][(A >> 8) & 0xFF])
        ^ S[2][(A >> 16) & 0xFF])
        + S[3][(A >> 24) & 0xFF];
    swap(A,B);               /* interchange values */
    }
/* some final operations here */
}
```

The algorithm looks simple, although the function of A computed using the S-boxes is slow because of the need to extract four bit fields and load four table entries. Each round can be executed on a Pentium in about 9 clocks, and there are no address setup penalties, but there is a pipeline stall of one clock due to the sequential nature of combining the S-box outputs, which could be alleviated by using xors to combine all the S-box entries. Using sixteen rounds, this equates to a throughput of about 18 clocks per byte.[6]

Blowfish is not attractive in dedicated hardware because of the large size of the tables involved. A simple implementation with sequential access to the four S-boxes would require about five clocks per round, while a version with parallel access could easily execute in two clocks per round, or four clocks per byte. The latter approach would almost certainly require on-chip RAM, making the cost quite high, and it is still considerably slower in hardware than a low-end DES hardware solution.

3.5 Khufu/Khafre

These algorithms, proposed by Ralph Merkle [Mer91a] and patented by Xerox [Mer91b], have some similarities to Blowfish, but each round does less, involving only a single S-box lookup, and thus is faster.

```
long  S[256];     /* Khufu: key dependent; Khafre: not */
long  A,B;                   /* values to be encrypted */

void Khufu(void)
    {
    int j;
    long tmp;
```

[6] CAST [AT93, Ada94] (designed by Carlisle Adams and patented by Northern Telecom [Ada96]) has a round function almost idential to Blowfish's. There are several variants of CAST; the current one seems to be a bit slower than Blowfish.

```
for (j=0;j<ROUNDS;j++)
    {
    tmp = B ^ S[A & 0xFF];
    B   = A <<< 8;
    A   = tmp;
    }
}
```

Because each round is so simple, there is no opportunity to pipeline the lookups completely. Thus, each round consists of five instructions and requires five Pentium clocks and the speed is 20 clocks per byte for 32 rounds. However, this time could be halved to 2.5 clocks per round if two independent blocks are encrypted simultaneously, which makes sense for encrypting a stream of blocks in ECB or for interleaving two independent feedback streams: 10 clocks per byte. The Khafre algorithm is similar to Khufu, but the S-box is not key dependent, and the key is XORed into the encryption block after every eight rounds, thus slowing it down only slightly from Khufu. These algorithms can be implemented fairly simply in hardware at one clock per round.

3.6 IDEA and DES

As a point of comparison, it is useful to present a brief discussion of the other block algorithms, not designed for software.

IDEA [LMM91, ML91] uses a total of eight rounds, following by a final transformation. Each round includes six loads of 16-bit subkeys, four multiplies, four adds, and six XORs. On a Pentium, which has a dedicated hardware multiplier that can complete a multiply in 10 clocks (although the multiply instruction cannot pair with any other instruction), each round executes in about 50 clocks, so the entire algorithm requires more than 400 clocks per block, or over 50 clocks per byte. There are no address setup penalties, and any penalties due to lack of parallelism are swamped by the multiplication time. It should be noted that the Pentium Pro can perform integer multiplies at a throughput of one multiply per clock, but with a latency of four clock cycles for each result. Thus, IDEA should run considerably faster (perhaps by a factor of three or four) on a Pentium Pro than on a Pentium.

Because there are no sizable tables involved, IDEA in hardware might be appealing, but multipliers are not inexpensive. The speed of hardware would depend dramatically on how much cost could be absorbed. Given enough dedicated multipliers, each round could be implemented in only a few clocks, giving speeds comparable to DES.

Estimated performance numbers for DES [NBS77] will also be helpful as a point of reference. These performance numbers are derived from a rather conventional approach to DES software implementation, using eight S-box tables of 64

entries each, including an assumption that the key can be pre-processed to build tables that speed implementation. It is quite possible that a different approach, perhaps involving much larger tables, could speed implementation considerably.

The initial and final permutations do not appear to allow for much parallelism but should each execute in less than 35 clocks on a Pentium. Each round of the algorithm can be executed in about 18 clocks (this assumes you do two DES S-box lookups in a single 12-bit table), with very good pipeline utilization on a Pentium. Thus, the overall time for a DES block to be encrypted is about 360 Pentium clocks, leading to a throughput of about one byte every 45 clocks. Similar calcuations for triple-DES give a throughput of one 108 clocks per byte.

3.7 Comparison of Results

The following table summarizes the results of the previous sections, with all clock counts being given for a Pentium processor:

Algorithm	Type	Clocks/round	# rounds	Clocks/byte of output
RC4	Stream cipher	n.a.	n.a.	7
SEAL	Stream cipher	n.a.	n.a.	4
Blowfish	Block cipher	9	16	18
Khufu/Khafre	Block cipher	5	32	20
RC5	Block cipher	12	16	23
DES	Block cipher	18	16	45
IDEA	Block cipher	50	8	50
Triple-DES	Block Cipher	18	48	108

4 Proposed Directions for Further Investigation

Having dissected several well-known algorithms, it is hoped that the general design principles for software optimization are now fairly clear. Up to this point, however, only minor changes to the structure of existing algorithms have been considered. This section presents a few basic structures for consideration in future algorithm design. In a sense, the proposed idea is to use "RISC" encryption: lots of rounds, each designed to execute very efficiently on a high-end CPU. Each round consists of simple transformations that are extremely efficient on today's popular CPU architectures: add, subtract, XOR, rotate by a constant amount, and table lookup (i.e., S-boxes). None of the particular algorithms outlined here has been seriously cryptanalyzed, but it is hoped that these proposals can serve as a springboard for further investigation and research.

Below is sample source code for a 128-bit block encryption algorithm called "Test 1." In each round, one of the four 32-bit words is updated, including a round-dependent rotation constant (which may also be key-dependent) and an S-box lookup using bits from another of the words. The rotation constants

are chosen within carefully designed constraints, as shown in the comments, to maximize diffusion of bits. For example, in the first round, note that $X3$ is merged into $X0$ both before and after the rotation. With properly selected rotation constants, each bit of $X3$ will affect each bit of $X0$ by the end of round 5, even ignoring the S-box lookups, and thus every bit of all three other words after seven rounds. In general, this "exponential" bit diffusion propagates a single bit change into 32 positions in five rounds, with the S-box lookups providing non-linearity. Observe that, for each set of five rounds, there are 1024 possible sets of rotation constants. After each ten rounds, a few rounds of a substantially different round function are inserted to inject some irregularity into the algorithm. The use of variables in the main round function is carefully chosen to allow full-speed encryption and decryption without pipeline stalls.

Each round can be performed in 3 Pentium clocks, with the intervening rounds each requiring 8 clocks each. With thirty rounds plus two intervening rounds, the entire block can be encrypted in about 9 clocks per byte. If the rotation constants are key-dependent, achieving this speed on a Pentium would require "compiling" a version of the code for each key, which can be easily accomplished as part of key initialization schedule.

```
                  /*  2a+1,4b+2,8c+4,16d+8,16            */
                  /*    ( --> exponential bit diffusion)  */
#define int ROTCNT[30]= {    1,   2,   4,    8, 16,
                            17,  26,  20,   24, 16,
                            11,   6,  12,    8, 16,
                            19,  14,  24,    8, 16,
                             5,  22,  12,   24, 16,
                            29,  18,  20,   24, 16
                     };
/*  Randomly generated from keys.  Every five rounds  */
/*  has 10 bits of freedom in ROTCNT[] values.        */


long S[1024];  /*  Randomly generated from keys  */
long P[8];     /*  Randomly generated from keys  */
long R[8];     /*  Randomly generated from keys  */
               /*              (five bits each)  */

void Test1Encrypt(long *X0,long *X1,long *X2,long *X3)
 {
    /*  could start out with some input whitening here  */

    /*  ten initial rounds  */
    X0 = ((X0 + X3) <<< ROTCNT[ 0]) ^ (S[X2 & MASK] + X3);
    X1 = ((X1 + X0) <<< ROTCNT[ 1]) ^ (S[X3 & MASK] - X0);
    X2 = ((X2 - X1) <<< ROTCNT[ 2]) ^ (S[X0 & MASK] + X1);
    X3 = ((X3 - X2) <<< ROTCNT[ 3]) ^ (S[X1 & MASK] - X2);
```

```
X0 = ((X0 + X3) <<< ROTCNT[ 4]) ^ (S[X2 & MASK] + X3);
X1 = ((X1 + X0) <<< ROTCNT[ 5]) ^ (S[X3 & MASK] - X0);
X2 = ((X2 - X1) <<< ROTCNT[ 6]) ^ (S[X0 & MASK] + X1);
X3 = ((X3 - X2) <<< ROTCNT[ 7]) ^ (S[X1 & MASK] - X2);

X0 = ((X0 + X3) <<< ROTCNT[ 8]) ^ (S[X2 & MASK] + X3);
X1 = ((X1 + X0) <<< ROTCNT[ 9]) ^ (S[X3 & MASK] - X0);

/*  pause after ten rounds to do something different  */
X0 =  (X0 << R[0]) ^ P[0];
X1 = ((X1 << R[1]) ^ P[1]) + X0;
X2 = ((X2 << R[2]) ^ P[2]) + X1;
X3 = ((X3 << R[3]) ^ P[3]) + X2;

/*  ten middle rounds  */
X2 = ((X2 - X1) <<< ROTCNT[10]) ^ (S[X0 & MASK] + X1);
X3 = ((X3 - X2) <<< ROTCNT[11]) ^ (S[X1 & MASK] - X2);

X0 = ((X0 + X3) <<< ROTCNT[12]) ^ (S[X2 & MASK] + X3);
X1 = ((X1 + X0) <<< ROTCNT[13]) ^ (S[X3 & MASK] - X0);
X2 = ((X2 - X1) <<< ROTCNT[14]) ^ (S[X0 & MASK] + X1);
X3 = ((X3 - X2) <<< ROTCNT[15]) ^ (S[X1 & MASK] - X2);

X0 = ((X0 + X3) <<< ROTCNT[16]) ^ (S[X2 & MASK] + X3);
X1 = ((X1 + X0) <<< ROTCNT[17]) ^ (S[X3 & MASK] - X0);
X2 = ((X2 - X1) <<< ROTCNT[18]) ^ (S[X0 & MASK] + X1);
X3 = ((X3 - X2) <<< ROTCNT[19]) ^ (S[X1 & MASK] - X2);

/*  pause after ten rounds to do something different  */
X0 =  (X0 << R[4]) ^ P[4];
X1 = ((X1 << R[5]) ^ P[5]) + X0;
X2 = ((X2 << R[6]) ^ P[6]) + X1;
X3 = ((X3 << R[7]) ^ P[7]) + X2;

/*  ten final rounds  */
X0 = ((X0 + X3) <<< ROTCNT[20]) ^ (S[X2 & MASK] + X3);
X1 = ((X1 + X0) <<< ROTCNT[21]) ^ (S[X3 & MASK] - X0);
X2 = ((X2 - X1) <<< ROTCNT[22]) ^ (S[X0 & MASK] + X1);
X3 = ((X3 - X2) <<< ROTCNT[23]) ^ (S[X1 & MASK] - X2);

X0 = ((X0 + X3) <<< ROTCNT[24]) ^ (S[X2 & MASK] + X3);
X1 = ((X1 + X0) <<< ROTCNT[25]) ^ (S[X3 & MASK] - X0);
X2 = ((X2 - X1) <<< ROTCNT[26]) ^ (S[X0 & MASK] + X1);
X3 = ((X3 - X2) <<< ROTCNT[27]) ^ (S[X1 & MASK] - X2);
```

```
X0 = ((X0 + X3) <<< ROTCNT[28]) ^ (S[X2 & MASK] + X3);
X1 = ((X1 + X0) <<< ROTCNT[29]) ^ (S[X3 & MASK] - X0);
}
```

Below is a stream cipher called "Test2." Again, the cryptographically appropriate number of rounds and the size of the S-box are not known. Each round consists of a fixed rotate, a table lookup, and addition, and an XOR. The indices into the S-box table ($r[0]$ and $r[1]$) are continually updated, and use of the two values alternates between rounds to avoid address setup penalties. In each round, two of the 32-bit variables are used to update the third variable, and then a three-way "swap" occurs. By unrolling this loop appropriately, no instructions are ever executed to effect the swap. After each set of ROUNDS inner loops, a single 32-bit quantity is output. Since only partial information on the internal variables is output each time (i.e., 32 out of 96 bits), it is assumed that a much smaller value of ROUNDS is acceptable, perhaps in the range of 2–6. Again, the inner loop can execute in only four clocks per round on a Pentium; thus, with ROUNDS set to three, the algorithm can output a byte in less than every three clocks, more than double the speed of RC4.[7]

```
long S[SBOX_SIZE];        /* initialized based on key */
long X,Y,Z;               /* initialized based on key */

void Test2Encrypt (long *p, long cnt)
    {
    long i,j,tmp;
    int r[2];
    r[0]=r[1]=0; /* could initialize based on keys also */

    for (i=0;i<cnt;i++,p++)
        {
        for (j=0;j<ROUNDS;j++)
            {
            r[j & 1] = (r[j & 1] + X) & (SBOX_SIZE-1);
            tmp = ((Z <<< ROTCNT) + Y) ^ S[r[j&1]];
            Z = X;
            X = Y;
            Y = tmp;
            }
        r[0] ^= r[1];
        *p ^= (X+S[r[0]]);              /* output */
        S[r[0]] = Y;              /* update S box */
        }
    }
```

[7] If the Pentium behaved as documented, this would be faster by one clock per round.

Clearly there are many possible variations on these themes, including varying the rotation amount as a function of the round number, changing which arithmetic and logical operations are performed and in which order, etc.

5 Conclusion

The nature of encryption algorithms is that, once any significant amount of security analysis is done, it is very undesirable to change the algorithm for performance reasons, thereby invalidating the results of the analysis. Thus, it is imperative to consider both security and performance together during the design phase. While it is impossible to take all future computer architectures into consideration, an understanding of general optimization guidelines, combined with exploratory software implementation on existing architectures to calibrate performance, should help achieve higher speed in future encryption algorithms. The authors believe that it is possible to develop secure stream ciphers that encrypt data at two clocks per byte, and secure block ciphers that encrypt data at ten clocks per byte. Certainly security is most important when designing an encryption algorithm, but don't go out of your way to make it inefficient.

References

[Ada94] C.M. Adams, "Simple and Effective Key Scheduling for Symmetric Ciphers," *Workshop on Selected Areas in Cryptography—Workshop Record*, Kingston, Ontario, 5–6 May 1994, pp. 129–133.

[Ada96] C.M. Adams, "Symmetric cryptographic system for data encryption," U.S. patent 5,511,123, 23 Apr 1996.

[AT93] C.M. Adams and S.E. Tavares, "Designing S-Boxes for Ciphers Resistant to Differential Cryptanalysis," *Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography*, Rome, Italy, 15–16 Feb 1993, pp. 181–190.

[BGV96] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Fast Hashing on the Pentium," *Advances in Cryptology—CRYPTO '96*, Springer-Verlag, 1996, pp. 298–312.

[CR95] D. Coppersmith and P Rogaway, "Software-efficient pseudorandom function and the use thereof for encryption," U.S. patent 5,454,039, 26 Sep 1995.

[Koc96] P. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," *Advances in Cryptology—-CRYPTO '96*, Springer-Verlag, 1996, pp. 104–113.

[LMM91] X. Lai, J. Massey, and S. Murphy, "Markov Ciphers and Differential Cryptanalysis," *Advances in Cryptology—CRYPTO '91*, Springer-Verlag, 1991, pp. 17–38.

[Mad84] W.E. Madryga, "A High Performance Encryption Algorithm," *Computer Security: A Global Challenge*, Elsevier Science Publishers, 1984, pp. 557–570.

[ML91] J.L. Massey and X. Lai, "Device for Converting a Digital Block and the Use Thereof," International Patent PCT/CH91/00117, 28 Nov 1991.

[Mer91a] R. Merkle, "A Fast Software Encryption Function," *Advances in Cryptology—CRYPTO '90 Proceedings*, Springer-Verlag, 1991, pp. 476–501.

[Mer91b] R. Merkle, "Method and apparatus for data encryption," U.S. patent 5,003,597, 26 Mar 1991.

[NBS77] National Bureau of Standards, NBS FIPS PUB 46, "Data Encryption Standard," National Bureau of Standards, U.S. Department of Commerce, Jan 1977.

[Riv95] R.L. Rivest, "The RC5 Encryption Algorithm," *Fast Software Encryption, Second International Workshop Proceedings*, Springer-Verlag, 1995, pp. 86–96.

[RC94] P. Rogaway and D. Coppersmith, "A Software-Optimized Encryption Algorithm," *Fast Software Encryption, Cambridge Security Workshop Proceedings*, Springer-Verlag, 1994, pp. 56–63.

[Sch94] B. Schneier, "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)," *Fast Software Encryption, Cambridge Security Workshop Proceedings*, Springer-Verlag, 1994, pp. 191–204.

[Sch96] B. Schneier, *Applied Crytography, 2nd Edition*, John Wiley & Sons, 1996.

This article was processed using the LaTeX macro package with LLNCS style