Performance of Protocols (Extended Abstract)

Michael Roe

Centre for Communications Systems Research

1 Introduction

NIST is currently running a contest to select a block cipher which will be the new "Advanced Encryption Standard". Many people have contributed to the evaluation of the candidate algorithms; however, much of the evaluation has concentrated on how fast the algorithms are, rather than other factors (such as their security) which might be more important.

This paper was presented at a security protocols conference, rather than a cryptography conference. Here the tradition is that we take a step back from discussing the internals of cryptographic algorithms and consider instead the context in which the algorithm is used. I think we should do this for discussions of performance as well as discussions of security. To the end user, what matters is the performance of the whole application. Is it worthwhile for cryptographers to spend a great deal of effort on micro-optimizing block ciphers? Will this have a significant effect on overall system performance?

2 Changes in technology

When I started doing research in cryptography, in the mid 1980's, performance of cryptographic algorithms was certainly an issue. The typical workstation was a Micro Vax connected to a dumb terminal: the CPU wasn't fast enough to support a windowing system. The cryptographic algorithm in use was DES, which was designed for hardware, not software implementation. Although we now know many tricks for doing DES fast in software, these weren't widespread in the mid-80's. A consequence of all this was in was worth spending some effort optimizing the cryptography.

CPUs are now much faster. Memory and network speeds have also increased, but not nearly as much as CPU speeds. Pure computation, such as is used in a block cipher, is cheaper in both absolute terms and relative to other tasks, such as writing the data to disc. Unlike DES, nearly all of the AES candidates are designed for high performance in software.

It could be argued that for most applications, nearly all the AES algorithms are fast enough. We have reached the point where cryptography is not a significant portion of the total CPU burden, and the relative speed of the algorithms no longer matters very much. The purpose of this paper is to collect some of the evidence for and against this argument.

3 The effect of APIs

When comparing the performance of two different cryptographic algorithms, we have to be very careful that we are comparing like with like. It is often the case that different implementations have different interfaces. (The AES reference implementations are an exception to this, as a common interface was defined). Surprisingly, the choice of interface matters. The overhead of the procedure call is quite large relative to the total amount of work the block cipher does, so the speed of the call matters.

At this point, protocol considerations intrude. An API that gives very fast performance in a benchmark may turn out to be useless in a real application, because it doesn't provide features which are needed:

– Often, we would like to have several keys in use at the same time. For example, different threads of control within a server might be talking to different clients, using different keys. This means that the key schedule (or a reference to it) should be a parameter to the encryption routine, not a global variable. This can effect both the time taken for the call itself, and the time taken for each access to the key schedule which the block cipher: Passing parameters can slow down the call, and indirect addressing can be slower than direct addressing.
– Often, we need to support dynamic negotiation of which algorithm to use. The block cipher of choice changes over time, and it is useful to be able to incrementally upgrade distributed systems one machine at a time. During transition you have to negotiate whether to use the old or the new algorithm. Export control and other political reasons mean that different clients of the same server may support different algorithms: again, dynamic negotiation is needed. In practice, this means that the call to the encrypt routine itself is indirectly addressed: it is entered by a jump to a subroutine whose address is held in a register, rather than jump to a subroutine at a fixed address. Such indirect calls are slower. More to the point, on many processors they disrupt the pipelining to a much greater extent that normal subroutine calls.

Block size also matters: a 128 bit block cipher will typically look faster than a 64 bit block cipher simply because half the number of subroutine calls are needed to encrypt the same amount of data — the subroutine call is a significant portion of the cost of the cryptography. An API that encrypts many blocks in one call will achieve greater performance than one that encrypts a single block per call.

Some machines are "big endian" and others are "little endian". Achieving interoperability between the two often involves exchanging the order of bytes within a word. Although some processors have a special instruction for doing this, it is typically not accessible from a high-level language. Like the subroutine call, the byte swap is a significant portion of the total CPU time. Some APIs attempt to hide the byte swap by pushing it outside the block cipher; they define the cipher as an operation on sequences of 32-bit words. Of course, the protocol implementation as a whole still ends up doing the byte swap, but benchmarks that only measure the cryptographic algorithm are unfairly inflated by excluding it.

Memory management strategies also make a huge difference:

- Service provider allocates, nobody deallocates.
  The memory leak is a common implementation strategy — it does not give good performance, at least not if you need to make a large number of calls. Memory accesses are much more expensive than register accesses; paging is even worse.
- Service consumer allocates and deallocates.
  The typical problem with this is that the caller doesn't know how much to allocate —- a protocol machine that doesn't know which algorithm is in use doesn't know how much space to allocate for a key schedule.
- Service provider allocates and deallocates.
  From a system point of view this is good, because the provider at least knows how big the buffers need to be. The downside is that it can result in far more calls to the memory allocator: one per key change, or even one per cipher block. Memory allocation is much more expensive than doing the actual encryption.

4 Chaining Modes

The choice of chaining mode can have surprising performance implications. In CBC mode, buffers need to be extended to a whole number of blocks on encryption and reduced to the real length on decryption. This may result in heap management calls which cost far more than the actual cryptography. Increasing the length of a buffer may require calls to heap management routines for the simple reason that the extended data won't fit, and a bigger buffer must be allocated. Reducing the length of a buffer may also cause heap management calls. This may come as a surprise to C programmers, as the usual C memory allocator remembers the original length of each object on the heap, even its size has subsequently changed. Memory allocators used by other languages can be less forgiving, and may need to be notified when a buffer's size changes.

As a result, length preserving modes such as Output Feedback (OFB) can give better performance than CBC mode. Although the number of encryption operations is roughly the same, fewer calls to heap management routines are needed.

5 Some Performance Measurements

This section presents two sets of performance measurements: firstly, the traditional cryptographic benchmark of the block ciphers in isolation, and secondly measurements of the block ciphers in a particular application.

The block ciphers were all implemented in the Ada programming language, rather than C. Many of the reasons for this choice of programming language are outside the scope of this paper — they were implemented for use by a project which I won't describe here. However, for the purposes of interpreting the benchmarks, it is important to realise that care needs to be taken when comparing them with other measurements of algorithms implemented in C. Such comparisons run the risk of confusing differences between the cryptographic algorithms with the differences between the compilers that were used.

However, relative comparison of these benchmarks is more meaningful: each implementation is in the same programming language, with the same API, the same coding style, and the same compiler used.

Figure 1 shows the speed in Mb/s of several applications, as measured by a program which encrypts a million 128-bit blocks under the same key. The machine used for measurement was a 266Mhz Pentium II running Linux. The compiler used was GNAT 3.11p at optimization level 3 with array bounds checking disabled. The test program does very little else apart from encrypt data; for example, it doesn't write the ciphertext to disk or an IO device. As it doesn't do anything between each encryption, most of the encryption routine will still be in the primary cache. This is a common set-up for measuring algorithm performance; however, it is not necessarily typical of what happens in real applications.

Algorithm	Encrypt (Mb/s)	Decrypt (Mb/s)
RC6	40.6	36.0
Rijndael	27.5	26.1
CAST 256	22.8	22.6
Serpent	11.9	10.6
Safer+	3.56	3.52

Fig. 1. Bulk Encryption Speed

The above table shows that there is nearly an order of magnitude difference in performance between the slowest and fastest algorithms, when they are measured in isolation.

Figure 2 shows the time taken to perform a remote procedure call with three different levels of cryptographic protection: no cryptography; data integrity provided by a MAC based on SHA-1; integrity and confidentiality provided by a MAC based on SHA-1 followed by encryption with RC6 in cipher block chaining mode. The RPC implementation used was GLADE. (Ada includes remote procedure calls as part of the specification of the programming language. GLADE is an implementation of these Ada RPCs). Two different types of remote procedure call were measured: one which takes an integer parameter and returns an integer result, and one which takes a string parameter and returns a string result. For timing the string RPC, both the parameter and the returned value were 45 bytes long. We expect the string RPC to take longer than the integer RPC because more data must be transmitted. This measurement was also taken with block ciphers other than RC6, but these are not shown in figure 2 because the differences between algorithms is less than the accuracy of the measurement. RC6, Rijndael and CAST 256 all result in an integer RPC taking 2.7 mS to two significant figures. This in itself is enough to provide evidence for the contention that was made at the beginning of this paper: the differences between cryptographic algorithms are small compared to the total amount of work, at least for all applications which use this particular RPC library.

Cryptography	Integer RPC (ms)	String RPC (ms)
None	1.2	1.5
SHA-1	2.5	2.8
SHA-1 + RC6	2.7	3.1

Fig. 2. Time taken for a remote procedure call

In order to measure the differences between algorithms, a more sensitive experiment was devised. The time taken for 10,000 remote procedure calls was measured 8 times. Figure 3 presents the mean, upper quartile and lower quartile of the distribution of the results. The purpose of presenting the upper and lower quartile is to give an indication of the accuracy of the measurement; other events taking placing within the same computer system can effect the timing, and it is important to separate this background noise from differences due to the choice of algorithm. In figure 3, the actual timings have been divided by 10,000 to give a figure in milliseconds per RPC.

Integer RPC			
Cryptography	Mean	Lower quartile	Upper quartile
SHA-1 + RC6	2.6660	2.6658	2.6661
SHA-1 + Rijndael	2.6997	2.6985	2.7013
SHA-1 + CAST 256	2.7172	2.7168	2.7176
SHA-1 + Serpent	3.0563	3.0557	3.0569
SHA-1 + Safer+	3.1466	3.1460	3.1470

String RPC			
Cryptography	Mean	Lower quartile	Upper quartile
SHA-1 + RC6	3.0534	3.0534	3.0536
SHA-1 + Rijndael	3.1214	3.1210	3.1217
SHA-1 + CAST 256	3.1403	3.1398	3.1403
SHA-1 + Serpent	3.5594	3.5590	3.5598
SHA-1 + Safer+	3.9573	3.9410	3.9414

Fig. 3. Time taken for integer and string RPC

A comparison of figures 1 and 3 shows that the performance of block ciphers measured in isolation gave a reasonably good prediction of their performance in the RPC application. Ciphers which were fast in isolation were also fast in the RPC application.

With the integer RPC, the call fits into 5 128-bit cipher blocks and the return value fits into 4 cipher blocks, so each RPC requires 9 encryptions and 9 decryptions. With the string RPC, the call fits into 8 blocks and the result into 7, resulting in 15 encryptions and decryptions. The actual timings in figure 3 are very close to those we would predict by converting the speed results of figure 1 into microseconds per block and multiplying by the number of blocks which are encrypted. One exception to this is the Serpent algorithm, which did rather worse in the RPC application than would have been predicted from its performance in isolation.

A remaining question is how much of computational cost of a protected RPC is due to computing a SHA-1 hash. Figure 4 shows the time in milliseconds needed to compute the SHA-1 hash of messages of various lengths. Very short (1 byte) messages take longer than 16 byte messages, because internally SHA-1 pads messages to a whole number of blocks, and this padding process takes a measurable amount of time. Apart from this effect, the time taken to compute a hash is approximately a constant amount plus an amount proportional to the size of the message.

Size (bytes)	Time (mS)	Speed (Mb/s)
1	0.0153	0.523
16	0.0156	8.21
32	0.0158	16.2
48	0.0160	24.0
64	0.0253	20.2
80	0.0263	24.3
128	0.0360	28.4
256	0.0573	35.7
512	0.100	41.0
1024	0.186	44.0
2048	0.357	45.9
16384	2.76	47.5
65536	11.04	47.5

Fig. 4. Time taken to compute SHA-1 hash

Comparison of figures 2 and 4 shows that for small messages the cost of computing the hash function is on the order of 1% of the cost of performing an RPC. This implies that replacing SHA-1 with a faster hash function will have little effect on the total time needed to perform a cryptographically protected RPC.

Comparison of figures 3 and 4 shows that although SHA-1 is faster than any of the AES block ciphers for large messages, some block ciphers are faster for short messages. In the RPC application, messages are short. When one of the faster block ciphers is in use, a CBC MAC computed using a block cipher would be a faster means of providing integrity than the MAC based on hash functions that was actually used. A third way of providing integrity is to compute a hash of the message, and then protect this hash using a CBC MAC. These results show that for short messages it is quicker to compute the MAC directly on the data.

6 Conclusions

These results show that for applications using a particular RPC library, the computational cost of encryption is small compared to the total computational cost of performing an RPC. We conjecture that this is also true for many other applications. A consequence of this is that a cipher designer should not further sacrifice speed for security in order to make their design look faster than the competition. Further improvements in block cipher performance will have negligible effect on applications as a whole.

Unfortunately, it is not the case that adding cryptographic protection has little impact on application performance. Adding cryptography nearly doubled the time taken to perform an RPC. However, the additional computational cost is not primarily due to the cryptographic primitives, the hash function and the block cipher. Rather, it is is due to the cost of adding another layer to the protocol stack, with the associated additional multiplexing, buffer management and header parsing. This effect is well known to opponents of the OSI seven layer model: additional layers often mean additional overhead.

Much work has been done on fast software implementation of block ciphers. These results show that this effort is misplaced. A greater effect on application performance could be achieved by developing fast implementations of the security protocol itself, not the block cipher on which it relies.