# Towards High Performance Cryptographic Software[1]

Erich Nahum[1], Sean O'Malley[2], Hilarie Orman[2], and Richard Schroeppel[2]

Department of Computer Science[1]
University of Massachusetts
Amherst, MA 01003

Department of Computer Science[2]
University of Arizona
Tucson, AZ 85721

TR 95 04

**Abstract**

Current software implementations of current cryptographic algorithms are orders of magnitude slower than required to secure a gigabit network. This paper examines three different approaches to improving the performance of cryptographic software: new algorithm design, parallelization, and algorithm independent hardware support. We believe that in combination these approaches could go a long way to improving cryptographic protocol performance without the inflexibility required for the current generation of cryptographic hardware support.

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This research supported in part by NSF under grant NCR-9206908, by ARPA under contract F19628-92-C-0089, by ARPA under contract DABT63-94-C-0002, and by NCSC under contract MDA 904-94-C-6110.

| Protocol | Performance |
|---|---|
| MD5 | 80 Mbits/sec |
| DES | 15 Mbits/sec |
| 3-DES | 5 Mbits/sec |
| RSA Crypt-Decrypt | 10 Kbits/sec |
| RSA Signature | 20 signatures/sec |
| Diffie-Hellman Key Exchange | 5 keys/sec |

Table 1: Current Cryptographic Algorithm Performance

# 1 Introduction

A great deal of effort is currently being expended to provide some level of Internet security. An important question is whether security can be provided at gigabit speeds. The standard set of algorithms required to secure a connection includes a bulk encryption algorithm such as DES [1], a cryptographic checksum such as MD5 [14], a key exchange algorithm (such as Diffie-Hellman key exchange) to securely distribute the DES key, and some form of digital signature algorithm to authenticate the parties (e.g., RSA [15]). The encryption and checksum algorithm must be applied to every packet going across a secure link, and therefore the performance of these algorithms directly affects the achievable bandwidth. Furthermore, there are some problems (such as sender authentication in multicast) that require the use of expensive algorithms such as RSA signatures on every packet. The other algorithms only need to be run at connection set-up time, so they only affect the overall bandwidth if the connections are short, the algorithms are particularly expensive, and/or the overhead of using these algorithms on busy servers reduces overall network performance. Thus there are two important performance metrics to consider: overall bandwidth and number of connections per second.

A straightforward approach to improving cryptographic performance is to implement cryptographic algorithms in hardware. This approach has been shown to improve cryptographic performance of single algorithms (e.g., DEC has demonstrated a 1 Gbit/sec DES chip [5]). Unfortunately there are several problems with this approach. First, a secure network system requires the efficient implementation of a suite of algorithms, not just DES. Hosts clearly need to be able to run both DES and MD5 efficiently, and servers, at least, need to run them all efficiently. Second, hosts need to implement Internet standards, and standards change. In fact, most Internet security standards are written to allow flexibility in algorithm selection. Third, security algorithms can be broken (e.g. knapsack crypto-systems, 129 digit RSA, 192 bit Diffie Hellman Key exchange, and any year now, DES). Hence, they may have to be changed on short notice. Fourth, cryptographic hardware is not ubiquitous, cheap or readily exportable. It is clear that adding more than one (and some people would say any) piece of cryptographic hardware to a given host is unlikely, if only for these reasons. Finally, implementing certain cryptographic algorithms (e.g. MD5) in hardware provides only limited increases in performance [19]. Thus we question whether the traditional approach of implementing cryptographic hardware is suitable for the current Internet environment.

Thus we need cryptographic software. Therefore let us consider the performance of a set of cryptographic algorithms when implemented in software. Table 1 gives the performance of a variety of standard cryptographic algorithms as implemented in software on a 175 MHz Dec Alpha 600. Clearly, implementing these algorithms in software on a modern RISC workstation does not provide sufficient bandwidth to drive a gigabit link. Furthermore, the connection establishment overhead is so high that overall performance on busy servers is likely to suffer. All of these algorithms are compute bound and their performance will scale nicely with increasing processor performance. Unfortunately, we believe that most, if not all, of this increased performance is already spoken for. Increasing network performance will require improvements in cryptographic software performance, and improved processor performance also helps an attacker. Thus, the strength (and hence the computational cost) of the cryptographic algorithms must be increased to keep up.

In this paper we will examine three different approaches to improving the performance of cryptographic software: new algorithm design, parallelization, and algorithm independent hardware support.

## 2 Secure Algorithm Design

An obvious way to improve the performance of cryptographic software is to develop new and faster algorithms. Unfortunately it is not sufficient for a security algorithm to be correct, it also must be secure. Showing that an algorithm is secure is a time consuming problem. This makes the creation of entirely new algorithms problematic: new algorithms are assumed to be insecure. We address this problem in three ways. First, there are a large number of existing security algorithms which, while not widely used, have at least been around long enough to have withstood some amount of attack. Second, existing security algorithms can often be implemented in a variety of different ways. Finally, it is often possible to make small modifications to an existing algorithm with a strong assurance that the resulting algorithm is not significantly weaker than the original.

An example of this approach is a set of modifications to the Diffie-Hellman key exchange algorithm(DHKX) that we (and others) have implemented. The DHKX algorithm [4] is a secure method for initiating a conversation between two previously un-introduced parties. It relies on exponentiation in a large group, and the software implementation of the group operation is usually computationally intensive. The algorithm has been proposed as an Internet standard [12], and as such its performance is of great importance. A straightforward implementation of DHKX uses the multiplication group of integers modulo $p$, where $p$ is a prime on the order of $2^{512}$ (this is the implementation timed in Table 1). However, the DHKX algorithm can also be implemented with the same level of security using the group of points on an elliptic curve over the Galois field $\mathbb{F}_{2^{155}}$[10]. By choosing an appropriate mapping from elliptic curve operations to the $\mathbb{F}_{2^{155}}$operations to the host machine instructions, one can gain significant improvements. In [17] we showed an improved method of computing reciprocals in $\mathbb{F}_{2^{155}}$which increases the overall performance of the DHKX algorithm by a factor of 6, bringing it down to 30 milliseconds per key exchange. A further increase in performance (about a factor of 2.5) can be achieved by precomputing a table of generator powers [3]. The resulting table size is small (in the low kilobytes) and as such is well worth the time-space trade off.

What is more important than the results on any specific algorithm is the approach. One can choose from a variety of algorithms in the literature. Then, in the case of number theoretic algorithms, one can choose the underlying mathematical structures for implementation. If careful implementation techniques for the structure representation and operators are used, performance improvements are sometimes possible.

## 3 Parallelism

Another approach to improving software encryption performance is to use parallelism. At least three types of parallelism can be used: per-connection, per-packet, and intra-packet (or functional) parallelism.

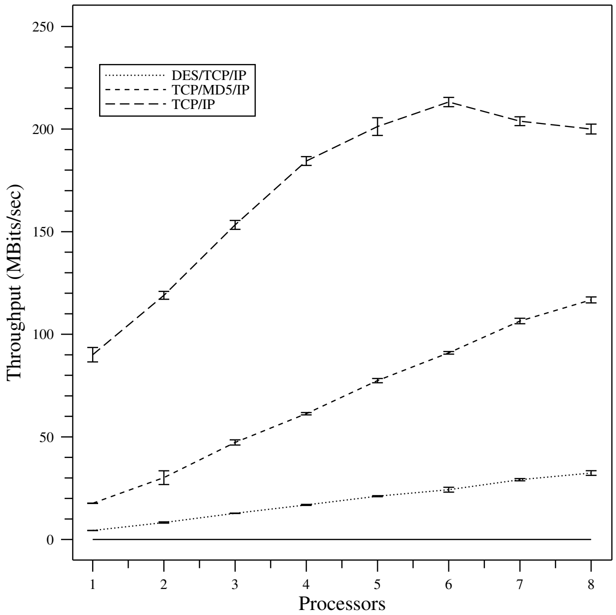


Figure 1: DES/MD5/TCP Throughputs

Connection-level parallelism is straightforward, but does not allow parallelism to improve the throughput of a single connection. Packet-level parallelism associates processing with each packet, regardless of the connection. Intra-packet parallelism associates multiple processing units with a single packet, and is only feasible if the encryption algorithm allows it. For example, DES with Electronic Code Book (ECB) allows separate eight-byte blocks to be encrypted in parallel, and thus could be parallelized at an intra-packet granularity. However, ECB is susceptible to simple-substitution code attacks and cut-and-paste forgery. Thus, most implementations use the Cipher-Block Chaining (CBC) mode of DES, where the output of each encryption is xor'ed into the next block of plaintext. Although each block cannot be encrypted in parallel with DES CBC mode, they can be decrypted in parallel.

To examine the impact of parallelism on cryptographic software, we used a version of the *x*-kernel [6] augmented to support packet-level parallelism [11]. The system runs in user space on Silicon Graphics R4400-based shared-memory multiprocessors. We ran a set of send-side throughput tests with DES and MD5 to see how well encryption protocols scale using packet-level parallelism. Given that the granularity of parallelism in this study is packet-sized, DES parallelism here means that separate packets are encrypted in parallel using CBC mode.

Figure 1 shows sending throughputs in Megabits per second for three protocol stacks: a TCP/IP stack, a TCP/MD5/IP stack, and a DES/TCP/IP stack[1]. These throughputs were measured on an 8-processor 100MHz Challenge machine, using a single TCP connection with 4 KB packets. Figure 2 shows the corresponding relative speedup for the three TCP stacks, where speedup is normalized relative to the appropriate stack's uniprocessor throughput. Each data point is the average of 10 runs, and throughput graphs include 90 percent confidence intervals. More details can be found in [11].

[1] For purposes of this study, we ignore for the moment that TCP does not preserve packet boundaries on the receive side. In this case, on the send side, packet boundaries are preserved.

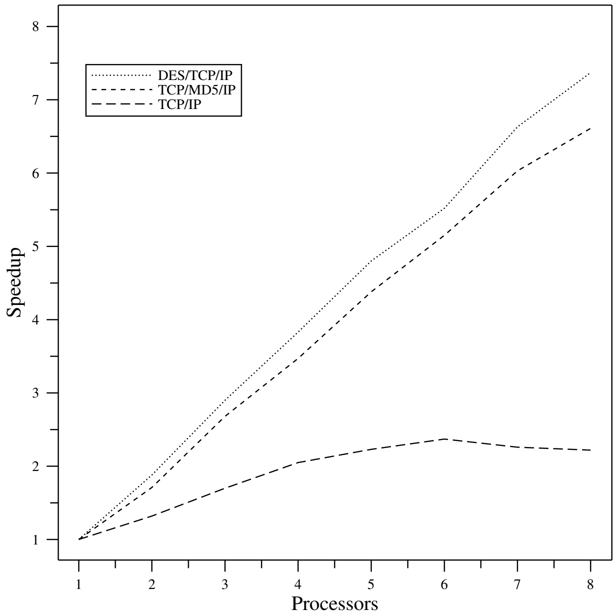


Figure 2: DES/MD5/TCP Speedup

Figure 1 illustrates the performance cost of doing cryptography in software. MD5 exacts roughly an order of magnitude in performance, and DES roughly two orders of magnitude. Figure 2 shows, however, that although the baseline TCP scales badly, speedup with encryption is close to linear, for the number of processors tested. This is because the encryption protocols are compute-bound, overshadowing any locking cost, and the encryption is done outside the scope of any locks. Similar linear speedups were observed for cryptographic UDP-based stacks, not shown due to space limitations.

Previous work [2, 11] has shown limited packet-level parallelism using a single TCP connection, barring any other protocol processing. Given that the locked component of manipulating the TCP connection-state limits the throughput to about 200 Mbits on this platform, we estimate that the TCP/MD5/IP stack would bottleneck at about 16 processors, and that the DES stack would scale to 40 processors. More compute-intensive protocols, such as triple-DES and RSA, should scale linearly as well.

This study is still preliminary, and many factors remain to be explored. We have examined only send-side throughputs, and receive-side processing may behave differently. Our results suggest, however, that software encryption protocol performance can be improved using parallelism.

# 4 Algorithm Independent Hardware Support

Instead of implementing entire algorithms in hardware we propose to improve the performance of a broad selection of cryptographic software. The basic idea is to do classic RISC processor (or co-processor) design on a large set of cryptographic software implementations. The idea is to add to a standard RISC instruction set only those instructions which significantly improve the overall performance of the test suite. Careful examination of current implementations of cryptographic software has identified three basic problems with implementations on modern RISC machines: operations on sub-wordsize units, operations on super-wordsize units, and operations of groups other than that of integers. We believe that the addition of a small number of new CPU instructions could significantly increase the performance of a wide variety of cryptographic algorithms.

For example providing a single instruction to better support arithmetic in the Galois field $\mathbb{F}_{2^{155}}$can significantly improve the performance of DHKX presented in section 2. The polynomials that make up these groups can be represented as bit vectors where each bit n represents $bit * x^n$. With this representation $+$ (and $-$) in this group can be implemented as $(n/64)$ 64 bit XOR's. Multiplication in the field is exactly analogous to integer multiplication with the addition operation replaced with XOR. The lack of carries makes this significantly easier to implement in hardware than integer multiply. A rough estimate is that a 64 bit XOR multiply implemented on the Alpha chip would take 8 cycles. In the algorithm used in Section 2, this would probably result in a 30% improvement in the DHKX running time. By redesigning the algorithm to take advantage of the cheap XOR-multiply operation, we believe that we could reduce the running time of the DHKX algorithm by a factor of 4. Note that this instruction would be useful for any security algorithm that uses $\mathbb{F}_{2^n}$and may have application beyond security in such areas as error correcting codes.

Is it practical to add instructions to a RISC processor? We think so. First, there is no real shortage of chip area on most modern RISC processors [7], and second, many modern super-scalar processors have more ALU's than are needed for the amount of instruction level parallelism found in most applications [18]. The real problem is the cost of the design work required to implement these instructions in very high clock-rate technologies. The question then is one of economics: is the market for such a chip large enough for the vendor to make a profit? The UltraSPARC and HP PA-7100LC may provide existence proofs that this approach is viable given a large enough market. The UltraSPARC design contains a set of special purpose instructions to speed up pixel operations and MPEG play [8]; a similar technique is used in the PA-7100LC chips [9]. This increases the performance of multimedia applications and reduces cost by eliminating the necessity of providing special purpose graphics hardware off chip. The market for crypto hardware may reach a size where this approach is profitable. If we can demonstrate that enhancing a standard CPU core by a minimal number of RISC-style instructions improves the performance of several security (and perhaps non-security) algorithms, then the viability of that market may well be increased.

# 5 Conclusion

Current software implementations of current cryptographic algorithms are orders of magnitude slower than required to secure a gigabit network. We have proposed three techniques to solve this problem. We believe that in combination these approaches could go a long way to improving cryptographic protocol performance without the inflexibility required for the current generation of cryptographic hardware support.

# 6 Acknowledgements

Joe Touch and David Yates provided valuable comments on earlier drafts of this paper.

# References

[1] A. N. S. I. (ANSI). American national standard data encryption standard. Technical report ANSI X3.92-1981, Dec. 1980.

[2] M. Björkman and P. Gunningberg. Locking effects in multiprocessor implementations of protocols. In *ACM SIGCOMM Symposium on Communications Architectures and Protocols*.

[3] E. Brickell, D. Gordon, K. McCurley, and D. Wilson. Fast exponentiation with precomputation (extended abstract). In *Lecture Notes in Computer Science **658***, pages 200–207, 1993.

[4] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, Feb. 1993.

[5] H. Eberle. A high-speed DES implementation for network applications. Technical Report 90, Digital Equipment Corporation Systems Research Center, Sept. 1992.

[6] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[7] N. Jouppi and S. Wilton. Tradeoffs in two-level on-chip caching. Technical Report Research Report 93/3, DEC WRL, Oct. 1993.

[8] L. Kohn, G. Maturana, A. Prabhu, and G. Zyner. The visual instruction set (VIS) in UltraSPARC. In *Compcon Spring 95*, pages 462–469, March 1995.

[9] R. B. Lee. Accelerating multimedia with enhanced microprocessors. *IEEE Micro*, 15(2):22–32, Apr. 1995.

[10] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

[11] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley. Performance issues in parallelized network protocols. In *First USENIX Symposium on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994.

[12] T. I. E. T. F. W. G. on Security for IPv4. Ipsec draft. Technical report, 1995.

[13] H. Orman, S. O'Malley, R. Schroeppel, and D. Schwartz. Paving the road to network security, or the value of small cobblestones. In *Proceedings of the 1994 Internet Society Symposium on Network and Distributed System Security*, Feb. 1994.

[14] R. Rivest. The MD5 message-digest algorithm. In *Network Information Center RFC 1321*, pages 1–21, Menlo Park, CA, Apr. 1992. SRI International.

[15] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryposystems. *Communications of the ACM*, pages 120–126, Feb. 1978.

[16] D. C. Schmidt and T. Suda. Measuring the performance of parallel message-based process architectures. In *Proceedings of the Conference on Computer Communications (IEEE Infocomm)*, Boston, MA, Apr. 1995.

[17] R. Schroeppel, H. Orman, and S. O'Malley. Fast key exchange with elliptic curve systems. Technical Report 95-03, Department of Computer Science, University of Arizona, Feb. 1995.

[18] M. D. Smith, M. Johnson, and M. A. Howowitz. Limits on multiple instruction issue. In *Proceedings Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 290–302, Boston MA, April 1989.

[19] J. Touch. Performance analysis of md5. Technical report, Submitted To ACM Sigcomm '95, 1995.