Parallelized Network Security Protocols

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

Department of Computer Science[1]
University of Massachusetts
Amherst, MA 01003
{nahum,yates}@cs.umass.edu

Department of Computer Science[2]
University of Arizona
Tucson, AZ 85721
{sean,ho,rcs}@cs.arizona.edu

Abstract

Security and privacy are growing concerns in the Internet community, due to the Internet's rapid growth and the desire to conduct business over it safely. This desire has led to the advent of several proposals for security standards, such as secure IP, secure HTTP, and the Secure Socket Layer. All of these standards propose using cryptographic protocols such as DES and RSA. Thus, the need to use encryption protocols is increasing.

Shared-memory multiprocessors make attractive server platforms, for example as secure World-Wide Web servers. These machines are becoming more common, as shown by recent vendor introductions of platforms such as SGI's Challenge, Sun's SPARCCenter, and DEC's AlphaServer. The spread of these machines is due both to their relative ease of programming and their good price/performance.

This paper is an experimental performance study that examines how encryption protocol performance can be improved by using parallelism. We show linear speedup for several different Internet-based cryptographic protocol stacks running on a symmetric shared-memory multiprocessor using two different approaches to parallelism.

1 Introduction

Security and privacy is a growing concern in the Internet community, due to the Internet's rapid growth and the desire to conduct business over it safely. This desire has led to the advent of several proposals for security standards, such as secure IP [2], Secure HTTP (SHTTP) [30], and the Secure Socket Layer (SSL) [14]. Thus, the need to use encryption protocols such as DES and RSA is increasing.

One problem with using cryptographic protocols is the fact that they are slow. An important question then 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 message digest such as MD5 [31], a key exchange algorithm such as Diffie-Hellman [7] to securely distribute a private key, and some form of digital signature algorithm to authenticate the parties, such as RSA [32]. The encryption and hash digest algorithms must be applied to every packet going across a link to ensure confidentiality, and therefore the performance of these algorithms directly affects the achievable throughput of an application. Furthermore, there are some services, such as strong sender authentication in scalable multicast algorithms, that currently require the use of expensive algorithms such as RSA signatures on every packet. For non-multicast services, the most expensive algorithms, RSA and Diffie-Hellman, need only be run at connection set-up time. 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.

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 [8]). Unfortunately there are several problems with this approach, that indicate why one would desire to do cryptography in software:

- Variability. Systems need to support suites of cryptographic protocols, not just DES. For example, both SSL and SHTTP support using RSA, DES, Triple-DES, MD5, RC4, and IDEA. Hardware support for all of these is unlikely.
- Flexibility. Standards change over time, and this is one reason why the IP security architecture and proposed

[1] This research supported in part by NSF under grant NCR-9206908, and by ARPA under contract F19628-92-C-0089. Erich Nahum is supported by a Computer Measurement Group Fellowship. David Yates is the recipient of a Motorola Codex University Partnership in Research Grant.

[2] This research supported in part by ARPA under contract DABT63-94-C-0002, and by NCSC under contract MDA 904-94-C-6110.

key exchange schemes are designed to handle multiple type of authentication and confidentiality algorithms.

- Security. Algorithms can be broken, such as knapsack crypto-systems, 129 digit RSA, and 192 bit DHKX. It is easier to replace software cryptographic algorithms than hardware.
- Cost. Custom hardware is not cheap, ubiquitous, or exportable.
- Performance. Certain algorithms, such as MD5 and IDEA, are designed to run quickly in software on current microprocessor architectures. Specialized hardware for them may not improve performance much. An analysis of this is given for MD5 [37].

Many approaches are available for improving software cryptographic support [23], including improved algorithm design and algorithm-independent hardware support. In this paper, we focus on how parallelism can improve software cryptographic performance.

A common approach to parallelism is through symmetric shared-memory multiprocessors. These machines are becoming more common, as shown by recent vendor introductions of machines such as SGI's Challenge [11], Sun's SPARCCenter [9], and DEC's AlphaServer [10]. The spread of these machines is due to a number of factors: binary compatibility with lower-end workstations, good price/performance relative to high-end machines such as Crays, and their ease of programming compared to more elaborate parallel machines such as Hypercubes. In addition, these machines make attractive server platforms, for example as secure HTTP (World-Wide Web) servers.

In this paper we demonstrate that parallelism is an effective vehicle for improving software cryptographic performance. We show linear speedup results of several different Internet-based cryptographic protocol stacks using two different approaches to parallelism. Our implementation consists of parallelized versions of the *x*-kernel [15] that run in user space on Silicon Graphics shared-memory multiprocessors. We give performance results on a 12 processor 100MHz MIPS R4400 SGI Challenge XL.

The remainder of the paper is organized as follows: In Section 2, we discuss several approaches to parallelism. Section 3 describes our implementation and experiments. In Section 4 we present our results in detail. Section 5 discusses related issues. In Section 6 we summarize our results.

2 Parallelism in Network Protocol Processing

Parallelism can take many forms in network protocol processing. Many approaches to parallelism have been proposed and are briefly described here; more detailed surveys

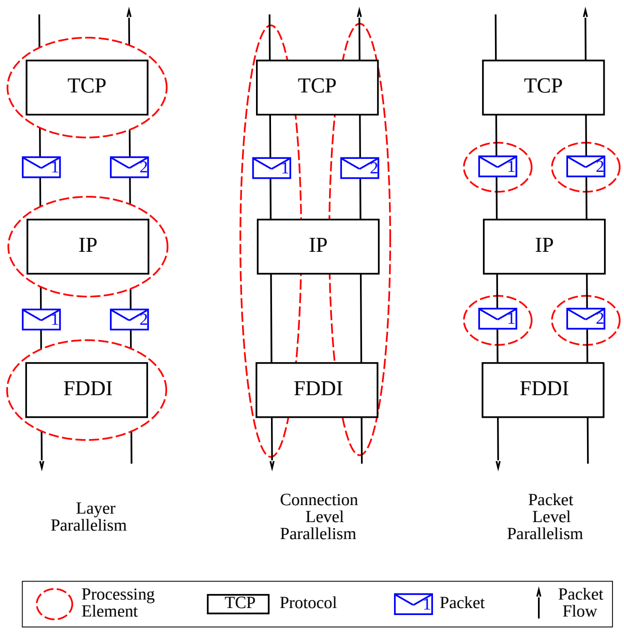

Figure 1: Approaches to Concurrency

can be found in [3, 13]. In general, we attempt to classify approaches by the unit of concurrency, or what it is that *processing elements* do in parallel. Here a processing element is a locus of execution for protocol processing, and can be a dedicated processor, a heavyweight process, or a lightweight thread. Figure 1 illustrates the various approaches to concurrency in host protocol processing. The dashed ovals represent processing elements. Messages marked with a number indicate which connection the message is associated with.

In *layered parallelism*, each protocol layer is a unit of concurrency, where a protocol layer is defined by the ISO reference model. Specific layers are assigned to processing elements, and messages passed between protocols through interprocess communication. The main advantage of layered parallelism is that it is simple and defines a clean separation between protocol boundaries. The disadvantages are that concurrency is limited to the number of layers in the stack, and that associating processing with layers results in increased context switching and synchronization between layers [5, 6, 34]. Performance gains are limited to throughput, mainly achieved through pipelining effects. An example is found in [12].

Connections form the unit of concurrency in *connection-level parallelism*, where connections are assigned to processing elements. Speedup is achieved using multiple connections, each of which is processed in parallel. The advantage of this approach is that it exploits the natural concurrency between connections, and only requires one lock acquisition, i.e. for the appropriate connection. Thus, locking is kept to a minimum along the "fast path" of data transfer. The disadvantage with connection-level parallelism is that no concurrency within a single connection can be achieved, which may be a problem if traffic exhibits locality [16, 19, 22, 27], i.e., is bursty. Systems using this approach include [29, 33, 34, 38].

In *packet-level parallelism*, packets are the unit of concurrency. Sometimes referred to as thread-per-packet or processor-per-message, packet-level parallelism assigns each packet or message to a single processing element. The advantage of this approach is that packets are processed regardless of their connection or where they are in the protocol stack, achieving speedup both with multiple connections and within a single connection. The disadvantage is that it requires locking shared state, most significantly the protocol state at each layer. Systems using this approach include [3, 13].

In *functional parallelism*, a protocol layer's functions are the unit of concurrency. Functions within a single protocol layer (e.g., checksum, ACK generation) are decomposed, and each assigned to a processing element. The advantage to this approach is that it is relatively fine-grained, and thus can improve latency as well as throughput. The disadvantage is that it requires synchronizing within a protocol layer, and is dependent upon the concurrency available between the functions of a particular layer. Examples include [17, 28, 25].

In *data-level parallelism*, the pieces of data are the units of concurrency, analogous to SIMD processing. Processing elements are assigned to the same function of a particular layer, but perform processing on separate pieces of data from the same message. An example would be computing a single message's checksum using multiple processors. The advantage to this approach is that it is the most fine-grained, and thus has the potential for the greatest improvement in both throughput and latency. The disadvantage is that processing elements must synchronize, which may be expensive. We are unaware of any work using this approach.

The relative merits of one approach over another depend on many factors, including the host architecture, the cost of primitives such as locking and context switching, the workload and number of connections, the thread scheduling policies employed, and whether the implementations are in hardware or software. Most importantly, they depend on the *available* concurrency within a protocol stack.

The most comprehensive study to date comparing different approaches to parallelism on a shared-memory multiprocessor is by Schmidt and Suda [34, 35]. They show that packet-level parallelism and connection-level parallelism generally perform better than layer parallelism, due to the context-switching overhead incurred crossing protocol boundaries using layer parallelism. In [35], they suggest that packet-level parallelism is preferable when the workload is a relatively small number of active connections, and that connection-level parallelism is preferable for large numbers of connections.

In this paper, we restrict our focus to connection-level and packet-level parallel approaches to network cryptographic protocols on shared-memory multiprocessors. More fine-grained approaches to parallelized security protocols, such as functional parallelism and data-level parallelism, are outside the scope of this work. We do discuss them in more depth in Section 5.

3 Implementation and Experiments

In this section we describe our implementation and experimental environment.

Our implementation consists of parallelized versions of the *x*-kernel [15] extended for packet-level parallelism [24] and connection-level parallelism [38]. Our parallel implementations run in user space on Silicon Graphics shared-memory multiprocessors using the IRIX operating system.

In this study, we examine the impact of cryptographic protocols under the respective paradigms, and show how parallelism improves cryptographic performance. The protocols we examine are those used in typical secure Internet scenarios. We briefly describe them here.

3.1 Protocols

The protocols used in our experiments are those that would be seen along the common case or "fast path" during data transfer of an application such as a secure World-Wide Web server. We focus on available throughput; we do not examine connection setup or teardown, or the attendant issues of key exchange. In these experiments, connections are already established, and keys are assumed to be available, as needed.

We use the terminology defined by the proposed secure IP standard [2]. *Authentication* is the property of knowing that the data received is the same as the data sent by the sender, and that the claimed sender is in fact the actual sender. *Integrity* is the property that the data is transmitted from source to destination without undetected alteration. *Confidentiality* is the property that the intended participants know what data was sent, but any unintended parties do not. Encryption is typically used to provide confidentiality.

TCP is the Transmission Control Protocol used by reliable Internet applications such as file transfer, remote login, and HTTP, the protocol used for retrieving hypertext documents in the World-Wide Web. IP is the network-layer protocol that performs routing of messages over the Internet. FDDI is the Fiber Distribute Data Interface, a fiber-optic token-ring based LAN protocol.

MD5 is a message digest algorithm used for authentication and message integrity. MD5 is a “required option” for secure IP; by required option we mean that an application’s use of MD5 in IP is optional, but an implementation must support use of that option. MD5 is also the default message digest algorithm proposed for SHTTP; and is also used in SSL. In our implementation we use the standard MD5 message digest calculation, rather than the keyed one. DES is the ANSI Data Encryption Standard, used for confidentiality, and is one of the required protocols used in secure IP, secure HTTP, and SSL. We use DES in cipher-block-chaining (CBC) mode. 3-DES is "triple DES," which runs the DES algorithm 3 times over the message. DES experiments can produce unrealistically good times on empty data buffers, since only a fraction of the 64KB Sbox[1] is exercised, which may lead to artificially good cache hit rates. To guard against this, we fill each buffer at the beginning of a test with a random sequence, with each test using a different initial seed value. Our protocols are taken from the cryptographic suite available with the *x*-kernel [26].

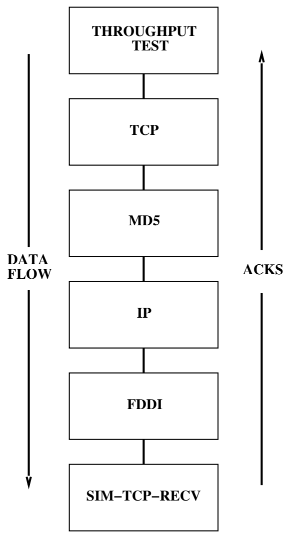

Figure 2: Sample Configuration: TCP/MD5/IP

3.2 Parallel Infrastructure

The implementations for packet-level parallel protocols and connection-level parallel protocols are described in detail in [24, 38]. We briefly outline them here.

In the packet-level parallel protocol testbed, packets are assigned to threads as they arrive, regardless of the connections they are associated with. Correctness and safety is maintained by placing locks around shared data structures, such as the TCP connection state or a protocol’s set of active associations, termed the *active map* in the *x*-kernel. Threads must contend with one another to gain access to shared data.

In the connection-level parallel testbed, connections are assigned to threads on one-to-one basis, called *thread-per-connection*. The entire connection forms the unit of concurrency, only allowing one thread to perform protocol processing for any particular connection. Arriving packets are demultiplexed to the appropriate thread via a packet-filter mechanism [21]. The notion of a connection is extended through the entire protocol stack. Where possible, data structures are replicated per-thread, in order to avoid locking and therefore contention.

The two schemes do have some implementation differences that are necessitated by their respective approaches to concurrency. However, wherever possible, we have strived to make the implementations consistent. For example, both use the same TCP uniprocessor source code base, which is derived from Berkeley’s Net/2 TCP [18] with BSD 4.4 fixes, but not the RFC1323 extensions [4].

[1] We use a double-Sbox implementation in our tests.

3.3 In-Memory Drivers

Since our platform runs in user space, accessing the FDDI adaptor involves crossing the IRIX socket layer, which is prohibitively expensive. Normally, in a user-space implementation of the *x*-kernel, a simulated device driver is configured below the media access control layer (in this case, FDDI). The simulated driver uses the socket interface to emulate a network device. To avoid this socket-crossing cost, we replaced the simulated driver with in-memory device drivers for the TCP protocol stacks. The drivers emulate a high-speed FDDI interface, and support the FDDI maximum transmission unit (MTU) of slightly over 4K bytes. This is similar to the approaches taken in [3, 13, 20, 34].

The drivers act as senders or receivers, producing or consuming packets as fast as possible, to simulate the behavior of simplex data transfer over an error-free network. To minimize execution time and experimental perturbation, the receive-side drivers use preconstructed packet templates. They do not calculate TCP checksums, MD5 hash values, or encrypt packets for decryption. Instead, in experiments that use a simulated sender, checksums, signatures, and decryptions are all performed, but the results are ignored, and assumed correct.

Figure 2 shows a sample test configuration, in this case a TCP/MD5/IP stack. Our test environment measures throughput through the network subsystem on the multiprocessor; it does not measure external factors such as latency across the wire to a remote host.

4 Results

We present results for packet-level and connection-level parallelism here.

4.1 Packet-Level Parallelism

Figure 3 shows the sender's throughput rates for our packet-level parallel (PLP) experiments. Times given are in megabits per second for several protocol stacks. Our baseline Internet stack consists of TCP/IP/FDDI, representing protocol processing without any security. A second stack is an Internet stack with MD5 between TCP and IP, representing the work done for an application that requires authentication and integrity but no confidentiality. Our third stack uses DES above TCP and MD5 below TCP, which supports both confidentiality and integrity. Our fourth stack is the same as the third, except that we use triple-DES instead of DES.

These throughputs were measured on our 12-processor Challenge machine, using a single TCP connection with 4 KB packets. For these and all subsequent graphs, each data point is the average of 10 runs, where a run consists of measuring the steady-state throughput for 30 seconds, after an initial 30 second warmup period. Throughput graphs include 90 percent confidence intervals.

Figure 3 quantifies the slowdown due to the use of cryptographic protocols. The baseline speed for the send-side TCP stack is roughly 138 Mbits/sec. Adding MD5 to the stack reduces throughput by nearly an order of magnitude, to a mere 18 Mbits/sec[2] . Adding DES on top of TCP reduces throughput nearly 2 orders of magnitude, to 4.6 Mbits/sec. Using Triple-DES is 3 times slower at 1.5 Mbits/sec.

Figure 4 shows the corresponding relative speedup for the send-side tests, where speedup is throughput normalized relative to the uniprocessor throughput for the appropriate stack. The theoretical ideal linear speedup is included for comparison. Previous work [3, 24] has shown limited performance gains when using packet-level parallelism for a single TCP connection, barring any other protocol processing, and this is reflected by the baseline TCP/IP stack's minimal speedup. This is because manipulating a TCP connection's state is large relative to the IP and FDDI processing and must occur inside a single locked, serial component. Of course, throughput can be improved by using multiple connections.

However, as more compute-intensive cryptographic protocols are used, while the throughput goes down, the relative speedup improves. For example, the MD5 stack achieves a speedup of 8 with 12 processors, and the DES and Triple-DES stacks produce very close to linear speedup. This is because the cost of cryptographic protocol processing, which outweighs the cost of TCP processing, occurs outside the scope of any locks.

[2] MD5 runs 30-50% slower on big-endian hosts [37], such as our Challenge.

Protocol Stack	Send Side	Inc. Cost	Recv Side	Inc. Cost
TCP/IP	236		189	
TCP/MD5/IP	1737	1500	1708	1519
DES/TCP/MD5/IP	8982	8746	9179	8990
3-DES/TCP/IP	21461	21225	21121	20932

Table 1: PLP Latency Breakdown (usec)

Figures 5 and 6 show the throughput and speedup respectively for the same stacks on the receive side. Again we observe successively lower throughputs but better relative speedups as more compute intensive cryptographic protocols are used. The speedup curve for 3-DES, which is the most compute-intensive, is essentially linear.

Table 1 gives the latency breakdown for the various protocol stacks. Latency here is calculated from the uniprocessor throughput, to gain an understanding of the relative cost of adding cryptographic protocols. The table includes total time and incremental overhead for adding a protocol in addition to the baseline TCP/IP processing time. The incremental cost for MD5 is about 1500 usec, for DES about 7200 usec, and for 3-DES about 21000 usec. Since these costs are incurred outside the scope of any locks, they can run in parallel on different packets.

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 30 processors. We expect that more compute-intensive protocols, such as RSA, would also scale linearly.

We also ran similarly configured UDP-based stacks, not shown due to space limitations. In general, the results were similar, except that single-connection parallelism with the baseline UDP stacks exhibited much better speedup that the single-connection baseline TCP stacks. However, as cryptographic processing is used, the differences in both throughput and speedup between TCP-based and UDP-based stacks essentially disappear.

4.2 Connection-Level Parallelism

Figures 7 and 8 show send-side throughput and speedup respectively for our Connection-Level Parallel (CLP) experiments. In these experiments, 12 connections are measured, and the number of processors is varied from 1 to 12. The throughput graphs here show aggregate throughput for all 12 connections. We use the same protocols and experimental

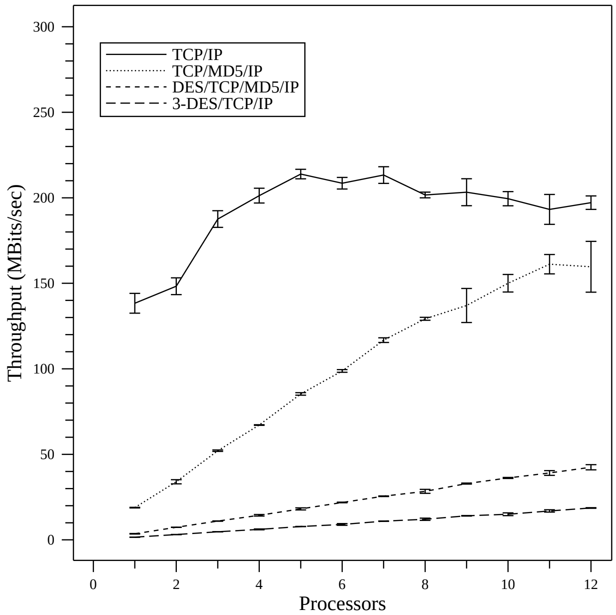

Figure 3: PLP Send-Side Throughput

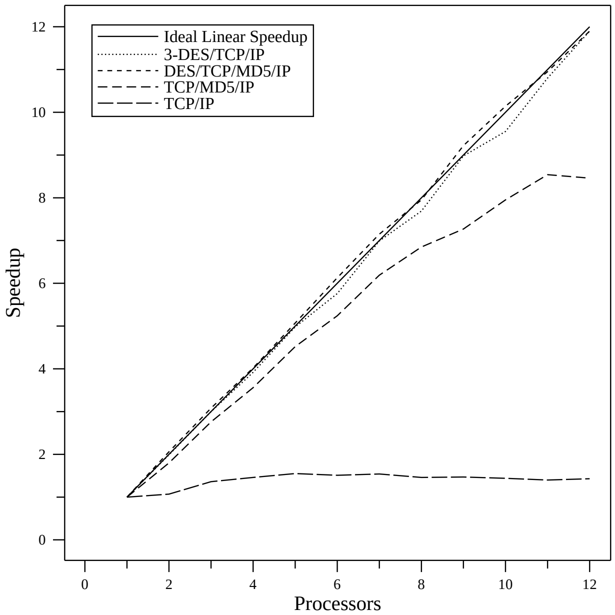

Figure 4: PLP Send-Side Speedup

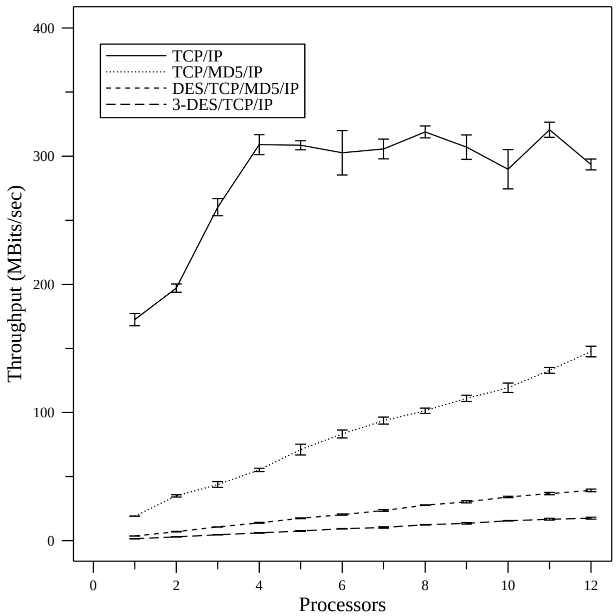

Figure 5: PLP Receive-Side Throughput

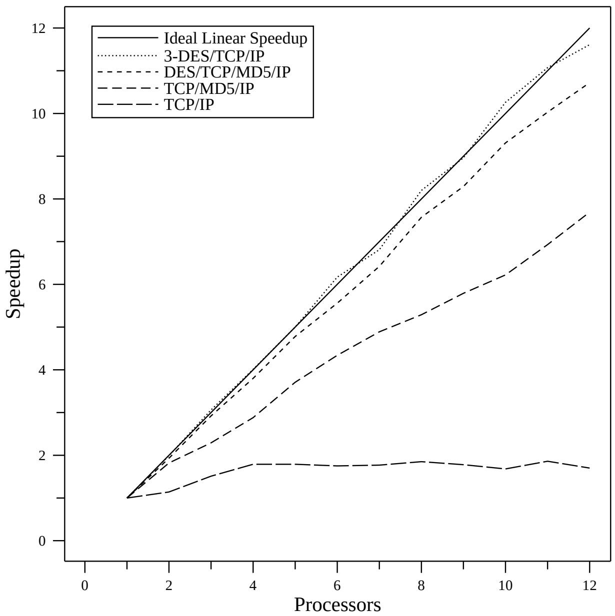

Figure 6: PLP Receive-Side Speedup

methodology as with the PLP experiments, with the one exception that in some cases, the warmup period is longer than 30 seconds.

As with packet-level-parallelism, the throughputs decline as more compute-intensive cryptographic operations are performed. Previous work [38] showed good speedup for CLP, barring any other protocol processing. Figure 8 illustrates this in the baseline case (i.e., the TCP/IP stack). CLP exhibits good speedup when the workload has a sufficient number of active connections because each connection is serviced by its own thread, and there is no explicit synchronization between threads. Figure 8 also shows that when any cryptographic processing is added to the scenario, speedup becomes essentially linear. This is because speedup for CLP depends on the ratio of compute-bound vs. memory bound processing. The more compute-bound an application is, the better the speedup will be. The more memory-bound an application is, the greater the likelihood that memory contention between processors will limit speedup. Since cryptographic protocols are compute-intensive, CLP stacks using them exhibit better speedup.

An interesting result is that while the two approaches to parallelism behave very differently in the baseline cases (i.e., the standard TCP/IP stacks), as more cryptographic processing is done, the more similar the schemes appear in both in terms of throughput, speedup, and latency. For example, in the send-side experiment using DES, both schemes have a throughput of roughly 50 Mbits/sec with 12 processors. Again, this is because the cryptographic processing, which is similar in the two schemes, vastly outweighs differences between the two approaches, as well as the differences in the experiments (i.e., single vs. multiple connections).

Figures 9 and 10 show that for the receive side, the baseline experiments show better throughput and speedup than for the send side. However, once cryptographic protocols are added, the same trends are exhibited.

Table 2 gives the appropriate latency breakdown for the connection-level parallel stack. Again, incremental costs are relative to the baseline TCP/IP stack. The incremental cost for MD5 in this case is about 1550 usec, for DES about 7100 usec, and for 3-DES about 21500 usec.

Figure 11 shows how connection-level parallelism scales with large numbers of connections for the send side. In this experiment, 12 processors were used, and the workload varied from 12 to 384 connections. Two forms of connection-level parallelism were used here: thread-per-connection, where the number of threads is equal to the number of connections, and processor-per-connection, where the number of threads equals the number of processors Note that at 12 connections the two schemes are equivalent.

Previous work [38] has shown that, barring any other protocol processing, thread-per-connection achieves higher aggregate throughput than processor-per-connection, but that

Protocol Stack	Send Side	Inc. Cost	Recv Side	Inc. Cost
TCP/IP	226		214	
TCP/MD5/IP	1772	1546	1769	1555
DES/TCP/MD5/IP	8694	8468	8985	8771
3-DES/TCP/MD5/IP	23150	22924	23237	23023

Table 2: CLP Latency Breakdown (usec)

processor-per-connection distributes the aggregate bandwidth more fairly between connections. Figure 11 shows that, as cryptographic protocol processing is added, the difference in performance between the thread-per-connection and processor-per-connection versions of CLP disappear. Again, this is because cryptographic computation overwhelms any costs due to memory referencing or increasing the parallelism by using more threads.

5 Discussion

In this paper we have shown how both packet-level and connection-level parallelism can be used to improve cryptographic protocol performance. We have not addressed functional parallelism or more fine-grained approaches to parallelized cryptography, such as using multiple processors to encrypt a single message in parallel. Such an approach would not only improve throughput, but might also reduce the latency as seen by an individual message.

For example, DES in Electronic Code Book (ECB) mode can be run in parallel on different blocks of a single message. However, DES using ECB is susceptible to simple-substitution code attacks and cut-and-paste forgery, both of which are realistic worries in computer systems which send large amounts of known text. Thus, most DES implementations use CBC mode, where a plaintext block is XOR'ed with the ciphertext of the previous block, making each block dependent on the previous one, and preventing a parallelized implementation. However, each 8 byte block of a message encrypted with DES in CBC mode could be decrypted in parallel, since computing the plaintext block requires only the key, the ciphertext block, and the previous ciphertext block.

In practice, DES CBC must be used with some form of message integrity check to thwart cut-and-paste forgeries. MD5 is not amenable to fine-grain parallelism, and this limits the opportunities for applying these methods. Some avenues for research include finding faster or parallelizable message integrity algorithms, and combining these with DES modes that allow finer-grain parallel encryption techniques, and especially modes that allow the sender and receiver to use different processing granularities.

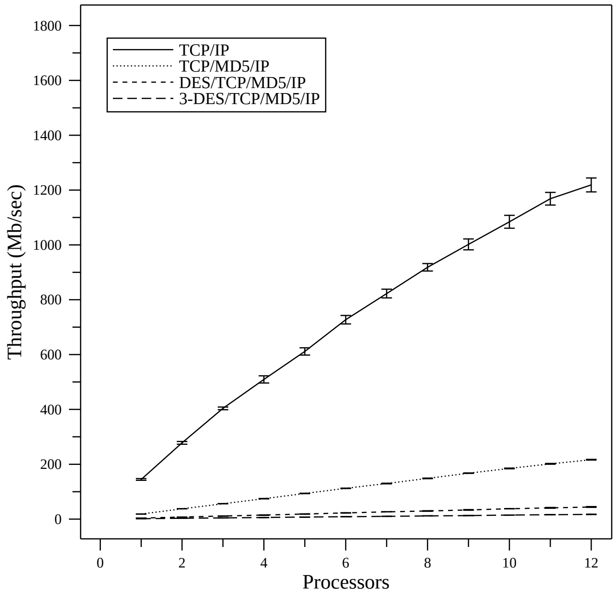

Figure 7: CLP Send-Side Throughput

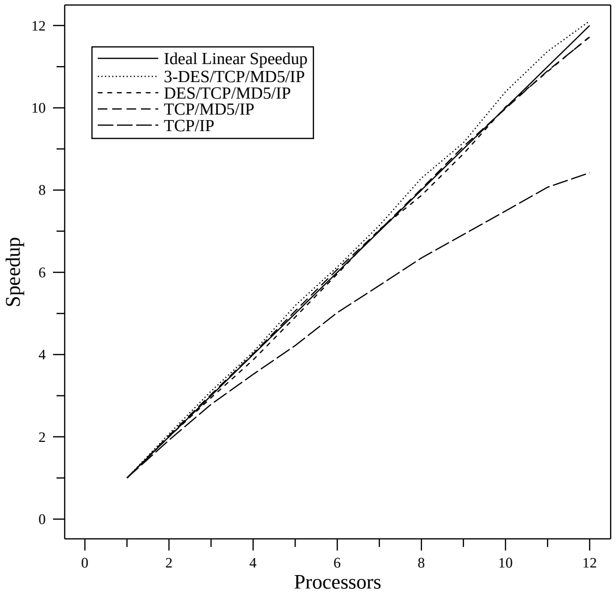

Figure 8: CLP Send-Side Speedup

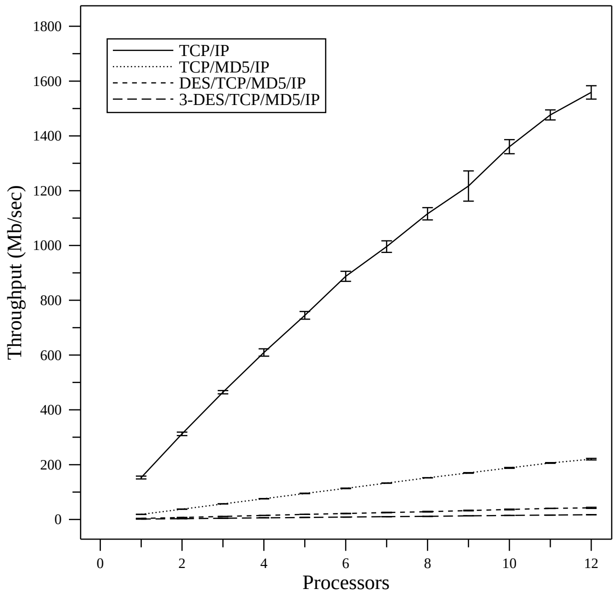

Figure 9: CLP Receive-Side Throughput

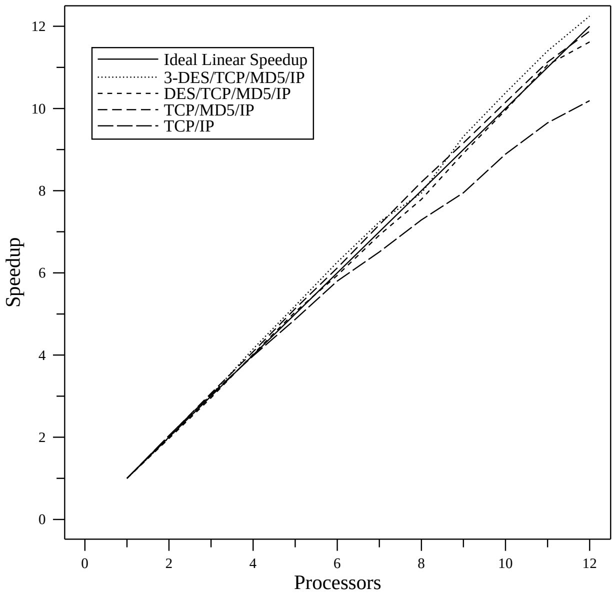

Figure 10: CLP Receive-Side Speedup

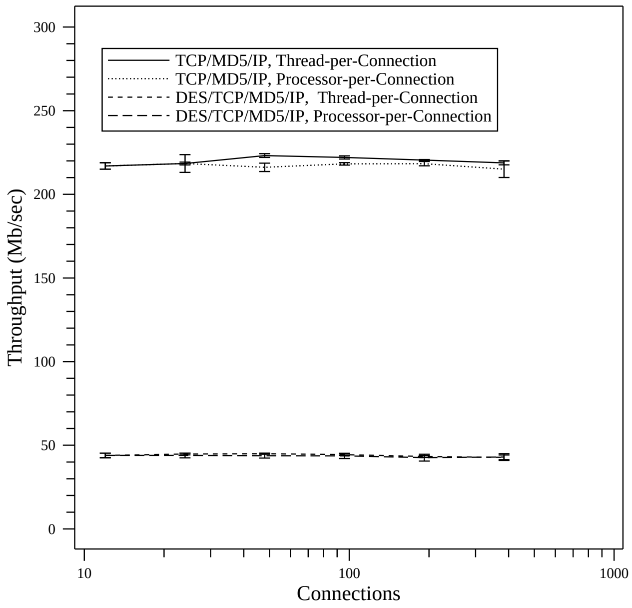

Figure 11: CLP Send-Side, Large Numbers of Connections

Although fine-grained approaches to parallelism may be attractive, both packet-level parallelism and connection-level parallelism have the important advantage that they can be used with a wide variety of protocols, and are easier to exploit on protocols that are already deployed. Fine grained approaches are more dependent on the algorithm of the protocol itself, and the opportunities for finding fine-grained concurrency seem more limited without developing new algorithms. Of course, an interesting question is whether an algorithm could be designed that could be run in parallel yet still have sufficient cryptographic strength. One inference seems clear: layer parallelism would be of minimal use to a secure server. Given the predominance of cryptographic processing times, layer parallelism would be essentially be limited to the throughput of a dedicated single processor.

Finally, we have not addressed adversarial issues. Parallelism is clearly useful to an attacker, who can use multiple processors to speed a brute-force cracking attempt.

6 Summary

We briefly summarize our findings as follows:

- Parallelism is an effective means of improving cryptographic performance, both using packet-level parallelism and connection-level parallelism.
- Under both approaches, relative throughput declines as more compute-intensive protocols are used. On the other hand, speedup relative to the uniprocessor case improves.
- In packet-level parallelism, speedup is essentially linear when DES or any more compute-intensive protocol is used.
- In connection-level parallelism, speedup is essentially linear when MD5 or any more compute-intensive protocol is used.

Due to the compute-bound nature of cryptographic protocols, we observe good scalability for parallelized network security protocols. Both packet-level and connection-level parallelism are appropriate vehicles for servers that need transfer large amounts of data securely. However, we plan to investigate more fine-grained approaches in future work.

Acknowledgments

Special thanks to Jim Kurose and Don Towsley for their advice, support, and comments on earlier drafts of this paper. Joe Touch also provided useful feedback.

References

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

[2] R. Atkinson. Security architecture for the Internet Protocol. Request for Comments (Draft Standard) RFC 1825, Internet Engineering Task Force, Aug. 1995.

[3] M. Björkman and P. Gunningberg. Locking effects in multiprocessor implementations of protocols. In *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, pages 74–83, San Francisco, CA, Sept. 1993.

[4] D. Borman, R. Braden, and V. Jacobson. TCP extensions for high performance. Request for Comments (Proposed Standard) RFC 1323, Internet Engineering Task Force, May 1992.

[5] D. D. Clark. Modularity and efficiency in protocol implementation. Request for Comments RFC 817, Internet Engineering Task Force, July 1982.

[6] D. D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 171–180, December 1985.

[7] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, Nov. 1976.

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

[9] M. C. et al. SPARCCenter 2000:Multiprocessing for the 90's! In *Proceedings IEEE COMPCON*, pages 345–353, San Francisco CA, February 1993.

[10] D. M. Fenwick, D. J. Foley, W. B. Gist, S. R. VanDoren, and D. Wissel. The AlphaServer 8000 series: High-end server platform development. *Digital Technical Journal*, 7(1):43–65, 1995.

[11] M. Galles and E. Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. Technical report, Silicon Graphics Inc., Mt. View, CA, May 1994.

[12] D. Giarrizzo, M. Kaiserswerth, T. Wicki, and R. C. Williamson. High-speed parallel protocol implementation. *First IFIP WG6.1/WG6.4 International Workshop on Protocols for High-Speed Networks*, pages 165–180, May 1989.

[13] M. W. Goldberg, G. W. Neufeld, and M. R. Ito. A parallel approach to OSI connection-oriented protocols. *Third IFIP WG6.1/WG6.4 International Workshop on Protocols for High-Speed Networks*, pages 219–232, May 1993.

[14] K. E. Hickman and T. Elgamal. The SSL protocol. Work in progress, Internet Draft (ftp://ds.internic.net/internet-drafts/draft-hickman-netscape-ssl-01.txt, June 1995.

[15] 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.

[16] V. Jacobson. Efficient protocol implementation. In *ACM SIGCOMM 1990 Tutorial Notes*, Philadelphia, PA, Sept. 1990.

[17] O. G. Koufopavlou and M. Zitterbart. Parallel TCP for high performance communication subsystems. In *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM)*, pages 1395–1399, 1992.

[18] S. J. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[19] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of Ethernet traffic. *IEEE/ACM Transactions on Networking*, 2(1):1–15, Feb. 1994.

[20] B. Lindgren, B. Krupczak, M. Ammar, and K. Schwan. Parallel and configurable protocols: Experience with a prototype and an architectural framework. In *Proceedings of the International Conference on Network Protocols*, pages 234–242, San Francisco, CA, Oct. 1993.

[21] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings 11th Symposium on Operating System Principles*, pages 39–51, Austin, TX, November 1987.

[22] J. C. Mogul. Network locality at the scale of processes. *ACM Transactions on Computer Systems*, 10(2):81–109, May 1992.

[23] E. Nahum, S. O'Malley, H. Orman, and R. Schroeppel. Towards high-performance cryptographic software. In *Proceedings of the Third IEEE Workshop on the Architecture and Implementation of High Performance Communications Subsystems (HPCS)*, Mystic, Conn, Aug. 1995.

[24] 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*, pages 125–137, Monterey, CA, Nov. 1994.

[25] A. N. Netravali, W. D. Roome, and K. Sabnani. Design and implementation of a high-speed transport protocol. *IEEE Transactions on Communications*, 38(11):2010–2024, Nov. 1990.

[26] 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.

[27] V. Paxson and S. Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[28] T. F. L. Porta and M. Schwartz. Performance analysis of MSP: Feature-rich high-speed transport protocol. *IEEE Transactions on Networking*, 1(6):740–753, Dec. 1993.

[29] D. Presotto. Multiprocessor Streams for Plan 9. In *Proceedings United Kingdom UNIX Users Group*, Jan. 1993.

[30] E. Rescorla and A. M. Schiffman. The secure hypertext transfer protocol. Work in progress, Internet Draft (ftp://ds.internic.net/internet-drafts/draft-ietf-wts-shttp-00.txt, July 1995.

[31] R. Rivest. The MD5 message-digest algorithm. Request for Comments (Informational) RFC 1321, Internet Engineering Task Force, Apr. 1992.

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

[33] S. Saxena, J. K. Peacock, F. Yang, V. Verma, and M. Krishnan. Pitfalls in multithreading SVR4 STREAMS and other weightless processes. In *Winter 1993 USENIX Technical Conference*, pages 85–96, San Diego, CA, Jan. 1993.

[34] D. C. Schmidt and T. Suda. Measuring the impact of alternative parallel process architectures on communication subsystem performance. *Fourth IFIP WG6.1/WG6.4 International Workshop on Protocols for High-Speed Networks*, Aug. 1994.

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

[36] B. Schneier. *Applied Cryptography*. John Wiley and Sons, Inc., New York, NY, 1994.

[37] J. Touch. Performance analysis of MD5. In *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, Boston MA, Aug. 1995.

[38] D. J. Yates, E. M. Nahum, J. F. Kurose, and D. Towsley. Networking support for large scale multiprocessor servers (extended abstract). In *Proceedings of the Third IEEE Workshop on the Architecture and Implementation of High Performance Communications Subsystems (HPCS)*, Mystic, Conn, Aug. 1995. A full version of this paper is available as Technical Report CMPSCI 95-83, University of Massachusetts.