Accelerating Application-Level Security Protocols

Matthew Burnside
Department of Computer Science
Columbia University
mb@cs.columbia.edu

Angelos D. Keromytis
Computer Science Department
Columbia University
angelos@cs.columbia.edu

***Abstract*— We present a *minimal* extension to the BSD socket layer that can improve the performance of application-level security protocols, such as SSH or SSL/TLS, by 10%, when hardware cryptographic accelerators are available in the system. Applications specify what cryptographic transforms must be applied to incoming and outgoing data frames, and such processing is applied by the operating system itself (exploiting hardware accelerators) when the application sends or receives data. Under this scheme, we can reduce the number of system calls and context switches by 50%, and the amount of data copying by 66%. We describe our prototype implementation for the OpenBSD system and quantify its performance implications. We conclude with a discussion of further possible performance improvements that our approach enables.**

Keywords: *Cryptography, SSL/TLS, operating system kernel, sockets, zero-copy.*

I. Introduction

Cryptographic protocols are a fundamental building block for securing distributed systems. With the increasing need to secure Internet traffic, a variety of such protocols has emerged and is seeing increasing use. SSL/TLS [1], SSH, and IPsec [2] are increasingly in use, as recent measurements indicate [3].

To further facilitate wide-spread adoption of such protocols, it is important to address specific concerns users of this technology have raised. These concerns can be distinguished in two broad categories: management and performance. While the former is an extremely important issue, in this paper we focus on performance. In particular, we examine two of the most widely-used network security protocols: TLS and SSH. More generally, we are interested in accelerating security protocols that are implemented as user-level processes, in contrast to protocols implemented inside the operating system kernel (*e.g.,* IPsec). Previous work in accelerating IPsec [4] is not applicable here because of the different set of constraints.

Other work has investigated performance improvements of TLS [5], [6], [7], [8] and offered recommendations on how to improve the performance of the session initialization phase of the protocol, which contains several heavy-weight public key operations. Similar improvements can be applied to the key exchange phase of SSH. Furthermore, recognizing the increasing importance of TLS, several hardware vendors have produced cryptographic accelerator cards that can be used both for the public-key (*e.g.,* RSA) and the data-encryption (using symmetric-key ciphers such as DES, RC4, *etc.*). operations. The OpenBSD Cryptographic Framework (OCF) [9] offers a general interface to such accelerators that can be used to improve the overall performance of such protocols.

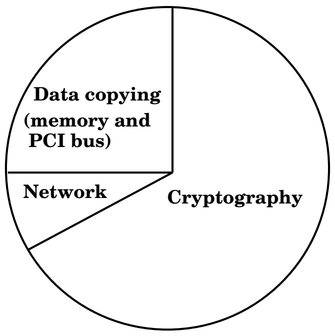

Fig. 1. **Cost breakdown for bulk-data transfer using TLS and a hardware accelerator.**

[4] investigated the overheads of the data-transfer component of TLS and determined that such acceleration can greatly improve the throughput of bulk data-transfer over the software-only case by up to 100% when such computationally-expensive algorithms as 3DES are used, and even up to 400% with slow CPUs. In [9], we determined that the key limiting factors to faster operation are the bandwidths of the PCI and the memory bus. Our cost-breakdown for the various costs, summarized in Figure 1, shows that over 20% of the processing time is spent copying data over these two buses. Such findings have been reported in the past for other systems [10], [11], [12], [13], and are by no means limited to cryptography or network I/O.

The solution to these problems is conceptually straightforward: integrate network and cryptographic processing, as shown in Figure 2, such that there are no diversions from the regular data path. Another proposal, when a separate accelerator card is used (as is most common in practice), is to minimize data copying between the user-level application (*e.g.,* the web server) and the kernel. Zero-copy I/O, whereby the kernel uses the MMU to re-map user-process memory pages in the kernel address space and vice versa, thus greatly reducing data copying and memory-bus contention, can be used to implement the latter. Unfortunately, implementing zero-copy I/O has great implications for all of the operating system and it requires extensive modifications to applications to achieve the best performance. Furthermore, zero-copy by itself cannot be used to take advantage of integrated network/crypto cards.

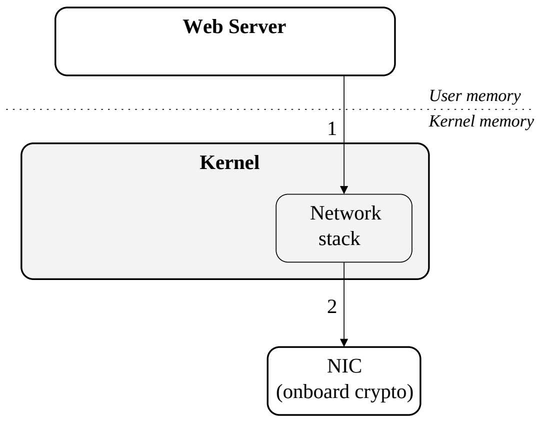

Fig. 2. **Integrated NIC / cryptographic accelerator card.**

We present a simple extension to the OpenBSD kernel that immediately improves the performance of TLS and other similar protocols by 10%, compared to the case of using the OCF directly from the web server. The necessary changes to the operating system kernel consisted of less than 80 lines of C code. Our scheme is to instrument the socket processing code to perform the needed cryptographic processing as data are sent to or received from the network. Applications simply instruct the kernel when to start and stop applying the transforms and provide the necessary keying material. Our extension can also be used in the presence of integrated network/crypto cards, by attaching a description of the necessary transforms to the *mbuf* using the OpenBSD *mbuf tags*. Although our implementation was done for OpenBSD, it is easily portable to other operating systems, especially those that implement System V STREAMS functionality (*e.g.,* Solaris).

The remainder of this paper is organized as follows. Section II gives a brief overview of related work. Section III presents the OpenBSD Cryptographic Framework (OCF) and Section IV describes our extensions to the OpenBSD kernel and gives a preliminary evaluation of its performance. We discuss potential improvements and future work in Section V and conclude in Section VI.

II. Related Work

There has been a considerable amount of work on the enhancement of system performance through the addition of cryptographic hardware [14]. This early work was characterized by its focus on the hardware accelerator rather than its implications for overall system performance. [15] began examining cryptographic subsystem issues in the context of securing high-speed networks, and observed that the bus-attached cards would be limited by bus-sharing with a network adapter on systems with a single I/O bus. A second issue pointed out in that time frame [16] was the cost of system calls, and a third [10], [11], [12], [13] the cost of buffer copying. These issues are still with us, and continue to require aggressive design to reduce their impacts.

As interest in security is currently in an upswing, recent work has been examining the overall performance impact of security technologies in real systems. Work by Coarfa, *et al.* [6] has focused on the impact of hardware accelerators in the context of TLS web servers using a trace-based methodology, and concludes that there is some opportunity for acceleration, but given the choice one might prefer a second processor as it also assists with the substantial (and perhaps dominant) non-cryptographic overheads. [4] provides some basic performance characterizations of IPsec as well as other network security protocols, and the impact acceleration has on throughput. The authors conclude that the relative cost of high-grade cryptography is low enough that it should be the default configuration.

[5] describes a technique for improving SSL handshake performance. It demonstrates that it is faster to do n SSL handshakes as a batch than n handshakes individually, based on a technique for batching RSA decryptions. It also shows a speedup factor of 2.5 for $n = 4$. It is important to note that this speedup only applies to the handshake portion of the SSL connection, not to the data transport itself. By caching session keys, the authors of [7] demonstrate a reduction in download time of secure web documents of between 15% and 50%. Again, this technique only accelerates the handshake portion of the SSL connection, without reducing the data transport time.

III. The OpenBSD Cryptographic Framework

The OpenBSD cryptographic framework (OCF) [9] is an asynchronous service virtualization layer inside the kernel, that provides uniform access to cryptographic hardware accelerator cards. It supports two classes of algorithms: symmetric (*e.g.,* DES, AES, MD5) and asymmetric (*e.g.,* RSA). Symmetric-algorithm (*e.g.,* DES, AES, MD5) operations are built around the concept of the *session,* so as to take advantage of session-caching features available in many hardware accelerators. Asymmetric algorithms are implemented as individual operations.

To use the OCF, other kernel subsystems (consumers) first create a session with the OCF specifying the algorithm(s) to use, mode of operation (*e.g.,* CBC, HMAC, *etc.*), cryptographic keys, initialization vectors. The OCF supports algorithm-chaining, *i.e.,* performing encryption and integrity-protection in one operation. Such combined operations are heavily used by almost all data-transfer security protocols, such as TLS, SSH, and IPsec. The OCF determines which card to use based on its capabilities, and creates the relevant state by invoking the driver.

For the actual encryption/decryption, consumers specify the data to be processed, a callback function, and various offsets that indicate where the encryption should start and end, where the message authentication code (MAC) should be placed, where the initialization vector can be found (if it is already present on the buffer) or where on the output buffer it should be written (if at all). Once the request is processed, the callback routine is called by the kernel. If an error has occurred, the callback routine is responsible for any corrective action. When multiple producers implement the same algorithms, the OCF can load-balance sessions across them.

A. The /dev/crypto Interface

To allow user-level processes to take advantage of hardware acceleration facilities, a `/dev/crypto` device driver abstracts all the OCF functionality and provides a command set that can be used by OpenSSL (or other software using the `/dev/crypto` interface directly). This interface is based on *ioctl()* calls. Similar to the OCF itself, this uses a session-based model, since the general case assumes that keys will be reused for a sequence of operations. After opening the `/dev/crypto` device and gaining a file descriptor `fd`, the caller requests that a new session be created for a certain cryptographic operation, and specifies all related parameters (*e.g.,* keys). A single session can support both a cipher and a MAC.

Once a session is established, blocks can be encrypted or decrypted using the `CIOCCRYPT` *ioctl()*. Each time this is used, the caller can specify a new IV or MAC information that they wish to fold into the operation. Input and output buffers are specified via separate pointers, but they can point to the same buffer for in-place encryption. Naturally, the data size provided by the caller must be rounded to the default block size of the algorithm being used. A data size limit of 262,140 bytes exists at the moment, to hide a similar limit found in some chipsets. The userland data blocks are copied into memory allocated inside the kernel. The OCF is then called to perform the operation using the initialization information stored in the application's `/dev/crypto` session. If the operation succeeds, the results are copied back to the application buffers. The cost of these copies is high for large block sizes.

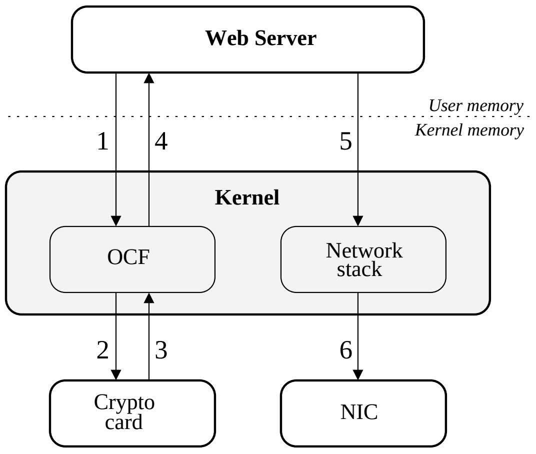

Fig. 3. **Encrypting and transferring a buffer.**

IV. Our Approach

In network-security protocols that use cryptographic accelerators, the user-level process that implements the protocol, *e.g.,* a web server serving HTTPS requests, issues one or more crypto requests via `/dev/crypto,` followed by a *write()* or *send()* call to transmit the data, as shown in Figure 3. The server uses the previously described`/dev/crypto` device driver to pass data to the OCF (step 1), which delegates the SSL encryption and MAC'ing to the cryptographic accelerator card (step 2). The card performs the requested operations and the data are returned to the server (steps 3 and 4). The server uses the *write()* system call to pass the data to the network stack (step 5), which transmits the data on the wire using a network interface card (step 6). Similarly, a *read()* or *recv()* is followed by a number of requests to `/dev/crypto.`

This implies considerable data copying to and from the kernel, and unnecessary process context switching. An alternative approach is to "link" some crypto context to a socket or file descriptor, such that data sent or received on that file descriptor are processed appropriately by the kernel. The impact of such data copying has been recognized in the past, as we saw in Section II, and has impacted TLS performance, as shown in [4]. Recalling Figure 1, as much as 25% of the overhead can be attributed to data copying. As cryptographic hardware improves, the relative importance of this overhead will increase.

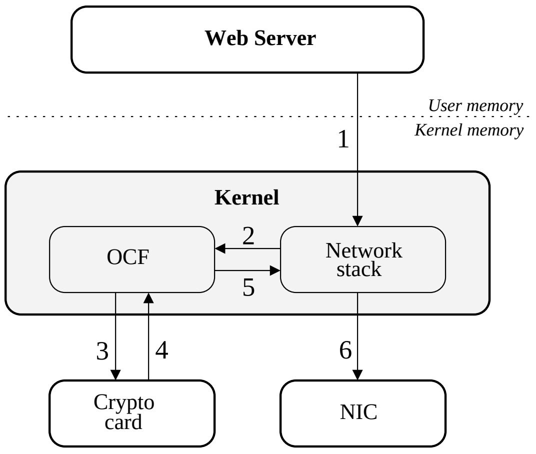

Fig. 4. **Encrypting and transferring a buffer, with socket layer extensions.**

Our approach reduces the per-buffer cost down to a single *write()* and two context switches. (This is the same penalty as sending a buffer over the network with no crypto at all.) The fundamental change is that the network stack is crypto-aware. Figure 4 demonstrates how a buffer is encrypted and transferred using our extensions to the socket layer; the web server passes the buffer to the network stack using *write()* (step 1), which passes the data directly to the OCF (step 2). The OCF delegates the crypto to the accelerator card (step 3), as before, but the data are returned to the network stack (step 5) rather than application memory. The network stack then finishes processing the data and transmits it (step 6).

A. Implementation

We implemented the extension by modifying the socket layer of the OpenBSD network stack. Using a new socket option, SO_CRYPT, an application-level crypto consumer defines the cryptographic transforms for each packet (*e.g.*, where the encryption should start and end, where the MAC should be placed, and so on). Then, when *sosend()* is called with SO_CRYPT set, *sosend()* passes the *mbuf* containing the data to be sent, along with the cryptographic transforms, to the OCF. At this time, *sosend()* calls *tsleep()*, and waits for OCF's callback. When the callback arrives, we replace the *mbuf* data with the newly encrypted data, and allow the control flow to continue with the network processing. The complete implementation comes to less than 100 lines of code.

The only complication arises from a quirk of OpenBSD's *write()* system call[1]: *sosend()* only receives at most 4088 bytes per invocation. Larger *write()* operations on a socket or file descriptor result in multiple invocations of *sosend()*. Since TLS frames can be up to almost $2^{16} - 2$ bytes, we must handle logical frames larger than the data buffer handed to *sosend()*.

We can detect the fact that a frame that spans multiple *write()* invocation is being transmitted by examining the TLS or SSH header, as discussed in Section IV-B. To solve the problem, we can use either of two approaches:

- The easiest solution involves incrementally computing the frame MAC from its component buffers, and linking the CBC encryption across different packets by keeping the last block of each for use as the initialization vector (IV) of the next. This way, we can process the TLS frame piece-meal. The drawback is that we incur a higher overhead than if we performed a single large transaction with the OCF.
- The best solution is to buffer data in the socket until a complete frame can be reconstructed, thus allowing a single request to be issued to the OCF.

In the interest of time, we decided to implement the first solution. This, unfortunately, has had a negative impact to our benchmarks for large transactions (8KB and 16KB), as we discuss in Section IV-C.

B. SSH and SSL/TLS

Since the OCF allows the composition of cryptographic transforms, the above scheme integrates seamlessly with SSH and SSL/TLS. Connection setup for both protocols is quite complex, as it must protect against various attacks. However, once the setup protocol is completed, the much simpler data exchange phase begins. The binary packet protocol for the data exchange phase of TLS using a CBC block cipher (such as DES or RC2) is [1]:

```
byte     type
byte     MajorVersionNumber
byte     MinorVersionNumber
uint16   length
byte[]   data
byte[]   MAC
byte[]   padding
byte     padding_length
```

[1]We have not investigated whether other BSDs exhibit the same behavior.

When the data exchange begins, we set the SO_CRYPTO option with the appropriate MAC and encryption algorithms and keys. The MAC covers from the beginning to the end of data, and is placed at the field called MAC. Encryption covers the data, MAC, the padding, and the padding_length. Similarly, with the binary packet protocol for SSH version 2:

```
uint32   packet_length
byte     padding_length
byte[]   data
byte[]   padding
byte[]   MAC
```

When the data exchange begins, we enable the SO_CRYPTO option in the socket, and set the offsets for the start of the MAC and encryption cipher to the first byte of data, and for the end to the last byte of padding. We set the offset for placing the MAC to the MAC field.

C. Evaluation

For our tests, we use two identical machines. The machines have 1.4 Ghz Pentium III processors on Tyan Thunder HEsl-T motherboards. These motherboards have three independent PCI buses: 32bit/33Mhz/5V, 64bit/66Mhz/5V, and 64bit/66Mhz/3.3V. The boards use 512MB of 133Mhz registered SDRAM and are based on the ServerWorks HESL chipset. For network testing, we use SysKonnect 9843 multimode fiber Gbit Ethernet cards. The machines are interconnected directly with a cross-over fiber cable. We placed a Gbit Ethernet card and a Broadcom 5820 cryptographic accelerator on the 64/66 bus of each machine. The peak performance of the 5820 card, as given by the manufacturer, is 310 Mbps of 3DES-SHA1 in IPsec (no other performance indications are given, but IPsec processing is very close in terms of cryptographic operations to that of TLS and SSH).

We ran three tests: encrypt a file in user space (with the OpenSSL EVP framework), encrypt a file using /dev/crypto, and encrypt a file in the kernel using our extension. In all cases, the encrypted data is then transmitted to a remote machine over a TCP connection. Table I contains the raw data; how many seconds it took to transfer the file in 4KB, 8KB, and 16KB packets[2], using each of the three methods. Table II shows that our approach gives a 6-10% improvement over /dev/crypto.

As we mentioned in Section IV-A, during the evaluation we discovered that OpenBSD fragments large *write()* operations into 4088 byte chunks before transmitting them. This likely reduces the benefits of our approach when transmitting packets larger than the 4088 byte limit.

V. Discussion and Future Work

The prototype implementation we described in Section IV does not currently handle incoming data decryption. Such data are passed on directly to the application. Implementing this feature is relatively simple: once the application turns on

[2]To be precise, we used multiples of the *write()* quantum, 4088 bytes, as discussed in Section IV-A.

TABLE I

Time to encrypt and transfer a 100MB file (in seconds), for 4k, 8k, and 16k packets. The file is encrypted with OpenSSL's EVP framework, the `/dev/crypto` device driver, or our in-kernel approach.

Scenario/Buffer size	4k	8k	16k
EVP	32.79	31.84	32.38
/dev/crypto	4.13	4.23	4.72
kernel	3.80	3.97	4.27

TABLE II

Percentage improvement of our in-kernel approach *vs.* encryption in user space and `/dev/crypto`.

Scenario/Buffer size	4k	8k	16k
EVP	88%	88%	87%
/dev/crypto	8%	6%	10%

socket encryption, we start examining the first few bytes of the incoming data stream, depending on the protocol type (*e.g.,* TLS or SSH, as indicated by the application). These include the total length of the incoming security protocol frame. The kernel will then wait until all the packets carrying data of that frame have arrived before passing them to the OCF for decryption and validation. Once the request is processed, the decrypted frame is passed on to the application.

While the implementation is as straightforward as the data-send case of our prototype, it offers only limited benefits. As was shown in [9], hardware accelerators can be very useful for overloaded servers but do not offer comparable benefits to less-loaded systems, *e.g.,* a workstation.

A similar situation to the multiple kernel crossing scenario is present in the use of the PCI bus: a host that is about to transmit a TLS, SSH, or IPsec packet must first DMA it over the PCI bus to the cryptographic accelerator, DMA it back to main memory, and finally DMA it to the NIC. This decreases the attainable PCI bandwidth to one-third of the theoretical maximum for the bus. If the NIC offers on-chip cryptography, we only need to perform one DMA transfer. However, it is possible to reduce the number of DMA transfers to two (instead of three), even when using a dedicated cryptographic accelerator, by doing card-to-card DMA from the accelerator to the NIC (and the other way around, on packet receipt).

Doing this requires support from the network stack — in particular, deferring of cryptographic operations until right before the packet must be transmitted to the network. In OpenBSD, we developed the *mbuf tags* as a way of attaching ancillary information to packets. This can be used as a signaling mechanism between the socket layer and the NIC driver or other kernel subsystem. We then need to modify the NIC driver to first DMA the packet to the accelerator, and then (once the request is completed) to arrange for a direct DMA transfer to the NIC itself. In the extreme case, we can include the hard drive to the DMA chain, such that data is simply

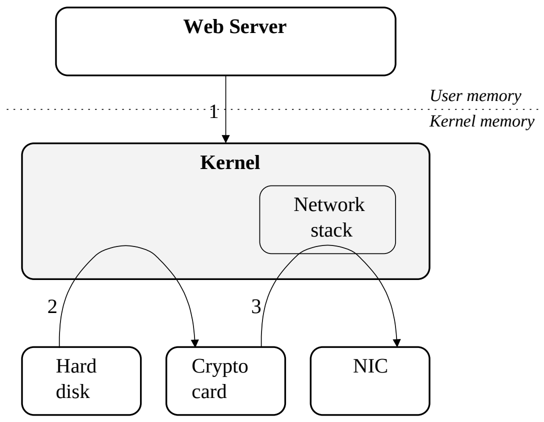

Fig. 5. **DMA chaining across multiple devices.**

DMA'ed between devices, as shown in Figure 5. In that case, the operating system's role becomes that of a flow-controller.

Another potential approach to reducing data copying overhead is to do "page sharing" of data buffers; when a request is given to */dev/crypto*, the kernel removes the page from the process's address space and maps it in its own. When the request is completed, it re-maps the page back to the process's address space, avoiding all data copying. This works well as long as */dev/crypto* remains a synchronous interface. If processes are allowed to have pending requests, accesses to that page while it is being shared with the kernel must be caught and handled, similar to the way copy-on-write of memory pages is handled. Operations that cross page boundaries also have to be dealt carefully. To take full advantage of page sharing, applications will have to be extensively modified to ensure that data buffers are kept in separate pages, and pages that are being shared with the kernel are not accessed while an encryption/decryption request is pending. Finally, page-sharing does not, by itself, take advantage of NICs with integrated cryptographic support, although it can be used to improve the performance in that scenario.

Another I/O performance bottleneck has been that of small requests. Although we did not examine its effects in this paper, previous work [9], [17] has demonstrated that they can have a significant negative impact on the performance. Since many cryptographic protocols (*e.g.,* SSH login), use small requests, the gains from cryptographic accelerators are smaller than one might hope for. There are several possible approaches: request-batching, kernel crossing and/or PCI transaction minimization, or simply use of a faster processor. These are more cost-effective solutions to deploying a hardware accelerator, as has already been pointed in the context of the TLS handshake [6].

VI. Conclusions

Cryptographic protocols are a fundamental building block for securing distributed systems. Although there has been an increase in their use for routine operations such as file transfer and remote login, many users express concern about their impact in performance. As a result, considerable effort has gone into accelerating various aspects of such protocols as TLS, which is widely used to protect web transactions. Most such work to date has focused on the initial handshake aspect of the protocol, while commercially-available cryptographic accelerators are not used to the best of their abilities. To address this problem, we proposed pushing into the operating system kernel some of the application-specific login common to many cryptographic protocols implemented at user level.

Our approach eliminates two redundant data copies between the kernel and the user-level process (*e.g.*, web server) with minimum modifications both to the kernel and the application. We implemented our scheme in the OpenBSD kernel, which provides support for hardware cryptographic accelerators. The implementation was remarkably straightforward, once we overcame some crypto card-specific problems. Our evaluation of the prototype shows an improvement in the data-transfer performance of TLS of approximately 10%. Furthermore, our scheme can easily be extended to permit use of network cards with integrated cryptographic acceleration.

Several possible improvements are in our future plans. First, we intend to extend our scheme to handle transparent data decryption. Second, we plan to investigate combining our system with zero-copy I/O. We believe there are several other optimizations that will allow us to overcome specific I/O deficiencies of the PC architecture, although probably none at such a low level of complexity.

ACKNOWLEDGEMENTS

This work was supported by DARPA and the Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-01-2-0537.

REFERENCES

[1] T. Dierks and C. Allen, "The TLS protocol version 1.0," RFC 2246, Jan. 1999. [Online]. Available: ftp://ftp.isi.edu/in-notes/rfc2246.txt

[2] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol," RFC 2401, Nov. 1998. [Online]. Available: ftp://ftp.isi.edu/in-notes/rfc2401.txt

[3] "OC48 Analysis – Trace Data Stratifi ed by Applications," http://www.caida.org/analysis/workload/byapplication/oc48/port˙analysis%˙app.xml.

[4] S. Miltchev, S. Ioannidis, and A. D. Keromytis, "A Study of the Relative Costs of Network Security Protocols," in *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, June 2002, pp. 41–48.

[5] D. Boneh and N. Shacham, "Improving SSL Handshake Performance via Batching," in *Proceedings of the RSA Conference*, January 2001.

[6] C. Coarfa, P. Druschel, and D. Wallach, "Performance Analysis of TLS Web Servers," in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, February 2002.

[7] A. Goldberg, R. Buff, and A. Schmitt, "Secure Web Server Performance Dramatically Improved By Caching SSL Session Keys," in *Workshop on Internet Server Performance, held in conjunction with SIGMETRICS*, June 1998.

[8] G. Apostolopoulos, V. Peris, and D. Saha, "Transport Layer Security: How Much Does it Really Cost?" in *INFOCOM: The Conference on Computer Communications, joint conference of the IEEE Computer and Communications Societies*, March 1999.

[9] A. D. Keromytis, J. L. Wright, and T. de Raadt, "The Design of the OpenBSD Cryptographic Framework," in *Proceedings of the USENIX Technical Conference*, June 2003.

[10] C. B. S. and J. M. Smith, "Hardware/Software Organization of a High-Performance ATM Host Interface," *IEEE Journal on Selected Areas in Communications (Special Issue on High Speed Computer/Network Interfaces)*, vol. 11, no. 2, pp. 240–253, February 1993.

[11] J. M. Smith and C. B. S. Traw, "Giving Applications Access to Gb/s Networking," *IEEE Network*, vol. 7, no. 4, pp. 44–52, July 1993.

[12] P. Druschel, M. B. Abbott, M. A. Pagels, and L. L. Peterson, "Network subsystem design," *IEEE Network*, vol. 7, no. 4, pp. 8–17, July 1993.

[13] J. Kay and J. Pasquale, "The Importance of Non-Data Touching Processing Overheads in TCP/IP," in *Proceedings of the ACM SIGCOMM Conference*, September 1993, pp. 259–269.

[14] A. G. Broscius and J. M. Smith, "Exploiting Parallelism in Hardware Implementation of the DES," in *Proceedings of CRYPTO*, August 1991, pp. 367–376.

[15] J. M. Smith, C. B. S. Traw, and D. J. Farber, "Cryptographic Support for a Gigabit Network," in *Proceedings of INET*, June 1992, pp. 229–237.

[16] C. Pu, H. Massalin, J. Ioannidis, and P. Metzger, "The Synthesis System," *Computing Systems*, vol. 1, no. 1, 1988.

[17] M. Lindemann and S. W. Smith, "Improving DES Coprocessor Throughput for Short Operations," in *Proceedings of the 10th USENIX Security Symposium*, August 2001, pp. 67–81.