The Case For Crypto Protocol Awareness Inside The OS Kernel

Matthew Burnside Angelos D. Keromytis
Department of Computer Science, Columbia University
{*mb,angelos*}*@cs.columbia.edu*

Abstract

Separation of control and data plane is a principle increasingly used to improve the performance of network protocols and applications, such as the Web. Use of security mechanisms, such as the SSL/TLS protocol, can negate these performance gains, since such mechanisms need to be located on the data path. We argue that the same principle of separation can be applied to security mechanisms, by removing the web server from the secure data path.

We present a *minimal* operating system extension that can improve the performance of web servers using SSL/TLS by up to 27%. Our intuition is that protocol framing and cryptographic transforms can be applied to incoming and outgoing data frames by the operating system under a policy specified by the web server. In this way, we can reduce the number of system calls and context switches to a small constant number, and the amount of data copying that involves the web server by 100%. We describe our prototype implementation for the OpenBSD operating system and quantify its performance implications.

1 Introduction

It is becoming increasingly common for modern system designers to enhance system performance by separating the system control and data planes. The intuition is that an application defines its control requirements, and the operating system or hardware mechanisms implement the requested movement or transformations of the data. This keeps the data in the fast path at all times. For example, the Apache webserver uses the *sendfile()* system call, which takes a file descriptor and a network socket and transfers the file directly over the socket, keeping all the data in kernel space. Apache makes a control decision, *(send this file to this socket)*, and the OS performs the data transfer without the file ever reaching the user-level process.

The cryptographic requirements of secure protocols often lead to a deviation from this fast path. An Apache server responding to HTTPS requests cannot use *sendfile()*, because the SSL/TLS libraries are implemented in user space. Even if the web server has a crypto accelerator card, the file must be copied into user space, dispatched to the accelerator card and returned to user space before it is passed to the network.

Our approach is conceptually straightforward: integrate network and cryptographic processing in the kernel so that there are no diversions from the fast data path. The result is minimization of data copying between the user-level application (*e.g.,* the web server) and the kernel. This has great advantages over similar proposals, such as 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 reducing data copying and memory-bus contention. 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.

We present an extension to the OpenBSD kernel that immediately improves the data-transfer performance of TLS and other similar protocols by 20% to nearly 30%. In our scheme, the kernel presents a new system call, similar to Linux's *sendfile()*, intended to be called by user-level processes. The system call takes a socket over which, for example, a TLS session exists, a file descriptor, and the various keying material associated with the TLS session. Then, in kernel memory space, it performs the necessary cryptographic transforms on the file (either in software or with a cryptographic accelerator card) and transfers it directly over the socket, bypassing user memory space entirely. Our implementation is easily portable to other operating systems, especially those that implement System V STREAMS functionality (*e.g.,* Solaris).

2 OpenBSD Cryptographic Framework

The OpenBSD cryptographic framework (OCF) [8] is an asynchronous service virtualization layer inside the kernel that provides uniform access to cryptographic hardware accelerator cards. It supports symmetric (*e.g.,* DES, AES) and asymmetric (*e.g.,* RSA) algorithms, as well as hash functions (*e.g.,* MD5). Symmetric-algorithm and hash function operations are built around the concept of the *session,* 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), cryptographic keys, initialization vectors. The OCF determines which card to use and creates the relevant state by invoking the driver.

For the actual encryption/decryption, consumers specify the data to be processed 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).

The OCF presents two APIs; one for cryptographic producers (*e.g.*, crypto cards) and one for cryptographic consumers (*e.g.*, application-level security protocols). The producer API allows a driver to register with the OCF the various algorithms it supports and any other device characteristics (*e.g.,* support for algorithm chaining, built-in random number generation, *etc*.). To use the OCF, consumers first create a session with the OCF and specify the algorithm(s) to use, mode of operation (*e.g.,* CBC), cryptographic keys, initialization vectors, and number of rounds (for variable-round algorithms). The request is queued and processing returns to the consumer, until a callback indicates that the operation is completed.

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.

3 Design

When a user-level process like Apache receives an HTTP request for a particular file, it issues a *sendfile()* system call to efficiently service the request, as shown in Figure 1. The web server cannot use *sendfile()*, though, if the request is HTTPS, since the SSL/TLS libraries are in shared libraries in user memory. In this case, when the web server process receives a request for a file, the file has to be read from disk into kernel memory and then copied into a buffer in user space. The buffer is then written to the cryptographic accelerator card using the */dev/crypto* interface to the OCF (so it is transfered back into kernel space). When the crypto operations are complete, the buffer is sent back into user space. Finally, the application writes the buffer to the network card, so, again, the buffer is transfered into kernel space. Figure 2 summarizes the data movement. The problem with this approach is that the data are copied unnecessarily into user memory space, and there are two context switches associated with each copy.

We eliminate the copying and context switching by transferring the data directly from disk to the crypto card, and then directly from the crypto card to the network card. In this case, the buffer is read from disk into kernel memory and written directly to the cryptographic accelerator card using the OCF's kernel API. When the OCF signals completion of the crypto operations, the buffer is passed to *sosend()* and thence to the network. The result is the initial and final context switches and no data copies, as shown in Figure 3.

When the file is larger than the buffer, our improvement is even greater. Consider a buffer of size n bytes and a file of size p bytes. The current state of affairs requires $4p/n$ data copies and $8p/n$ context switches. For $p = 10n$, this means the n-byte buffer will get copied 40 times and there will be 80 context switches. In our scheme, the buffer is copied zero times and there are only two context switches.

3.1 Implementation

Our implementation consists of two relatively simple modifications to the OpenBSD kernel. The first is the addition of a system call similar to Linux's *sendfile*. The system call takes a file descriptor *fd* and a socket *sck* and copies data from *fd* to *sck*. Note this copying is all done within the kernel so the system call does not waste time copying the data to and from user space.

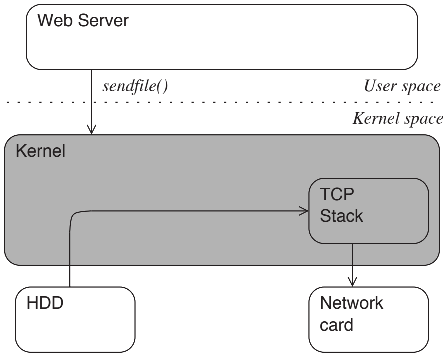

Figure 1: Apache's default file transfer behavior, with no crypto.

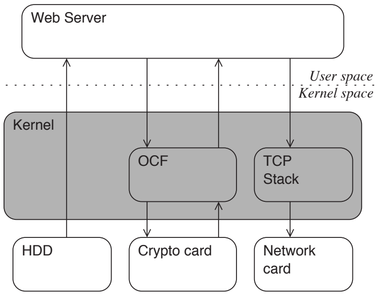

Figure 2: Current mechanism for encrypting and transfering a file. Note the four user/kernel space crossings, each also encompasses two context switches.

The second modification changes the socket layer of the OpenBSD network stack. We add a new socket option, SO_CRYPT, that allows a crypto-consumer to define cryptographic transforms for each packet sent over a socket (*e.g.*, where the encryption should start and end, where the MAC should be placed, and so on). When *sosend()* is called with the SO_CRYPT flag set, *sosend()* passes the data (in the form of an *mbuf*) to the OCF. Then *sosend()* calls *tsleep()* and waits for OCF to indicate the completion of the cryptographic operations. When the operation com-

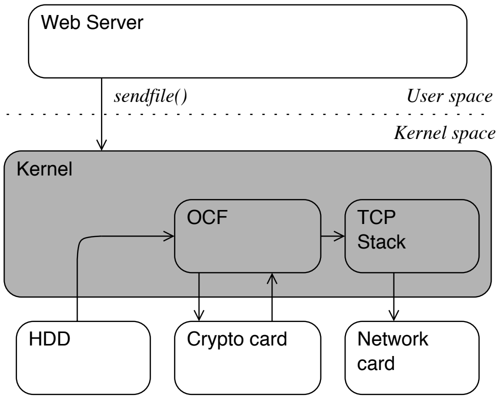

Figure 3: Encrypting and transfering a buffer with *sendfile()* and `SO_CRYPT`.

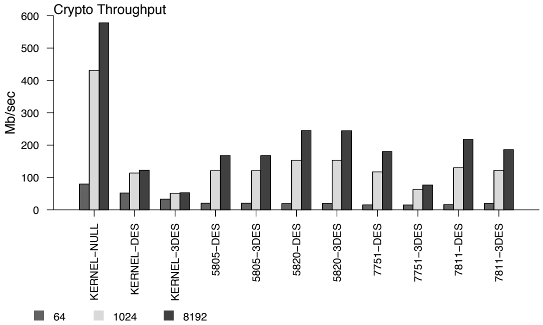

Figure 4: Crypto-hardware performance. The *KERNEL-NULL* bar indicates use of the *null* encryption algorithm. The *KERNEL-DES* and *KERNEL-3DES* bars indicate use of the software DES and 3DES implementations in the kernel. The remaining bars indicate use of the various hardware accelerators. The vertical axis unit is Mbits/second.

pletes, the new encrypted data are substituted into the *mbuf* and control flow returns to the default network processing. By passing *sendfile()* a socket with `SO_CRYPT` set, all network *and* crypto processing takes place within the kernel and the data are never copied into user space.

When an application such as a web server responding to HTTPS requests receives a request for a file (with a descriptor *fd*) over a socket (with a descriptor *so*), the web server enables `SO_CRYPT` on the socket and sets the necessary transforms and keying material for TLS or SSL, as required. Then it calls *sendfile(fd, so)*. The file *fd* is read into a buffer *buf* and each time the buffer fills, *sendfile()* calls *sosend(so, buf)*. Since `SO_CRYPT` has already been set on *so*, the cryptographic operations are handled seamlessly. The file, which would have been copied to and from user space repeatedly, is now *never* copied into or out of user space.

3.2 Evaluation

We evaluate our system by comparing it with the traditional approach. In the traditional approach, we *read()* the file from disk, use the */dev/crypto* interface to the OCF to perform the cryptographic transforms, and then *write()* the file to the network socket (thus, each data buffer is copied between user space and kernel space four times). Our approach uses *sendfile()* with `SO_CRYPT` and eliminates all user-kernel space crossings.

To determine the raw performance of OCF, we use a single-threaded program that repeatedly encrypts and decrypts a fixed amount of data with various symmetric-key algorithms, using the */dev/crypto* interface. We run the test against a variety of accelerators, as well as using the kernel-resident software implementation of the algorithms. We vary the amount of data to be processed per request across experiments. To measure the overhead of OCF without the cryptographic algorithms, we added to the kernel a *null* algorithm that simply returns the data to the caller without performing any processing. Figure 4 shows the results.

We can make several observations on this graph. First, even when no actual crypto is done, the ceiling of the throughput is surprisingly low for small-size operations (64 bytes). In this case, the measured cost consists of the overhead of system call invocation, argument validation, and crypto-thread scheduling. As larger buffers are passed to the kernel, the throughput increases dramatically, despite the increasing cost of memory-copying larger buffers in and out of the kernel. When we use 1024-byte buffers, performance in the no-encryption case jumps to 420 Mbps; for 8192-byte buffers, the framework peaks at about 600 Mbps.

Figure 5 shows a comparison over the OCF between HTTP, HTTP over IPsec, and `ssl(3)` as used by HTTPS. We used `curl(1)` to transfer a large file from the server to the client. As is evident, HTTPS imposes a significant performance overhead compared to plaintext HTTP. Figure 6 provides insight on the latency overhead induced by HTTPS. We used `curl(1)` to transfer a very small file (15 bytes) from the server to the client. We timed 1,000 consecutive transfers.

Figure 7 shows the results for the two schemes operating on files of size 1MB, 10MB and 100MB. We ran the tests between two Dell PowerEdge 2650s, each with 1GB of RAM, over Gigabit Ethernet. The sending machine was equipped with a Soekris Engineering vpn1201 cryptographic accelerator card, and encrypted each file using 3DES. Each test case was run multiple times, and the first run of case was discarded, so that only those runs on a "hot" cache were included. As the figure demonstrates, by partitioning application-level data plane from the control plane, performance gains approach 30% for all size file transfers. This gain is due entirely to the elimination of data copies between kernel and user memory space.

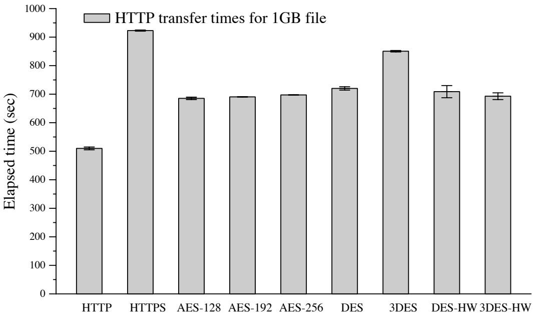

Figure 5: Large file transfer using http (1^{st} bar), https, and http/IPsec. The file is read and stored in the Unix FFS.

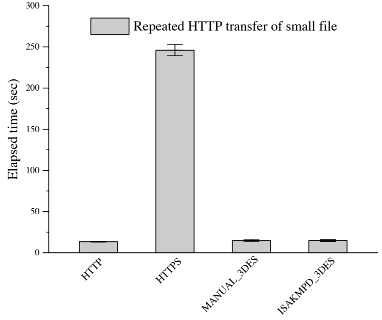

Figure 6: Small file transfer using http, https, and http over IPsec, on a host-to-host network topology. We timed 1000 transfers of the file.

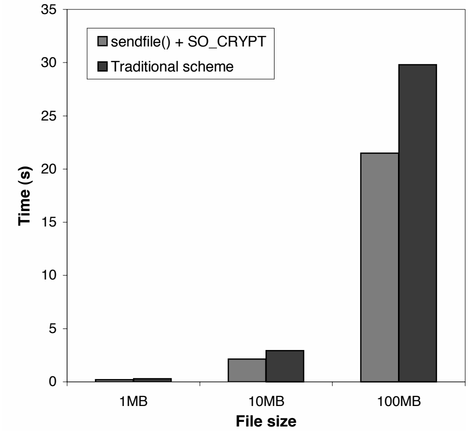

Figure 7: Comparison of files transfers using our scheme vs. the traditional scheme. The improvement over the traditional scheme on all three file sizes is approximately 27%.

4 Related Work

There has been a considerable amount of work on the enhancement of system performance through the addition of cryptographic hardware [3]. This early work was characterized by its focus on the hardware accelerator rather than its implications for overall system performance. [12] began examining cryptographic subsystem issues in the context of securing high-speed networks, and observed that the busattached 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 [10] was the cost of system calls, and a third [11, 5, 7] 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*. [4] 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. [9] 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.

[2] 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 [6] 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.

Other work has investigated performance improvements of TLS [2, 4, 6, 1] 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 operations.

5 Conclusions

Cryptographic protocols are a fundamental building block for securing the Internet. Although their use for routine operations such as file transfer and remote login has become quite common, there remain concerns about their impact in performance, particularly on the server side. The result is that there has been considerable effort accelerating various aspects of protocols like 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 separating the web server application-level control and data planes, removing the web server entirely from the security-enhanced data path.

Our approach eliminates all unnecessary data copies between the kernel and the user-level process with minimum modifications to the kernel and the application. We implemented our scheme in the OpenBSD kernel, which provides support for hardware cryptographic accelerators. The implementation was straightforward with little in the way of pitfalls or hurdles. Our evaluation of the prototype shows an improvement in the data-transfer performance of TLS of 27%. Additionally, only incremental changes are required to extend our scheme to include use of network cards with integrated cryptographic acceleration. We intend to extend our scheme to handle transparent data decryption, and exploit the conceptual parallels between the user/kernel space crossings and the use of the PCI bus. This will allow DMAing of data directly from disk to the cryptographic accelerator card and directly from there to the network card.

References

[1] G. Apostolopoulos, V. Peris, and D. Saha. Transport Layer Security: How Much Does it Really Cost? In

Proceedings of IEEE INFOCOM, March 1999.

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

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

[4] 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.

[5] P. Druschel, M. Abbott, M. Pagels, and L. Peterson. Network subsystem design. *IEEE Network*, 7(4):8–17, July 1993.

[6] 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.

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

[8] 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.

[9] 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*, pages 41–48, June 2002.

[10] C. Pu, H. Massalin, J. Ioannidis, and P. Metzger. The Synthesis System. *Computing Systems*, 1(1), 1988.

[11] J. M. Smith and C. B. S. Traw. Giving Applications Access to Gb/s Networking. *IEEE Network*, 7(4):44–52, July 1993.

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