Cryptography As An Operating System Service: A Case Study

ANGELOS D. KEROMYTIS
Columbia University
JASON L. WRIGHT and THEO DE RAADT
OpenBSD Project
and
MATTHEW BURNSIDE
Columbia University

Cryptographic transformations are a fundamental building block in many security applications and protocols. To improve performance, several vendors market hardware accelerator cards. However, until now no operating system provided a mechanism that allowed both uniform and efficient use of this new type of resource.

We present the OpenBSD Cryptographic Framework (OCF), a service virtualization layer implemented inside the operating system kernel, that provides uniform access to accelerator functionality by hiding card-specific details behind a carefully designed API. We evaluate the impact of the OCF in a variety of benchmarks, measuring overall system performance, application throughput and latency, and aggregate throughput when multiple applications make use of it.

We conclude that the OCF is extremely efficient in utilizing cryptographic accelerator functionality, attaining 95% of the theoretical peak device performance and over 800 Mbps aggregate throughput using 3DES. We believe that this validates our decision to opt for ease of use by applications and kernel components through a uniform API and for seamless support for new accelerators.

We are grateful to Global Technologies Group, Inc. for providing us with two XL-Crypt (Hifn 7811) boards, one Hifn 6500 reference board, and one Hifn 7814 reference board. We are also grateful to Network Security Technologies, Inc. for providing us with two Hifn 7751 boards, one Broadcom 5820 board, and two Broadcom 5805 boards. In addition, Network Security Technologies funded part of the original development of the device-support software.
Part of this work was supported by DARPA and the Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-01-0537.
Part of his work appeared in Proceedings of the USENIX Annual Technical Conference, San Antonio, TX, June 2003, pp. 165-178 and in Proceedings of the 11th IEEE International Conference on Networks, Sydney, Australia, Sept.-Oct. 2003, pp. 455-460.
Authors' addresses: A. D. Keromytis, Department of Computer Science, Columbia University, M.C. 0401, 1214 Amsterdam Avenue, New York, NY 10027; email: angelos@cs.columbia.edu; T. de Raadt, 812 23rd Avenue SE, Calgary, Alberta T2G 1N8, Canada; J. L. Wright, 3857 Beech Down Dr., Chantilly, VA 20151; M. Burnside, Department of Computer Science, Columbia University, M.C. 0401, 1214 Amsterdam Avenue, New York, NY 10027.

Furthermore, our evaluation points to several bottlenecks in system and operating system design: data copying between user and kernel modes, PCI bus signaling inefficiency, protocols that use small data units, and single-threaded applications. We identify some of these limitations through a set of measurements focusing on application-layer cryptographic protocols such as SSL. We offer several suggestions for improvements and directions for future work. We provide experimental evidence of the effectiveness of a new approach which we call *operating system shortcutting.* Shortcutting can improve the performance of application-layer cryptographic protocols by 27% with very small changes to the kernel.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Cryptographic Controls; D.4.8 [**Operating Systems**]: Performance—*Measurements*

General Terms: Security, Performance

Additional Key Words and Phrases: Encryption, authentication, hash functions, digital signatures, cryptographic protocols

1. INTRODUCTION

Today's computing systems are used for applications such as electronic commerce, tele-collaboration of various types, and evolving peer-to-peer systems that often contain sensitive information. Security in these systems depends on several mechanisms that utilize cryptographic primitives as a basic building block. Such cryptographic primitives can be very complex [Broscius and Smith 1991] because the design of these systems is intended to impede simple, brute-force, computational attacks. This complexity drives the belief that strong security is fundamentally incompatible with good performance which, in turn, leads to favoring performance over cryptography by minimizing use of the latter. However, the foundation for this belief is often software implementation [Feldmeier and Karn 1990] of algorithms that were originally intended for efficient hardware implementation. Although modern encryption algorithms such as AES were designed with performance on general CPUs in mind, they remain relatively heavyweight compared to other computational tasks typically found on a server or workstation.

To address this issue, vendors have been marketing hardware acceleration boards that implement several cryptographic algorithms used by security protocols and applications. In some ways, this mirrors the evolution of high-performance graphics processing units (GPUs) to match the needs of the computer-gaming community. Note that, despite the increasing performance of CPUs, GPUs are considered essential for serious gaming (or other graphics-intensive applications), both because they often outperform the system processor and, perhaps more importantly, because the CPU can be used to complete other tasks while the GPU is handling the graphics-rendering part of the application.

In this environment, GPUs and applications must conform to industry-standard APIs such as DirectX and OpenGL. In the case of cryptography, modern operating systems lack the necessary support to provide *efficient* access to similar functionality to applications and the operating system itself through a *uniform* API that abstracts away hardware details. As a result, accelerators are often used directly through libraries linked with applications typically requiring device-specific knowledge by the applications and preventing the operating system itself from easily utilizing such hardware.

We present the OpenBSD Cryptographic Framework (OCF), a service virtualization layer implemented inside the operating system kernel, that provides uniform access to accelerator functionality by hiding device-specific details behind a carefully-designed API. The abstraction introduced allows us to easily support new hardware accelerators and enables applications to use any such accelerator without device-specific knowledge. Furthermore, this intermediate layer does not unduly impact performance which is common when such abstractions are introduced.

The OCF has been in use with OpenBSD [de Raadt et al. 1999] for over three years (since OpenBSD 2.8) and has proven stable and efficient in practice, although it continues to evolve in response to new requirements[1]. The OCF has also been ported to FreeBSD and NetBSD with a port under development for Linux. It offers features such as load-balancing across multiple accelerators, session migration, and algorithm chaining. We describe the changes we made to the OpenBSD kernel and applications to take advantage of the OCF. This article should serve as a good introduction to newcomers as well as veterans of operating system design and development of the complexity of introducing a major new mechanism to a relatively widely-used and stable operating system.

We evaluate the impact of the OCF in a variety of micro-benchmarks, measuring overall system performance, application throughput and latency, and aggregate throughput when multiple applications use the OCF. Our evaluation shows that, despite its addition in the system as a device/service virtualization layer, the OCF is extremely efficient in utilizing cryptographic accelerator functionality, attaining 95% of the theoretical peak device performance. In another configuration, we were able to achieve a 3DES aggregate throughput of over 800 Mbps, by employing a multithreaded application and load-balancing across multiple accelerators.

A secondary observation from our work is that small data buffers should be processed in software, if possible, freeing hardware accelerators to handle larger requests that better amortize the system and PCI transaction costs. On the other hand, multithreading results in increased utilization of the OCF, improving *aggregate* throughput. We make recommendations for future directions in architectural placement of cryptographic functionality, operating system provisions, and application design, and discuss several improvements and promising directions for future work. We believe that our observations will be valuable to operating system designers as they should be applicable to a large class of application environments.

Perhaps more important than the micro-benchmarks, however, is the confirmation that the use of hardware accelerators can remove contention for the CPU and thus improve overall system responsiveness and performance for unrelated tasks. Our experiments allowed us to determine that the limiting factor for high-performance cryptography in modern systems is often data copying and the PCI bus. Thus, when deciding what hardware accelerators to use in a

[1]Public-key algorithm support and the */dev/crypto* interface were introduced in a later version.

particular system, the best choice may be neither the fastest accelerator nor the one with the best price/performance ratio. Instead, system designers need to evaluate such additional hardware in the context of their system using real (or at least realistic) workloads. Furthermore, we expect our findings to be applicable in noncryptographic contexts, for example, in media stream processing, Web server data flows, and so on. We believe that our approach offers an attractive model for introducing similar new features and support for large classes of new hardware devices in a legacy system and should thus be of practical interest to developers and researchers.

Finally, we briefly evaluate one of our suggestions for future operating system design which we call *operating system shortcutting*. This consists of introducing a small amount of application-specific logic in the operating system kernel which allows the application to remove itself from the performance-critical data path. Our proof-of-concept implementation of the scheme for an SSL-enabled Apache Web server shows that performance of static pages and files can improve by up to 27%.

The take-away lessons of our work are as follows.

—It is possible to introduce generic support for new classes of computation-offload devices in legacy operating systems through carefully designed APIs and abstractions as discussed in Section 3. Such APIs need to take into consideration the limitations of the operating system (for example, lack of threading support inside the OpenBSD kernel) and the underlying hardware to achieve satisfactory performance.

—Implementing such an API inside the kernel allows for a wide variety of protocols and applications to take advantage of the new facilities often with minimal modifications. In the case of OCF, application-level support consisted of implementing a pseudodevice (discussed in Section 4.2) and the necessary interface in the OpenSSL library. Here, the existance and almost universal use of a library such as OpenSSL allowed us to easily introduce support for hardware accelerators to all user-level applications as discussed in Section 4.2.1.

This approach allowed us to easily utilize such hardware both for in-kernel security protocols (such as IPsec) and for applications (e.g., SSL/TLS). Contrast this to the way modern graphics cards are supported by windowing systems (at least in unix-like systems), where all of the device-specific support is implemented as part of user-level drivers and libraries (e.g., as part of the X server); in this environment, the operating system itself cannot easily take advantage of advanced graphics capabilities, for example, for data stream processing [GPG 2003; Macedonia 2003; Thompson et al. 2002; Cook et al. 2005].

—Despite the introduction of an intermediate layer between producers and consumers of cryptographic services, it is possible to minimize the performance impact (and, in fact, make the system much more efficient), at the cost of increased complexity and extensive code reengineering (e.g., in the case of IPsec as discussed in Section 4.1).

—If the resulting system is well designed and implemented, new performance limitations will be exposed as a result of stressing different aspects of the overall system architecture. In our case, the new limiting factors turned out to be the memory and PCI bus throughput as shown in Section 5. Although we had not planned ahead for this specific scenario, we were lucky to be able to trivially augment the operating system kernel such that we could achieve a considerable performance improvement for specific usage scenarios by attempting to mitigate these limitations as discussed in Section 7.

—Finally, although we do not stress this point in this article, careful API design allows for rapid, parallel development of the different system components. Specifically for OCF, we developed the device drivers, core OCF functionality, and IPsec modifications in parallel. Likewise, we were able to augment OCF, develop */dev/crypto*, and introduce the necessary code to OpenSSL in parallel, bringing all the pieces together for debugging and integration at the end of the development process. This approach is particularly useful in an open-source environment where different developers may be contributing to different aspects of the system at various times; partitioning of development through clean APIs allows for a smoother, more efficient and more fault-tolerant[2] process.

1.1 Organization

Section 2 discusses related work. Section 3 describes the OCF's design and implementation, while Section 4 discusses its use by various subsystems and applications. In Section 5, we evaluate the framework's performance, and in Section 6, we discuss some of the results and potential improvements and future work. Section 7 discusses a prototype of OS shortcutting and its evaluation; this is meant as a proof of concept rather than a complete validation of the approach. Section 8 concludes the article.

2. RELATED WORK

As interest in security is currently in an upswing, recent work has focused on examining the overall performance impact of security technologies in real systems. Work by Coarfa et al. [2002] 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 since it also assists with the substantial (and perhaps dominant) noncryptographic overheads. Miltchev et al. [2002] 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. In Gupta et al. [2004], the authors examine the benefits of using elliptic curve-based public-key cryptosystems which they show can improve HTTPS performance by 13%–30% in

[2]That is, a process that tolerates developers dropping out or disappearing for arbitrary amounts of time often without warning.

realistic workloads with more benefits to be had as servers move to larger key sizes. The authors Shirase and Hibino [2004] present a hardware architecture for accelerating elliptic curve operations.

Boneh and Shacham [2001] describe 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 Goldberg et al. [1998] 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.

There has been a considerable amount of work on the enhancement of system performance through the addition of cryptographic hardware [Broscius and Smith 1991]. This early work was characterized by its focus on the hardware accelerator rather than its implications for overall system performance. Smith et al. [1992] 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 [Pu et al. 1988] was the cost of system calls, and a third [Traw and Smith 1993; Smith and Traw 1993; Druschel et al. 1993; Kay and Pasquale 1993] the cost of buffer copying. These issues are still with us and continue to require aggressive design to reduce their impacts.

Smyslov [1999] describes an API to cryptographic functions, the main purpose of which is to separate cryptographic libraries from applications, thus allowing independent development. Our service API is similar at a high level, although several differences were dictated by the need to support actual hardware accelerators and allow it to be used efficiently by protocols such as IPsec and SSL as we discuss in Section 3. Other work includes the Microsoft CryptoAPI [Microsoft Corporation 1998], GSS-API [Linn 1997], and IDUP-GSS-API [Adams 1998], PKCS #11 [RSA Laboratories 1997], SSAPI [National Security Agency 1997], and the CDSA [The Open Group 1999]. These are primarily intended for use by applications that also require authentication, authorization, key management, and other higher-level security services. Our work focuses on low-level cryptographic operations, providing a simple abstraction layer that does not significantly impact performance compared to a device-specific approach.

Gutmann [2000] describes an open-source cryptographic coprocessor, focusing on protecting keys and other sensitive information from tampering by unauthorized applications. The author extends the *cryptlib* library to communicate with the coprocessor. While he discusses several options for hardware acceleration and identifies some potential performance bottlenecks, it is mostly a qualitative analysis. This work is extended in Gutmann [1999] which presents a comprehensive cryptographic security architecture, again focusing primarily on preserving the confidentiality of users' (and applications') cryptographic keys, similar work is discussed in McGregor and Lee [2004]. We are interested in a much simpler problem: how to accelerate cryptographic operations in a

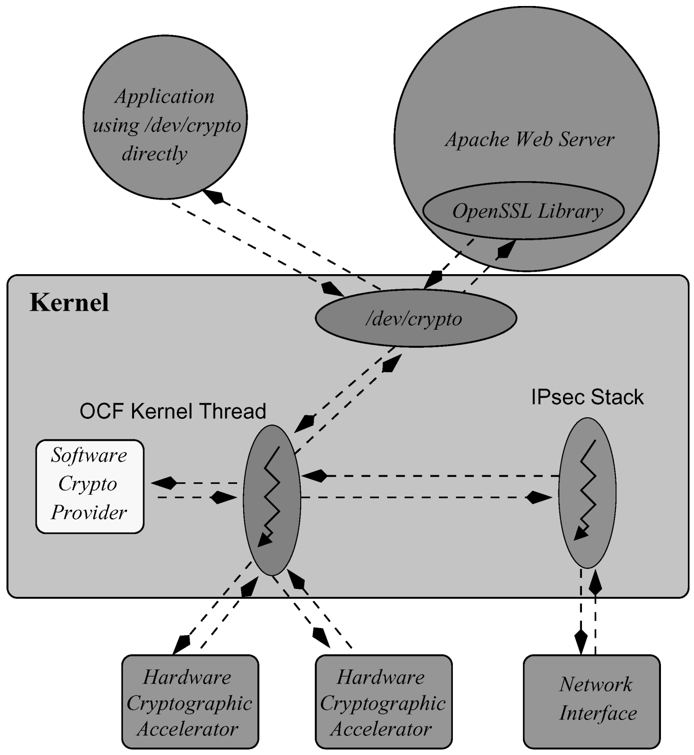

Fig. 1. The OpenBSD cryptographic framework structure.

general purpose operating system using hardware available in the market and with minimal modifications to the kernel, libraries, and applications.

NetBSD uses the *dmover* facility which provides an interface to hardware-assisted data movers. This can be used to copy data from one location in memory to another, clear a region of memory, fill a region of memory with a pattern, and perform simple operations on multiple regions of memory such as XOR without intervention by the CPU.

3. THE CRYPTOGRAPHIC FRAMEWORK

The OpenBSD cryptographic framework (OCF), depicted in Figure 1, is an *asynchronous service virtualization* layer inside the kernel that provides uniform access to hardware cryptographic accelerator cards. The OCF implements two APIs for use by other kernel subsystems, one for use by *consumers* (other kernel subsystems) and another for use by *producers* (crypto-card device drivers). The OCF supports two classes of algorithms: symmetric (e.g., DES, AES, keyed-MD5, HMAC-SHA1) and asymmetric (e.g., RSA, DSA).

Symmetric-algorithm[3] (e.g., DES, AES, MD5, compression algorithms, etc.) operations are built around the concept of the *session* since such algorithms

[3]Technically, hash functions such as MD5 and compression algorithms such as LZS are not symmetric (key) algorithms. We group them with algorithms such as AES and DES for simplicity in our discussion and because most hardware accelerators use the same API for all such algorithms.

are typically used for bulk-data processing, and we wanted to take advantage of the session-caching features available in many accelerators. Asymmetric algorithms are implemented as individual operations: no session caching is performed. Session creation and teardown are synchronous operations.

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.). The device driver also registers four callback functions that the OCF uses to initialize, use, and teardown symmetric-algorithm sessions and to issue asymmetric-algorithm requests. The drivers can also selectively deregister algorithms or remove themselves from the OCF (e.g., when a PCMCIA card is ejected). Any sessions using the defunct driver (or algorithm) are migrated to other cards on an on-demand basis (i.e., as the next request for that session arrives). Registration and deregistration can occur at any time; typical device drivers do so at system initialization time. Drivers notify the OCF as individual requests are completed by the accelerators. A brief description of the API is given in Appendix A.

In addition to any hardware drivers, a software-crypto pseudodriver registers a number of symmetric-key algorithms when the system boots. The pseudodriver acts as a last-resort provider of crypto services; any suitable hardware accelerator will be treated preferably. However, the kernel does not implement asymmetric algorithms in software for performance reasons; we shall see in Section 4.2 how we handle these. Using a generic API for crypto drivers allows us to easily add support for new cards. We briefly discuss these drivers in Section 3.1.

To use the OCF, 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, and number of rounds (for variable-round algorithms). The OCF supports algorithm-chaining, that is, performing encryption and integrity protection in one operation. Such combined operations are used by practically all data transfer security protocols. At session-creation time, the OCF determines which card to use based on its capabilities and creates a session by calling its *newsession* method provided at device registration time. When the session is not needed, the OCF frees any allocated resources.

For the actual encryption/decryption, consumers pass to the OCF the data to be processed, a copy of the parameters used to initialize the session, consumer provided opaque data, and a callback function. The data can be provided in the form of *mbufs* (linked lists of data buffers used by the network subsystem to store packets) or as a collection of potentially noncontiguous memory blocks (which subsumes the case of a single contiguous data buffer). Although *mbufs* are a special case of noncontiguous memory blocks, we added special support to allow for some processing optimizations when using software cryptography. Furthermore, the issuer of a request can specify whether encryption should be done in place, or if the encrypted data must be returned on a separate buffer. Various offsets indicate where to start and end the encryption, where to place the message authentication code (MAC), and where to find the initialization vector (if already present on the buffer) or where to write it on the output buffer.

The request is queued, and the OCF API routine immediately returns to the consumer. The *crypto* kernel thread is periodically invoked by the scheduler and dispatches all pending requests to the appropriate producers. It also handles all completed requests by calling the specified callback functions. It then returns to sleep, waiting for more requests. As a result of the OpenBSD kernel architecture (common in most non-SMP kernels), the thread is not preemptable by user processes, although hardware interrupts are still handled. Currently, the thread must operate at a high priority to avoid synchronization problems. When using the software pseudodriver, this can cause significant latency in application scheduling and in low-priority kernel operations, although the same problem manifested before the migration to OCF when encryption was done in-band with IPsec packet processing.

Once the request is processed, the crypto thread calls the consumer supplied callback routine. If an error has occurred, the callback is responsible for any corrective action. Session migration is implemented by recreating the session using the initial session parameters which accompany all requests as we already mentioned. A specific error code[4] is indicated to the callback routine which reissues the request after recording the new session number to be used so that subsequent requests are correctly routed. Including the initialization data in each request also allows us to easily integrate cards that do not support the concept of session: the driver simply passes all necessary information (data, algorithm descriptions, and keys) to the card with each request. The opaque data are simply passed back to the consumer unmodified by the OCF; they are used to maintain any additional information for the consumer that is relevant to the request. We shall see an example in Section 4.1.

Asymmetric operations are handled similarly, albeit without support for the concept of session. The parameters in this case include an array of parameters, containing the algorithm-specific big-integers.

When multiple producers implement the same algorithms, the OCF can load-balance sessions across them. This is currently implemented by simply keeping track of the number of sessions active on each producer. At session setup, the OCF picks the producer with the smallest number of active sessions. The software pseudodriver is currently never used in load-balancing. We evaluate the effectiveness of this simple scheme in Section 5.4. We discuss possible future improvements in Section 6.4.

3.1 Device Drivers

The drivers for the various crypto devices must be able to cope with a wide variety of hardware design decisions (and bugs) made by the manufacturers. These drivers register the algorithms supported by the device and export the appropriate callback functions to the OCF.

The *hifn* driver supports the Hifn 7751, 7811, and 7951 chips and contains around 3,000 lines of code and definitions. The driver supports the symmetric operations and hashes available on all these chips. Additionally, it supports the random-number generators available on the 7811 and 7951, but does not

[4]The symbolic code **EAGAIN** is used for this purpose.

support the public key unit on the 7951; the latter was clearly designed for SSL server implementations as it requires a large amount of CPU-intensive initialization which can be precomputed and used repeatedly on a server but not a client. All these chips support copying-through header and trailer data to the destination buffer and include full support for scatter-gather I/O. Unfortunately, there is no easy way to coalesce interrupts on this chip which generates one interrupt per operation, resulting in considerable system overhead[5].

The *nofn* driver supports the Hifn 7814, 7851, and 7854 chips (also known as HIPP1 packet processors). Currently, there is no support for the symmetric unit on these chips. Fitting these into the current framework is not currently done because they are designed to replace almost all of the IPsec processing (IV generation, MAC checking, replay window handling, etc.). In the future, we intend to add support for the IPsec unit by adding a combined class algorithm and checking for this in IPsec. On the other hand, the public-key unit is almost exactly the same as the Hifn 6500 described in the following.

The *lofn* driver supports the Hifn 6500 chip which contains a public-key unit and a random-number generator. This chip is essentially a simple big-number arithmetic logic unit (i.e., it is an ALU capable of performing operations on 1024-bit registers). Unlike all of the other chips, the 6500 is not a bus-master (i.e., has no support for DMA); instead, registers exist within its PCI memory-mapped address space. Because of the expense of modular exponentiations, the somewhat higher overhead of writes to these I/O addresses is still small compared to doing the exponentiation in software.

The *ubsec* driver which supports the Broadcom 5801, 5802, 5805, 5820, 5821, and 5822 chips, consists of slightly less than 3,000 lines of code and definitions. The symmetric-crypto units on all of the chips are very similar, but the 580*x* series and 582*x* series require different formatting for the big numbers on the asymmetric unit. These chips support interrupt coalescing by chaining several commands together and scatter-gather I/O. Unlike Hifn, these chips do not poll main memory.

We have a variety of other device drivers in various stages of completion. We are aware of other and more modern products from a variety of vendors which we hope to support in the future.

4. USE OF THE OCF IN OPENBSD

In this section, we discuss how we extended parts of OpenBSD to make use of the OCF services.

4.1 IPsec

The IP Security (IPsec) Architecture [Kent and Atkinson 1998], as specified by the Internet Engineering Task Force (IETF), is comprised of a set of protocols that provide data integrity, confidentiality, replay protection, and authentication at the network layer. The data encryption/authentication protocols, AH

[5]Another important detail is that all of the Hifn symmetric crypto chips poll their descriptor rings in main memory for data to process.

and ESP, reside at the lowest level of the IPsec architecture. These are the wire protocols, used for encapsulating the IP packets to be protected. They simply provide a format for the encapsulation; the details of the bit layout are not particularly important for the purposes of this article. Outgoing packets are authenticated, encrypted, and encapsulated just before being transmitted, and incoming packets are decapsulated, verified, and decrypted immediately upon receipt. These protocols are typically implemented inside the kernel for performance and security reasons.

IPsec was the first consumer of the OCF services. The original implementation of the OpenBSD IPsec was described in Keromytis et al. [1997]. Here, we give a brief overview and then describe the modifications we had to make to it to enable it to use of the OCF.

In the OpenBSD kernel, IPsec is implemented as a pair of protocols sitting on top of IP. Thus, incoming IPsec packets destined to the local host are processed by the appropriate IPsec protocol through the protocol switch structure used for all protocols (e.g., TCP and UDP). The selection of the appropriate protocol is based on the protocol number in the IP header. The SA needed to process the packet is found in an in-kernel database using information retrieved from the packet itself. Once the packet has been correctly processed (decrypted, integrity-validated, etc.), it is requeued for further processing by the IP module accompanied by additional information (such as the fact that it was received under a specific SA) for use by higher-level protocols and the socket layer.

Outgoing packets require somewhat different processing. When a packet is handed to the IP module for transmission, a lookup is made in the Security Policy Database (SPD) to determine whether that packet needs to be processed by IPsec. The decision is made based on the source/destination addresses, transport protocol, and port numbers. If IPsec processing is needed, the lookup will also specify what type of SA(s) to use for IPsec processing of the packet. If no suitable SA exists, the key-management daemon is notified to acquire one. Otherwise, the packet is processed by IPsec and requeued for transmission. The packet also carries an indication as to what IPsec processing has already occurred to it in order to avoid processing loops. In the original IPsec implementation, all cryptographic operations were done in-band with packet processing. This meant that a lot of time was spent performing symmetric-key encryption in the kernel.

To make use of the OCF, we split the input and output processing paths. For example, let us consider the case where the kernel determines (by consulting the SPD) that a packet must be IPsec-protected. After handling generic IPsec encapsulation issues, this routine calls the appropriate wire protocol output routine. In the ESP protocol processing, the original processing routine was broken up into two routines, *esp_output()* and *esp_output_cb()*. The former does all the data marshaling and ESP header manipulation, constructs a crypto request, passes it to the OCF, and simply returns. Execution returns to the network stack (where the decision to apply IPsec was made) with an indication that the operation was successful.

Once the OCF processes the request, it calls *esp_output_cb()*, a pointer to which is included in the request itself. The callback routine completes the ESP protocol processing by checking for any errors in the crypto processing (requeuing the request if the OCF indicated so), completes IPsec bookkeeping, and requeues the packet for transmission. The network stack will then perform a new SPD lookup (making sure no IPsec loops occur by examining the list of SAs that have been already applied to the packet). If necessary, the output processing cycle will occur again. Eventually, the kernel will pass the packet to a network driver for actual transmission.

The cases for output AH and IPcomp processing are similar. Input processing is also similar. The kernel first locates the appropriate SA in the kernel SA database and calls the IPsec routine that validates the ESP header fields, constructs a crypto request, passes it to the OCF and returns. Once the request is processed, the OCF will call the corresponding callback routine which will verify the packet integrity (by comparing the value on the packet with that computed by the accelerator), remove the ESP header, perform further sanity and security checks on the decrypted packet, and requeue it for further processing by the IP layer. AH and IPcomp input processing is similar as is the case of IPsec over IPv6.

Input ESP and AH processing offer one example of use of the opaque data passed with each crypto request discussed in Section 3. All the cryptographic accelerators that support message authentication (MAC) algorithms only offer a forward-compute mode. That is, the card can only compute the MAC on the packet, and it is up to the operating system to verify its validity by comparing it with the received value. Thus, we use the opaque data to store the MAC value from the packet and instruct the OCF to write the new MAC value in the appropriate location in the packet—the operation is exactly the same as the output case. In the callbacks, we simply do a bytewise comparison of the computed value (stored on the packet) and the received value (stored as opaque data in the request itself).

While the code was not very complicated, there were several minor headaches as a result of this asynchronous processing model. For example, one problem was communicating MTU information through arbitrarily many IPsec SAs to the TCP layer so as to correctly fragment application data and avoid fragmentation at the IP layer. We could not simply update the appropriate data structures with the correct MTU value after the packet had been encapsulated once since we could not peek inside the encryption. Fortunately, we keep a record of which SAs have been applied to a packet during input and output processing. Thus, on receipt of the appropriate ICMP message, or when the IP layer indicates that the packet is too large to be transmitted without fragmentation, the list of SAs is traversed and each SA is updated with the correct MTU value based on its position in the SA chain (i.e., the first SA on output will advertise a smaller MTU than the last one, the difference is the ESP headers and encryption padding). The next packet that tries to traverse the chain will encounter a correct MTU value.

4.2 /dev/crypto

Building on our experience with the IPsec implementation, we turn our attention to exporting the OCF services to user-level applications. A */dev/crypto* device driver exists which abstracts all the OCF functionality and provides a command set that can be used by OpenSSL (or any other software that uses */dev/crypto* directly).

The interface exported through */dev/crypto* is based on *ioctl()* calls and is thus fully synchronous (i.e., applications can only have one request pending)—in the future, we intend to allow processes to issue multiple requests. Both symmetric and asymmetric operations are permitted using this framework; we will first describe the symmetric component.

Similar to the underlying OCF, 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). Similar to the OCF, a single session supports both a cipher and a MAC as we are simply exporting the same functionality available to the kernel. The kernel returns a session identifier that can then be reused repeatedly for subsequent operations. When the session is no longer needed, it can be revoked. Many sessions can be requested against a single file descriptor *fd*; all sessions follow a particular *fd* through *fork()* and *exec()* calls and are not otherwise visible to other processes. Obviously, the last *close()* on *fd* destroys all the sessions.

If the request cannot be satisfied using hardware accelerators, the kernel will return a specific error code[6] so that the caller can fall back to a software implementation. We considered adding an *ioctl()* that describes the abilities of the available hardware, allowing an application to determine if the needed algorithm is supported by looking at a list. However, numerous other variables exist (key sizes, block sizes, alignment) which might be difficult to describe. For the time being, we have punted on this issue. However, when first called, the OpenSSL engine will enumerate all OCF-supported algorithms. It does so by trying to create session for each algorithm it supports in software and caches the result. If an algorithm is not provided by the OCF, the library will use its software implementation (in reality, the kernel will admit that it supports cryptographic algorithms that it implements in software, and OpenSSL will make use of them as if they were implemented by hardware unless a system-wide configuration variable is set to prohibit this which is the default setting).

Once a session is established, blocks can be encrypted or decrypted using additional *ioctl() calls*. 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. In the future, we may support larger blocks by splitting operations into smaller chunks.

The user level data blocks are copied into memory allocated inside the kernel address space. The OCF is then called to perform the operation using the

[6]The symbolic error code **EINVAL** is used.

initialization information stored in the application's */dev/crypto* session. If the operation is successful, the results are copied back to the application buffers. Obviously, the cost of these two copies is higher for larger block sizes as we shall see in Section 5.4. In the future, we hope to use page flipping for larger blocks when the kernel memory subsystem supports this.

For asymmetric operations, no session is required. A different *ioctl()* is used in an atomic fashion for each individual operation. Five operations are provided, supporting different versions of modular exponentiation (a building block for many public-key algorithms), DSA processing, and Diffie-Hellman computation. Each of these has an operation-specific number of input and output parameters which are always a packed byte array of big integers. The particular format we chose for these parameters makes it easy to interface to OpenSSL bignums and to most of the early hardware we had access to.

Presently, OpenBSD lacks cloning devices. Therefore a cumbersome procedure for opening */dev/crypto* must be followed. After the initial *open()* call, the caller must use *ioctl()* to retrieve a file descriptor (*fd*) to use, then perform all operations against this replacement *fd*. This replacement *fd* is a unique per-process descriptor, while the initially-opened one would naturally be shared between all callers. Without such semantics, the *fork()* and *_exit()* system calls do not exhibit the expected semantics with respect to file-descriptor inheritance and closing. Just as bad, we would end up with all processes able to see and use each other's keys. When cloning devices are implemented in OpenBSD, we will change the user-level code (mostly OpenSSL) to no longer use this complicated procedure, but the kernel will retain it for backward compatibility. While writing this code, we ran into numerous strange and difficult resource management issues for session teardown.

It should also be noted that applications using */dev/crypto* must ensure they use *ioctl()* with the *F_SETFD* command on the crypto descriptor to ensure that the close-on-exec flag is set. Otherwise, child processes will inherit unwanted descriptors which is both a security and a resource exhaustion concern. resources (OCF sessions and kernel memory) may also be held for arbitrarily long periods of time, for example, when SSH spawns a new shell after a user login. This would result in starvation for other applications and/or the kernel.

4.2.1 *OpenSSL Enhancements.* In the past, programmers using OpenSSL (or its predecessor, SSLeay) directly called the generic crypto routines as they existed for each algorithm. More recently, programmers have been encouraged to use the EVP layer for dealing with symmetric algorithms. This provides a session-based model much like the */dev/crypto* layer described in the previous section. Applications like OpenSSH, *mod_ssl* (the Apache SSL module we use) and *sendmail* have matured to use these interfaces.

Newer OpenSSL code bases contain an engine component. This allows asymmetric algorithms to be directed to a hardware driver; a number of stub functions are provided which typically interface with vendor-specific shared libraries to actually do the operation on the vendor's accelerator. Many of these subsystems interact badly and do not consider the effects of *chroot()* or other strange Unix behaviors, resulting in weak security models. Since we run Apache in a *chroot()*'ed environment in which there exists no */dev/crypto* device, we modified it to perform all necessary initializations prior to being sandboxed. We wrote our own engine modules that interacts directly with */dev/crypto,* without any of these surprises. Symmetric operations from the EVP layer are directly mapped into OCF requests. One major weakness is that the EVP layer has no concept of bundling algorithms. Thus, protocols that use encryption and MAC on a message, such as TLS and SSH version 2, sequentially issue two separate requests to */dev/crypto* through the EVP layer, resulting in unnecessary context switches, data copying, and DMA transactions. Thus, the EVP layer currently does not pass MAC operations to the OCF.

Despite the existance of the direct */dev/crypto* interface, we believe that libraries such as OpenSSL will remain the main mechanism through which OCF is accessed for several reasons. First, the wide availability (and portability) of OpenSSL means that application developers are not locked into any specific operating system vendor interface. Second, developers need to anticipate that their software may operate in different environments with or without hardware accelerators. Using OpenSSL, such software can make use of cryptographic acceleration where available, while maintaining the ability to easily and transparently (for both developers and users) fall back on using the software implementation of the same algorithms. Finally, there exists considerable software that has already been written for OpenSSL; it is unrealistic to expect such software to be rewritten. Extending OpenSSL is a convenient way of allowing these applications to use the OCF transparently.

4.3 Swap and Filesystem Encryption

While OpenBSD supports swap-space encryption [Provos 2000] and the Transparent Cryptographic Filesystem (TCFS) [Gattaneo et al. 2001], neither of these currently utilize the OCF. There is no fundamental reason why this is the case, and we intend to convert them accordingly as time permits.

5. PERFORMANCE EVALUATION

In this section, we analyze the performance of the cryptographic framework. We have ran a series of micro-benchmarks that allowed us to determine the limits of the framework and potential directions for improvement. We use the OCF for simple cryptographic tasks, comparing different cryptographic accelerators with the case of pure software encryption, and provide a cost breakdown. We also attempt to quantify the benefits to be had by the system at large when offloading cryptographic operations to hardware accelerators. Furthermore, we evaluate the load-balancing feature of OCF by simultaneously using multiple accelerators on the same machine. Finally, we provide some indications on the gain in performance for cryptographic protocols that make use of the OCF; a more extensive analysis of the latter may be found in Miltchev et al. [2002].

5.1 Testbed

For our tests, we use two identical machines. The machines have 1.4Ghz 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. We placed the crypto card being tested either on the 64bit/66mhz/3.3V bus or the 32bit/33Mhz/5V bus as appropriate for the card. The crypto cards we used are:

—Broadcom 5805 reference design board (32bit),
—Broadcom 5820 reference design board (64bit),
—GTGI XL-Crypt (based on the Hifn 7811 chip) (32bit),
—NETSEC 7751 (based on the Hifn 7751 chip) (32bit),
—Hifn 6500 reference design board (32bit),
—Hifn 7814 reference design board (64bit).

The Hifn datasheet gives a peak performance for the 7751 chip of 62 Mbps for encryption and 110 Mbps decryption when using IPsec with 3DES/SHA1/LZS (LZS is a data compression algorithm). When the 3DES engine alone is used, both encryption and decryption throughput are 83 Mbps. Broadcom's Web site places the peak performance of the 5820 chip at 310 Mbps of 3DES-SHA1 when used in IPsec. Furthermore, they claim 800 1024-bit RSA signature computations per second. In mid-2003, the most expensive of these cards represented an investment of less than 20% of overall system price.

For network testing, we used SysKonnect 9843 multimode fiber 1-Gigabit Ethernet cards for all tasks except monitoring. No switches were used; instead, we connected the two hosts directly with fiber.

We used vanilla OpenBSD 3.3, with the default compiler settings for the kernel and applications. The GCC version we used (default with OpenBSD 3.3) was 2.95.3.

5.2 OCF Throughput

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 all the hardware accelerators listed in the previous section 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. The results can be seen in Figure 2.

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.

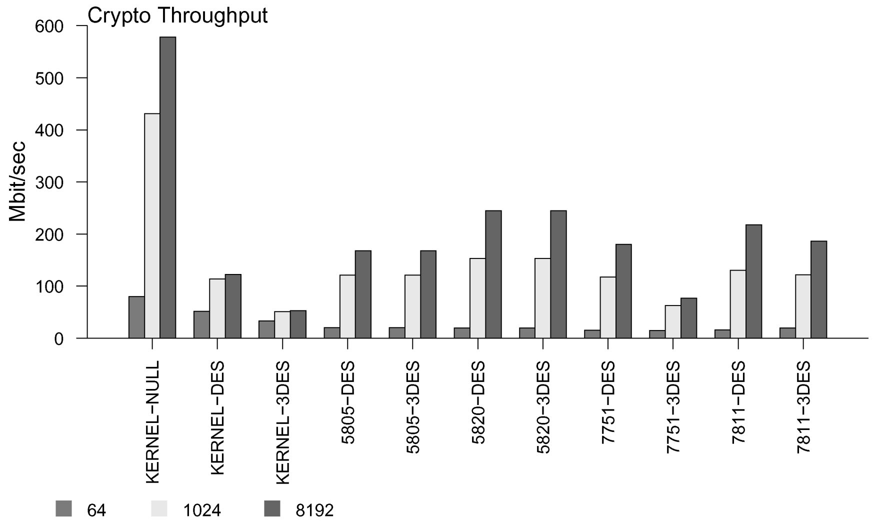

Fig. 2. Crypto-hardware performance. The *KERNEL-NULL* bar indicates use of the *null* encryption algorithm. The *KERNEL-DES* and *KERNEL-3DES* bars indicate use of the software DES and 3DES implementations in the kernel. The remaining bars indicate use of the various hardware accelerators. The vertical axis unit is Mbits/second.

Notice, however, that this peak corresponds to a single process issuing crypto requests. This process is blocked after each request, the scheduler context-switches to the crypto thread (which was blocked waiting for requests), the *null* algorithm executes and the completed request is passed back to the */dev/crypto* driver which wakes up the blocked user-level process. If many processes are issuing requests, the crypto thread's request queue will contain multiple requests. When we run multiple processes, each will queue a request (and be blocked by */dev/crypto*); the crypto thread will process all these requests in a flurry of activity and cause all processes to wake up in synchrony. The crypto thread will then go back to sleep, while each of the processes will issue another request. This cycle repeats for the duration of the experiment. As a result, more processes using the OCF result in increased aggregate throughput, simultaneously increasing the average processing latency.

These buffer sizes are close to the typical sizes of requests issued by some of the most commonly used applications.

—*SSH* keyboard input results in many small requests (so we are close to the 64-byte case); responses from the server are larger, but not considerably so. When X forwarding is used, we can occasionally get larger buffers.

—*SCP* / *SFTP* issue larger requests; OpenSSH, a popular implementation, uses requests of 4KB.

—*SSL* / *TLS* also issue large requests. The maximum size of an SSL record is 16KB, but can be less if (optional) compression is used.

—*IPsec* processes packets at the network layer. Such traffic is trimodal [Claffy et al. 1998]: about 40% of packets are 40–60 bytes (the vast majority of these are being TCP acknowledgments), with the remainder split between

576 bytes (TCP MSS when no Path MTU Discovery is used) and 1460 bytes (when Path MTU Discovery is used).

When we use real cryptographic algorithms, we notice that the performance of DES done in software is close to that of no encryption for small packet sizes; even 3DES performance is just half of the no encryption case. If we use larger buffer sizes, the performance of software crypto done in the kernel (the KERNEL-$*$ labeled bars) degrades rapidly. When we use hardware accelerators, we notice two different trends. For small buffers, the performance degrades with respect to the software case. This indicates that the additive costs of system call invocation, OCF processing, and the 2 PCI transactions (to/from the crypto cards) dominate the cost of doing crypto. However, as we move to larger buffer sizes, performance quickly improves as these overheads are amortized over larger buffers despite the fact that more data has to be copied in and out of the kernel and over the PCI bus. Thus, to improve the performance of the system when applications issue large numbers of small requests, either request-batching should be done, a faster processor should be used, or the number of user/kernel crossings should be minimized. When larger buffers are being processed, it pays to use some cryptographic accelerators, although not all such cards are equal in terms of performance.

Notice that the performance of DES and 3DES is the same in each of the 5805 and 5820 cards these cards really implement only 3DES in Encrypt-Decrypt-Encrypt (EDE) mode and emulate DES by loading the same key in one of the Encrypt and the Decrypt engines (effectively canceling each other out). In contrast, the 7751 seems to implement two separate crypto engines for DES and 3DES, or uses a shortcut in its 3DES engine. The 7811 seems to implement different engines as well, but the performance difference between the two is not as pronounced.

Similarly, we measure the performance of OCF for public-key operations. In this case, there are no kernel-resident software public-key algorithms. We count the number of RSA signature generations and verifications per second, for different accelerators and key sizes (512 to 4096 bits as supported by of the each cards). The results are shown in Figures 3 and 6. Similar results are shown for the DSA algorithm in Figures 4 and 5.

The Hifn 6500 and 7814 are geared more towards slower, embedded applications so the fact that their performance is considerably worse than software is not surprising. The number of verifications is much larger than the number of signature generations in unit time. This is because, as with most crypto libraries, OpenSSL opts for small values for the public part of the RSA key (typically, $2^{16} + 1$) and correspondingly large values for the private key. This causes the public-key operations (encryption and verification) to be much faster than the private-key operations even though they are, in principle, the same operation (modular exponentiation).

Another interesting observation is that the RSA sign throughput is higher in the software case (see Figure 3). This happens because the CPU on the crypto-card is slower than the host CPU and optimized for bit operations which is as useful for public-key cryptography. So the anomaly in Figure 3 is actually

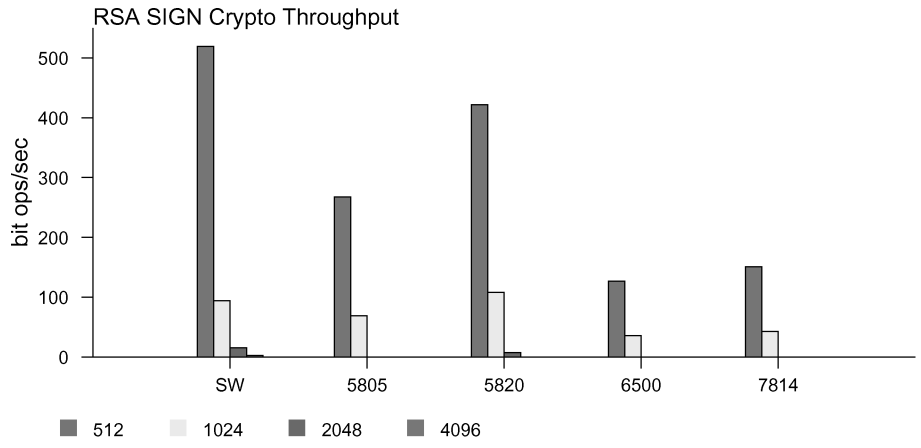

Fig. 3. RSA signature generation. The horizontal axis indicates the modulus size in bits. The vertical axis indicates the number of operations per second. Note that none of the hardware accelerators supports 4096-bit keys; we give the software case for completeness.

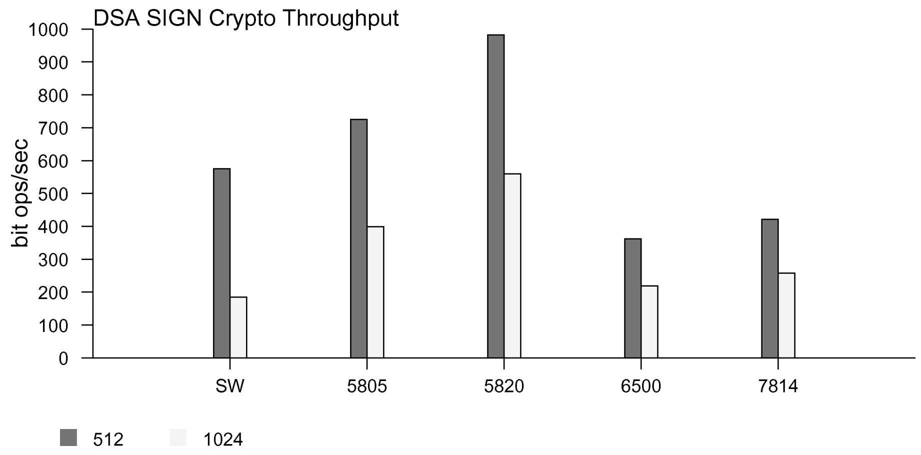

Fig. 4. DSA signature generation. For each cryptographic accelerator, we tested with two modulus sizes, 512 and 1024 bits; respectively. The vertical axis indicates number of operations per second.

expected. However, as we mentioned in Section 5.1, Broadcom claims that the 5820 can perform 800 RSA signature operations per second with 1024-bit keys. In our case, we only see slightly over 100. There are two explanations for this. First, we are underutilizing the 5820: there is only one thread issuing RSA sign operations which is blocked waiting termination of each request. Once the card computes the signature, it has to wait for the crypto framework to wake up the blocked process, then for the scheduler to context-switch to it and the process to issue an *ioctl()* call to get the results and then another *ioctl()* call to issue the next request which is placed on the crypto thread's queue. Finally, the scheduler has to context-switch to the crypto thread. During all this time, the accelerator is idle since there is no other process using it. The second reason for the higher vendor-stated performance is that the tests they performed used the

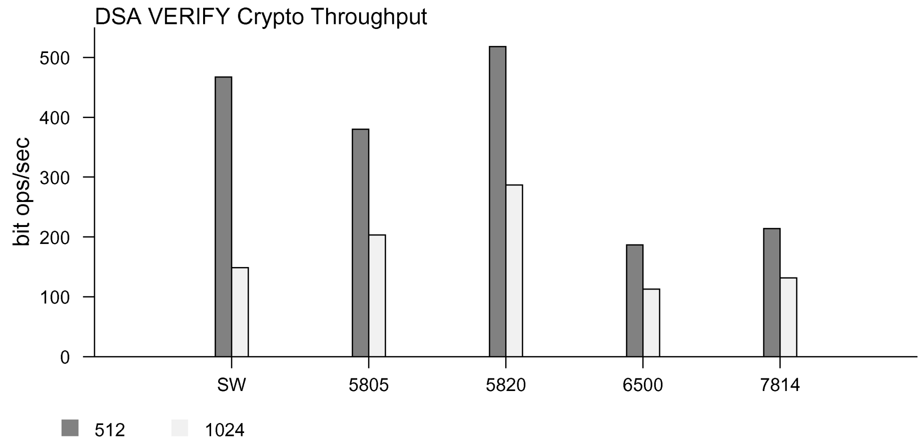

Fig. 5. DSA signature verification. For each cryptographic accelerator, we tested with two modulus sizes, 512 and 1024 bits, respectively. The vertical axis indicates number of operations per second.

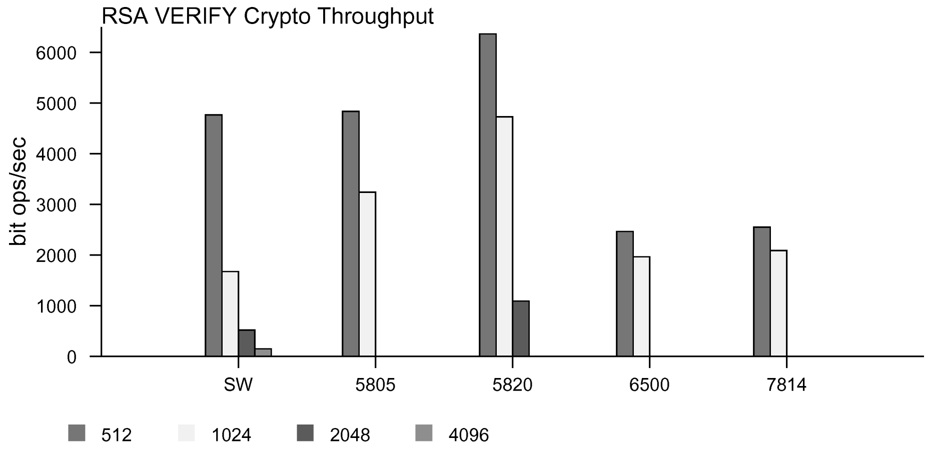

Fig. 6. RSA signature verification. The horizontal axis indicates the modulus size in bits. The vertical axis indicates operations per second. Note that none of the accelerators supports 4096-bit keys; we give the software case for completeness.

CRT parameters for the RSA operations which make RSA processing considerably faster. However, for implementation reasons, our OpenSSL engine does not use CRT parameters yet.

5.3 System-Wide Effects

To determine the system-wide benefits of offloading cryptographic processing, we run multiple threads (up to 24) of the *openssl speed* benchmark with various algorithms, while, at the same time, we run a simple CPU-intensive job. The CPU hog process consists of a small program that performs 2^{32} function calls, each function call performing an integer-multiply operation. The elapsed time for the CPU hog process was recorded for each (algorithm, number of threads) tuple. As we see in Figure 7, the crypto accelerators very effectively eliminate

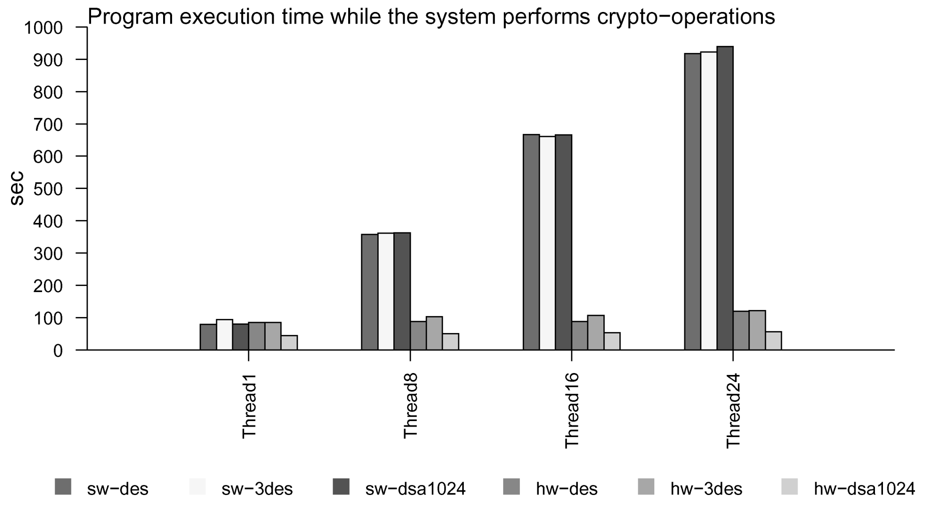

Fig. 7. Program execution time while multiple threads perform crypto operations in parallel. The bars show the elapsed time in seconds for executing the CPU-bound process for different algorithms and numbers of threads.

Table I. Crypto-Request Load-Balancing Using a Quad-Hifn 7751 Card on a PCI 64bit/66Mhz bus

# Threads	16 bytes	64 bytes	256 bytes	1024 bytes	8192 bytes	16384 bytes
1	3 Mbps	11.4 Mbps	33 Mbps	59 Mbps	79 Mbps	80 Mbps
2	5.5 Mbps	18.4 Mbps	56 Mbps	111 Mbps	154 Mbps	160 Mbps
3	6.4 Mbps	23.2 Mbps	71 Mbps	152 Mbps	229 Mbps	238 Mbps
4	6.8 Mbps	25.7 Mbps	81 Mbps	182 Mbps	292 Mbps	299 Mbps
32	7.3 Mbps	27.5 Mbps	94 Mbps	249 Mbps	313 Mbps	320 Mbps

contention for the otherwise shared resource, the CPU, whether the crypto performed is symmetric (DES, 3DES) or asymmetric (DSA with 1024-bit keys). The execution time for the hog process remains constant, regardless of the number of threads of execution.

5.4 Load Balancing

We are also interested in determining how well the OCF can load-balance crypto requests when multiple accelerators are available and the aggregate throughput that can be achieved in that scenario. We use a custom-made card by Avaya that contains four Hifn 7751 chips that can be used as different devices through a PCI bridge resident on the card. We use multiple threads that issue encryption requests for 3DES, and vary the buffer size across different runs. The results are shown in Table I. As we can see, performance peaks in the case of 32 threads and 16KB buffers at 320 Mbps which is over 96% of the maximum rated throughput of four Hifn 7751 chips. The card was installed on the 64bit/66Mhz PCI bus, but because the chip is a 32bit/33Mhz device, the maximum bus transfer rate is 1.056Gbps. At our peak rate, we use over 640 Mbps of the bus, 320 Mbps for data in each direction (to and from the card), plus the transfer initialization

Table II. Crypto Request Load-Balancing Using Four 5820 Cards on a PCI 64bit/66Mhz bus

# Threads	16 bytes	64 bytes	256 bytes	1024 bytes	8192 bytes	16384 bytes
1	5.4 Mbps	18.9 Mbps	62 Mbps	152 Mbps	301 Mbps	255 Mbps
32	9.9 Mbps	37 Mbps	120 Mbps	410 Mbps	759 Mbps	802 Mbps

commands and descriptor ring probing, etc., thus utilizing over 60% of the PCI bus. Notice that because the card uses a PCI bridge, a 2-cycle latency is added on each PCI transaction.

The card was installed on the 64bit/66Mhz bus because the system's 32bit/33Mhz bus exhibited surprisingly bad performance probably because many other system components are found on that bus and likely cause contention. Since the machine is operating as it normally would while this test is being run, the scheduler is active, and two clock interrupts are received at 100 and 128Hz, respectively. Other devices are also generating their own interrupts.

Another possible cause is an artifact of the i386 *spl* protection method: a regular *spl* subsystem disables the interrupts from a certain class of devices at the invocation of an *splX()* call. For instance, calling *splbio()* blocks reception of interrupts from all devices which are in the "bio" class of devices. On the i386, the registers used to do interrupt blocking (found on the programmable interrupt controller, also known as the PIC) are located on the 8Mhz ISA bus which is what OpenBSD uses for interrupt management (as opposed to the APIC).

Worse yet, some operations on this device require a 1 *usec* delay before taking effect. To partially mitigate this extremely high overhead, the i386 kernel interrupt model instead makes the vectors for blocked interrupt routines point to a single depth queuing function which does the actual interrupt blocking at the time of reception. When the *spl* is lowered again, the original interrupt handler is called. However, the 8Mhz ISA bus still had to be accessed. This has the effect of further reducing the available bandwidth on the PCI bus. One small buffer benchmark generated over 62,000 interrupts/sec; we believe that the *spl* optimization is failing under such load.

Using four 5820 cards on a 64bit/66Mhz PCI bus allows us to achieve even higher throughput as shown in Table II. We show only the 1 and 32-thread tests; the rest of the measurements followed a similar curve as the quad-7751. Performance peaked at over 800 Mbps of crypto throughput. Using the same analysis as before, we are using in excess of 1.6Gbps of the fast PCI bus which has a throughput of 4.22Gbps, achieving slightly over 38% utilization of the bus. As we mentioned in Section 5.1, the vendor rates this card at 310 Mbps. Thus, the maximum theoretical attainable rate would be 1.24Gbps. We achieve 64.5% utilization of the four cards in this case. A rough sampling of CPU utilization during these large block benchmarks on both cards showed around 10,000 interrupts/second, which is substantial for a PC.

Investigating further, we determined that all four 5820 cards were sharing *irq* 11. Thus, it is possible that the culprit is the *spl* optimization previously mentioned, at least for the small buffer sizes: the *vmstat* utility shows us anything from 50,000 to 60,000 interrupts/second when processing buffers of 16 to 1024 bytes. Furthermore, because of a quirk in the processing of shared *irq* handlers, some cards experience slightly worse interrupt-service latency: shared *irq* handlers are placed in a linked list. If multiple cards raise the interrupt at the same time, the list will be traversed from the beginning for each interrupt raised, and each *irq* handler will poll the corresponding card to determine if the interrupt was issued by it. However, fixing this quirk or moving the cards on different *irq*'s did not significantly improve throughput.

When we use 8192-byte buffers, the interrupt count drops to 12,000 which the system can handle. In each of these cases, the system spends approximately 65% of its time inside the kernel. Most of this cost can be attributed to data copying. However, as we move to larger buffer sizes, we find the system spending 89% of its time in the kernel and only 1.9% in user applications for the case of 16KB buffers. The number of interrupts in this case is only 5,600 which the system can easily handle. The problem here is that there is considerable data copyin/copyout between the kernel and the applications. Aggravating the situation, while such data copying is in progress, no other thread can execute, causing a convoy effect: while the kernel is copying a 16KB buffer to the application buffer, interrupts arrive that cause more completed requests to be placed on the crypto thread's completed queue. The system will not allow the applications to run again before all completed requests are handled which cause more data copying. Thus, the queue will almost drain before applications will be able to issue requests again and refill it. We intend to further investigate this phenomenon.

Fundamentally, the data copyin/copyout limitation is inherent in the memory subsystem. We measured its write-bandwidth to be approximately 2.4Gbps. Using the crypto cards, we are in fact doing 3 memory-write operations for each data buffer: one copyin to the kernel, one DMA from the card to main memory, and one copyout to the application. Notice that data DMA'ed in from the card is not resident in the CPU cache as all such data is considered suspect for caching purposes. In addition, there is an equal amount of memory reads (copyin, DMA in from the card, copyout). Each of these transfers represents an aggregate of 800 Mbps. When we ran the same test with three 5820 cards, performance improved slightly to 841.7 Mbps in the case of 16KB buffers, achieving over 90% utilization of the three cards. In this case, the memory subsystem is still saturated, but the cards can more easily get a PCI-bus grant and perform the DMA.

One interesting problem we ran into with this experiment was that the *openssl speed* test was broken when used with many threads. Each block size was run for 3 seconds by each thread, but it took several seconds for all 32 threads to start. By increasing the time for testing each block to one minute, we amortized this thread startup overhead over a longer period of time.

5.5 File Transfer

Measuring the performance of the OCF outside the context of any specific applications allowed us to determine how effectively it can take advantage of hardware accelerators. However, cryptography is often used in the context of a real application whose workflow may not allow complete utilization of system

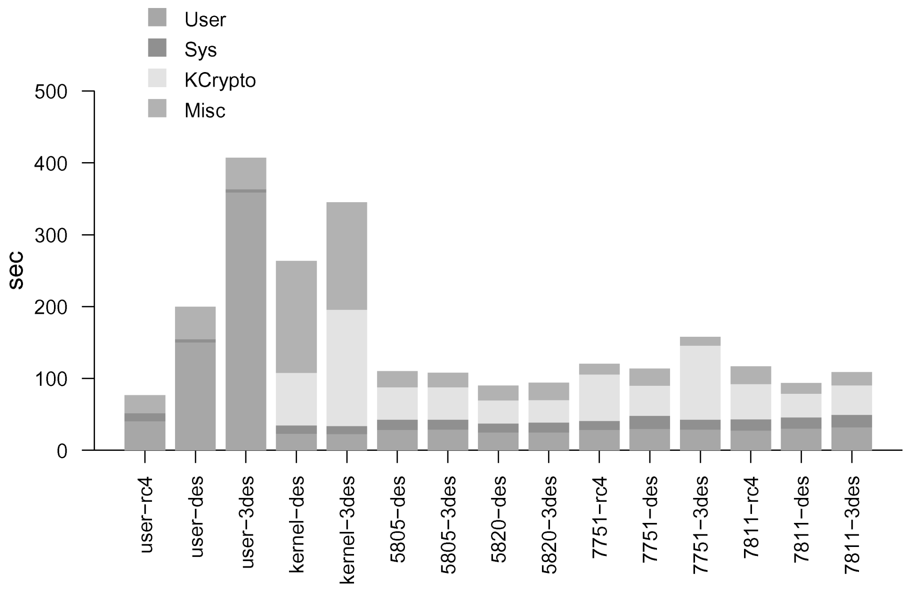

Fig. 8. File transfer using SSL. The bars show the elapsed time for transferring a 1GB file in seconds.

resources. Thus, we need to also determine how suitable the OCF model is to the needs of common cryptographic applications. Although this is an open-ended question, it is possible to make some early observations by using some representative applications that make heavy use of cryptography. To that end, we used TLS and SFTP to transfer a 1GB file between two hosts. A preliminary evaluation using OCF with IPsec can be found in Miltchev et al. [2002]. Figures 8 and 9 show the elapsed time for the transfer, for TLS and SFTP, respectively.

For the TLS test, we used the *openssl* utility from the OpenSSL 0.9.7-beta3 release. We used the OpenSSH 3.5 protocol 2 SFTP implementation. Both of these make use of the OpenSSL cryptographic library which uses the */dev/crypto* interface. Although the two protocols differ slightly in the number and type of public-key operations performed during initialization, any difference in the overhead is amortized over the processing and transfer of such a large file. We recorded wall-clock time spent in user mode and system time (which includes system call handling and the */dev/crypto* device driver processing), spent in the crypto thread as well as time spent for each operation on the crypto card (including the two DMA transfers over the PCI bus). We also report miscellaneous time, which is the total wall clock time minus the system and user time, spent in the crypto thread and time spent waiting on the crypto operation to be performed in hardware. Miscellaneous costs primarily consist of the cost of network communication itself.

For TLS, the following cipher suites were tested: *EDH-DSS-DES-CBC-SHA*, *EDH-DSS-DES-CBC3-SHA*, and *DHE-DSS-RC4-SHA*. The symmetric algorithms we used were DES, 3DES, and RC4, respectively. In all cases, SHA1 was used in the message authentication code. In addition, each exchange involved a DSA signature/verification on either side and a Diffie-Hellman key

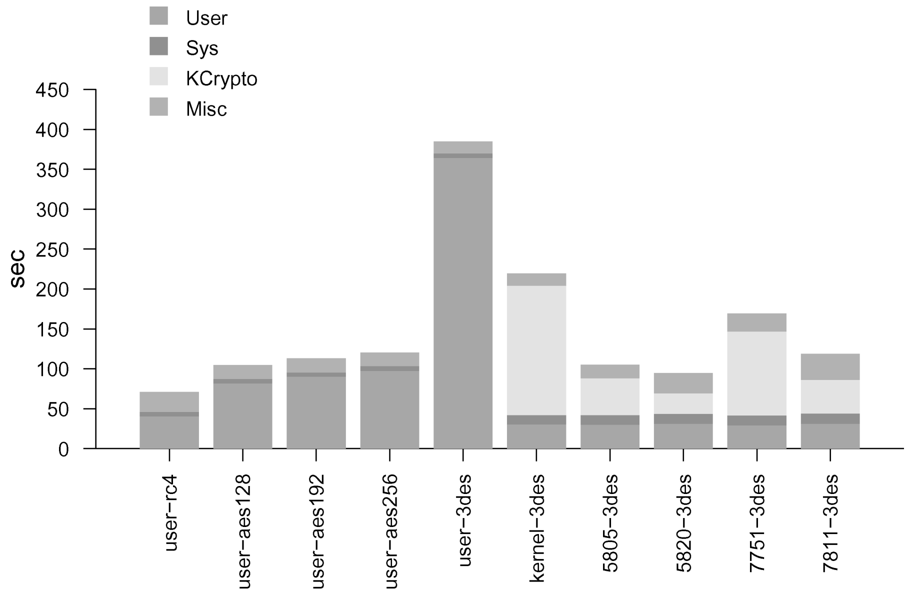

Fig. 9. File transfer using SFTP. The vertical bars show the elapsed time for transferring a 1GB file in seconds.

exchange. For SFTP, we tested the following ciphers: *AES128-CBC*, *AES192-CBC*, *AES256-CBC*, *3DES-CBC*, and *ARCFOUR* (as RC4 is called in the SSH protocol). Again, SHA1 was used for message authentication. The AES measurements are only included here for completeness; although the OCF supports AES in software, there are as of yet no commercially available hardware accelerators for AES for which the specifications are available to us. RC4 was also included as a baseline for TLS performance: RC4 is a fairly lightweight stream cipher which imposes very little performance overhead even when implemented entirely in software.

The *user-∗* bars indicate encryption done exclusively in user-level context; the *kernel-∗* bars indicate use of software encryption in the kernel (in this as well as the hardware cases only the encryption is done by the OCF, per our discussion in Section 4.2.1). The remaining bars indicate use of the various cryptographic accelerators. We notice that the KCrypto slice (which indicates the amount of time taken by the crypto thread itself) is noticeable in the *kernel-** and hardware accelerator tests. In the former, the bulk of the KCrypto processing is due to algorithm execution; in the latter, most of the cost is in data marshaling and unmarshaling, before and after sending to the crypto card.

Notice that *kernel-des* is slower than *user-des* (which can be explained in terms of system call and data-copying overheads), but *kernel-3des* is faster than *user-3des*. Although these same overheads apply here as well, 3DES is approximately 3 times more expensive than DES (there are certain parts of the DES computation that can be skipped in an optimized 3DES implementation). Because the OCF kernel thread is non-preemptable as we mentioned in Section 3, once it starts processing a 3DES request, it is not interrupted by another process (although it can be interrupted by hardware interrupts). Thus, the difference in performance between the two bars shows the overhead of the periodic scheduler invocation and context switching to other jobs in the ready queue (the only other jobs in our system were daemons processes with little or no work to do).

6. DISCUSSION

Following our evaluation of the OCF in the previous section, we give some thoughts on improvements and future directions.

6.1 Cryptography in the Kernel

As we saw in the previous section, the influence of multithreading on performance is strong which suggests that busy servers can make better use of hardware cryptography than clients. This supports the observations of Dean et al. [2001] that it may make sense to make cryptography a shared network service to achieve the best cost/performance in a secure system. Notice that, within the boundaries of one host (operating system instance), this is precisely what the OCF does. We should also mention that use of a threaded model for applications involves an obvious security vs. implementation complexity trade-off.

Although the performance of individual applications may not improve drastically when using an accelerator, it appears that the *aggregate* performance of a number of applications (as may be the case in a system with many remote login sessions, a busy Web server, or a VPN gateway) does improve as a result of increased utilization. Furthermore, hardware accelerators can give a performance boost to the rest of the system as shown in Figure 7. Very simply, they eliminate contention for the CPU which is a resource shared by all applications and the operating system itself. Thus, while throughput is not drastically improved (and may in fact degrade in certain scenarios) with use of hardware acceleration, overall system utilization improves because the main CPU is left to perform other tasks.

6.2 System Architecture

As we saw in Section 5.4, data copying and the PCI bus quickly become the limiting factors. In practice, the situation is even worse since cryptography is used in conjunction with either network security protocols in which case the network interface card (NIC) contends for a slice of the PCI bandwidth, or with filesystem encryption in which case the storage device claims a portion of the bus. This situation suggests that, for maximum performance, cryptographic support must be provided by the individual devices (e.g., NICs, disk controllers, etc.). Alternatively, cryptographic support must be located elsewhere in the system architecture (e.g., attached to the main CPU[7], the system "north bridge" as the video subsystem is) or the memory subsystem. Any of these approaches, if

[7]As of late 2004, at least one vendor provided a proprietary extension through a new instruction to the Pentium processor that used AES circuitry located inside the CPU. We believe this to be a very promising direction for minimizing or even eliminating cryptographic processing overheads.

implemented correctly, will improve application performance by reducing contention for the PCI bus but, at the same time, will create new challenges for operating systems that have to support these new devices such as session migration and fail-over (which the OCF supports by design as we discussed in Section 3).

Although the OCF does not directly take advantage of NICs that support IPsec-processing offloading since they are not general-purpose cryptographic accelerators, we have extended the IPsec stack to use them. The cards of this type we are familiar with are 100 Mbps full-duplex Ethernet, and it seems reasonable to assume that they can achieve this performance given our results with dedicated cryptographic processors. Unfortunately, at the time this article was written, we did not have enough information to write a device driver that could take advantage of such features. We are also not aware of any commercially available hard drive controllers that provide built-in encryption services.

6.3 The Effect of Small Requests

The nature of the challenge for operating systems and their support for cryptography is clear. On every measurement, without exception, small-sized operations fare much worse than those performed on large data buffers. In some cases, buffer size influences performance more than the choice between hardware or software cryptography. This suggests that the per-operation overhead is very high, and this is clear from the larger data sizes which get close to the throughput advertised by the board manufacturer that we presume is the best-case. In this respect, our findings confirm those of Lindemann and Smith [2001]. Since many cryptographic protocols are transactional in nature rather than bulk transfers, these small data operations will be the common case. Energy should be spent on reducing the overhead of such cases.

As we mentioned in Section 5.2, there are several possible approaches including request-batching, kernel crossing, and/or PCI transaction minimization, or simply use of a faster processor. These are more cost effective solutions than deploying a hardware accelerator. In situations where bulk data transfer is the norm (as may be the case in the various Storage Area Network technologies currently under consideration), cryptographic accelerators can drastically improve performance especially for the more expensive algorithms such as 3DES. Unfortunately, there were no commercially available hardware accelerators for AES supported by OpenBSD so we cannot compare the software and hardware cases for that algorithm. However, recent attacks against AES make the continued use of 3DES in many environments likely.

6.4 Other Optimizations and Future Work

In our evaluation of OCF, we noticed a few inefficiencies and potential improvements to the system.

Smarter load-balancing. The load-balancing currently done in OCF, as discussed in Section 3, is very simple. It performs load-balancing of sessions by keeping a record of the active sessions per producer and selecting the least-loaded one. However, not all sessions are equivalent in terms of processing requirements: an FTP-over-IPsec session will use the OCF more heavily than a telnet-over-IPsec one. Furthermore, the current scheme does not perform load-balancing for public-key operations. Finally, all producers of crypto services are considered equal in terms of performance. All these issues point to several potential improvements that can be made to the OCF.

For example, drivers can state their peak performance (experimentally measured, using the vendor provided numbers or measured at system boot time), and the OCF can keep a record of the number of operations actively pending on each driver. However, this requires sessions to be simultaneously established on all these cards, and since these cards have a limited amount of memory for session caching, this approach is perhaps not optimal for a very busy system. One potential solution is to allow the OCF to do dynamic load-balancing of sessions, replicating and tearing them down on additional cards based on their measured traffic by maintaining session information internally. Asymmetric operations are easier to load-balance because they do not depend on the concept of the session. An additional benefit of implementing load-balancing in this way is that we can let the software driver handle small requests, reducing latency, and use the hardware producers for larger requests. One complication to this is that many cards (e.g., Hifn) do not export internal state such as IVs or intermediate MAC results which makes such session sharing difficult.

Algorithm-chaining across cards. It is possible that an OCF consumer needs to chain together a number of cryptographic algorithms but no hardware producer implements all these. Currently, this would cause the session to be established on the software pseudodriver (which implements all algorithms). However, by maintaining session information inside the OCF, it is possible to create virtual sessions across multiple (hardware and software) producers. In this case, the OCF will issue multiple sequential requests to the various producers, invoking the consumer-specified callback routine at the end. We have a prototype of this, but we need to further evaluate the performance implications and trade-offs of doing multiple PCI transactions.

Asymmetric Multiprocessing (AMP) support. There is an increasing number of multiprocessor systems. Most of these underutilize the secondary processor as many modern tasks are I/O-limited. Furthermore, it seems likely that the first version of SMP support for OpenBSD will be very coarse grained: only one processor (and process) can be inside the kernel at a time. An alternative approach is to designate the secondary processor as a dedicated cryptographic accelerator that registers with the OCF as such. No special support by the OCF is necessary, and we are currently working toward an implementation of this.

OpenSSL support algorithm-chaining with OCF. As we mentioned in Section 4.2, TLS and SSH use the OCF at the granularity of the algorithm. That is, if both an encryption and a message authentication algorithm have to be applied on an outgoing message, there will be two distinct calls to the OCF via */dev/crypto.* (The same situation holds for incoming messages.) Since the OCF supports algorithm-chaining, there is no reason why OpenSSL cannot take advantage of this to reduce the number of user/kernel crossings. This requires modification of the TLS implementation in OpenSSL and of OpenSSH to support this algorithm-chaining. While this is purely an implementation matter, the complexity of the OpenSSL code is a significant deterrent to progress in this direction.

Minimize the number of user/kernel crossings and data copying. In most practical uses of the OCF (especially in protocols like TLS or SSH), an application issues one or more crypto requests via */dev/crypto,* followed by a *write()* or *send()* call to transmit the data. Similarly, a *read()* or *recv()* call is followed by a number of requests to */dev/crypto.* This implies considerable data copying to and from the kernel and potentially unnecessary context switching back and forth. An alternative approach is to link some crypto context to a socket or file descriptor (when doing application-level file encryption) such that data sent or received on that file descriptor are processed appropriately by the kernel. For example, a TLS implementation might construct a data record and simply *write()* it to the socket (one data copy and kernel crossing) only to have the kernel pass it to the OCF for processing before actually passing it on to TCP for transmission. This requires some discipline by the application, which must set the state on the socket and only *write()* an appropriately formatted record, as well as some support in the kernel to decode incoming TLS or SSH frames for processing by the OCF before passing them on to the application.

Another potential approach 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 done, the kernel remaps 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 multiple 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. An alternative is to block any process that tries to access such pinned-down pages until the crypto request is completed. Obviously, pages that are shared between processes can cause similar problems even in the current mode of operation. Operations that cross page boundaries also have to be dealt carefully.

Minimize the number of DMA transfers. A similar situation to the multiple kernel crossing scenario just described is present in the use of the PCI bus: a node that is about to transmit an IPsec packet must first DMA it to the cryptographic accelerator, DMA it back to main memory, and finally DMA it to the NIC. This decreases the attainable PCI bandwidth to a third of the theoretical maximum for the bus. The same situation holds in the case of file system encryption. If the NIC (or the storage device) offers on-chip cryptography, we only need one DMA transfer. However, it is possible to reduce the number of DMA transfers to two (instead of three) even when we have a separate 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 IPsec stack—in particular, deferring of cryptographic operations until right before the packet must be transmitted to the network. Fortunately, this is the exact same functionality that the IPsec stack must implement if it supports NICs with integrated crypto. Fortunately, the OpenBSD IPsec stack supports this feature. 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. Again, we believe this is feasible and should improve the performance of IPsec by more than 30%. In Section 7, we describe a similar optimization for application-level cryptographic protocols that achieves a similar speedup.

Emulation of NIC-level TLS/SSH support. Finally, it may be possible to combine the socket with crypto support and the DMA reduction scheme discussed in the previous two items to improve the performance of TLS and SSH by deferring crypto processing until the packet reaches the NIC. In this case, output TCP checksumming must be deferred until after the accelerator has processed the packet. Fortunately, the OpenBSD IP stack supports offloading this computation to the NIC and many modern NICs offer this option. Furthermore, the NIC must receive a jumbo packet with the complete application-layer frame (TLS or SSH record) not just individual TCP-fragmented packets. This will considerably complicate the situation as the TCP and IP layers will try to fragment the packet based on the connection or interface MTU, respectively.

We have two potential solutions to this. One approach is to allow TCP to do the fragmentation and have all the packets that contain a record marked as such and shepherded through the network stack in a bundle. Since most cryptographic accelerators support scatter-gather I/O, it may be possible to combine the data portion of these packets for crypto processing and then perform scatter-DMA to the NIC for multiple packets. The second approach is to prevent TCP and IP from doing fragmentation; the NIC driver will receive a jumbo frame which it can pass to the crypto accelerator for processing. When that is done, it can do scatter-DMA to the NIC, while fixing up the TCP and IP headers on the fly (or have them precomputed) and have the NIC do the TCP and IP header check-summing. Whether either of the two approaches is feasible depends on the capabilities of the DMA engine, the NIC, and the cryptographic accelerator. We will avoid speculation on their performance or complexity of implementation at this point.

7. OPERATING SYSTEM SHORTCUTTING

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 Web server 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

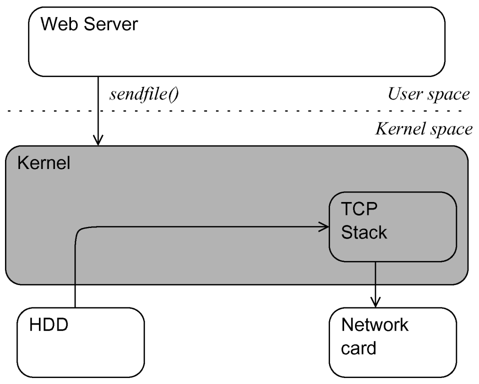

Fig. 10. Apache's default file transfer behavior with no crypto.

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 remap 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 the entire 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.

7.1 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 10. 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 11 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

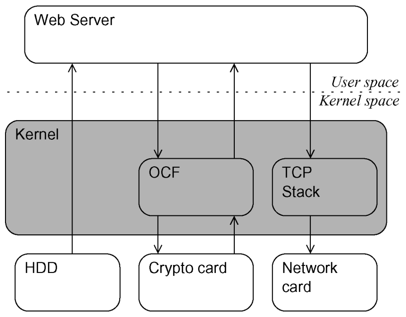

Fig. 11. Current mechanism for encrypting and transfering a file. Note the four user/kernel space crossings, each also encompasses two context switches.

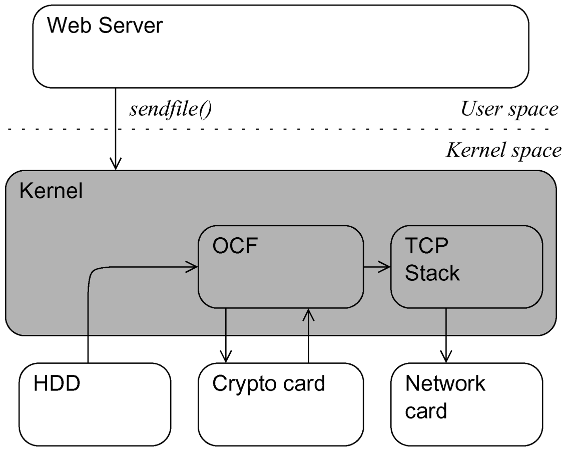

Fig. 12. Encrypting and transfering a buffer with *sendfile()* and *SO_CRYPT*.

API. When the OCF signals completion of the crypto operations, the buffer is passed to *sosend()* and then to the network. The result is the initial and final context switches and no data copies as shown in Figure 12.

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.

7.2 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.

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, etc.). 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 completes, 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.

7.3 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 (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.

Figure 13 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 the application-level data plane from the control plane, performance gains approach 27% for all size file transfers. This gain is due entirely to the elimination of data copies between kernel and user-memory space.

7.4 Further Discussion

The current implementation does not handle incoming data decryption. Such data are passed on directly to the application. Implementing this feature is relatively simple: once the application turns on 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.

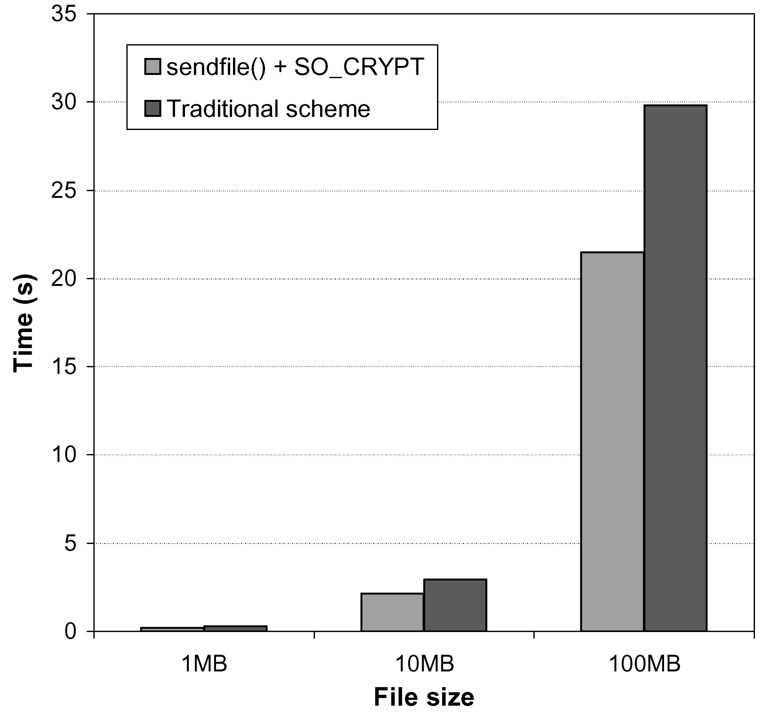

Fig. 13. Comparison of files transfers using our scheme vs. the traditional scheme. The improvement over the traditional scheme on all three file sizes is approximately 27%.

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) as shown in Figure 14.

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 are simply DMA'ed between devices as shown in Figure 14. In this case, the operating system's role becomes that of a flow-controller.

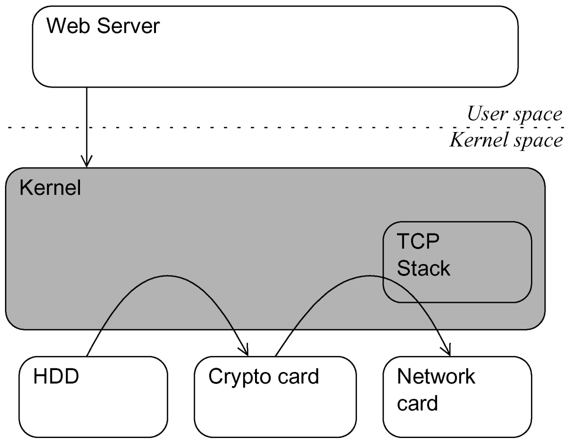

Fig. 14. DMA chaining across multiple devices.

8. CONCLUSIONS

We presented the OpenBSD Cryptographic Framework (OCF), a service virtualization layer implemented inside the kernel, that provides uniform access to cryptographic hardware accelerator cards by hiding card-specific details behind a carefully designed API. Other kernel subsystems and user-level processes can use the API with symmetric and asymmetric algorithms. The OCF offers several other features such as load-balancing, session migration, and algorithm-chaining.

Our performance evaluation demonstrated the OCF's ability to utilize available accelerators to within 95% of their peak performance. This validates our decision to design for ease of use by applications and seamless support for new accelerators over a device-specific approach which should be able to fully utilize that device's capabilities. In addition, we demonstrated aggregate (across several concurrent applications) throughput for 3DES encryption in excess of 800 Mbps. Furthermore, use of hardware accelerators can remove contention for the CPU and thus improve overall system responsiveness and performance for unrelated tasks.

Our evaluation also allowed us to determine that the limiting factor for high-speed cryptography in modern systems is data copying and the PCI bus. Furthermore, small data-buffers should be processed in software, freeing hardware accelerators to handle larger requests that better amortize the system and PCI transaction costs. On the other hand, multithreading results in increased utilization of the OCF, improving *aggregate* throughput. We made recommendations for future directions in architectural placement of cryptographic functionality, operating system provisions, and application design, and discussed several improvements and promising directions for future work.

We evaluate one of our recommendations, operating system shortcutting which eliminates all unnecessary data copies between the kernel and the user-level process with minimum modifications to both the kernel and the application. 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.

APPENDIX A: OCF KERNEL API

—*int32_t crypto_get_driverid();*
int crypto_register();
int crypto_kregister();
int crypto_unregister();
Used by device drivers to register and unregister symmetric and asymmetric algorithm support with the OCF.

—*void crypto_done();*
void crypto_kdone();
Called by device drivers on completion of a request (symmetric and asymmetric, respectively).

—*int crypto_newsession();*
Called by consumers of cryptographic services (such as the IPsec stack) that wish to establish a new session with the framework. On success, the first argument will contain the Session Identifier (SID). The second argument contains all the necessary information for the driver to establish the session (keys, algorithms, offsets, etc). The third argument indicates whether only hardware acceleration is acceptable.

—*int crypto_freesession();*
Called to disestablish a previously-established session.

—*int crypto_dispatch();*
Called to process a request, encapsulated in its only argument. The various fields in that structure contain:

(1) The SID.
(2) The total length in bytes of the buffer to be processed,
(3) The total length of the result which for symmetric crypto operations will be the same as the input length.
(4) The type of input buffer as used in the kernel *malloc()* routine. This will be used if the framework needs to allocate a new buffer for the result (or for reformatting the input).
(5) The routine that the OCF should invoke upon completion of the request whether successful or not.
(6) The error type, if any errors were encountered. If the **EAGAIN** error code is returned, the SID has changed. The consumer should record the new SID and use it in all subsequent requests. In this case, the request may be resubmitted immediately. This mechanism is used by the framework to perform session migration (move a session from one driver to another because of availability, performance, or other considerations).

(7) A bitmask of flags associated with this request. Currently, the only flag defined is **CRYPTO_F_IMBUF** which indicates that the input buffer is an mbuf chain.
(8) The input and output buffers. The input buffer may be an mbuf chain or a contiguous buffer (as identified by the flags). The output buffer will be of the same type.
(9) A pointer to opaque data. This is passed through the crypto framework untouched and is intended for the invoking application's use.
(10) A linked list of operation descriptors which indicate what operations should be applied, and in what sequence, to the input data. The descriptors indicate where each operation should start, the length of the data to be processed, where on the output buffer the results should be placed, the key material to be used, and various operation-specific flags (e.g., what Initialization Vector to use for CBC-mode encryption).

—*int crypto_kdispatch();*
Similar to *crypto_dispatch()*, for public-key operations.

ACKNOWLEDGMENTS

Bob Beck and Markus Friedl helped with numerous OpenSSL integration issues we faced, since the engine code we required was unreleased. Bob also wrote the first working OpenSSL engine interfacing with */dev/crypto*. Markus helped with regression tests to ensure that */dev/crypto* operation was correct. Jonathan Smith and Sotiris Ioannidis provided valuable comments and insights. Sam Leffler adapted the OCF to the FreeBSD kernel. We would also like to thank Patrick McDaniel for providing high-quality shepherding of this article.

REFERENCES

ADAMS, C. 1998. Independent data unit protection generic security service application program interface (IDUP-GSS-API). RFC 2479. (Dec).

BONEH, D. AND SHACHAM, N. 2001. Improving SSL handshake performance via batching. In *Proceedings of the RSA Conference*.

BROSCIUS, A. G. AND SMITH, J. M. 1991. Exploiting parallelism in hardware implementation of the DES. In *Proceedings of the Crypto Conference* (Santa Barbara). 367–376.

CLAFFY, K., MILLER, G., AND THOMPSON, K. 1998. The nature of the beast: Recent traffic measurements from an Internet backbone. In *Proceedings of the ISOC INET Conference*.

COARFA, C., DRUSCHEL, P., AND WALLACH, D. 2002. Performance analysis of TLS Web servers. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)* San Diego, CA.

COOK, D., IOANNIDIS, J., KEROMYTIS, A., AND LUCK, J. 2005. CryptoGraphics: Secret key cryptography using graphics cards. In *Proceedings of the RSA Conference, Cryptographer's Track (CT-RSA)*. 334–350.

DE RAADT, T., HALLQVIST, N., GRABOWSKI, A., KEROMYTIS, A. D., AND PROVOS, N. 1999. Cryptography in OpenBSD: An overview. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*. 93–101.

DEAN, D., BERSON, T., FRANKLIN, M., SMETTERS, D., AND SPREITZER, M. 2001. Cryptology as a network service. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

DRUSCHEL, P., ABBOTT, M. B., PAGELS, M. A., AND PETERSON, L. L. 1993. Network subsystem design. *IEEE Network 7,* 4 (July) 8–17.

FELDMEIER, D. C. AND KARN, P. R. 1990. UNIX password security—Ten years later. In *Proceedings of the Crypto Conference*. 44–63.

GATTANEO, G., CATUOGNO, L., SORBO, A. D., AND PERSIANO, P. 2001. The design and implementation of a transparent cryptographic filesystem for UNIX. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*.

GOLDBERG, A., BUFF, R., AND SCHMITT, A. 1998. Secure Web server performance dramatically improved by caching SSL session keys. In *Workshop on Internet Server Performance (held in conjunction with SIGMETRICS'98)*.

GPG 2003. Available at www.gpgpu.org.

GUPTA, V., STEBILA, D., FUNG, S., SHANTZ, S. C., GURA, N., AND EBERLE, H. 2004. Speeding up secure Web transactions using elliptic curve cryptography. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*. 231–239.

GUTMANN, P. 1999. The design of a cryptographic security architecture. In *Proceedings of the 8th USENIX Security Symposium*.

GUTMANN, P. 2000. An open-source cryptographic coprocessor. In *Proceedings of the 9th USENIX Security Symposium*.

KAY, J. AND PASQUALE, J. 1993. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of the ACM SIGCOMM Conference*. 259–269.

KENT, S. AND ATKINSON, R. 1998. Security architecture for the Internet protocol. RFC 2401 (Nov).

KEROMYTIS, A. D., IOANNIDIS, J., AND SMITH, J. M. 1997. Implementing IPsec. In *Proceedings of Global Internet (GlobeCom)*. 1948–1952.

LINDEMANN, M. AND SMITH, S. W. 2001. Improving DES coprocessor throughput for short operations. In *Proceedings of the 10th USENIX Security Symposium*. 67–81.

LINN, J. 1997. Generic security service application programming interface. RFC 2078. (Jan).

MACEDONIA, M. 2003. The GPU enters computing's mainstream. *IEEE Computer*. 106–108.

MCGREGOR, J. P. AND LEE, R. B. 2004. Protecting cryptographic keys and computations via virtual secure coprocessing. In *Proceedings of the Workshop on Architectural Support for Security and Anti-virus (WASSA)*. 11–21.

Microsoft Corporation 1998. *Microsoft Cryptographic Application Programming Interface (CryptoAPI)*, 2nd Ed. Microsoft Corporation.

MILTCHEV, S., IOANNIDIS, S., AND KEROMYTIS, A. D. 2002. A study of the relative costs of network security protocols. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, Monterey, CA. 41–48.

NATIONAL SECURITY AGENCY. 1997. *Security Service API: Cryptographic API Recommendation*. Updated and Abridged Ed. Cross Organization CAPI Team (July).

PROVOS, N. 2000. Encrypting virtual memory. In *Proceedings of the USENIX Security Symposium*.

PU, C., MASSALIN, H., IOANNIDIS, J., AND METZGER, P. 1988. The synthesis system. *Computing Syst. 1,* 1.

RSA LABORATORIES. 1997. PKCS #11: Cryptographic token interface standard, version 2.01.

TRAW, C. B. S. AND SMITH, J. M. 1993. Hardware/software organization of a high-performance ATM host interface. *IEEE J. Select. Areas Comm.* (Special Issue on High Speed Computer/Network Interfaces) *11,* 2 (Feb). 240–253.

SHIRASE, M. AND HIBINO, Y. 2004. An architecture for elliptic curve cryptograph computation. In *Proceedings of the Workshop on Architectural Support for Security and Anti-virus (WASSA)*. 120–129.

SMITH, J. M. AND TRAW, C. B. S. 1993. Giving applications access to Gb/s networking. *IEEE (Network) 7,* 4 (July), 44–52.

SMITH, J. M., TRAW, C. B. S., AND FARBER, D. J. 1992. Cryptographic support for a gigabit network. In *Proceedings of INET*. 229–237.

SMYSLOV, V. 1999. Simple cryptographic program interface (Crypto API). RFC 2628. (June).

THE OPEN GROUP 1999. *Common Data Security Architecture (CDSA)*, 2nd Ed. The Open Group.

THOMPSON, C., HAHN, S., AND OSKIN, M. 2002. Using modern graphics architectures for general-purpose computing: A framework and analysis. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Micro Architecture (MICRO-35)*. 306–317.

Received September 2004; revised July 2005; accepted August 2005