Transparent Multi-core Cryptographic Support on Niagara CMT Processors*

James Hughes, Gary Morton, Jan Pechanec,
Christoph Schuba, Lawrence Spracklen, and Bhargava Yenduri

Sun Microsystems, Inc.
10 Network Circle
Menlo Park, CA 95025 – USA
{Firstname.Lastname}@Sun.COM

Abstract

How cryptographic functionality has been implemented and made available in application scenarios has evolved over time. Pure software implementations were the obvious first choice, followed by dedicated hardware devices, be it co-processors or hardware accelerators accessible on the main bus.

This paper examines aspects of making the next step in this evolution work, namely the use of dedicated cryptographic hardware that's part of multi-core system CPUs. While the inclusion of cryptographic accelerator functionality in the processor chip is not new, this paper investigates the question of how to transparently combine such multi-core cryptographic processor support with higher level software stacks in a commodity operating system that also needs to perform well if such hardware support is not present.

We explore this question in the context of the UltraSPARC T1 and T2 processor family, Chip Multi-Threaded (CMT) processors that have hardware cryptographic accelerators integrated on-chip with 8-core support for symmetric and asymmetric cryptographic and secure hash operations. The paper presents how a software infrastructure, the Solaris Cryptographic Framework, transparently takes advantage of these chip features and presents a brief comparative study of their performance.

1 Introduction

High performance and high security cryptography has moved from obscure government requirements to mainstream machines with the killer application being scalable electronic commerce. The ability for a web server to maintain and communicate concurrently over thousands of SSL sessions is a recurring requirement. Even enterprises without these levels of workload are interested in making sure their infrastructure can scale when faced with success and rapid growth or the need to temporarily accommodate flash crowds. There are other applications scenarios with similar characteristics in the context of encrypted transactions or encrypted storage. These examples apply in industries beyond government, such as the health care or telecommunications industries.

How cryptographic functionality has been implemented and made available in such application scenarios has evolved over time. Pure software implementations were the obvious first choice, followed by dedicated hardware devices, be it co-processors (e.g., FPGA-based) or hardware accelerators accessible on the main bus (e.g., the Sun Cryptographic Accelerator 6000 PCIe Card.) The latter approach became especially popular for high volume web transactions that needed to be encrypted via SSL.

This paper examines aspects of making the next step in this evolution work, namely the use of dedicated cryptographic hardware that's part of multi-core system CPUs. While the inclusion of cryptographic accelerator functionality in the processor chip is not new, this paper investigates the question how to transparently combine such multi-core cryptographic processor support with higher level software stacks.

1.1 The (Open)Solaris Cryptographic Framework

The Solaris Cryptographic Framework (CF) (see [8]) provides cryptographic services to users and applications through commands, a user-level programming interface, a

*Published at the Second International Workshop on Multicore Software Engineering (IWMSE09), Vancouver, Canada, May 2009.

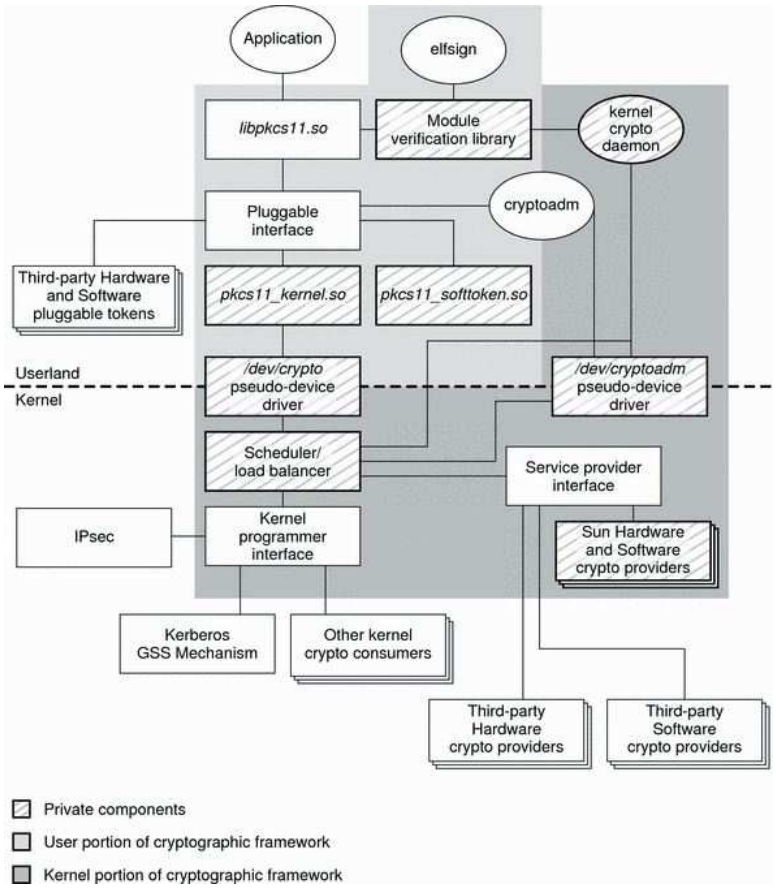

Figure 1. The(Open)Solaris Cryptographic Framework

kernel programming interface, and user-level and kernel-level frameworks. The Cryptographic Framework provides these services to applications and kernel modules in a manner that is seamless to the end user, and brings direct cryptographic services, such as file encryption and decryption, to the end user. Figure 1 depicts the architecture of the framework with its userland and kernel components.

The user-level framework is responsible for providing cryptographic services to consumer applications and the end user commands. The kernel-level framework provides cryptographic services to kernel modules and device drivers. Both frameworks give developers and users access to optimized cryptographic algorithms. The programming interfaces are a front end to each framework. A library or a kernel module that provides cryptographic services can be plugged into one of the frameworks by the system administrator. This architecture makes the plug-in's cryptographic services available to applications or kernel modules. Using the Cryptographic Framework has many benefits. Applications and kernel services do not need to re-implement complete cryptographic functions but can use existing implementations, reducing code duplication and thereby reducing the number of bugs. In addition, the cryptographic implementations can be optimized for the available hardware, including hardware accelerators. The resulting performance benefits are directly available to the userland and kernel services. Access to the framework in user space is primarily through the PKCS#11 library.

1.2 The UltraSPARC cryptographic accelerators

Starting with the introduction of Sun's UltraSPARC T1 processor in 2004, Sun's Chip Multi-Threaded (CMT) processors (see [9]) have had hardware cryptographic accelerators integrated on-chip. The rationale for moving the accelerators on-chip was not only the cost savings (no requirement for an add-on accelerator card), but also, in an attempt to reduce the high CPU utilization, I/O bandwidth, and I/O latency overheads typically associated with off-chip accelerator cards. These overheads tend to make the use of off-chip cards problematic for the effective acceleration

of bulk ciphers, especially for small or moderately sized packets. Employing *high-performance* off-chip accelerators, using e.g., Hypertransport or Front-Side-Bus connectivity, may help reduce some of these issues. However, repeatedly moving the data off and on chip is less efficient than using on-chip accelerators that are tightly coupled with the processor cores.

On the UltraSPARC T1, each of the processor's eight cores has an associated cryptographic accelerator that is targeted at offloading (and accelerating) public-key cryptographic operations. In essence, these accelerators, termed modular arithmetic units (MAU), perform the modular exponentiation operations that lie at the heart of algorithms such as RSA and Diffie-Hellman.

With the introduction of the UltraSPARC T2 in 2007, the processor's on-chip cryptographic support was further extended. Support for Elliptic Curve Cryptography (ECC) – both prime and binary polynomial fields – was added to the MAU. Additionally, a per-core unit to accelerate symmetric ciphers (RC4, DES, 3DES, AES) and hash operations (MD5, SHA-1, SHA-256) was also introduced. The two sub-units can operate in parallel, such that each core's accelerator can concurrently perform, for example, a public-key operation and a bulk cipher operation. These operations also happen in parallel to instruction processing on the remainder of the core. In other words, each core's accelerator can concurrently perform an RSA operation and an AES operation, in addition to the 8 hardware threads per core that can continue to execute unimpeded.

Interaction with the cipher/hash units is via memory-based control word queues, with each core's accelerator having its own private queue. To offload an operation to the accelerator, software inserts a control word at the end of the queue and informs the accelerator of the update. The control word provides the accelerator with all the information required to perform the requested operation, i.e., pointers to src, dst, keys, IV's and length information. As a result, the accelerator is stateless, giving it maximum key agility, an important feature in application spaces with thousands of simultaneous connections (e.g., Secure Web, Secure VoIP). Additionally, given this light-weight interface to the hardware, the overheads associated with offloading an operation to the accelerator can be small, allowing even operations that take little time to be offloaded cost-effectively.

It is possible to interact with the accelerator in a synchronous or an asynchronous manner, such that, if desired, it is possible to perform other useful processing on the core while the cryptographic operation is being performed on the accelerator; this feature provides an additional level of parallelism that is not achieved when ISA customization is used to achieve cryptographic acceleration.

Discrete on-chip accelerators can be particularly effective on CMT processors, where execution resources are more limited and instruction-based cryptographic acceleration may be a less compelling feature. Furthermore, there is also the opportunity to tightly integrate discrete accelerators with secure keystores (e.g., the PKCS#11 softtoken keystore), ensuring that, unlike instruction-based acceleration, sensitive key information does not need to reside in easily accessible non-privileged user address spaces.

2 Transparent Hardware-accelerated Software Stacks

The Niagara* T1 and T2 cryptographic drivers plug into the Solaris Cryptographic Framework as hardware providers. This approach makes them transparently accessible to any user of the Cryptographic Framework, in both kernel and user space. From a user application perspective, the T2 cryptographic capabilities are accessible to OpenSSL through the PKCS#11 engine shipped with the Solaris Operating System, to Java, and to native PKCS#11 users given that the Cryptographic Framework supports all of those APIs.

The user-level framework has two special plug-in shared libraries: pkcs11_kernel.so and pkcs11_softtoken.so. The pkcs11_kernel.so shared library interacts with the kernel-level framework to take advantage of hardware providers. It uses the ioctl interface of the pseudo device driver, /dev/crypto, to invoke operations in the kernel-level framework. The pkcs11_softtoken.so shared library provides a software implementation of all standard cryptographic algorithms.

The Cryptographic Framework offers a metaslot service which gives a user transparent access to providers registered on that system. A user of metaslot on a T2 system would use the T2 cryptographic hardware for the algorithms which it supports and another provider for those which the hardware does not handle. In other words, the user-level framework uses the pkcs11_kernel.so library first, thus assuring that applications benefit from any hardware providers present on the system. If no hardware providers are present, it uses pkcs11_softtoken.so plus any other plug-in library added by the administrator.

The PKCS#11 API offered by the user-level framework supports only a synchronous mode of operation. The API offered by the kernel-level framework supports both synchronous and asynchronous modes of operations. In the asynchronous mode, the cryptographic operation is initiated and control is returned to the caller. The cryptographic operation continues concurrently with other activity of the caller. This mode is useful for callers in interrupt context.

The cryptographic hardware is accessed via a queue interface. There are separate queues for each core as

Niagara is an alternate name for UltraSPARC processors.

well as for symmetric and asymmetric cryptographic algorithms. As new requests come into the cryptographic driver, they are enqueued on the appropriate core's cryptographic queue. The hardware processes requests in FIFO order, and the requesting threads are notified when their jobs have completed.

On larger, CMT systems, lock contention is an obvious issue. To reduce contention, the cryptographic driver code implements locking on a per core basis, which reduces contention down to the 8 strands on that core. Having strands on the same core vie for locks and resources is beneficial in that they share the same caches, avoiding some cache coherency issues that can occur otherwise. When cryptographic requests arrive at the cryptographic drivers, the request is scheduled on a core by (i) using the core that thread is currently running on (binding it to a CPU to avoid migration) or (ii) dispatching the request on a thread already bound on a particular core. Both options guarantee that the submitting thread is running on the desired core prior to grabbing the per core resource related locks.

For asymmetric cryptographic requests, the ncp driver[†] schedules requests in a round robin fashion across all cores. The overhead of the cryptographic operation/computation itself is significantly larger than the overhead to schedule on a new core so this strategy improves performance by attempting to use the engines in parallel. For symmetric cryptographic workloads, requests are scheduled on the core that the submitting thread is running on. In general, the overhead to migrate the submitter from one core to another was found to be larger than the time to process the requests. The Solaris Operating System thread scheduling provides as a side effect reasonable load balancing for cryptographic workloads when assigning cryptographic requests to the accelerator of the executing core. Attempts to try and improve overall performance by additionally load balancing just the cryptographic workloads across separate cores have been unsuccessful. This result is in part because a thread can only interact with the accelerator associated with the core on which the thread is executing. Accordingly, while the memory-based queue can be updated, it is necessary for a hyperprivileged interprocessor interrupt (i.e., cross call) to inform the remote accelerator of the update.

A Cryptographic Framework client can submit multiple asynchronous cryptographic requests without waiting for the completion of an earlier request. The Cryptographic Framework internally maintains a pool of threads that handle these requests in parallel. This architecture results in improved throughput because each thread is typically scheduled on a separate core by the Solaris scheduler which results in each request being handled by a separate cryptographic unit.

[†]The abbreviation *ncp* stands for Niagara (i.e., T1) cryptographic provider; *n2cp* for Niagara 2 (i.e., T2) cryptographic provider.

2.1 Resisting side-channel attacks

Another advantage of using the UltraSPARC T2 hardware accelerators is that many of the more practical side-channel attacks are impossible. For instance, many cache-based attacks are not feasible (no use of cache-based lookup tables) and a number of timing attacks are similarly thwarted (symmetric cipher performance is constant) (see [2]). While timing attacks against RSA (see [3]) are not eliminated, they are made more difficult, because the decreased computation time and the timing vagaries introduced by asynchronous operation make timing noise more problematic for such attacks to succeed. While protection against these attacks can be added in software if necessary, there is typically a non-negligible performance penalty.

2.2 Accelerators and virtualization

On the UltraSPARC CMT processors, there are multiple threads (4 on the T1, 8 on the T2) per core, and the core's hardware accelerator is shared between them. With virtualization (Logical Domains (LDOMs) for SPARC), each of these separate threads can be running a separate OS instance. Accordingly, access to the accelerator is hyperprivileged. Currently, the software does not support sharing a core's accelerator between multiple domains. However, as the accelerator is under the sole control of the Hypervisor, secure sharing of the accelerator, with strictly enforced per-domain QoS standards, could be introduced in future versions of LDOMs – each OS has its own virtual copy of the in-memory queue in privileged address space. The Hypervisor maintains the single real version of the queue in hyperprivileged space and copies requests from the virtual queues into the real queue, facilitating sharing and QoS enforcement.

3 UltraSPARC T2 accelerator performance

The UltraSPARC T2 cryptographic accelerators operate at the same frequency as the UltraSPARC cores (1.4GHz) and each deliver up to 5.5Gb/s of AES-128 throughput (for a total of 44Gb/s per processor). The accelerators request data directly from the processors on-chip Level-2 cache, and support multiple outstanding data requests, allowing data to be streamed directly from DRAM memory without performance impact. The peak performance delivered on an 8-core processor for a variety of additional algorithms is illustrated in Tables 1 and 2.

To put these numbers in perspective, a 2.7GHz quad-core x64 processor is capable of delivering an aggregate of 4.2Gb/s of AES-128 throughput, when using all four cores. This analysis illustrates the benefits of hardware cryptographic acceleration, i.e., a 10-fold performance improve-

Algorithm Performance	(Gb/s)
AES-128	44
AES-192	36
AES-256	31
3DES	27
RC4	83
MD5	41
SHA-1	32
SHA-256	41

Table 1. Aggregate performance delivered by 8 UltraSPARC T2 cryptographic accelerators for bulk ciphers and secure hash operations

Algorithm Performance	(sign operations / sec)
RSA-1024	37,000
ECCp-160	52,000
ECCb-163	92,000

Table 2. Aggregate performance delivered by 8 UltraSPARC T2 cryptographic accelerators for public-key operations

ment when running at roughly half the clock speed. Further, given the cryptographic processing is offloaded to the accelerators on the T2 processor, there are still significant idle CPU cycles available to perform useful processing with the results from the cryptographic operations; this capacity is in stark contrast to the situation with software cryptographic processing on x64 processors, where 100% of cycles on all 4 cores are being utilized.

4 Harnessing Multiple Cores In Real Applications

There are a few things to consider when making the decision about whether to use the Cryptographic Framework (CF), and whether to harness multiple cryptographic cores. Applications stand to benefit from using multiple cryptographic cores, if, e.g., multiple instances of the application (i.e., multiple processes) are running or if the application is multi-threaded.

4.1 Repetitive Operations

The first question to address is which cryptographic operations to offload to the CF and which operations to process in independent software library implementations, such as OpenSSL (see [5]) or NSS (see [4]). Obviously, there is little benefit to offload rare operations, while there is a lot of benefit to offload operations that are frequently repeated by, e.g., running multiple instances of the same application. The web server is an example of an application that can greatly benefit from the offloading of public key cryptographic operations, needed for SSL connection handshakes, to the CF. A web server expects a large number of SSL connections (that require RSA/DSA/DH operations) that may be short lived and that may not process much data.

A typical example that won't greatly benefit from speeding up the public key cryptographic operations is an application that implements the SSH protocol. Any SSH implementation uses such operations during the initial key exchange, possibly for the user authentication as well, and sporadically during an optional key re-exchange. Additionally, SSH connections are often long lived. The overhead of the initialization of an SSH connection is an order of magnitude greater in comparison with the time needed to generate a shared secret through the Diffie-Hellman key exchange protocol plus several RSA/DSA signature operations. However, SSH data links can be used for backups or large data transfers. In that case, offloading the symmetric cryptographic operations and message digests may greatly speed up the whole process. Figure 2 illustrates the execution time breakdown for the SPECweb05 banking benchmark on an UltraSPARC processor, without cryptographic hardware support.

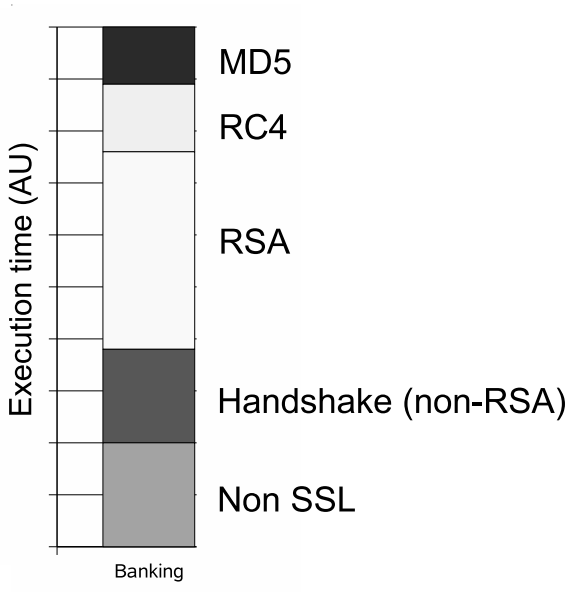

Figure 2. The execution time breakdown for the SPECweb05 benchmark on an UltraSPARC processor, without cryptographic hardware support.

Additionally, given this light-weight interface to the hardware, the overheads associated with offloading an operation to the accelerator can be quite small, allowing even short duration operations to be cost-effectively offloaded – as illustrated in Figure 3.

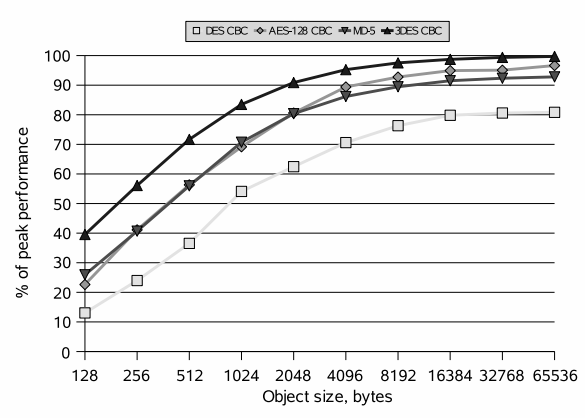

Figure 3. Performance of the UltraSPARC T2 cryptographic accelerator for various object sizes and ciphers.

These overheads tend to make use of off-chip cards problematic for the effective acceleration of bulk ciphers, especially for small or moderately sized packets (which can be problematic, given the prevalence of these packet sizes in some application spaces, as illustrated in Figure 4).

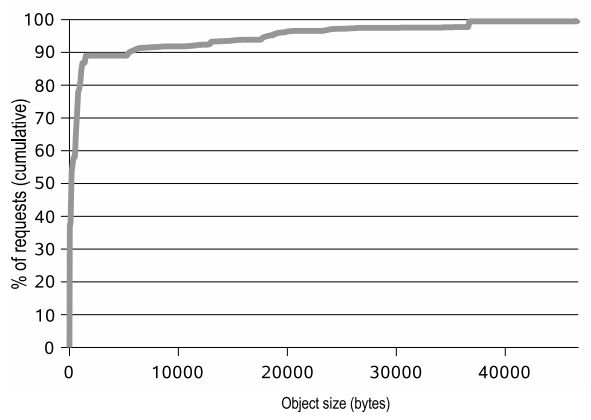

Figure 4. Size of server responses for SPECweb05 banking benchmark.

4.2 Packet Size Matters

When processing symmetric cryptographic and digest operations the important issue is the size of the data blocks that are processed. The CF has its own overhead before the data block is actually transferred to a hardware cryptographic accelerator, and the accelerator itself has an overhead that is not related to the data block size. There is a break-even point for every algorithm that determines what data block size is the size when the software library implementation is of the same speed as if offloading the block to the CF.

There is a small example with the OpenSSL speed(1) program. Tables 3 and 4 are used to illustrate how to determine the break-even point for the AES CBC mode with 128 bit keys. The comparison is for the use of OpenSSL either via the native AES implementation built with the Sun Studio compiler or via the cryptographic operations offloaded to the CF through the PKCS#11 engine[‡] on an UltraSPARC T5220 machine.

According to these experiments a number of observations can be made. The break-even point for AES CBC mode with 128 bit key is between 384 and 512 bytes. Processing of 16 byte data blocks using the CF is an order of magnitude slower than using the native OpenSSL AES code. Finally, the operational overhead of the Cryptographic Framework on small data blocks is what determines the overall time spent.

Transferring large amounts of data is the only situation when the CF can help with respect to symmetric cryptographic and digest operations. The size of packets or processed blocks of data is usually far enough behind the break-even point for such algorithms. However, which blocks are processed through the cryptographic operations may not be immediately clear without detailed knowledge of the underlying protocols and algorithms used. The SSH protocol version 2 encrypts the packet length as well. Therefore, on the receiving side the first cipher block needs to be decrypted to obtain the length of the additional data that are to be read before the MAC checksum can be computed. If a 512 byte long SSH packet is decrypted with two operations, the OpenSSL speed's output above demonstrates that the time spent in those cryptographic operations is roughly twice the time spent for the decryption of the whole 512 byte data block at once.

4.3 Not All Cryptographic Operations Can Be Parallelized

In an SSH application using multiple threads to encrypt and decrypt transferred data in parallel data chunks need to be processed independently. This requirement cannot be satisfied in one of the most widely used cipher mode

[‡]PKCS#11 *engine*: the OpenSSL PKCS#11 engine serves as a liaison between OpenSSL and the CF so that existing applications using OpenSSL can, with minimal changes, make use of the CF hardware providers. How exactly the PKCS#11 engine works is out of the scope of this paper, see [7], [6], and [1] for more information.

$ openssl speed -evp aes-128-cbc
Doing aes-128-cbc for 3s on 16 size blocks: 2288268 aes-128-cbc's in 2.99s
Doing aes-128-cbc for 3s on 64 size blocks: 703001 aes-128-cbc's in 3.00s
Doing aes-128-cbc for 3s on 256 size blocks: 186333 aes-128-cbc's in 3.00s
Doing aes-128-cbc for 3s on 384 size blocks: 125058 aes-128-cbc's in 3.00s
Doing aes-128-cbc for 3s on 512 size blocks: 94113 aes-128-cbc's in 3.00s
Doing aes-128-cbc for 3s on 640 size blocks: 75442 aes-128-cbc's in 3.00s
Doing aes-128-cbc for 3s on 768 size blocks: 62952 aes-128-cbc's in 3.00s
...

Table 3. Performance of native T2 AES CBC mode with 128 bit keys.

$ openssl speed -evp aes-128-cbc -engine pkcs11 -elapsed
Doing aes-128-cbc for 3s on 16 size blocks: 121829 aes-128-cbc's in 2.99s
Doing aes-128-cbc for 3s on 64 size blocks: 120523 aes-128-cbc's in 3.00s
Doing aes-128-cbc for 3s on 256 size blocks: 116682 aes-128-cbc's in 3.00s
Doing aes-128-cbc for 3s on 384 size blocks: 113612 aes-128-cbc's in 3.00s
Doing aes-128-cbc for 3s on 512 size blocks: 109206 aes-128-cbc's in 3.00s
Doing aes-128-cbc for 3s on 640 size blocks: 105578 aes-128-cbc's in 3.00s
Doing aes-128-cbc for 3s on 768 size blocks: 102965 aes-128-cbc's in 3.00s
...

Table 4. Performance of T2 AES CBC with 128 bit keys accessed through the PKCS11 interface.

currently used, cipher block chaining (CBC). Using CBC, every block depends on the previously encrypted cipher block, making parallelization impossible. However, another widely used cipher mode is the counter mode (CTR). This mode encrypts successive values of a counter, and the resulting data stream is xor'ed with the data to be transferred. Since the counter is predictable one can generate and encrypt different parts of the same counter sequence independently, enabling the ability to take advantage of multiple cryptographic cores.

Using the counter mode can also help with the problem of decrypting the packet length cipher block before decrypting the rest of the packet, which uses two CF operations. In counter mode, the counter sequence can be decrypted in advance in large data chunks, reducing the overhead of the CF operation.

4.4 SSH/SCP/SFTP With the Cryptographic Framework Support

The SCP and SFTP protocols implemented in SunSSH (see [10]) use SSH transfer to transfer data and do not perform any cryptographic operations themselves. Therefore, the cryptographic operations issued by the SSH client and SSH server are all that matter. The SunSSH client and server already make use of the OpenSSL PKCS#11 engine (see [1]) through which the CF hardware providers can be used. Our tests show that the speed transfer on UltraSPARC T2 machines dropped to roughly 40% of the baseline measurement, a 2.5-fold speed-up. SunSSH with the PKCS#11 engine support is available as opensource, with the latest OpenSolaris distribution, and is planned to be delivered with the upcoming Solaris 10 Update 7 release as well.

The next step is to add multi-threading support which should get further speed gains. SunSSH already uses AES CTR mode as the default mode. Once the multi-threaded support is available, even connections made by older SunSSH clients will benefit from parallelism on the server side, because clients select their cipher mode from the list of ciphers offered by the server.

4.5 AES-CTR performance through the CF with the PKCS#11 API

The peak numbers (see Table 1) for the hardware cores represent the maximum performance. However, more important from the developer's point of view is how much data can be processed by her code. This capacity can be illustrated writing a simple application written with the PKCS#11 API, using AES-CTR with all 128/192/256 bit keys. AES with CBC gives the same result, the actual mode of operation make no difference in this example. The application starts a given number of threads, with every thread initializing the encryption with the C_EncryptInit call, following by multiple C_EncryptUpdate operations. Every thread performs 50000 operations with 16KB data buffer.

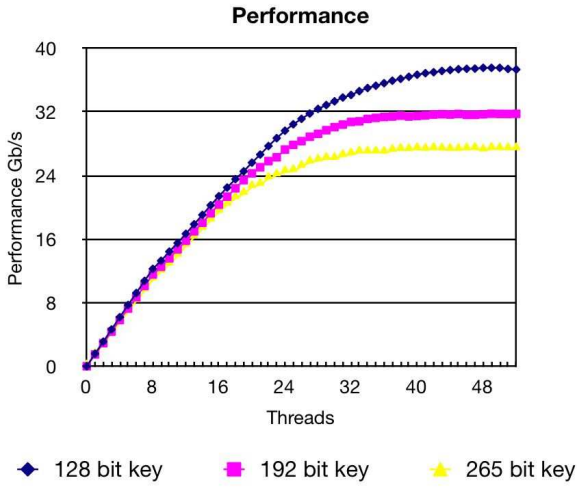

Figure 5. Performance of AES in CTR mode with varying keys sizes and number of application threads.

To avoid unnecessary overhad the application does not generate or read any data. The buffer is filled with random data.

The tests was run 10 times for each key length and the results displayed in Figure 5 are the average of all the test runs for the respective key length. The test machine was UltraSPARC T5120 with 8 cores installed with Solaris Nevada 106. It took 50 threads to get to the peak of roughly 37, 32, and 27 Gbit/sec (4.6, 4, and 3.4 GB/sec) of data encrypted. There were no attempts to further tune the application or the system to find out whether the peak is the maximum data throughput that can be achieved from the application linked to the libpkcs11 library.

5 Conclusions

Based on its consumer-provider architecture, the Solaris Cryptographic Framework is an ideal candidate to present how multi-core functionality can be provided transparently to applications, in both the user space as well as the kernel of a commodity operating system. Multi-Core hardware-accelerated cryptographic functions, as available in the UltraSPARC T1 and T2 processor family, plug into this framework as providers from below. Depending on the functions provided they can be accessed synchronously or asynchronously.

We presented just how the multi-core functionality can be hidden, yet yield tremendous performance gains, as witnessed by our algorithmic performance characterization and our analysis of ssh, scp, and sftp performance, with or without the use of PKCS#11 interfaces.

Interesting work remains to be done. E.g., while our approach utilizes the hardware support in a stateless mode, maintaining state in the cryptographic cores could lend itself to higher performance. This gain would come at the cost of less scheduling flexibility under the requirement to maintain strict data separation. In future hardware, providing the ability to remotely launch operations on other core's accelerators, would make it much easier for software to load balance operations across accelerators, and could even make distributing single operations (e.g., large ECB or CTR operations) across accelerators an interesting option.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. Thanks also to the authors of the documentation for the Solaris Cryptographic Framework who generously gave us text for introducing the architecture of the (Open)Solaris Cryptographic Framework in section 1.1.

References

[1] *engine(3), OpenSSL's manual page for ENGINE cryptographic module support*.

[2] D. J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[3] P. C. Kocher. Timing attacks on implementations of diehellman, rsa, dss, and other systems. pages 104–113. Springer-Verlag, 1996.

[4] Netscape Corp. NSS, the Network Security Services, http://www.mozilla.org/projects/security/pki/nss.

[5] OpenSSL. The open source toolkit for SSL/TLS, http://www.openssl.org.

[6] Pechanec, Jan and Schuba, Christoph and Phalan, Mark. New Security Features in OpenSolaris and Beyond. In *Proceedings of OpenSolaris Developer Conference*, Prague, Czech Republic, 2008.

[7] RSA Laboratories. PKCS11: Cryptographic Token Interface Standard, http://www.rsa.com/rsalabs/pkcs.

[8] P. Sangster, V. Bubb, and K. Belgaied. The Solaris Cryptographic Framework. In *BigAdmin System Administration Portal*, 2005.

[9] L. Spracklen and S. Abraham. Chip Multithreading: Opportunities and Challenges. In *Proceedings of the* 11^{th} *Int'l Symposium on High-Performance Computer Architecture (HPCA-11)*, 2005.

[10] Sun Microsystems, Inc. SunSSH, http://www.opensolaris.org/os/community/security/projects/SSH.