# USING THE CRYPTOGRAPHIC ACCELERATOR OF THE ULTRASPARC® T1 PROCESSOR

Ning Sun, Performance, Availability, and Architecture Tools Engineering

Pallab Bhattacharya, Performance, Availability, and Architecture Tools Engineering

Sun BluePrints™ OnLine—March 2006

Part No 819-5782-10
Revision 1.0, 3/2/06
Edition: March 2006

© 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, CA 95054 USA

All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California.

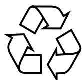
Sun, Sun Microsystems, the Sun logo, Sun BluePrints, SunSolve, SunSolve Online, docs.sun.com, Java, UltraSPARC, Sun Fire, and Solaris are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries.

UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a). DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS HELD TO BE LEGALLY INVALID.



Please
Recycle

# Table of Contents

Cryptography and the Secure Sockets Layer . . . . . 1
UltraSPARC T1 Processor . . . . . 2
Introduction to the Secure Sockets Layer Protocol . . . . . 3
Solaris Cryptographic Framework Overview . . . . . 4
  Terminology . . . . . 4
  Solaris Cryptographic Framework Components . . . . . 5
  Administration Utilities . . . . . 6
  Provider Selection Policy . . . . . 6
UltraSPARC T1 Cryptographic Provider . . . . . 7
  General Metaslot Configuration . . . . . 7
  General NCP Configuration . . . . . 8
  Confirming NCP Operation . . . . . 9
Using NCP with Apache . . . . . 9
Using NCP with the Sun Java™ System Web Server 6.1 Software . . . . . 11
Using NCP with Java Technology Applications . . . . . 13
Kernel SSL Proxy and NCP . . . . . 15
  How the Solaris Kernel SSL Proxy Works . . . . . 15
  The UltraSPARC T1 Processor and KSSL Advantage . . . . . 17
  Configuration . . . . . 17
Performance . . . . . 19
  RSA Performance . . . . . 19
  HTTPS Throughput on the Sun Fire T2000 Server . . . . . 21
  SPECweb2005 World Record . . . . . 22
About the Author . . . . . 23
Acknowledgments . . . . . 23
References . . . . . 24
Related References . . . . . 24
Ordering Sun Documents . . . . . 24
Accessing Sun Documentation Online . . . . . 24
Appendix A—Introduction to Cryptography and the Secure Sockets Layer . . . . . 25
  Cryptography . . . . . 25
  Secure Sockets Layer . . . . . 25

# Using the Cryptographic Accelerator of the UltraSPARC® T1 Processor

Businesses in every industry are concerned about secure communications and data privacy. Typically, these tasks are accomplished through the utilization of the Secure Sockets Layer (SSL). Unfortunately, SSL processing is compute-intensive and can create performance bottlenecks for a variety of commercial workloads. To address these concerns, organizations can take advantage of several Sun technologies that work together to mitigate the performance bottlenecks associated with SSL encryption and decryption. The Solaris Cryptographic Framework (SCF) provides cryptographic services for kernel-level and user-level consumers, as well as several software encryption modules. Based on the SCF, a new SSL proxy (KSSL) kernel module offloads SSL processing from user applications, enabling them to transparently take advantage of powerful hardware accelerators, like those available in Sun's new UltraSPARC® T1 processor, that speed up SSL processing.

This Sun BluePrints™ article demonstrates how the combination of the Solaris™ 10 Operating System (Solaris 10 OS) and the UltraSPARC T1 processor can be used to create a high performance, secure Web site. It provides a brief overview of SSL technology, as well as an introduction to the Solaris Cryptographic Framework. Configuration details are included for common security applications, such as Apache, the Sun Java™ System Web Server, and secure Java™ technology applications, enabling these programs to utilize NCP and KSSL technology. A performance study of secure Web applications is also included.

## Cryptography and the Secure Sockets Layer

Cryptography is the study of mathematical techniques focused on information security, including confidentiality, data integrity, entity authentication, and data origin authentication. A set of compute-intensive mathematical algorithms typically comprise a cryptography implementation, and can be used by applications when encrypting, decrypting, and hashing data. Some implementations also include authentication and verification techniques.

One well-known application of cryptography technology is the implementation of the Secure Sockets Layer (SSL) protocol. Originally developed by Netscape Communications, SSL is a set of rules governing authentication and encrypted communication between servers and clients. As the growth of secure Web site deployment has risen rapidly in recent years, the SSL protocol has emerged as the *de facto* standard for secure electronic commerce (e-commerce). Indeed, the SSL protocol is now built into all popular Web browsers. Due to the compute-intensive nature of SSL technology, it is anticipated that many of these sites struggle as demand rises, as a large volume of SSL traffic can impact the performance of even the most powerful, general purpose Web servers.

## UltraSPARC T1 Processor

Traditional system designs focus on speeding a single thread of execution, with most processors providing a combination of two threads per core, and two cores per chip. Sun's Throughput Computing initiative represents a paradigm shift that aims to maximize the overall throughput of key commercial workloads. Chip multi-threading (CMT) processor technology is key to this approach. The UltraSPARC T1 processor combines chip multiprocessing (CMP) and hardware multithreading (MT) with an efficient instruction pipeline to enable chip multithreading. The resulting processor design provides multiple physical-instruction execution pipelines and several active thread contexts per pipeline. Indeed, the UltraSPARC T1 processor raises the limits set by traditional designs, providing four threads per core, and eight cores per chip with the ability to execute eight instructions in parallel. With 32 threads per chip, the UltraSPARC T1 processor takes thread-level parallelism to a new level.

Traditional data center applications, like Web servers and on-line transaction processing systems, move vast quantities of data in to, and out of, a server. As a result, simply running these applications on systems with fast CPUs and massive instruction level parallelism is not enough to help application performance. Today, memory speed continues to lag behind CPU speed. Consequently, traditional system designs often waste approximately 75 percent of a CPU cycle waiting for a memory transaction to complete. During this time, the execution unit can be kept busy if another thread is ready to execute. This idea is at the heart of the UltraSPARC T1 processor design. With four threads per core, the core can be kept busy approximately 85 percent of time when running a traditional enterprise workload.

Figure 1 illustrates the design of the UltraSPARC T1 processor. In this design, each core contains an 8 KB data cache and a 16 KB instruction cache. A four bank, 3 MB unified L2 cache is shared by the eight cores. Each core contains a single pipeline, and a mechanism to switch between the four threads of a core such that a new thread is scheduled on the pipeline at each clock cycle in a round robin manner. Four DDR2 channels provide a maximum memory configuration of 32 GB. A single Floating Point Unit (FPU) is shared by the eight cores. The eight cores, L2-cache, FPU, and memory controllers are connected via an on-chip crossbar interconnect. To help speed the Rivest Shamir Adleman (RSA) and Digital Signal Algorithm (DSA) operations needed for SSL processing, each core contains a modular arithmetic unit (MAU) that supports modular exponentiation and multiplication.

Note that the UltraSPARC T1 processor is SPARC Version 9 compliant, and can run SPARC Version 7, 8, and 9 binaries without source code modification or recompilation.

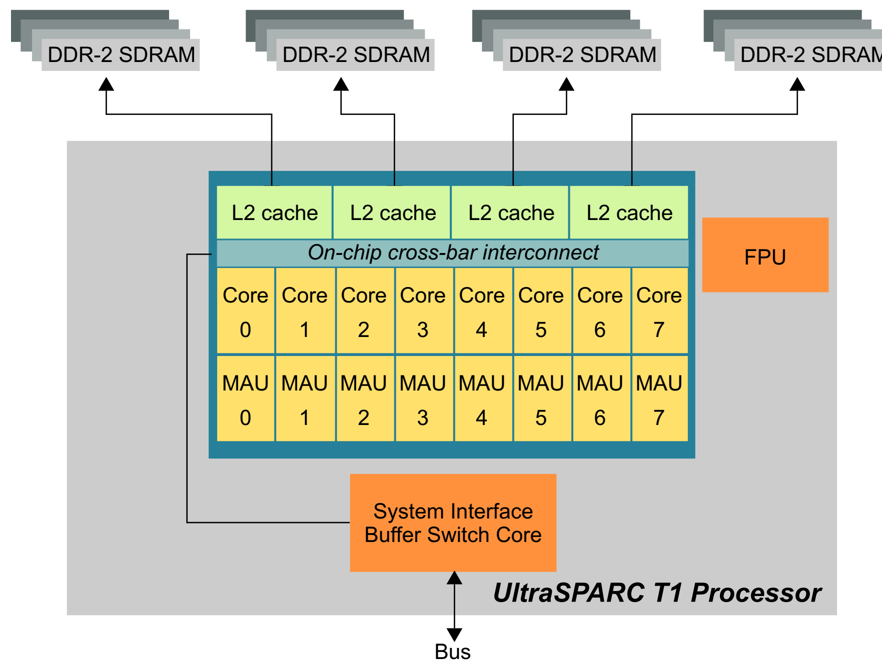



*Figure 1. UltraSPARC T1 processor design*

The eight MAUs, one for each core, are the heart of the processor's hardware cryptographic capabilities. The Niagara Crypto Provider (NCP) device driver in the Solaris 10 OS harnesses the power of these MAUs.

NCP supports hardware assisted acceleration of RSA and DSA cryptographic operations. NCP is enabled in the Solaris Cryptographic Framework, a set of cryptographic services for kernel-level and user-level consumers. The framework also includes several software encryption modules, and gives users the ability to specify the software encryption library or hardware encryption sources an application can use, such as NCP. Note that the NCP device driver is preconfigured in the Solaris Cryptographic Framework on Sun systems with UltraSPARC T1 processors.

## Introduction to the Secure Sockets Layer Protocol

The Secure Sockets Layer (SSL) protocol relies on a RSA public key cryptographic system for authenticating and securing key exchanges. Each key consists of two parts, called the *public* and *private* keys. Web sites make public keys available to clients, while private keys are known only to servers. During SSL transactions, clients encrypt messages using the server's public key. The server decrypts the message using the private key associated with that public key. Typically, a private key is 1,024 bits long.

The SSL protocol consists of two phases. In the first phase, a client and server agree on a common set of symmetric encryption algorithms and keys to use for security and authentication. In the second phase, these agreed upon mechanisms are used during the actual communication of data messages. Typically, clients, such as Web browsers, initiate the SSL handshake (first phase) after a TCP connection is

established. This phase involves a CPU-intensive mathematical operation by the server to recover a secret message shared with the client. Both the client and server perform further operations to derive the encryption, decryption, and message authentication keys from the shared secret.

When a large number of secure clients connect to a secure server, the decrypting of the shared secret during the handshake phase consumes a significant portion of available CPU cycles. If the decryption of the shared secret can be offloaded to a specialized device, such as a hardware cryptographic accelerator (commonly known as SSL accelerator cards or SSL co-processors), the server's CPU resources can focus on handling the business logic of running applications. Another technique, used by SSL proxy appliances and SSL proxy switches, terminates the SSL connection outside the server. This technique enables these appliances to assume the responsibility of cryptographic operations, enabling the server to spend valuable CPUs cycles on the business logic of applications.

Sun systems with UltraSPARC T1 processors do not require the use of additional SSL cards or co-processors. Furthermore, the Solaris 10 OS provides in-kernel SSL termination, a technique that offers greater security than SSl termination outside the server. The remainder of this article describes the security features available in the Solaris 10 OS and how they can be used to build highly scalable, secure, and robust Web sites.

## Solaris Cryptographic Framework Overview

A consistent framework for application-level and kernel-level cryptographic operations, the Solaris Cryptographic Framework can help increase security and performance while giving applications access to the same hardware accelerators used by the operating system kernel. Based on the PKCS#11 public key cryptography standard created by RSA Security, Inc., the Solaris Cryptographic Framework provides a mechanism and API whereby both kernel- and user-based cryptographic functions can be executed by the same optimized encryption software or transparently use hardware accelerators configured on the system. This framework brings the power of advanced, streamlined encryption algorithms and hardware acceleration to user-level C and Java™ programming language-based applications.

### Terminology

Cryptographic services utilize the following terms:

- *Consumer*, an application, library, or kernel module that uses or calls cryptographic services.
- *Provider*, an application, library, or kernel modules that provides cryptographic services to consumers through the framework.
- *Mechanism*, an entity that implements a cryptographic operation based on an algorithm. Multiple mechanisms may be based on the same algorithm, such that the same algorithm can be used for authentication in one mechanism and for encryption in another.
- *Device*, a hardware or software functional unit that can perform cryptographic operations.
- *Token*, a collection of mechanisms in a record format. A token represents the device in abstract form. Tokens that represent pure software are referred to as *soft tokens*.

- *Slot*, the connecting point for applications to use cryptographic services. A token is plugged into a slot.
- *Key*, a parameter used by an algorithm implementation to produce a specific ciphertext from plaintext using encryption. The number of bits in a key determine the strength of the encryption.
- *Symmetric key cryptography*, a category of encryption in which the same key is used for encryption and decryption. DES, RC4, and AES are examples of symmetric key cryptography.
- *Asymmetric key cryptography*, a type of encryption that uses a pair of keys (a public key, and a private key) for encryption and decryption. One key is used for encryption, and the other for decryption. RSA and DSA are examples of asymmetric key cryptography.
- *Certificate*, a collection of identifying information bound together with a public key and signed by a trusted third-party to prove its authenticity. Certificates and private key objects are stored in the token.
- *Keystore*, a special type of cryptographic device capable of persistent storage for token objects.

See Appendix A for more information about cryptography and the SSL protocol.

**Solaris Cryptographic Framework Components**

Figure 2 describes the Solaris Cryptographic Framework. Consisting of a user-level framework, and a kernel-level framework, the SCF has the following components:

- *libpkcs11.so*, the interface user applications can link to in order to call functions in the user-level framework.
- *Pluggable interface*, a PKCS#11-based interface that enables user-level providers to be plugged into the user-level framework.
- *pkcs11_softtoken.so*, user-level cryptographic mechanisms provided by the Solaris OS.
- *pkcs11_softtoken_extra.so*, a user-level library that is identical to *pkcs11_softtoken.so*, but supports stronger keys with bigger key sizes.
- *$HOME/.sunw/pkcs11_softtoken*, the default file system location of the per-user keystore in the Solaris OS.
- *pkcs11_kernel.so*, provides the hardware accelerated algorithms in the kernel-level framework to the user-level framework. Note that it does not contain any cryptography, and does not expose the software-only kernel implementation to the user-level framework.
- *Scheduler and load balancer*, the kernel software responsible for coordinating use, load balancing, and dispatching of the kernel cryptographic service requests.
- *Kernel programmer interface*, the interface for kernel-level consumers of cryptographic services. IPSec and Kernel SSL (KSSL) are example consumers.
- *Service provider interface (SPI)*, an interface that enables kernel providers to be plugged into the kernel-level framework. This includes hardware- or software-based cryptographic services.

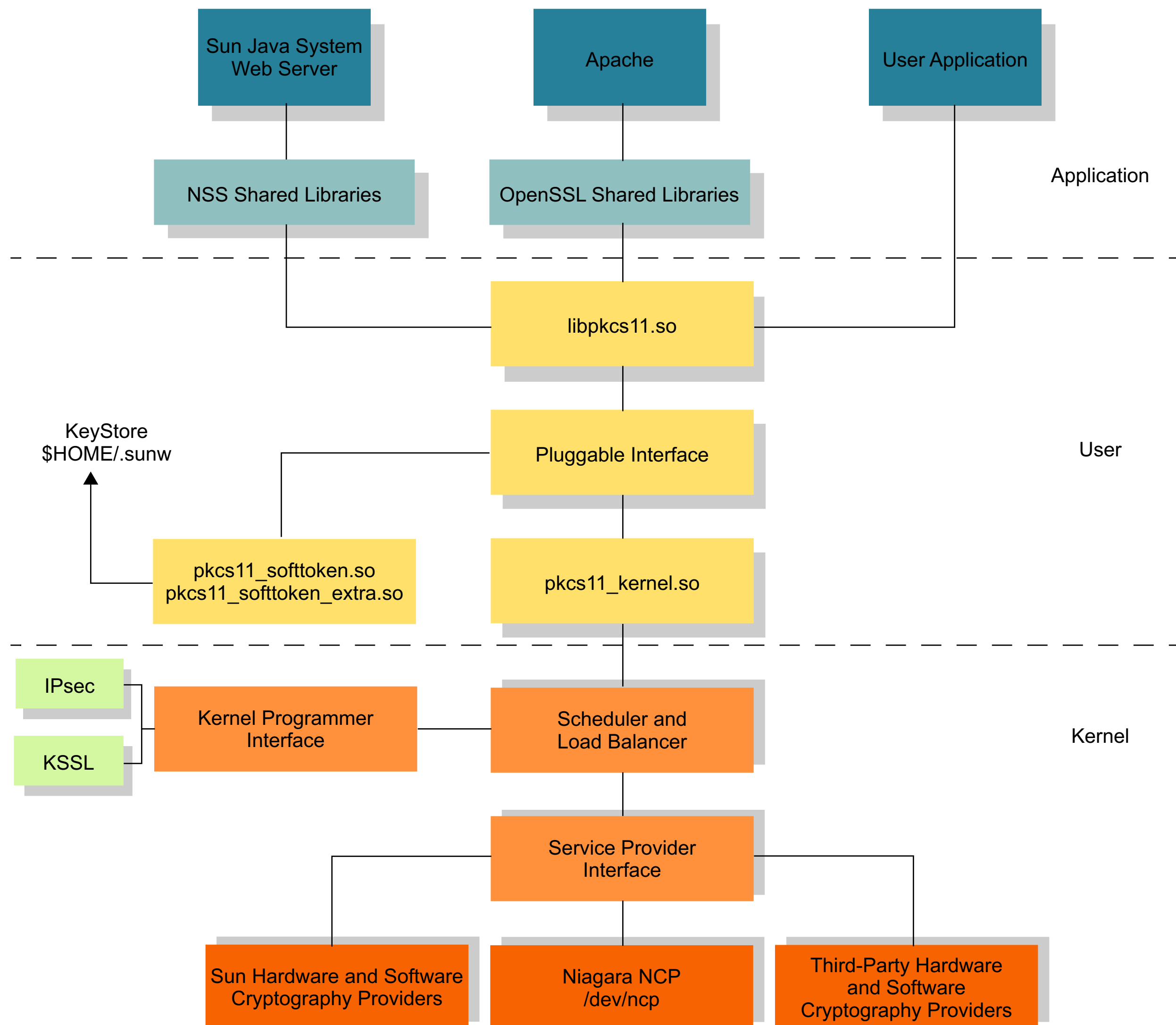



*Figure 2. An overview of the Solaris Cryptographic Framework*

### Administration Utilities

The Solaris Cryptographic Framework provides two administration utilities:

- `cryptoadm(1M)`, a tool that provides administration for both the user-level and kernel-level frameworks. This tool can be used to install hardware and software providers, enable and disable mechanisms for providers, and display cryptographic provider information.
- `pktool(1)`, a tool for managing softtoken objects stored in the keystore. The `pktool setpin` command changes the passphrase (password) used to authenticate a user to the keystore.

### Provider Selection Policy

A cryptographic provider can be plugged into the user-level or kernel-level framework if it implements the pluggable cryptographic provider interface based on PKCS#11. Since more than one provider is generally

available, the user-level framework exposes a *metaslot* to applications. This metaslot is an abstract slot which acts as the single connection point for the mechanisms provided by all cryptographic providers inserted into the user-level framework. A provider can be inserted into the metaslot with the following command:

```
# cryptoadm install provider=providername
```

A mechanism can have multiple providers. In this case, the order in which providers are inserted into the metaslot determines how lists are searched when a particular mechanism is requested. For example, if *pkcs11_kernel.so* appears before *pkcs11_softtoken.so* in the list, a provider from the kernel-level framework (if available) is chosen over the softtoken provider (*pkcs11_softtoken*) in the user-level framework. This is the default operation in the Solaris OS. To ensure a particular mechanism from a provider is chosen, delete the mechanism from all preceding providers in the metaslot list. The following command can be used to set the metaslot preference for token adoption per mechanism.

```
# cryptoadm enable metaslot [mechanisms=mechanism-list] [token=token-label]
```

## UltraSPARC T1 Cryptographic Provider

RSA operation is an important component of the SSL full handshake. Each core of the UltraSparc T1 processor has a Modular Arithmetic Unit (MAU) which supports RSA and DSA operations. RSA operations utilize a compute-intensive algorithm that can be offloaded to the MAU. Indeed, the MAU is capable of sustaining 14,000 RSA operations per second. Moving RSA operations to the MAU speeds SSL full handshake performance and frees the CPU to handle business computation. For example, we have observed a 20 percent to 40 percent improvement in overall throughput in an environment performing approximately 9 percent full handshakes when RSA operations are offloaded to the MAU.

In the context of the Solaris Cryptographic Framework, the MAU is implemented as a Service Provider (`ncp(7D)`), and all eight units are visible as a single device (`/dev/ncp0`) to consumers. Because this device implementation continues to process requests as long as one MAU remains functional, it is highly available.

NCP provides RSA and DSA services, and is preconfigured in the Solaris OS and made available to applications. The following sections describe how to use the administrative tools of the Solaris Cryptographic Framework to enable and use NCP.

### General Metaslot Configuration

The metaslot is easily configured using the following steps.

1. The *$HOME/.sunw* directory is created as a result of this process. If the application owner does not have a login (such as `nobody`), set the `SOFTTOKEN_DIR` environment variable to a directory with read and write permission by the application owner before proceeding.

```
SOFTTOKEN_DIR=path; export SOFTTOKEN_DIR
```

2. Initialize the user's default keystore by logging in to the system as the application owner.

```
# pktool setpin
Enter token PIN:
Enter new PIN:
Re-enter new PIN:
```

### General NCP Configuration

1. Verify the `ncp` and `pkcs11_softtoken.so` are available as cryptographic providers by using the `cryptoadm` command. Be sure to run the command as the `root` user. Because NCP is a kernel provider, `pkcs11_kernel.so` should appear before `pkcs11_softtoken.so` in the output. This ordering ensures the framework checks the availability of kernel providers before requesting the `softtoken` implementations.

```
# cryptoadm list -p

User-level providers:
=====================
/usr/lib/security/$ISA/pkcs11_kernel.so: all mechanisms are enabled. random is enabled.
/usr/lib/security/$ISA/pkcs11_softtoken.so: all mechanisms are enabled. random is enabled.

Kernel software providers:
==========================
des: all mechanisms are enabled.
aes256: all mechanisms are enabled.
arcfour2048: all mechanisms are enabled.
blowfish448: all mechanisms are enabled.
sha1: all mechanisms are enabled.
md5: all mechanisms are enabled.
rsa: all mechanisms are enabled.
swrand: random is enabled.

Kernel hardware providers:
==========================
ncp/0: all mechanisms are enabled.
```

2. If stronger encryption (greater than 1,024 bit encryption) is desired, install `pkcs11_softtoken_extra.so` using the `cryptoadm` command. Note the `SUNWcry` package must be installed to make `pkcs11_softtoken_extra.so` available on the system.

```
# cryptoadm uninstall provider=/usr/lib/security/\$ISA/pkcs11_softtoken.so
# cryptoadm install provider=/usr/lib/security/\$ISA/pkcs11_softtoken_extra.so
```

3. Make sure the following mechanisms are disabled, or disable them in the metaslot.

```
# cryptoadm disable provider=/usr/lib/security/\$ISA/pkcs11_softtoken.so \
    mechanism=CKM_SSL3_PRE_MASTER_KEY_GEN, \
    CKM_SSL3_MASTER_KEY_DERIVE, \
    CKM_SSL3_KEY_AND_MAC_DERIVE, \
    CKM_SSL3_MASTER_KEY_DERIVE_DH, \
    CKM_SSL3_MD5_MAC,CKM_SSL3_SHA1_MAC
```

### Confirming NCP Operation

The next few sections describe how to configure applications to run with NCP, including Apache, the Sun Java System Web Server, and Java technology applications. Once the application is operational using the Solaris Cryptographic Framework, a set of `kstat` commands can be used to confirm NCP is involved in the operation. The `kstat` command below displays the number of RSA public key decryptions performed using NCP since the last system boot. A positive and growing value indicates NCP is operational.

```
# kstat -n ncp0 -s rsaprivate
```

## Using NCP with Apache

Standard OpenSSL libraries are not built to support the Solaris Cryptographic Framework. However, the default installation of the Solaris 10 OS provides a PKCS#11-based OpenSSL implementation. The Solaris 10 OS includes the following OpenSSL libraries:

- */usr/sfw/lib/libcrypto.so* and */usr/sfw/lib/libssl.so* (for 32-bit applications)
- */usr/sfw/lib/sparcv9/libcrypto.so* and */usr/sfw/lib/sparcv9/libssl.so* (for 64-bit applications)

The *libcrypto.so* shared library calls the *libpkcs11.so* library to bridge OpenSSL-based cryptographic applications with NCP. To take advantage of NCP, applications must use, or be linked to use, the set of OpenSSL libraries delivered with the Solaris 10 OS in the */usr/sfw* directory. The standard OpenSSL library available from *http://www.openssl.org* cannot take advantage of NCP technology.

The following `openssl` command checks the PKCS#11 capability of installed OpenSSL libraries.

```
# /usr/sfw/bin/openssl engine pkcs11
(pkcs11) PKCS #11 engine support
```

More detailed information can be found by using the `openssl` command with the `-c` and `-t` options.

```
# /usr/sfw/bin/openssl engine -c -t
(pkcs11) PKCS #11 engine support
[RSA, DSA, DH, RAND, DES-CBC, DES-EDE3-CBC, AES-128-CBC,RC4, MD5, SHA1]
[available]
```

The following command runs an `openssl` speed test for RSA operations performed by the NCP.

```
# /usr/sfw/bin/openssl speed -engine pkcs11 rsa
```

The Apache Web Server software should be deployed with the ability to use these OpenSSL libraries. The following options must be supplied to the `configure` command, in addition to standard options, in order to compile and build the Apache Web Server software and enable these capabilities.

```
# CFLAGS='-DSSL_EXPERIMENTAL -DSSL_ENGINE' ./configure -enable-ssl --enable-rule=SSL_EXPERIMENTAL \
--with-ssl=/usr/sfw
```

If the Apache Web Server software was built to use `mod_ssl.so` but the above flags were not used, the `mod_ssl.so` library must be recompiled to include `SSLCryptoDevice`. The following options should be used when recompiling the `mod_ssl.so` library. More information on installing and configuring the Apache software is available at *http://apache.org*

```
# CFLAGS='-DSSL_EXPERIMENTAL -DSSL_ENGINE' ./configure -with-apxs=/usr/local/apache2/bin/apxs \
--enable-rule=SSL_EXPERIMENTAL -with-ssl=/usr/sfw
```

Once the software is built and installed, a private key and certificate must be installed for the Apache Web Server software. The following steps outline the process for installing a key and certificate.

1. Generate an RSA private key in Privacy-Enhanced Mail (PEM) format. The key is placed in the */etc/apache/ssl.key/server.key* file.

   ```
   # /usr/sfw/bin/openssl genrsa -des3 -out /etc/apache/ssl.key/server.key 1024
   ```

2. Generate the certificate request using the key generated.

   ```
   # /usr/sfw/bin/openssl req -new -key /etc/apache/ssl.key/server.key -out \
     /etc/apache/ssl.csr/certreq.csr
   ```

3. Send the certificate (*certreq.csr*) to a certificate authority, such as Verisign, to get the certificate signed.
4. Save the signed certificate in the */etc/apache/ssl.csr/server.crt* file.
5. Edit the *httpd.conf* file and add (or replace) the following lines.

   ```
   SSLCertificateFile /etc/apache/ssl.csr/server.crt
   SSLCertificateKeyFile /etc/apache/ssl.key/server.key
   ```

6. Edit the *ssl.conf* file and add (or replace) the following lines.

   ```
   SSLCryptoDevice pkcs11
   ```

7. Start the server using the `apachectl` command

```
# /usr/apache/bin/apachectl startssl
```

8. Send a request from a Web browser and check to see the software is working.

The following should be considered when configuring the Apache Web Server software:

- Use of the Apache Web Server software version 2.0.55 is recommended for Sun systems with UltraSPARC T1 processors.
- The Apache process should not run as the `root` user as a security precaution. Performance may suffer if the software is run as the `root` user and the `pkcs11` engine is enabled.

## Using NCP with the Sun Java™ System Web Server 6.1 Software

The cryptographic module in the Sun Java System Web Server software is PKCS#11 compliant. As a result, the software can plug into any PKCS#11 compliant cryptographic provider, such as the Solaris Cryptographic Framework. The following steps describe how to configure the Sun Java System Web Server software to use the NCP. Note that the Sun Java System Web Server software maintains a database called *your_webserver_instance/../alias/secmod.db* to hold configuration and registry information about security modules.

1. Set the environment variables for the Sun Java System Web Server software.

```
# cd webserver_instance
# ./start -shell
# cd webserver_instance/../alias
# PATH=webserver_instance/../bin/https/admin/bin:$PATH;export PATH
```

2. Display the list of modules registered with the Sun Java System Web Server software.

```
# modutil -dbdir . -nocertdb -list

Using database directory ....

Listing of PKCS #11 Modules
-----------------------------------------------------------
  1. NSS Internal PKCS #11 Module
         slots: 2 slots attached
        status: loaded

         slot: NSS Internal Cryptographic Services
        token: NSS Generic Crypto Services

         slot: NSS User Private Key and Certificate Services
        token: NSS Certificate DB

  2. Root Certs
        library name:

webserver_instance/../bin/https/lib/libnssckbi.so

         slots: 1 slot attached
        status: loaded

         slot:
        token: Builtin Object Token
-----------------------------------------------------------
```

3. Register the */usr/lib/libpkcs11.so* PKCS11 library with the Sun Java System Web Server software, and enable the slot named `Sun Metaslot`. If a 64-bit version of the Sun Java System Web Server software is used, replace */usr/lib/libpkcs11.so* with */usr/lib/sparcv9/libpkcs11.so* in the command.

```
# modutil -dbdir . -nocertdb -add "Solaris Cryptographic Framework" \
  -libfile /usr/lib/libpkcs11.so -mechanisms RSA
```

4. Enable the slot named `Sun Metaslot`.

```
# modutil -dbdir . -nocertdb -disable "Solaris Cryptographic Framework"
# modutil -dbdir . -nocertdb -enable "Solaris Cryptographic Framework" -slot "Sun Metaslot"
```

5. Verify the slot named `Sun Metaslot` was added correctly.

```
# modutil -dbdir . -nocertdb -list

Using database directory ....

Listing of PKCS #11 Modules
-----------------------------------------------------------
  1. NSS Internal PKCS #11 Module
         slots: 2 slots attached
        status: loaded

         slot: NSS Internal Cryptographic Services
        token: NSS Generic Crypto Services

         slot: NSS User Private Key and Certificate Services
        token: NSS Certificate DB

  2. Solaris Cryptographic Framework
        library name: /usr/lib/libpkcs11.so
         slots: 2 slots attached
        status: loaded

         slot: Sun Metaslot
        token: Sun Metaslot

         slot: ncp/0 Crypto Accel Asym 1.0
        token: ncp/0 Crypto Accel Asym 1.0

  3. Root Certs
        library name:

your_webserver_instance/../bin/https/lib/libnssckbi.so

         slots: 1 slot attached
        status: loaded

         slot:
        token: Builtin Object Token
-----------------------------------------------------------
```

6. Generate the key and certificate request and get the certificate signed by a Certificate Authority and installed for the `internal` token. If this token was configured previously, skip this step and proceed to step 7 below. See *http://docs.sun.com/source/819-0130-10/agcert.html* for information on configuring the `internal` token.
7. The Sun Java System Web Server software is bundled with the `pk12util` utility. This utility enables users to export certificates and keys from the internal store and import them into an external PKCS#11 based cryptographic service provider.
8. Export the certificate and key from the `internal` store into a PKCS#12 formatted file.

```
# pk12util -o certpk12  -d . -n Server-Cert  -P  your_webserver_instance
Enter Password or Pin for "internal": password_used_in_setpin
Enter password for PKCS12 file: new_password_for_your_cert_file
```

9. Import the key and certificate in the `Sun Metaslot`.

```
# pk12util -i certpk12 -d . -h "Sun Metaslot"
Enter Password or Pin for "Sun Metaslot": password_used_in_setpin
Enter password for PKCS12 file: new_password_for_your_cert_file
```

10. Verify the certificate and key were successfully imported.

```
# certutil -L -d . -h "Sun Metaslot"
Enter Password or Pin for "Sun Metaslot":
Sun Metaslot:Server-Cert                                     u,u,u
```

11. Configure the Sun Java System Web Server instance to use the metaslot by prepending the original token name with `"Sun Metaslot:"`. For example, replace `"Server-Cert"` with `"Sun Metaslot:Server-Cert"` in the *server.xml* file for `"servercertnickname"`.
12. Finally, restart the Sun Java System Web Server software. Be sure to specify the password for `Sun Metaslot`, which should be the password specified in the `setpin` command earlier in the process. If the `SOFTTOKEN_DIR` environment variable was used earlier in this procedure, be sure to set it to the same value before starting the server.

## Using NCP with Java Technology Applications

The first release of the Security API in the Java™ 2 Platform Standard Edition 1.1 Developer Kit (JDK 1.1) introduced the Java Cryptography Architecture (JCA), a framework for accessing and developing cryptographic functionality for the Java platform. It includes a provider architecture that supports multiple, interoperable cryptography implementations. As part of the Java Secure Socket Extension (JSSE), the Java Cryptography Extension (JCE) extends the JCA API to include APIs for encryption, key exchange, and Message Authentication Code (MAC). The JDK version 1.4 and later includes a JSSE provider named SunJSSE which comes pre-installed and pre-registered with the JCA.

In the Java™ 2 Platform, Standard Edition (J2SE) version 1.5, the SunJSSE provider uses JCE exclusively for all of its cryptographic operations. As a result, the SunJSSE provider can automatically take advantage of JCE features and enhancements, including new support for PKCS#11 via the Sun PKCS#11 provider. This enables the SunJSSE provider in J2SE 1.5 to use hardware cryptographic accelerators for significant performance improvements. Therefore, the use of hardware cryptographic accelerators is automatic if:

- JCE is configured to use the Sun PKCS#11 provider
- The Sun PKCS#11 provider is configured to use the underlying accelerator hardware

Unlike many other providers, the Sun PKCS#11 provider does not implement cryptographic algorithms. It is simply a bridge between the Java JCA and JCE APIs, and the native PKCS#11 cryptographic API, translating calls and conventions between the APIs. This means that Java technology applications that call standard JCA and JCE APIs can take advantage of algorithms provided by underlying PKCS#11 implementations without modification.

On Sun systems with UltraSPARC T1 processors, the default Java Virtual Machine (JVM) and J2SE 1.5 components are preconfigured to use the SunPKCS#11 Provider. The mapping is set up by the following line in the *java-home/jre/lib/security/java.security* file.

```
#
# List of providers and their preference orders (see above):
#
security.provider.1=sun.security.pkcs11.SunPKCS11 \
${java.home}/lib/security/sunpkcs11-solaris.cfg
security.provider.2=sun.security.provider.Sun
security.provider.3=sun.security.rsa.SunRsaSign
<...> (other providers skipped)
```

The *java-home/jre/lib/security/sunpkcs11-solaris.cfg* file configures the SUn PKCS#11 Provider to utilize the Solaris Cryptographic Framework. Mechanisms to be handled by applications can be disabled from the SUn PKCS#11 Provider using this file. Sample file content follows.

```
# Configuration file to allow the SunPKCS11 provider to utilize
# the Solaris Cryptographic Framework, if it is available
name = Solaris
description = SunPKCS11 accessing Solaris Cryptographic Framework
library = /usr/lib/$ISA/libpkcs11.so
handleStartupErrors = ignoreAll
attributes = compatibility
disabledMechanisms = {
  CKM_MD2
  CKM_MD5
  CKM_SHA_1
  CKM_SHA256
  CKM_SHA384
  CKM_SHA512
  CKM_DSA_KEY_PAIR_GEN
  CKM_RSA_PKCS_KEY_PAIR_GEN
  CKM_MD5_RSA_PKCS
  CKM_SHA1_RSA_PKCS
  CKM_DH_PKCS_KEY_PAIR_GEN
  CKM_DH_PKCS_DERIVE
    <...> (other mechanisms disabled)
  }
```

## Kernel SSL Proxy and NCP

The Kernel SSL proxy implements the SSL protocol such that a non-SSL application server is able to handle SSL-based client requests. All SSL processing is performed in the kernel. As a result, server programs need only send and receive data in clear text. Figure 3 provides an overview of the Kernel SSL Proxy.

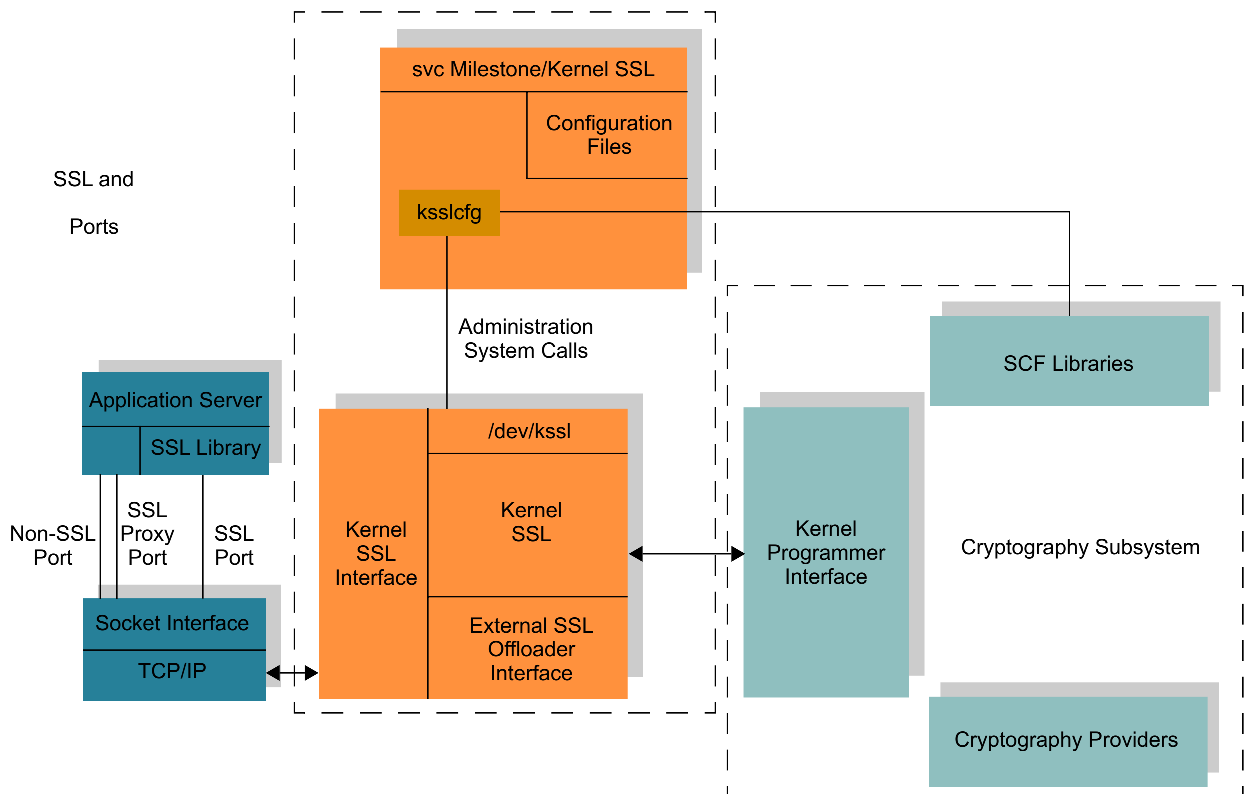



*Figure 3. The Kernel SSL Proxy*

**How the Solaris Kernel SSL Proxy Works**

The Solaris Kernel SSL (KSSL) proxy implementation adds a kernel module (`kssl`) which is responsible for the server-side SSL protocol. This module acts as the SSL proxy server, and is responsible for providing a proxy port to the application server and listening on the SSL port. It also manages keys and certificates, manages SSL session state information, and is responsible for the SSL handshake with clients. The SSL handshake, or SSL alert, is handled asynchronously without the application server's knowledge or involvement. However, the encryption, decryption and message digest is still performed in the context of the application server. The kernel SSL proxy does not implement the full set of cipher suites defined by the Secure Sockets Layer/Transport Layer Security (SSL/TLS) protocol.

For the SSL Proxy module to be active on a given SSL port, such as port 443, the application server must listen on a proxy port, such as port 8080. Three listeners can be configured in the application server:

- One listening on a regular clear text port, such as HTTP port 80.
- One listening on the SSL proxy port through which the `kssl` module sends and receives clear text payloads for secure clients.
- A third port for managing encrypted data at the user level, acting as a user-level SSL server. In other words, the user-level SSL server configuration need not be changed or disabled when configuring

KSSL. In this type of configuration, client connections requesting a cipher suite that is not supported by the KSSL are forwarded to the user-level SSL server as a fallback mechanism. For this mechanism to succeed, the user-level SSL server and KSSL must listen on the same SSL port.

Based on these considerations, three different configurations can be derived:

- One SSL proxy port for clear text communication between the Kernel SSL proxy and the application, and one port for the SSL-protected traffic between the Kernel SSL proxy and remote SSL clients.
- The same configuration as above, with the addition of a fallback listener in the application on the same SSL port for handling the cipher suites not supported by the Kernel SSL proxy. In this case, the application server also behaves as a secure server.
- For application servers that want to offer a mixture of clear text and SSL protected services, an additional listener can be added on a clear text port that is independent from the proxy port. This is a typical configuration for e-commerce deployment.

When configured, the `kssl` module maintains a table of four-tuples, (IP address, SSL port, proxy port, SSL parameters), where the SSL parameters include the private key, server certificate, acceptable cipher suites, and other SSL session related parameters. The user-level `ksslcfg` command configures the `kssl` module by adding and removing entries from this table.

When a user-level application calls a normal `bind()` operation, the kernel inspects the IP address and port specified in the function call. Several outcomes are possible as a result of this inspection:

- If the IP address and port do not match those configured with the `ksslcfg` command, data for this endpoint will not be intercepted or handled by the `kssl` module.
- If the port matches the SSL port configured via the `ksslcfg` command, then the port is used as the fallback port for forwarding all SSL requests that cannot be handled to the user-level SSL server.
- If the port matches the proxy port configured with the `ksslcfg` command, then this binding endpoint acts as the proxy port. The `kssl` module sends and receives clear text data through this endpoint with the application binding to this endpoint. At this time, the `kssl` module starts listening for incoming SSL requests on the SSL port. The kssl module guarantees that the application binding to the SSL proxy endpoint will not be attacked at that endpoint with a clear text message, because the proxy port is not accessible from outside the server.

When a client makes a new SSL connection to the endpoint on which the `kssl` module is listening, the client offers a list of cipher suites that it is willing to negotiate. If at least one the cipher suites can be handled by the `kssl` module, the handshake is completed with the client and the application is notified of an incoming connection at the proxy port. Once the application accepts the new connection, all cryptographic operations on the application payload are handled by the `kssl` module.

When the application performs a `read()` operation, the `kssl` module works in the context of the application to verify the MAC, decrypt the payload, and strip out the SSL header and tail of the incoming record and copy the plaintext payload to the user-buffer supplied as an argument to the `read()` system call. The process is similar for a `write()` operation. The `kssl` module uses the application context to

encrypt and compute the MAC of the outgoing message before actually sending out the encrypted message. In the event no cipher suite matches, the client is notified by the `kssl` module. If the application is also listening on the SSL port endpoint, all connection information is passed to the application as a fallback mechanism for the `kssl` module. At that point, the `kssl` module is no longer interested in the connection.

### The UltraSPARC T1 Processor and KSSL Advantage

Since the `kssl` module is built using the Solaris Encryption Framework kernel APIs, it can take advantage of the NCP module for RSA operation during a handshake. Regular SSL proxy devices terminate the TCP connection on the device and originate a new TCP connection to the application server. If the SSL proxy device is outside the application server but within the same data center, the plain text data moving between the proxy device and the application server hardware can be intercepted, posing a security threat. If the SSL proxy device is an I/O card on the application server, it may not be able to manage a large quantity of SSL sessions due to limited resource availability on the card, such as memory resources.

The combination of Sun systems with UltraSPARC T1 processors,. NCP, and the Solaris OS enable the `kssl` module to take advantage of all the features provided by the operating system and the Solaris Cryptographic Framework. As will be demonstrated later in this article, Sun systems with UltraSPARC T1 processors configured to use `kssl` offer a robust, scalable, and secure application server solution.

### Configuration

The `ksslcfg(1M)` command manages the Service Management Facility (`smf(5)`) instances for the kernel SSL proxy module. It creates and deletes instances of the *svc://network/ssl/proxy* Service Management Facility service. Each instance is unique for a host and SSL port, and specifies the proxy port and SSL parameters to be used by the kernel proxy when the service instance is started. Availability for a given SSL port can be checked with the `svcs` command:

```
# svcs | grep kssl
online         21:08:34 svc:/network/ssl/proxy:kssl-INADDR_ANY-443
```

By default, the `ksslcfg` command creates an instance for SSL port 443, proxy port 8080, and the `INADDR_ANY` IP address. The command assumes the certificate and key file is located in the */etc/kssl/keypair.pem* file (in PEM format), and that passwords for the private key are stored in the */etc/kssl/passphrase* file in clear text.

There are two primary reasons to create a SMF instance:

- Create configuration information for the proxy that is persistent across reboots.
- Provide a service with the Fault Management Resource Identifier (FMRI) that can be started, stopped, and observed by SMF tools, such as the `svcs` and `svcadm` commands, and that can be used by other SMF-enabled services to express dependencies.

If users want to configure the SMF instance manually after each system boot, the service can be disabled using either the `svcadm` or `ksslcfg` command.

```
# svcadm disable svc:/network/ssl/proxy:kssl-INADDR_ANY-443
OR
# ksslcfg delete 443
```

Following is the usage information for the `ksslcfg` command:

```
Usage:
ksslcfg create -f pkcs11 [-d softtoken_directory] -T token_label -C
certificate_subject -x proxy_port [options] [server_address] ssl_port
ksslcfg create -f pkcs12 -i certificate_file -x proxy_port [options]
[server_address] ssl_port
ksslcfg create -f pem -i certificate_file -x proxy_port [options]
[server_address] ssl_port
options are:
        [-c ciphersuites]
        [-p password_file]
        [-t ssl_session_cache_timeout]
        [-u username]
        [-z ssl_session_cache_size]
        [-v]
ksslcfg delete [-v] [server_address] ssl_port
```

The `create` option configures the `kssl` module with a port-pair (SSL port and proxy port). The `delete` option removes a previously configured port-pair. The `ssl_port` is the port to which clients connect, such as HTTPS port 443. The application server must be configured to listen on the `proxy_port`, such as port 8080. If the application server binds to a specific IP address, or a set of IP addresses, then the `ksslcfg` command should be executed with the `server_address` argument, once for each address.

During configuration, the `kssl` module expects a certificate. The certificate can be passed to the `kssl` module during execution of the `ksslcfg` command. The `ksslcfg` command can work with multiple file formats. Note that the `-f` option specifies the file format.

The password for the keypair file is stored in clear text format in a file. This file name is passed as an argument to the `-p` option of the `ksslcfg` command. This option is almost always mandatory when configuring the `kssl` module.

To use the certificate and key from a previously configured softtoken directory, such as *$HOME/.sunw*, use the following ksslcfg command.

```
# ksslcfg create -f pkcs11 [-d softtoken_directory] -T token_label -C  \
   certificate_subject -x proxy_port [options] [server_address] server_port
```

The following steps illustrate how to use a PKCS12 formatted private key and certificate.

1. Create the combined key and certificate file in PKCS12 format. Note that the certificate and key can also be exported from a previously existing store, such as from the store managed by the Sun Java System Web Server software, using the `pk12util` command as described earlier in this article.

```
# /usr/sfw/bin/openssl pkcs12 -export -inkey path_to_private_key_pem_file -in \
   path_to_certificate_pem_file -out keypair_file_in_pk12_format
```

2. Run the `ksslcfg` command.

```
# ksslcfg create  -f pkcs12  -i keypairfile_in_pk12_format -x proxy_port \
               [options] [server_address] server_port
```

The following steps illustrate how to use a key and certificate PEM formatted file.

1. Concatenate the private key and certificate in a single file.

```
# cat private_key.pem certificate.pem > keypair.pem
```

2. Run the `ksslcfg` command.

```
# ksslcfg create -f pem -i keypair.pem -x proxy_port [options] \
   [server_address] server_port
```

## Performance

The following sections demonstrate how the Sun systems with UltraSPARC T1 processors running the Solaris OS work together to deliver superior SSL performance.

### RSA Performance

The `openssl` speed test utility was used to measure the performance of RSA private key generation on a Sun Fire™ T2000 system. The results were compared with those from the same test performed on a Sun Fire™ V40z server running four 2.2 Ghz AMD Opteron processors, and a workstation with two 3.6 GHz Intel Xeon processors with HT enabled.

The following `openssl` commands were used during the test:

```
openssl speed rsa1024 -elapsed -multi n
openssl speed -engine pkcs11 rsa1024 -elapsed -multi n
```

The first command uses the RSA implementation in `openssl`. The second command routes RSA jobs from `openssl` via the PKCS11 interface to the Solaris Cryptographic Framework, enabling hardware accelerators to be used, if available. The `-multi` *n* option was used to run multiple `openssl` speed tests in parallel. This was done in an effort to obtain the maximum throughput from each system by distributing the load on multiple processors and keeping them occupied as much as possible.

Table 1 summarizes the `openssl` speed tests results, detailing the maximum RSA (1024-bit) throughput delivered by each system under test. Using built-in cryptographic capabilities, a single UltraSPARC T1 processor can perform over 13,000 RSA operations per second, more than seven times that of two Intel Xeon 3.6 GHz HT processors.

*Table 1. RSA throughput delivery results*

| System | Operating System | PKCS11-SCF | RSA-1024 Private (Operations/Second) |
|---|---|---|---|
| Sun Fire T2000 Server (Eight 1.2 GHz UltraSPARC T1 Processors) | Solaris OS | Yes — NCP | 13,263.8 |
| Sun Fire V20z Server (Four 2.2 GHz AMD Opteron processors) | Solaris OS | Yes | 3,510.0 |
| Sun Fire V40z Server (Four 2.2 GHz AMD Opteron processors) | Solaris OS | No | 1,672.4 |
| Workstation (Two 3.6 GHz HT Intel Xeon processors) | Linux 2.4.21 (64-bit) | No | 1,780.9 |

On a Sun Fire V20z server configured with four 2.2 GHz AMD Opteron processors, RSA performance with the PKCS11 engine is more than twice that of the `openssl` default algorithm implementation. The reason—with the PKCS11 engine RSA is handled by the Solaris Cryptographic Framework, which includes a highly optimized software implementation for RSA. Figure 4 details the throughput obtained under different numbers of parallel `openssl` speed tests using the `-multi` *n* option.

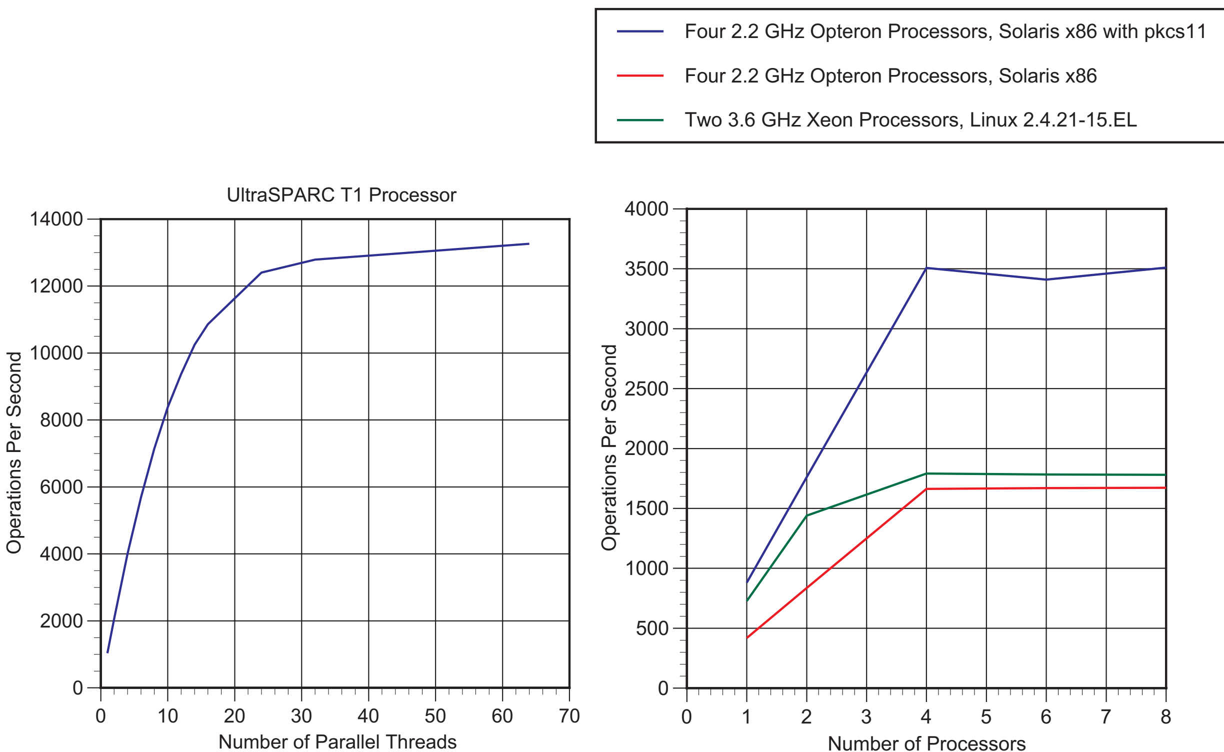


*Figure 4. RSA performance on systems with UltraSPARC T1 processors and AMD Opteron processors*

**HTTPS Throughput on the Sun Fire T2000 Server**

The performance benefits realized from using NCP and KSSL by Web servers when handling HTTPS requested was also studied. A modified SPECweb99_SSL workload (*http://www.spec.org/web99ssl/*) was used to perform entirely static HTTPS requests.

Designed by the Standard Performance Evaluation Corporation, the SPECweb99_SSL is an industry standard benchmark that measures Web server performance. The benchmark was retired in October, 2005. Because it uses a modified workload, the results presented here cannot be compared with other modified or unmodified SPECweb99_SSL benchmarks performed outside the scope of this document.

The Sun Java System Web Server 6.1 SP5 (64-bit) software and Apache Web Server 2.0.55 (compiled in prefork mode) were used in the study. Table 2 summarizes the results. Note that a Sun Fire T2000 Server that uses both KSSL and NCP outperformed the Apache software by 82 percent, and the Sun Java System Web Server 6.1 SP5 (64-bit) doubled its performance than when run without these capabilities. Note that NCP by itself also improves performance.

*Table 2. HTTPS throughput results*

| KSSL | NCP | SJSWS 6.1 SP5 64-bit (Operations/Second) | Apache 2.0.55 Prefork (Operations/Second) |
|---|---|---|---|
| Yes | Yes | 13,558 | 9,406 |
| No | Yes | 9,548 | 5,726 |
| No | No | 6,736 | 5,288 |

**SPECweb2005 World Record**

The latest benchmark available from the Standard Performance Evaluation Corporation (SPEC), the SPECweb2005 benchmark measures Web server performance. The benchmark contains multiple standardized workloads: Banking (HTTPS), E-commerce (HTTP and HTTPS), and Support (HTTP). One component of the benchmark focuses on SSL performance. Two sub-workloads use SSL, with the Banking workload consisting entirely of HTTPS requests. More information on the SPECweb2005 benchmark can be found at *htpp://www.spec.org/osg/web2005/*.

Table 3 and Figure 5 compare the Sun Fire T2000 Server with other systems. Results are based on *http://www.spec.org/osg/web2005/results/web2005.html* as of Jan 26, 2006. Sun Fire T2000 systems using both the kernel SSL proxy and NCP achieved the highest result to date.

*Table 3. SPECweb2005 results and sub-metrics for different systems as of January 26, 2006*

| System | Configuration | Result |
|---|---|---|
| UltraSPARC T1000 Server | One processor with eight cores running at 1.2 GHz | 14,001 |
| IBM p5 550 | Two processors with four cores running at 1.9 GHz SMT | 7,881 |
| Dell PowerEdge 2850 | Two processors with four cores running at 2.8 GHz HT | 4,850 |
| IBM eServer xSeries x346 | Two processors with two cores running at 3.8 GHz HT | 4,348 |

| | | |
|---|---|---|
| IBM eServer xSeries BladeCenter HS20 | Two processors with two cores running at 3.8 GHz HT | 4,177 |
| IBM eServer xSeries x306M | One processor with two cores running at 3.0 GHz | 2,151 |
| Dell PowerEdge 850 | One processor with two cores running at 3.2 GHz | 1,744 |
| Dell PowerEdge 750 | One processor with one core running at 2.8 GHz HT | 848 |

| System | Banking | E-Commerce | Support |
|---|---|---|---|
| UltraSPARC T1000 Server | 21,500 | 21,500 | 13,160 |
| IBM p5 550 | 12,240 | 11,820 | 7,500 |
| Dell PowerEdge 2850 | 8,750 | 6,800 | 4,250 |
| IBM eServer xSeries x346 | 7,950 | 5,150 | 4,450 |
| IBM eServer xSeries BladeCenter HS20 | 7,140 | 4,695 | 4,820 |
| IBM eServer xSeries x306M | 3,870 | 2,640 | 2,160 |
| Dell PowerEdge 80 | 5.050 | 870 | 2,675 |
| Dell PowerEdge 750 | 950 | 850 | 1,675 |

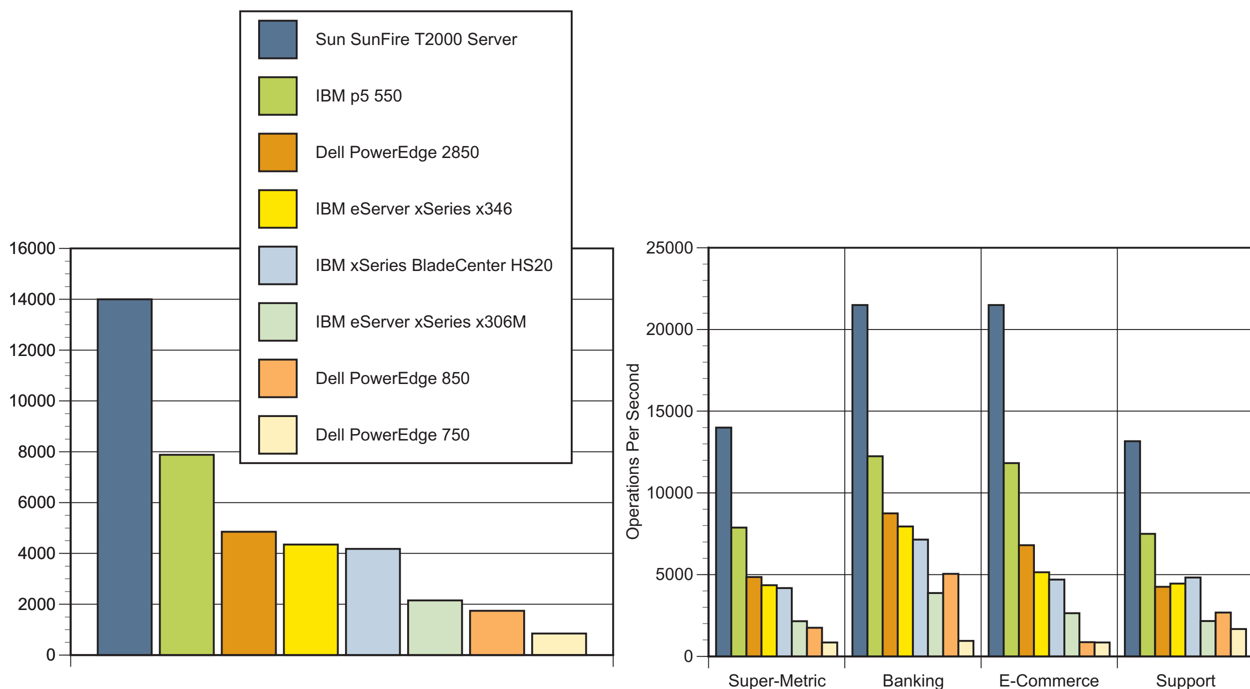



*Figure 5. SPECweb2005 results and sub-metrics for different systems as of January 26, 2006*

## About the Authors

A member of Sun's Performance, Availability, and Architecture Engineering (PAE) group, Ning Sun is currently at work on several Web and application server performance projects. Her technical focus centers on the performance of networks, Java technology, and servlet and EJB containers. Ning has extensive

background in the development of industry standard benchmarks, including SPECweb, SPECjAppServer, and TPC-W.

Pallab Bhattacharya has years of experience in the performance analysis and tuning of applications and application servers, with a focus on servlets and JSP. He has helped shape several industry standard benchmarks, including TPC-W and SPECweb2005, as well as other nonJ2EE related benchmarks. Furthermore, Pallab is a key contributor in the development of Sun Java™ Enterprise System products.

## Acknowledgments

The authors thank the following for their advice and contributions to this article:

- Denis Sheahan, Horizontal Systems Group
- Kais Belgaied, Solaris Security Group
- Karen Tung, Solaris Security Group
- Chi-Chang Lin, Performance, Availability, and Architecture Engineering Group
- Keith Bierman, Performance, Availability, and Architecture Engineering Group

## References

Introduction to Cryptography PGP 6.5.1
`http://www.pgpi.org/doc/pgpintro/`

Security APIs, SPIs, and Frameworks for the Solaris OS
`http://developers.sun.com/solaris/articles/security_apis/security_apis.html`

Solaris Security for Developers Guide
`http://docs.sun.com/app/docs/doc/816-4863/`

Solaris Security for Developers Guide, Chapter 8: Introduction to the Solaris Cryptographic Framework
`http://docs.sun.com/app/docs/doc/816-4863/6mb201vgm`

## Related References

Handbook of Applied Cryptography, Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone

PKCS #11 v2.20: Cryptographic Token Interface Standard, RSA Laboratories, June 28, 2004.
`ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2020/pkcs-11v2-20.pdf`

Standard Performance Evaluation Corporation
`http://www.spec.org`

SPECweb2005 Benchmark
`http://www.spec.org/osg/web2005/`

SPECweb2005 Benchmark Results
`http://www.spec.org/osg/web2005/results/web2005.html`

## Ordering Sun Documents

The SunDocs[SM] program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals through this program.

## Accessing Sun Documentation Online

The `docs.sun.com` web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com/`

To reference Sun BluePrints OnLine articles, visit the Sun BluePrints OnLine Web site at: `http://www.sun.com/blueprints/online.html`

# Appendix A
# Introduction to Cryptography and the Secure Sockets Layer

## Cryptography

Cryptography is a collection of algorithms that uses mathematics to encrypt and decrypt data. A cryptographic algorithm (cipher), is a set of complex mathematical functions used in the encryption and decryption process. It works in combination with a key (a number, sentence, or word) to encrypt the plain text. The encrypted text is called *ciphertext*. The security strength of the ciphertext is dependent on two factors:

- The strength of the cryptographic algorithm
- How secret the key can be kept

Symmetric-key encryption uses the same key for encryption and decryption. The Data Encryption Standard (DES) is an example of symmetric-key encryption. Sometimes this key is referred to as the Secret Key, and the technique is referred to as Secret Key Encryption. Symmetric-key encryption poses a problem in key distribution between the various trusted parties—the key can be leaked during transmission. The problem of key distribution is addressed by Public Key Cryptography. It is an asymmetric scheme (asymmetric-key encryption) that uses a pair of keys. These keys include a *public key* which encrypts data, and a corresponding *private key* for decryption. Only the public key is published. The private key is kept secret. Anyone with a copy of the public key can encrypt information, however the data can only be decrypted by the owner of the private key.

Since keeping the key secret is no longer an issue, the focus is on making the key pairs as unique as possible. This requires generating large key values. Commonly used keys are 1,024 bits long, and stronger encryption mechanisms use 2,048 bit or 4,096 bit keys. RSA and DSA are two algorithms used in key generation.

Cryptography often involves a one way hashing function that takes variable length input messages of any length and produces a fixed length output The hash function ensures a different hash value is produced if the information is changed in any way. This process is called the message digest. The message digest is then encrypted with the private key of the sender, and is called a digital signature. The verification of a digital signature involves decrypting the signature using the sender's public key, and then comparing the resultant message digest (hash value) with another hash of the original message which was sent with the digital signature.

## Secure Sockets Layer

The Secure Socket Layer (SSL) is a TCP/IP based application of cryptography built around the concept of public key cryptography. Developed by Netscape Communications as an application layer protocol above TCP/IP but just below other application layer protocols like HTTP, SSL has been universally accepted as the protocol for authenticated and encrypted communication between clients and servers. The SSL protocol uses a combination of public key and symmetric key encryption. An SSL session always begins

with an exchange of messages called the SSL handshake. The handshake allows the server to authenticate itself with the client using public key techniques. It then allows the client and server to agree on the symmetric keys that will be used for encryption, decryption, and tamper detection during the session that follows. The RSA key exchange algorithm is commonly used with the common cipher suites, including:

- DES and 3DES
- DSA
- MD5, the message digest algorithm developed by Rivest
- RC2 and RC4, Rivest encryption ciphers developed for RSA Data Security
- RSA, a public-key algorithm for both encryption and authentication developed by Rivest, Shamir, and Adleman
- SHA-1, the Secure Hash Algorithm, a hash function used by the U.S. Government

Multiple cryptographic algorithms, and multiple vendors implementing these algorithms, pose an interoperability challenge. Even though vendors may agree on the basic cryptographic techniques, compatibility between implementations is not guaranteed. Interoperability requires strict adherence to agreed upon standards. To help this effort, RSA Laboratories worked with representatives of industry, academia, and government to develop a family of standards called Public Key Cryptography Standards, or PKCS.

There are multiple PKCS standards. Each standard addresses a set of interoperability issues. PKCS#11 specifies an API, called the cryptographic token interface (Cryptoki), to devices which hold cryptographic information and perform cryptographic functions. Cryptoki follows a simple object-based approach, addressing the goals of technology independence (any kind of device) and resource sharing (multiple applications accessing multiple devices), presenting to applications a common, logical or abstract view of the device called a cryptographic *token*.

The PKCS11 functionality is split roughly into three parts:

- Administrative operations, including functions like login, session management, and more
- Object management operations, such as create, destroy objects, and more
- Cryptographic operations, including digest, encrypt, and more

The main objects in the PKCS #11 API are slots, tokens and PKCS11 objects. Token is a general term for devices or placeholders which hold cryptographic information (PKCS11 objects like keys or certificates) and perform cryptographic functions (like digital signatures, random number generation or encryption) after having opened a session. A slot is a container which can potentially hold a token.

PKCS#12 (Personal Information Exchange Syntax Standard) specifies a portable format for storing or transporting a user's private keys, certificates, or miscellaneous secrets.

**Sun Microsystems, Inc.** 4150 Network Circle, Santa Clara, CA 95054 USA **Phone** 1-650-960-1300 or 1-800-555-9SUN **Web** sun.com