SSL acceleration test bench

analysing[1] SSL accelerating test methods

Maurits van der Schee & Stéfan Deelen

February 5, 2007

[1]Research for the University of Amsterdam in cooperation with SURFnet

Abstract

In this document methods for answering the question what the actual added value of an SSL accelerator to a web server is, are analysed. These test methods should be easy to perform and should enable it to compare results of different accelerators and make a statement about their performance. The scope of this research is limited to: Open source operating systems, Apache, OpenSSL and the RSA + 3DES cipher suite.

We, two students, performed research to three testing methods. The first method is the OpenSSL's built-in benchmark. That benchmark tests the performance of the crypto library used by Apache. Second, we tested the response time of a single SSL request in a single session. Last we used Httperf [15] to test the number of connections that the server can handle concurrently.

A gray box test is a black box test with a design that is based on the knowledge of algorithms, internal states, architectures, or other high level descriptions of program behaviour (see [16]). Since in [16] is also stated that "gray box testing is integral to the effective testing of Web applications", we did research into the internal structure of the software and algorithms we used.

OpenSSL speed benchmark is a white box testing method that only tests the costly RSA decryption step in the handshake part of the SSL connection. We found that the following factors influences the RSA crypto performance: Hardware optimisation for certain algorithms and block sizes, driver implementation and algorithm balance.

We validated Httperf as our main black box test tool by analyzing its behavior with an ip packet analyzer. We also ran some tests on both Ab (Apache bench) and Httperf and found no difference in their results. Our calculation shows that there should be a difference of 7 ms between the situation with and without accelerator, caused by the RSA decryption in the handshake. Our measurements only shows a 2 ms difference. In the limited research time we couldn't find a reason for this.

We used Httperf to test concurrent connections. The knowledge of the SSL implementation [1] internals was used to alter the settings (gray box testing). We used a 0 byte file to focus on handshake and switched to HTTP 1.0 to avoid keep-alive (and thus connection limits). We disabled caching on the client and server side to simulate connections from different hosts. In our test method we use Autobench [5] to distribute and automate series of Httperf tests. In our method we plot graphs from which you can read a saturation point. This saturation point can be zoomed into and the concurrent number of connections that a secured web server can handle can be read from it.

We didn't do any throughput testing although this is very important in looking at the added value of an accelerator. Just like other (maybe better) testing software we didn't evaluate it is part of future work.

We conclude three things: 1, gray box testing is needed to find the actual added value of an accelerator; 2, choices in algorithm, operating system and drivers may multiply (!) performance; and 3, we developed a method that enables easy and comparable tests, within a limited scope. Future work may prove whether this method is useful for other scopes or not.

Contents

1 Introduction

1.1 Situation

An SSL accelerator offers general-purpose CPU offloading for intensive SSL processing. Specialised hardware security processors are used in Web servers and front-end equipment to process the heavy SSL computations much faster than a general-purpose CPU can do.
Due to an increasing use of SSL server certificates, the demand for SSL accelerators within SURFnet increases. Therefore SURFnet is looking for a method that enables fast and easy evaluation of the performance of SSL accelerators. SURFnet asked for developing a test bed and method for evaluating SSL hardware accelerators, which can easily be deployed and generates appropriate reproducible test results. For further reading about the initial project definition see Chapter 8. This first chapter will give a brief overview of SSL acceleration, the research objectives, the problem space, and the research approach.

1.2 Research question and problem space

As stated in the initial project definition the objectives for the developed test method are,

1. to give, through tests, a realistic insight in the actual added value, in terms of performance
2. to make it possible to compare several accelerators.

To achieve the above goals, an answer to the following research question has to be found:

How should we measure the performance of an accelerator in an Apache web environment to reach the above mentioned objectives?

This research is limited to Apache with SSL support on open source operating systems. Within these operating systems the OpenSSL software delivers cryptographic support to applications like SSH, Apache, etc. Unfortunately we couldn't limit the problem space to a specific kernel on a specific operating system. Due to a lack of support for accelerator drivers it appears not possible to specify a kernel for the test bed for testing future accelerators.
For our research we build the required web server on a FreeBSD 6.2 which includes the OpenBSD Cryptographic Framework OCF [14] in the OpenSSL package. OCF contains an extensive set of hardware drivers for cryptographic accelerators.

1.3 Document layout

In this document several methods are described which are potentially useful for evaluating accelerators. Chapter 2 introduces the test methods which will described in detail in the following chapters. To gain an understanding of the value of the vendor specifications, it is attractive to investigate OpenSSL speed testing and measure the speedup factor, which is the performance increase that is caused by addind an accelerator. Chapter 3 describes research to the speed test method, which can be applied at the OpenSSL hardware driver level.

Another approach is to analyze the characteristics of the establishment of a single SSL session. This is described in chapter 4.

Measuring the performance of the web server from the perspective of the client with accelerator against without accelerator seems to reveal everything we desire. It reveals the speedup factor, it enables comparing accelerators, and it gives an insight in the maximum number of SSL handshakes per second in a static situation or even in a real-life setup. Real-life performance depends on user-behavior or application specifics (file sizes, etc). Chapter 5 describes this test method.

Chapters 6 and 7 represent future work and conclusions.

1.4 SSL Background and Definitions

The Secure Socket Layer protocol is applied between TCP and the application layer. SSL provides authentication, confidentiality and integrity of data when it is for example applied to secure HTTP, the most popular application environment for SSL.

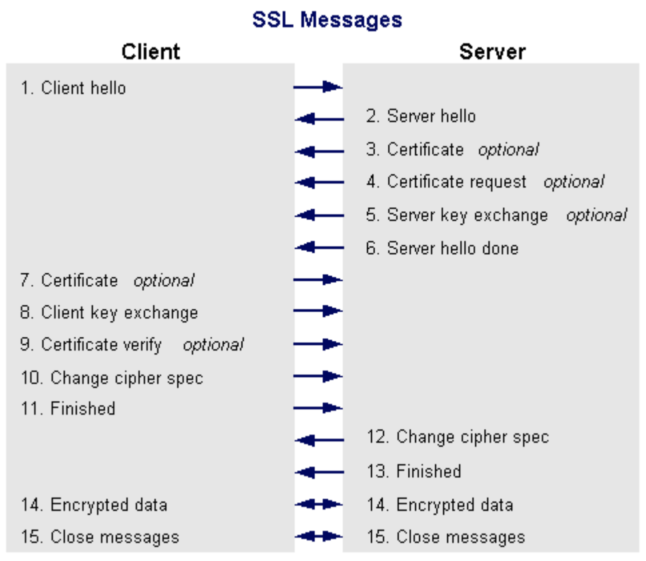

Figure 1: Message flow during a SSL handshake.

A simplified explanation about the SSL protocol must be written out here:
When there is a need for establishing an SSL session, a TCP connection is opened to the server, in which during comprehensive SSL-handshaking a 32

byte SSL session-ID is generated and used by the server. A symmetric key, which is mapped to the unique session-ID will be computed and exchanged using the CPU expensive asymmetric RSA algorithm. The most common SSL-handshaking only uses server authentication with SSL server certificates, causing a computational imbalance in which the server performs the intensive RSA private-key heavy-duty part of the SSL handshake [11]. After session establishment, the application layer protocol is able to start transferring secured data. In the newer SSLv3 version it became possible to re-use old session-ID's which were not removed after closing the earlier TCP connection. With this work-around feature called SSL caching, the old key material is temporarily stored, which decreases the server CPU impact for there is less SSL processing.
For more specific information regarding SSL we refer to the modssl documentation pages [1] and for TLS (successor) the IETF pages [4].

The performance of SSL is proportional to the number of unique SSL transactions within a period of time stated by the rate metric TPS, Transactions Per Second. It is difficult to determine exactly the period between starting and successfuly establishing the SSL session, but somehow this TPS value should be measured. It fits in the scope of this research to put SSL in the HTTPs context. With HTTP, the maximum Request Per Second value, rates the maximum number in time it takes between opening a TCP connection and applying a "HTTP GET" Request. We know that the GET Request can only be seen or measured after the SSL session successfuly established. If we can couple the Get Request to an SSL transaction then TPS will equals RPS, which enables us to measure easily the SSL performance. For an impression of the impact on server performance caused by adding SSL to HTTP, imagine a configuration in which for every "HTTP GET" Request users apply at a secured web server, a unique SSL session has to be established. Similar tests pointed out by comparing that it is possible to reduce the performance of the HTTP server with a factor 1000! [6] (of course this "worst case" SSL configuration isn't very realistic).
SSL accelerators prevent potential performance problems by guaranteeing a determined performance when applying its SSL specific tasks. It is typical that these tasks are completely handled by the accelerator, even if the general purpose CPU is in idle-state and the accelerator is overloaded (!).
There are two types of accelerator implementations to deal with in our research approach: A PCI or SCSI card which contains a co-processor that performs part (or all) of the SSL processing, or a standalone appliance which completely offloads the SSL tasks of the general-purpose CPU.

2 Research approach

2.1 Introduction

Factors that influence SSL performance are various, for example the type of kernel or kernel version, the used SSL ciphers, SSL caching, the performance of the general-purpose CPU, also the client system is a potential performance limit, and not the least factor: The presence of hardware acceleration inside -, or in front of the web server. Our approach comprises finding the most appropriate comparison tests. For example comparing the HTTP transaction speed with or without SSL or comparing the SSL handshake duration with or without accelerator generates interpretable results. In every test method is making sure that the hardware accelerator (including the driver) is the performance bottleneck essential for correct test results. The researched methods for testing accelerators are described below.

2.2 Test Method 1

OpenSSL speed testing enables to compare different accelerators. This method gives direct insight in the combined driver and accelerator performance in terms of maximum processing speed by executing a crypto algorithm maximum number per second. This is an interesting accelerator evaluation method for test objective number 2. This test method only works for internal PCI accelerators and is applied below the Apache application, where SSL TPS rates remains unknown. Compared to speed testing without accelerator a theoretical speedup factor is revealed. Hypothetically, when measuring the SSL TPS value via another method comparing these results to OpenSSL speed results might be useful. Therefore OpenSSL speed testing generates relevant information for our research. Of course the TPS rate value, which is about complete SSL-handshakes, is lower then the measured OpenSSL speed of a specific crypto algorithm (which is part of the SSL-handshake).

2.3 Test Method 2

Our theoretic idea of single session measurement measures the duration of one SSL-handshake with and without accelerator under un-stressful circumstances. Captured traffic nearby the server reveals the TCP connection time in which the SSL-handshake has taken place.

2.4 Test Method 3

Regardless whether the accelerator is applied internal or external, this method approaches from the client perspective the secured web server as a black box remotely [16]. Clients need to be able to overwhelm the server, so this method often requires several clients which cooperate in stressing the server. For reliable results checking whether not the server but the clients are in control is

essential in this method.
While we researched this method a decision had to be made: Do we want to measure the added value in a real-life situation by approaching a real-life configured server set-up including specific (virtual) user behavior? Or will we evaluate the accelerator by reconfiguring the blackbox and locking the (variable) factors, which otherwise potentially obfuscate the interpreted number of SSL-handshakes?
We decided to choose the last option. For both objective number 1 and 2 this is an interesting evaluation method. This method involves establishing concur-

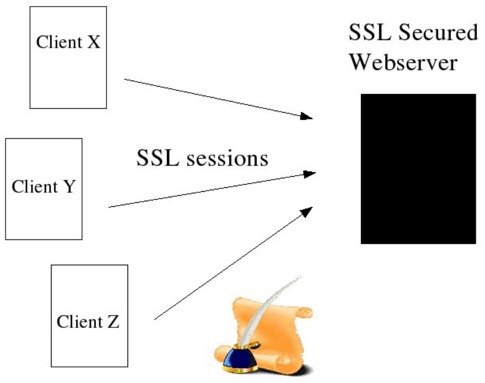

Figure 2: Simple representation of the test bed.

rent SSL sessions, pushing the black box to its limits and measuring the TPS rate on the outside. We enabled measuring the TPS rate on the outside by coupling the TPS rate to the RPS rate. That way we knew that each ”HTTP GET” client request results in a unique SSL-handshake. By disabling SSL caching and applying a work-around, in which the involved clients uses HTTP1.0, we surely knew a TCP connection closes immediately after the GET request (which is typically inefficient behavior with HTTP1.0 [2]) and last but not least, surely knew that the SSL-handshake had to be done all over again in the next TCP connection. From this ”TPS” rate, we didn’t subtract the time TCP handshaking takes neither the ”HTTP GET” interpreting time. So this ”TPS” value is strictly taken an approach to the actual TPS value.
The proposed test bed facilitates this test method, which includes several clients. The proposed clients and software are easy to set up, they will be used to overwhelm the server. While increasing the load step by step, the clients are sampling the actual ”HTTP GET” request rate vs. the demanded ”HTTP GET” request rate. This information is directed to a management client, which - after plotting the results - provides good insight in the saturation point of the accelerator.

3 OpenSSL speed Test Method

> "In the Web context, one of the main overload factors is the direct consequence of expensive Public Key operations performed by servers as part of each SSL handshake. Since most SSL-enabled servers use RSA, the burden of performing many costly decryption operations can be very detrimental to server performance"[11]

OpenSSL is the most popular SSL implementation for Apache and since we focus on Apache we also focus on OpenSSL. OpenSSL has a built-in benchmark. You can use it to measure the performance of cryptographic operations performed by OpenSSL. It can be invoked by executing the `openssl speed` command. Specific asymmetric and symmetric test suites can be given as an extra parameter. You can also test a specific algorithm. To test RSA 1024 bit (the RSA standard in SSLv3) you should execute: `openssl speed rsa1024`

NOTE: When using multiple processors you should add the -multi n flag, where n is the number of processors in the system. When using an accelerator you should add the -elapsed flag to make the benchmark calculate good averages.

3.1 Choosing an algorithm

> "The goal of the SSL/TLS handshake is the establishment of a shared client-server key. The most important component of this process is the clients encryption of a (randomly selected) key under the servers RSA public key. The ciphertext is then transmitted to the server which decrypts it and extracts the key. The core of the overexertion problem is the RSA decryption operation" [11]

We said most SSL-enabled server use RSA, but is RSA a smart choice? In RSA signs are computational expensive and verifies are relatively cheap (up to 19 times). In [11] we read that this imbalanced arrangement is clearly beneficial for computationally challenged clients, however, that it is detrimental to server's connection throughput and general availability. When no precautions are taken a single client machine can easily overload an SSL server by sending consecutive SSL initiation requests. We verified this in our tests.

In other algorithms (like DSA) there is a different balance between encryption and decryption. Even within RSA this balance can be changed, this is called rebalanced RSA. It shifts the workload to the encryptor. It is a variant of an earlier technique by Wiener (see [17]).

> "Specifically, d is chosen to be close to n such that both d mod (p-1) and d mod (q-1) are small integers. The resulting public exponent e also becomes close to n, which is much larger than typical values (i.e., e = 3; 17; 9 or 65537). It is in fact so large that Microsoft Internet Explorer (IE) cannot accept it; IE allots a maximum of 32 bits for the public exponent e. Rebalanced RSA offers the theoretical speedup of 3,6 but the actual speedup is 3,2." [11]

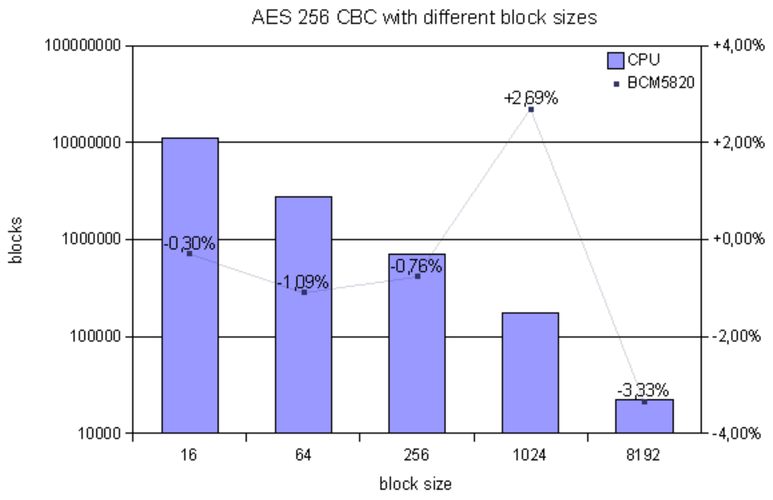

Figure 3: Actual performance in blocks of the CPU and relative performance of the accelerator compared to the performance of the CPU.

Last it is important to note that accelerators are optimized (or only capable) of certain algorithms and certain block sizes. When an algorithm isn't supported the CPU does all the work and no offloading takes place. When an algorithm is supported it is often optimized for a certain block size. In figure 3 you can see the accelerator performs better or worse than the CPU depending on the block size. So choose your handshake algorithm with care, because it may have a big influence on the performance of SSL.

3.2 Hardware influence

We noticed that the performance of an accelerator doesn't seem to depend much on the CPU speed of the machine it is put into. We tested the BCM5820 [12] on different machines and saw that it performed almost equal on all machines as long as the hardware requirements are met. We tried to place the card in a faster machines and we concluded that our hardware was adequate, because we saw no increase in performance.

Note that if you put a slow accelerator in a fast machine the OpenSSL speed benchmark might produce lower scores than without the accelerator (it may slow down SSL). In such a case you might argue that it is better to not use the accelerator, but you should not forget that the accelerator offloads the SSL cryptographic operations and that the CPU is available for other tasks. We measured only a few percent CPU load under heavy stress. Therefore an accelerator that is slower than the CPU in the OpenSSL benchmark is not useless. Especially when you have a dynamic CPU intensive website.

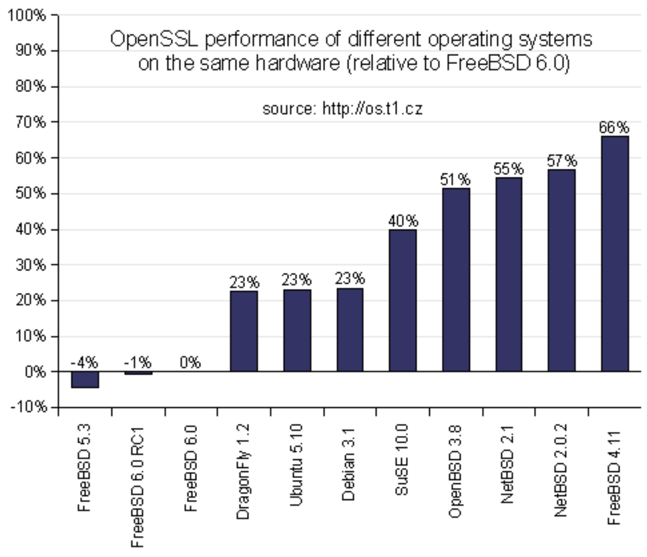

Figure 4: OpenSSL speed tests on different operating systems on the same hardware.

3.3 Operating system influence

On [7] we found openssl speed results (AES-256 CBC algorithm only) for different operating systems that were measured in November 2005. You can see the relative performance we calculated in figure 4. We wanted to know whether the results for the RSA-1024 signs would differ as much as these results. Because RSA-1024 signs are larger operations than AES-256 CBC, we expected a smaller influence of the kernel on the performance. Surprisingly we saw a big difference in the RSA-1024 score on the OpenSSL speed benchmark on different operating systems on the same hardware. We saw that the score of Ubuntu 6.06 LTS was approximately 20% higher compared to the score of FreeBSD 6.2. We are not able to explain this difference, but we showed that the choice of operating system can make a big difference in the performance of OpenSSL.

3.4 Driver influence

We tested the 'ubsec' open source driver for the Broadcom 5820 under FreeBSD 6.2. Broadcom promised via product specifications that this 5820 hardware accelerator is capable of 800 RSA-1024 signs / second. We weren't able to achieve more than 100 signs /second on FreeBSD and over 200 on Redhat 7.2. Broadcom told us personally that the FreeBSD driver wasn't well designed. After contacting Broadcom a tech sales representative said it didn't use the CRT parameters and that we had to stress it with multiple threads to achieve maximum performance. It didn't help us to achieve more than 100 signs / second. That is roughly the same amount that was measured in [13].

Broadcom worked together with RedHat to provided support for this device in the default RedHat 7.2 installation. We tested under RedHat 7.2 with this driver loaded and it performed roughly twice as good as FreeBSD with their driver loaded (97 vs 195 RSA-1024 signs measured with openssl speed). It probably uses the CRT parameters Broadcom told us about. With CRT-RSA one can expect a speed-up by a factor of 4 compared to plain RSA according to [10]. Although we weren't able to reach the promised 800 RSA-1024 signs / second, we can conclude that drivers can make a big difference in performance of an accelerator.

4 Single Session Test Method

We chose Httperf as the main tool for our benchmarking method. To make sure Httperf does what we expect it to do we tried a single request with and without accelerator. We captured the packets using tcpdump [tcpdump] and analyzed those with Wireshark [9] (see figure 5 and 6).

```
-------------------------------------------------------------------------------
|Time     | client                     server | Comment                       |
-------------------------------------------------------------------------------
|0,000    |         40885 > https [SYN]       | TCP: 40885 > https [SYN]      |
|         |(40885)  ------------------>  (443)|                               |
|0,000    |         https > 40885 [SYN,       | TCP: https > 40885 [SYN, ACK] |
|         |(40885)  <------------------  (443)|                               |
|0,000    |         40885 > https [ACK]       | TCP: 40885 > https [ACK]      |
|         |(40885)  ------------------>  (443)|                               |
|0,001    |         Client Hello, [Pack       | SSLv3: Client Hello           |
|         |(40885)  ------------------>  (443)|                               |
|0,020    |         Server Hello, [Pack       | SSLv3: Server Hello           |
|         |(40885)  <------------------  (443)|                               |
|0,020    |         40885 > https [ACK]       | TCP: 40885 > https [ACK]      |
|         |(40885)  ------------------>  (443)|                               |
|0,043    |         Client Key Exchange       | SSLv3: Client Key Exchange    |
|         |(40885)  ------------------>  (443)|                               |
|0,043    |         Change Cipher Spec        | SSLv3: Change Cipher Spec     |
|         |(40885)  ------------------>  (443)|                               |
|0,043    |         Encrypted Handshake       | SSLv3: Encrypted Handshake    |
|         |(40885)  ------------------>  (443)| Message                       |
|0,043    |         https > 40885 [ACK]       | TCP: https > 40885 [ACK]      |
|         |(40885)  <------------------  (443)|                               |
|0,052    |         Change Cipher Spec,       | SSLv3: Change Cipher Spec,    |
|         |(40885)  <------------------  (443)| Encrypted Handshake Message   |
|0,053    |         Application Data, [       | SSLv3: Application Data       |
|         |(40885)  ------------------>  (443)|                               |
|0,053    |         Application Data, [       | SSLv3: Application Data       |
|         |(40885)  <------------------  (443)|                               |
|0,053    |         Encrypted Alert, [P       | SSLv3: Encrypted Alert        |
|         |(40885)  <------------------  (443)|                               |
|0,053    |         https > 40885 [FIN,       | TCP: https > 40885 [FIN, ACK] |
|         |(40885)  <------------------  (443)|                               |
|0,053    |         Encrypted Alert, [P       | SSLv3: Encrypted Alert        |
|         |(40885)  ------------------>  (443)|                               |
|0,053    |         40885 > https [FIN,       | TCP: 40885 > https [FIN, ACK] |
|         |(40885)  ------------------>  (443)|                               |
|0,053    |         https > 40885 [ACK]       | TCP: https > 40885 [ACK]      |
|         |(40885)  <------------------  (443)|                               |
-------------------------------------------------------------------------------
```

Figure 5: Packet flow (accelerator disabled).

We compared the packet flow with the theoretical packet flow (figure 1) and with a packet flow generated by a real browser and we saw no unexpected packets or differences. We also compared the results we got from Httperf with the results we got from AB (Apache Benchmarking tool, see [8]). We noticed no difference between those two benchmarks.

We calculated a difference of 7 ms (see calculations below) in the decryption of the symmetric key between the situation with and without accelerator. Surprisingly we didn't see this extra latency in the results of Httperf. We did measure a 2 ms difference in the total transaction between the situation with and without accelerator in the server. We enabled debugging in the driver to verify that the device was actually in use. We didn't have enough time to find out why the performance was different than expected.

$$\text{Sign duration} = 1/\text{sign rate} \tag{1}$$

$$1/\text{sign rate unaccelerated} = 1/307 = 3\text{ms} \tag{2}$$

$$1/\text{sign rate accelerated} = 1/97 = 10\text{ms} \tag{3}$$

$$10 - 3 = 7\text{ms difference} \tag{4}$$

```
--------------------------------------------------------------------------------
|Time     | client                     server | Comment                          |
--------------------------------------------------------------------------------
|0,000    |         40865 > https [SYN]       | TCP: 40865 > https [SYN]         |
|         |(40865)  ------------------>  (443)|                                  |
|0,000    |         https > 40865 [SYN,       | TCP: https > 40865 [SYN, ACK]    |
|         |(40865)  <------------------  (443)|                                  |
|0,000    |         40865 > https [ACK]       | TCP: 40865 > https [ACK]         |
|         |(40865)  ------------------>  (443)|                                  |
|0,001    |         Client Hello, [Pack       | SSLv3: Client Hello              |
|         |(40865)  ------------------>  (443)|                                  |
|0,021    |         Server Hello, [Pack       | SSLv3: Server Hello              |
|         |(40865)  <------------------  (443)|                                  |
|0,022    |         40865 > https [ACK]       | TCP: 40865 > https [ACK]         |
|         |(40865)  ------------------>  (443)|                                  |
|0,044    |         Client Key Exchange       | SSLv3: Client Key Exchange       |
|         |(40865)  ------------------>  (443)|                                  |
|0,044    |         Change Cipher Spec        | SSLv3: Change Cipher Spec        |
|         |(40865)  ------------------>  (443)|                                  |
|0,044    |         Encrypted Handshake       | SSLv3: Encrypted Handshake       |
|         |(40865)  ------------------>  (443)| Message                          |
|0,044    |         https > 40865 [ACK]       | TCP: https > 40865 [ACK]         |
|         |(40865)  <------------------  (443)|                                  |
|0,054    |         Change Cipher Spec,       | SSLv3: Change Cipher Spec        |
|         |(40865)  <------------------  (443)| Encrypted Handshake Message      |
|0,055    |         Application Data, [       | SSLv3: Application Data          |
|         |(40865)  ------------------>  (443)|                                  |
|0,055    |         Application Data, [       | SSLv3: Application Data          |
|         |(40865)  <------------------  (443)|                                  |
|0,055    |         Encrypted Alert, [P       | SSLv3: Encrypted Alert           |
|         |(40865)  <------------------  (443)|                                  |
|0,055    |         https > 40865 [FIN,       | TCP: https > 40865 [FIN, ACK]    |
|         |(40865)  <------------------  (443)|                                  |
|0,055    |         Encrypted Alert, [P       | SSLv3: Encrypted Alert           |
|         |(40865)  ------------------>  (443)|                                  |
|0,055    |         40865 > https [FIN,       | TCP: 40865 > https [FIN, ACK]    |
|         |(40865)  ------------------>  (443)|                                  |
|0,055    |         https > 40865 [ACK]       | TCP: https > 40865 [ACK]         |
|         |(40865)  <------------------  (443)|                                  |
--------------------------------------------------------------------------------
```

Figure 6: Packet flow (accelerator enabled).

We changed from HTTP 1.1 to HTTP 1.0 to force the server to disconnect after a "HTTP GET" request. Otherwise Httperf would keep the connection open (Keep-alive) for 5 seconds after the "HTTP GET". We noticed that this keep alive happened only when the requested file exceeded 8 kilobyte. The connection should be closed immediately because this decreases the chance to hit a TCP connection limit.

5 Black Box Test Method

5.1 Introduction

This method can be applied to an internal or external accelerator. The client stresses the server and counts the maximum number of SSL-handshakes per seconds it achieves. As mentioned earlier, in this document the number of SSL-handshakes per second is referred to as ”RPS”.

This method requires a test bed. This test bed is briefly described in chapter 2, but is detailed described in chapter 9 (Appendix B). Figure 2 in Chapter 2 visualizes the test bed. There are two important specifications of this test bed:

1. each initiated client TCP connection is a trial to initiate an SSL session.
2. each successful ”HTTP GET” Request confirms a successful establishment of the SSL session.

5.2 Required tools

This method uses the benchmark tools **httperf** [15], and **autobench** [5]. The autobench “man page” offers the visual description shown in figure 7:

```
             +-----------------+
             | autobench_admin |
             +--------+--------+
                      |
                      |
      +---------------+---------------+
      |               |               |
      |               |               |
+-----+------+  +-----+------+  +-----+------+
| autobenchd |  | autobenchd |  | autobenchd |
| clientX    |  | clientY    |  | clientZ    |
+-----+------+  +-----+------+  +-----+------+
      |               |               |
      |               |               |
      +---------------+---------------+
                      |
                      |
               +------+-----+
               | web server |
               | under test |
               +------------+
```

Figure 7: Autobench visual description.

Httperf initiates a fixed number of TCP connections per second with a configurable fixed number of ”HTTP GET” requests within one TCP connection.

During the test httperf samples the (TCP) connection rate, the (HTTP) request rate and the reply rate. Each 5 seconds a sample is taken. It is recommended to run tests long enough so at least thirty samples are obtained of accurate results. The httperf tool only measures at a fixed rate.

The collected information is stored as a table in ASCII format in a tab separated file and can be plotted with the included `bench2graph` tool. We've used Open Office to generate our charts, because we felt `bench2graph` was too limited

Autobench runs a series of httperf tests, changing its parameters in such a way that it increases the load every run. The results can be plotted and enable you to read the saturation point from the graph.

Autobench is also capable of controlling a group of clients. This feature can be used to stress a server in a distributed manner. Distributed autobench installs a daemon on each client to control httperf. The results of each run are sent to the autobench admin program that collects all sample information.

5.3 Example results

If the tools are installed at the client systems [appendix B], each participating client has to be put in listen mode by starting the "autobenchd" daemon. At the management client the distributed tool "autobench_admin" has to be started from the command line with the correct options. For example the following options might reveal the saturation point if it is somewhere between 30 and 40 RPS:

```
autobench_admin --single_host --host1
ssl-test1.wind.surfnet.nl --port1 --uri1 /0k
 --clients 192.87.110.43:4600,192.87.110.44:
 4600,192.87.110.96:4600 --low_rate 26
 --high_rate 45 --rate_step 1 --num_call 1
  --num_conn 200 --timeout 5
  --file dist-test-bench.tsv
```

In this distributed autobench example 20 httperf tests will be executed sequentially. The first httperf test will start with initiating 26 TCP connections per second, until a subtotal of 200 TCP connections is reached. Then the next httperf test will start with 27 TCP connections. The option "num_call" sets the number of "HTTP GET" requests which is 1 in our case. When there is no response after 5 seconds, the TCP connection is closed. The duration of the test can take a while. The results will be better if httperf takes more samples. You can make a httperf test run longer by increasing "num_conn". Consequence is that autobench will also longer (about 30 minutes is a realistic duration). The host name may be an ip address. The server certificate doesn't have to be pop-up free.

If the saturation point is completely unknown, a quick test with large steps can roughly indicate this. With these results the options can be refined by running autobench_admin again for accurate results.

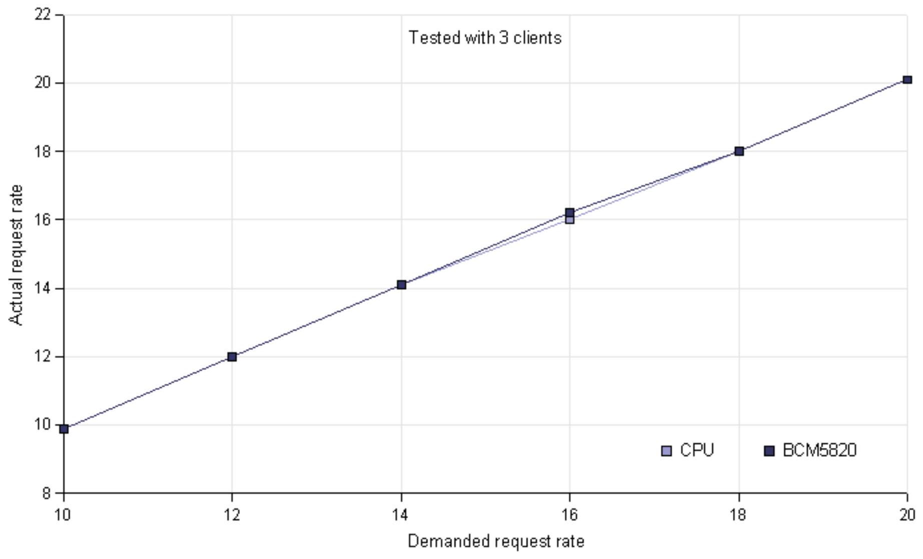

Figure 8: First quick test showing no saturation point (straight line).

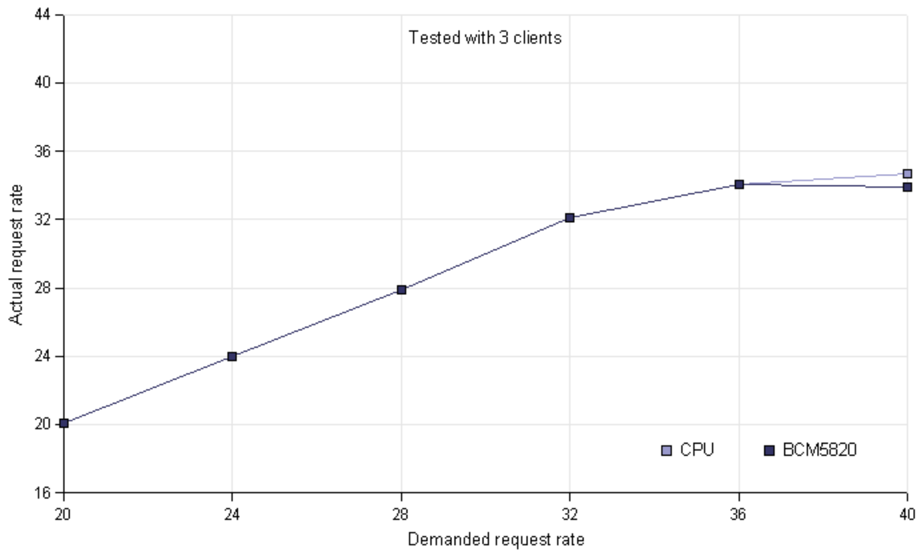

Figure 9: Second quick test showing a saturation point (line is bending horizontal).

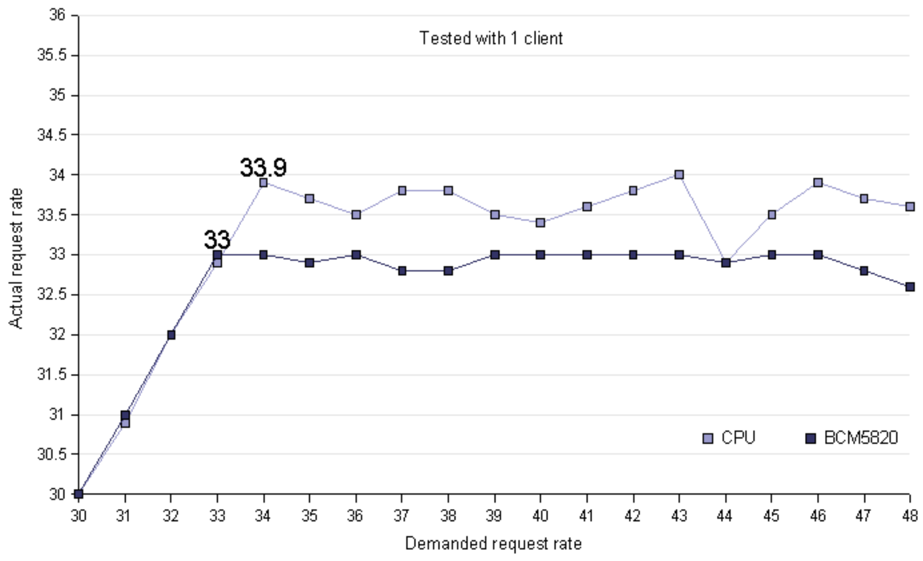

Figure 10: Recalculating the area around the saturation point in more detail using a single client. The maximum rate (number of connections per second) is shown.

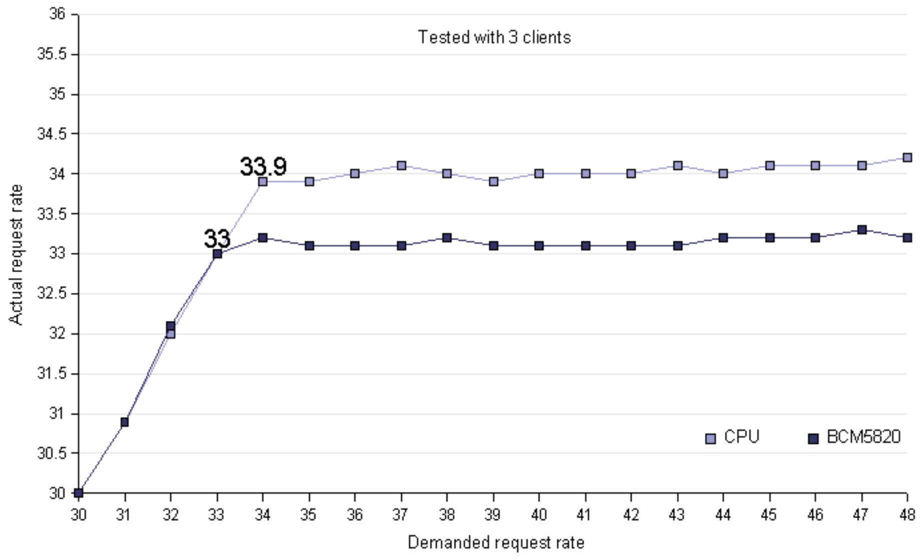

Figure 11: Recalculating the area around the saturation point in more detail using three clients. The maximum rate (number of connections per second) is shown.

5.4 Expected performance

In our case the vendor specifications states that the accelerator is capable of 800 1024 bits RSA signatures per second. OpenSSL rates a limit of 97 signatures per second (due to other limiting factors). Finally, the measured TPS rate via this black box method (in which each SSL-handshake involves a 1024 bits RSA signature) shows that the maximum is about 37 full SSL-handshakes per second.

5.5 How many clients?

To determine the minimum number of clients we stressed the server with several clients, analyzed the position of the saturation point and removed one client. We did this several times until removing a client caused the saturation point to move. This indicates that you've reached the minimum number of clients. To generate the desired results, it is important to check the minimum number of clients to make sure the clients are in control.

In most situations SSL is configured without client certificates; only the server authenticates itself. Due to the RSA imbalance, the server has more load than the client and fewer clients are required for stressing a secured than an unsecured web server. In our case we showed that one client was enough for stressing the secured web server, with or without the accelerator in use. This is normal if no precautions are taken (see [11]).

6 Future work

We have made a lot of choices that determined the focus of this research. We've only looked at the handshake part of a SSL connection. You could also test the throughput of an accelerator. Probably with the same tools and the same testbed, but one may encounter different problems.

We focused on OpenSSL with Apache on open source operating systems, but one could be interested in testing SSL acceleration with, for example, IIS on Windows to find the best environment for a specific accelerator in specific circumstances.

To test how many users can actually use a web server concurrently you should try to emulate these users as good as possible. Application specific test scripts can be written and executed against a real web application to get insight in the amount of users that can use a specific secure web application concurrently.

The tools that were used in our testbed are appropriate, but we were not able to test and use all available tools. We are especially interested in the difference between the SSLperf [3] tool the tools we used. There may also be other interesting software that we didn't look at.

To automate our test method you could improve autobench in such a way that it can search for a saturation point. One would have to program software to interpret the output delivered by autobench.

During the development of our test method we used a four year old SSL accelerator. We've read about more powerful accelerators. Those may require a larger and even more distributed approach. One could develop a special distributed test client linux distribution that is bootable from a live CD. It may be tuned and configured to test SSL servers. Another approach may be to rent CPU time and bandwidth from companies with large cluster systems.

7 Conclusions

We have shown that vendor specifications cannot always be trusted. Performance is influenced heavily by the choice of algorithm, the optimizations in the algorithm, the balance in the algorithm, the operating system the server runs on and the quality of the available drivers. The performance implications by these choices shouldn't be underestimated. Changing a few things can multiply (!) the performance of your secure web server.

For testing PCI card SSL off-loaders you can use the OpenSSL speed test. It gives a good indication of the performance of a specific algorithm on a specific machine. This method cannot be used on external SSL offloaders and doesn't give a good indication of the actual maximum number of SSL transactions per second a secure web server can handle.

The best test method to determine the maximum TPS rate for internal and external offloaders is to do a distributed black box test on the server. The results of this method are the easiest to interpret among the other tests. If the specific application you want to use the accelerator for is known, you could change the parameters of autobench to reflect an actual situation. That can make this method even more valuable.

8 Appendix A, Project definition

Context

SURFnet is a high-grade computer network specially reserved for higher education and research in the Netherlands. To achieve a better understanding of the added performance value of SSL accelerators for their services, the SURFnet organisation proposed the following initial project definition for students at the University of Amsterdam: ”Develop a test method for testing the quality and performance of SSL accelerators and HSMs”. We, two students from the University of Amsterdam, will perform this desired research and development during a period of four weeks.

Problem space

After doing a week of pre-research, we decided in consensus with Jan Meijer of SURFnet to exclude the HSM (Hardware Security Module) part from our problem space. We felt that HSM was a different subject that required a different approach and was only loosely related to SSL accelerators.

For narrowing down the subject of our research on SSL accelerators, we decided to focus on HTTPS connections in Apache with OpenSSL, a popular application of SSL accelerators.

Goals

The problem with SSL accelerators is that the manufacturer specifications do not provide enough information to predict the added value that such a device has for the application it is used for. The developed test method should therefore:

1. give, through tests, a realistic insight in the actual added value, in terms of performance
2. make it possible to compare several accelerators.

Research question

The following research question wich is separated in following sub questions reflects our approach:

How should we measure the performance of a hardware accelerator in an Apache web environment to reach goal 1 and 2?

Approach and Subquestions

OpenSSL contains a benchmarking tool which can be used for measuring the performance of a SSL accelerator. We think that the performance as stated by the OpenSSL benchmark doesn't differ enough from the performance stated by the manufacturer to make it a subject of our research. We will test this assumption and if it proves to be correct we want to measure the performance of the web server as a whole.

This allows us to produce a single test method that can be used for both SSL accelerator PCI cards and stand-alone SSL off loader appliances. By measuring the performance with and without accelerator we can calculate the added value of the accelerator.

The performance of an SSL accelerated HTTPS application depends on many factors that we will describe in our research. In our tests we try to keep as much factors as constant as possible to generate comparable results. For our tests we will try to define scenarios that are common to HTTPS applications. This should provide a good insight in the added value of the accelerator in different HTTPS scenarios.

Our goal is to develop a test method for SSL accelerators which probably requires a test bed. Therefore we will describe what the specifications of this test bed are.

By working out these subquestions, we hope to give our approach some structure:

- Should we measure the performance by doing isolated tests on the accelerator and its driver or should we test the performance of the entire web server (with and without accelerator)? **Explanation:** We will argue that goal 1 cannot be reached by only looking at the performance of the accelerator and its driver. We will try to explain, with goal 2 in mind, what may cause differences between OpenSSL performance and the performance stated by the manufacturer.

If we test the performance of the entire web server:

- How do we validate our performance results? **Explanation:** The performance of a web server has a weakest link principle. We must make sure that we are not hitting client, network or other limits when measuring performance. We must make sure that the SSL encryption by OpenSSL is the weakest link.
- Should we express the resulting performance of the accelerator as a relative (related to the performance of the same system without accelerator) or absolute performance (e.g. transactions per second)? **Approach:** We will do our tests on faster and slower machines to look at the influence of the systemspecs on the performance results of an accelerator.
- Should we measure the maximum performance or should we measure the performance at certain load levels? **Approach:** We will base our decision on research done by others.

If we test the maximum performance of the entire web server:

- When do we say that the performance is at its top? **Approach:** We will discuss the following points (we might find more): When are connections disconnected? (TCP timeout); when are errors (like too busy) served? (Apache limit); when does the response time exceeds a certain limit? (usability).
- Which performance metrics should we use and how should we interpret these metrics? **Approach:** We will discuss the following metrics (we might find more): Transactions per second; requests per second; virtual users (application specific); response time; concurrent connections; CPU load (lower is better); memory usage (lower is better); at which point are no more connections accepted? (connection timeout).
- How to setup a test bed and what open source tools can be used to retrieve the metrics described earlier? **Approach:** We might discuss these or similar software packages and describe how to use them: Jmeter; Httperf and Autobench; OpenSTA
- If we need a distributed test environment, how can we implement this? **Approach:** We might discuss how to use: Jmeter distributed; Amazon distributed services, "Elastic Cloud" (virtual computing) with S3 (storage)

Planning

This research project will take about four (4) weeks. The result of the first week is scoping and focussing the project definition. In the next two weeks we will try to follow the approaches described above.
The last week will be used to write the report detailing the results of our project and preparing for the presentation given on the 7th of Februari at the University of Amsterdam in which we will defend our conclusions.

9 Appendix B, Guide for Test Bed Set-up

Required for executing Method no. 3

Introduction

In this appendix the setup of a FreeBSD based testbed for SSL accerators is ”spelled out”. For verifying the functionality of this test bed, we used an accelerator of Broadcom, PCI model type 5820. We’ve first tried Linux kernel vs 2.6 as server OS (the Ubuntu distro): It appears that this kernel doesn’t support required drivers for Broadcom 5820 accelerators. FreeBSD kernels contains the OCF, which is a kernel-included piece of software - it stands for ”openbsd crypto framework” - which includes a set of device drivers for crypto hardware devices.

Also, Surfnet advised FreeBSD as server OS for it’s performance and stability.

At the client side it doesn’t mather what OS is used, we have used 3x Ubuntu 6.06 (server), and 1x FreeBsd 6.1. The drawing of the testbed layout can be found at the beginning of this document. First the server installation is demonstrated, the client installation follows.

Server installation

Configuring the FreeBsd Apache SSL server with accelerator..

Setup your network and hardware

Patch the server on a locally switched 1 Gbit/s network segment.
Plug-in the (PCI) accelerator.

Install FreeBSD from CD, follow the sysinstall dialogue

Fetch the latest portsnap

We advise using the latest packages. These are required. When the server is connected via the network with a functional (dns) configuration, use:
portsnap fetch

Install and configure required packages

For easy working: Make sure the server is remote accessible and install the package ”nano” and ”bash”. (Configure ”/etc/ssh/sshd_config” for remote management).

Install Apache 2.x. (Apache version 2.2 includes the required mod_ssl package)

- cd usr/ports/www/apache22/
- make WITH_EXPERIMENTAL_MODULES=yes
- make install

Configure /etc/rc.conf for adding Apache:

```
host_name# vi /etc/rc.conf
defaultrouter="[default_gateway]"
hostname="[host_name]"\\
ifconfig_em0="inet [ip_address] netmask [net_mask]"
inetd_enable="YES"
linux_enable="YES"
sshd_enable="YES"
usbd_enable="YES"\\
apache22_enable="YES"
```

Configure /etc/resolvċonf for name resolution:
host_name# vi resolv.conf
search [domain]
nameserver [name server 1]
nameserver [name server 2]

Edit the Apache configuration file '/usr/local/etc/apache22/Includes/httpd-ssl.conf' add:

SSLCryptoDevice cryptodev
ServerName ssltest1ẇindṡurfnetṅl
remark rules with 'SSLSessionCache' and add:
SSLSessionCache nonenotnull

Generate a server certificate

It is good common practice to create a valid popup free server certificate. Therefore it is needed to configure a correct period "Valid from..to" period within the certificate is valid. Also the "Issued to" name should match the URL of the website (Ip address in our case). Also, the client browser should be able to verify the identity via the "Certification Path". To work around this (missing) path, we install the certificate in the local 'trusted' certificate repository. We followed the certificate generation procedure below:

Generate a key-pair

```
openssl genrsa -des3 -out server.key 1024
```

Generate a Certifcate Signing Request via a local question/answer dialogue

```
openssl req -new -key server.key -out server.csr
```

As stated above, to create a pop-up free certificate it is necessary to match the "Issued to" name with the URL of the website. In the dialogue this is the "Common Name" (eg, YOUR name) []:192.87.110.94. Our testsite is n't advertised in DNS, therefore we use the server ip address as the common name.

Generate a Self-signed certificate:

```
openssl x509 -req -days 600 -in server.csr -signkey server.key -out server.crt
```

Now we are almost ready. In the Apache configuration file (/usr/local/etc/apache/httpd.conf) the - for public use generated - info is needed according to the following configuration statements:

```
SSLCertificateKeyFile /usr/local/etc/apache22/server.key
SSLCertificateFile/usr/local/etc/apache22/server.crt
```

Before we copy the created files to this location, we'll remove the encryption of the server.key file. Our server has crashed several times during openssl speed test's (in the case of multi threading situation, and when a rsa 4096 bits keysize performance test was applied). We experienced that the server didn't selfrecover after a crash due to a required pass phrase for starting SSH and Apache-SSL. The reason for this behavior was the encryption of the above generated private key: Only a non-stored password enabled the use of this key by SSL and SSH.

Therefore we remove the encryption with:

- cp server.key server.key.org
- openssl rsa -in server.key.org -out server.key

So now copy the key, the certificate and the file 'httpd-ssl.conf' to the appropriate location:

- cp server.key /usr/local/etc/apache22/server.key
- cp server.crt /usr/local/etc/apache22/server.crt
- cp /usr/local/etc/apache22/extra/httpd-ssl.conf /usr/local/etc/apache22/Includes/httpd-ssl.conf

Load the required kernel modules (drivers)

Attended after each reboot:

- kldload ubsec
- kldload cryptodev
- kldload accf_http

Or unattended during each reboot:

- cp /boot/GENERIC/ubsec.ko /boot/modules

Add the following statements in the file '/boot/loader.conf':

- accf_http_load="YES"
- ubsec_load="YES"
- cryptodev_load="YES"

Restart the system

Client Setup

Installating clients & tools for distributed stress testing at a SSL black box.

We used Ubuntu 6.06 clients (server OS). This OS is simple to set-up and offers easy package management. *The next steps are required:*

- iso-cd install Ubuntu 6.06 server
- edit sources.list conform the ubuntu.org advice
- install the following packages: apt-get install ssh, apt-get update, apt-get upgrade, apt-get update, apt-get install build-essential, apt-get install httperf gawk gnuplot rdate
- download and install the autobench tool:

wget http://www.xenoclast.org/autobench/downloads/autobench-2.1.2.tar.gz
tar zxvf autobench-2.1.2.tar.gz, cd autobench-2.1.2, make, make install

Important autobench settings !!

For using httperf's -ssl flag by autobench edit the file .autobench.conf and ad the following statements: (the file autobench.conf is created after the first runtime in)
httperf_ssl = NULL
httperf_ssl-ciphers = choose an OpenSSL cipher
httperf_ssl-no-reuse = NULL
httperf_http-version = 1.0

We choose http1.0 for the reason that http1.1 within httperf don't closes the TCP connection immediately after the HTTP GET requist: It waits for 5 seconds which diffuses the results.

It is adviceable to choose an appropriate common used ssl-cipher. Check OpenSSL for the possibilities.

This client is now ready for testing!!!

It is adviceable to minimize any potential risks for faulty interpretations.. We created reliable results (regarding data transfer) by connecting the Ubuntu clients in the same Gbps Ethernet switch as the server system.

References

[1] modssl documentation, chapter 2: Introduction, 1998. `http://www.modssl.org/docs/2.8/ssl_intro.html` [Online; accessed 02-February-2007].

[2] Key differences between http/1.0 and http/1.1, 1999. `http://www.research.att.com/~bala/papers/h0vh1.html` [Online; accessed 02-February-2007].

[3] delivering the world's fastest hp-ux 11i ssl performance with the intel itanium 2 processor family, 2002. `http://h20338.www2.hp.com/hpux11i/downloads/5981-2318EN.pdf` [Online; accessed 02-February-2007].

[4] Transport layer security (tls) extensions, 2003. `http://www.ietf.org/rfc/rfc3546.txt` [Online; accessed 02-February-2007].

[5] Autobench: A perl wrapper around httperf for automating benchmarking, 2004. `http://www.xenoclast.org/autobench/` [Online; accessed 02-February-2007].

[6] Performance impact of using ssl on dynamic web applications, 2004. `http://www.bsc.es/media/389.pdf` [Online; accessed 02-February-2007].

[7] Simple os comparison, 2005. `http://os.t1.cz/` [Online; accessed 02-February-2007].

[8] ab - apache http server benchmarking tool, 2007. `http://httpd.apache.org/docs/2.0/programs/ab.html` [Online; accessed 02-February-2007].

[9] Wireshark: The world's most popular network protocol analyzer, 2007. `http://www.wireshark.org/` [Online; accessed 02-February-2007].

[10] Johannes Bl&#246;mer, Martin Otto, and Jean-Pierre Seifert. A new crt-rsa algorithm secure against bellcore attacks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 311–320, New York, NY, USA, 2003. ACM Press.

[11] C. Castelluccia, E. Mykletun, and G. Tsudik. Improving secure server performance by rebalancing ssl/tls handshakes, 2005.

[12] Broadcom corporation. Bcm5820 product brief, 2006. `http://www.broadcom.com/collateral/pb/5820-PB06-R.pdf` [Online; accessed 02-February-2007].

[13] A. Keromytis, J. Wright, and T. de Raadt. The design of the openbsd cryptographic framework, 2003.

[14] Samuel J. Leffler. Cryptographic device support for freebsd (awarded best paper!). In *BSDCon*, pages 69–78. USENIX, 2003.

[15] David Mosberger and Tai Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998.

[16] Hung Quoc Nguyen, Bob Johnson, Robert Johnson, and Michael Hackett. *Testing Applications on the Web: Test Planning for Mobile and Internet-Based Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[17] Michael J. Wiener. Cryptanalysis of short rsa secret exponents (abstract). In *EUROCRYPT*, page 372, 1989.