Optimizing TLS for High–Bandwidth Applications in FreeBSD

Randall Stewart
Netflix Inc.
100 Winchester Circle
Los Gatos, CA 95032
USA
Email: rrs@netflix.com

John-Mark Gurney
Consultant
Oakland, CA
USA
Email: jmg@freebsd.org

Scott Long
Netflix Inc.
100 Winchester Circle
Los Gatos, CA 95032
USA
Email: scottl@netflix.com

Abstract—**Transport Layer Security (TLS) is becoming increasingly desirable and necessary in the modern Internet. Unfortunately it also induces heavy penalties on application CPU performance for both the client and server. In this paper we examine the server-side performance implications on CPU computational and data-movement overhead when enabling TLS on Netflix's OpenConnect Appliance (OCA [1]) network. We then explore enhancements to FreeBSD to reduce the costs that TLS adds when serving high volumes of video traffic. Finally we describe recent changes and future improvements to FreeBSD's OpenCrypto Framework that can be used to further improve performance.**

I. Introduction

Transport Layer Security [2] (TLS) is becoming an operational requirement in today's unfriendly Internet. It provides both encryption and authentication to any application that enables it; but as with many improvements it also comes at a high cost in terms of additional CPU cycles. Up until recently Netflix has not enabled TLS on its OpenConnect Appliances (OCA).

An OCA is a FreeBSD-based appliance that serves movies and television programming to Netflix subscribers. Confidential customer data like payment information, account authentication, and search queries are exchanged via an encrypted TLS session between the client and the various application servers that make up the Netflix infrastructure. The actual audio and video content session is not encrypted. At first glance, this might seem like a glaring oversight, but the audio and video objects are already protected by Digital Rights Management (DRM) that is pre-encoded into the objects prior to them being distributed to the OCA network for serving. The addition of TLS encryption to these objects was previously not considered to be a high priority requirement.

Evolving market forces as well as the changing landscape of the internet [3] have caused us to re-evaluate our view on TLS. The computational cost of TLS serving is high, so with this in mind Netflix launched a small pilot project to explore what impacts enabling TLS would have on its products. We also started to examine recent innovations in FreeBSD for ways that we might be able to reduce the costs of TLS.

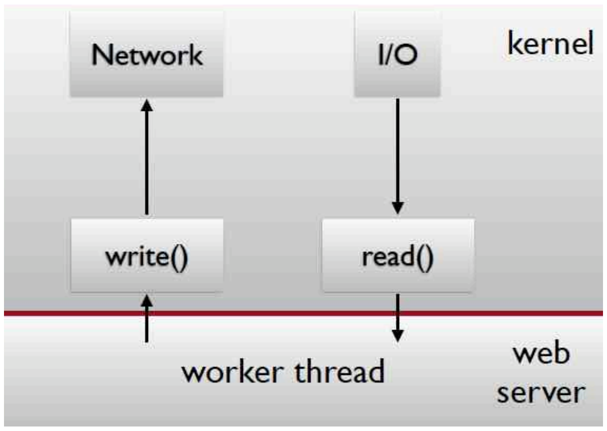

Fig. 1. Classic Web Serving

II. The idea

The Netflix OpenConnect Appliance is a server-class computer based on an Intel 64bit Xeon CPU and running FreeBSD 10.1 and Nginx 1.5. Each server is designed to hold between 10TB and 120TB of multimedia objects, and can accommodate anywhere from 10,000 to 40,000 simultaneous long-lived TCP sessions with customer client systems. The servers are also designed to deliver between 10Gbps and 40Gbps of continuous bandwidth utilization. Communication with the client is over the HTTP protocol, making the system essentially into a large static-content web server.

A traditional web server will receive a client request for an object stored on a local disk, allocate a local buffer for the object data via the malloc(3) library call, then issue a read(2) system call to retrieve and copy the contents of the object into the buffer, and finally issue a write(2) system call to copy the buffer contents into a socket buffer which is then transmitted to the client. This process usually involves two or more data copies handled directly by the CPU as well as some associated consumption of CPU cache and memory bandwidth. This simple data flow model (see Fig 1) works well and is easily maintainable for low-bandwidth needs, but is taxing on the CPU for high bandwidth applications. Early tests in the OCA development process showed that the server

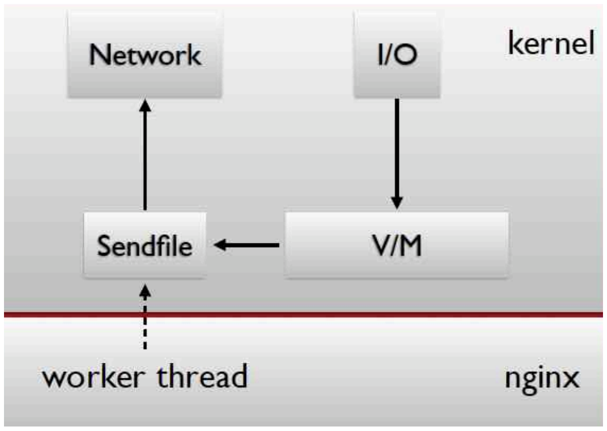

Fig. 2. Optimized Nginx Web Serving

was typically unable to serve more than 4-5Gbps of bandwidth under this model. Over time, more advanced hardware has brought with it improvements in serving performance, but the simplicity of this model continues to penalize the potential performance.

Nginx employs a novel solution that uses the sendfile(2) system call to perform a zero-copy data flow from disk to socket (see Fig 2). When the client request comes in, Nginx issues the sendfile call on both the file descriptor and the socket descriptor. The kernel then retrieves the object into its VM page cache subsystem and passes a set of pointers to those cache pages to the socket buffer for transmission. No additional buffers or data copies are required, and the data never leaves the kernel memory address space. This relieves the CPU of the burden of doing copies, greatly reduces pressure on memory cache and bus resources, eliminates a system call, and shortens the code pipeline for the data path. Nginx improved this model by utilizing the AIO subsystem to asynchronously prime the VM cache with disk reads prior to issuing the sendfile call. This avoided forcing the Nginx thread to wait on the disk I/O to be performed synchronously by sendfile, but it increased the complexity of data flow and risked that data might be flushed from the cache before it could be put into the socket. Even with this complexity, the savings resulted in 20% or more better performance on the same hardware.

Netflix further improved this model by making the sendfile system call asynchronous [4]. This allows Nginx to dispatch multiple requests for multiple clients in parallel, with each request proceeding on its own as disk I/O completes. This further optimized the data path and allowed Nginx to scale to more simultaneous connections.

Regular sendfile, along with faster hardware, allowed us to jump from less than 10Gbps to 30Gbps of service capacity per server. Moving to asynchronous sendfile allowed us to achieve 40Gbps and beyond on the same hardware. It's important to note that this scheme also lends itself to inserting other operations into the disk-to-socket data flow without requiring complex modifications to the web server or requiring that the

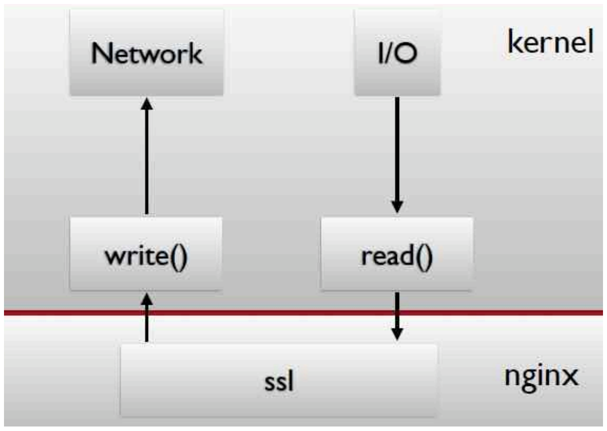

Fig. 3. Classic SSL Web Serving

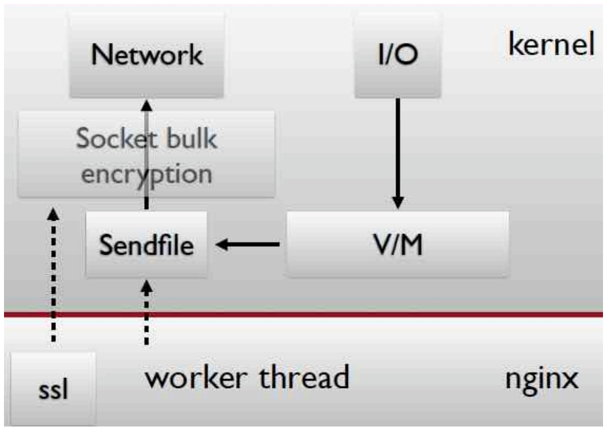

Fig. 4. In-Kernel SSL Web Serving

data be copied into the application space.

TLS functionality has traditionally been performed in the application space as a library to the web server (see Fig 3). As the web server prepares objects for transmission to the client, they are passed to the TLS library for encryption, after encrypting the data the TLS library writes it to the socket handle. Along with this bulk encryption functionality, the TLS library also handles session setup, management, and key generation and exchange with the client (See RFC5288 [5] for details on TLS). This scheme fits well into the traditional simple data flow model presented above. Unfortunately, it is incompatible with the sendfile model since that model does not allow the data to enter the application space. The processing overhead of TLS combined with the loss of the zero-copy optimization of sendfile resulted in our serving capacity dropping by a factor of 2.5x to 3x in our initial testing, whereas other business partners who were not using sendfile were seeing a drop of only 2x in theirs.

In order to retain the benefits of the sendfile model but also implement TLS functionality, we designed a hybrid TLS scheme whereby session management would stay in the application space, but the bulk encryption would be inserted into the sendfile data pipeline in the kernel (see Fig 4). TLS session negotiation and key exchange messages are passed from Nginx to the TLS library, and session state resides in the library's application space. Once the TLS sessions is set up and appropriate keys are generated and exchanged with the client, those keys become associated with the communication socket for the client and are shared into the kernel. For sockets that are flagged for encryption, the TLS bulk encryption happens, via a trip through the Open Crypto Framework, as the data pages are added to the socket buffer. For sessions that require encryption services not available in the Open Crypto Framework, Nginx reverts to the traditional read+send model and performs the encryption in the application space. This ultimately allows for a modular selection of encryption protocols and algorithms that best suit the requirements of the client and the available resources of the server.

III. Recent changes to FreeBSD Open Crypto Framework

The Open Crypto Framework provides a consistent API to software and hardware-based cryptographic engines. It features a modular design that allows the support of new accelerators, such as the AES-NI processor instructions [6], without changing the original code. Though the Open Crypto Framework was usable as-is, there were a number of issues in both performance and algorithm support that were required to be fixed before it was ready to be deployed.

A. Sessions

The Framework was originally written to accelerate Internet Protocol Security [7] (IPsec). The code was written assuming that there would only be a small handful of security associations (SA). For each SA there is a key, and the Framework requires a session for each key. For a normal IPsec gateway, there are only a few SAs and hence only a few sessions. For each client connection there will be a distinct key, requiring an individual Framework session. Therefore on a busy server the number of sessions will be very large (as much as 20-40,000 connections).

When a session is created, an integer session identifier is used. This isn't necessarily a bad thing if an efficient data structure is used to handle looking the identifier up. Though a linked list ($\mathcal{O}(n)$) was used, so it was a major performance issue. Modifications were made such that a pointer to the session is used instead of the identifier allowing the bypass of the lookup stage.

B. Cipher Mode Support

Since the Framework was originally imported, only the ciphers Camila (2007) and AES-XTS (2010) have been added. In 2008, AES-GCM was standardized as part of TLS in RFC5288 [5]. In previous modes, an HMAC using either SHA1 or SHA256 was used to authenticate that the data had not been tampered with. The AES-GCM cipher mode is an Authenticated Encryption with Associated Data (AEAD) cipher which integrates both encryption and authentication. AES-GCM uses AES-CTR for encryption and GMAC for authentication. The mode allows encrypting and authenticating in a single pass avoiding walking the data a second time.

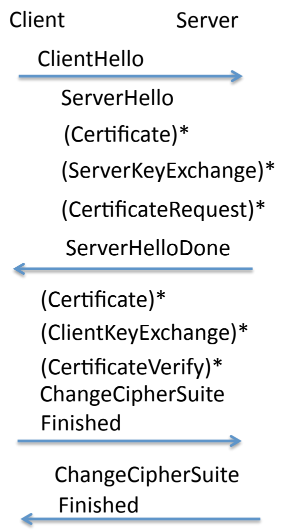

Fig. 5. An initial TLS exchange

The addition of the AES-GCM cipher when combined with AES-NI and CLMUL [8] instructions available on modern AMD64 processors allow encryption and decryption at around 1GBps per core.

IV. Harnessing the Open Crypto Framework from within the kernel

How does one go about using the sendfile() call with AESNI and TLS? First one must consider the actual handshake flow as seen in Fig 5; note that the messages with the '*" are optional. The key messages that we wish to focus on are the ChangeCipherSuite (CCS) and Finished message. The CCS message is the "last" message that uses the previous cipher (the initial cipher is the NULL cipher). Sending it causes the internal state machine within TLS to change so that the next message output (the Finished message) is encrypted with the new cipher. This presumes that the keys were generated by the sender and sent down into its encryption mechanism at that time as well (this is what all of the previously exchanged messages in the diagram were used for: generating the keys). What is interesting to note here is that each side sends the CCS when it is ready, when you consider that with the fact that messages in the TCP stream may arrive out of order, adding TLS for both sending and receiving adds a lot of complexity to the kernel.

Normally the client's CCS and Finished message arrives before the server sends its CCS and Finished message. This has some distinct implications for something running in the kernel that is going to be encrypting and sending data over a TCP connection or receiving and decrypting data. On the receiving side, its possible that the keys are not quite in place when messages that are encrypted with those keys arrive which means the kernel must buffer the data for some time until the keys are available for use. On the sending side the transmission of the CCS must be coordinated with the engaging of the new cipher for the next message. Other questions also abound here such as:

1) Will the kernel need to process all the various forms of key exchange to generate the keys?
2) How will the kernel handle re-keying?
3) Not all ciphers are supported by the Open Crypto Framework, so how does one interwork with TLS library to cover ciphers not yet supported by the kernel?

After studying the problem for quite some time the approach decided upon was to exclude decryption (the receive side) and defer that work for some future time. Since the main goal was to be able to continue to use the sendfile system call, one could narrow the problem down to a sender side problem only. This then would simplify the design making the actual impact to the kernel much smaller. To handle the handshake, rekeying and ciphers not yet supported, it was decided to lean heavily on the TLS library. The design that took shape was to let all of the key exchanges and normal SSL processing occur as usual. When the keys were ready, have the TLS library send them to the kernel and let the kernel do the encryption part, while all the other parts of TLS would continue to be executed by the TLS library. The TLS library would continue to frame its messages and submit framed but un-encrypted messages to the kernel. The kernel would then use the keys given earlier to encrypt and send the data. For the sendfile system call, the kernel would add in a framing layer as well as the encryption. This would then allow the TLS library to do all of the processing of keys and TLS state and to handle re-keying, should one of the sides initiate that procedure and yet still allow the sendfile system call to operate.

To make all of the pieces come together a number of changes would be needed:

- Changes to the kernel to add two socket options. One which tells the caller if the requested cipher is supported and the second which accepts the keying material and starts the kernel looking for the CCS message. Once the CCS message is seen by the kernel, the new cipher suite would then be engaged for subsequent sends. Note that once the socket options are enabled any message sent without a TLS framing would be considered invalid and would cause an error return to the caller.
- Importing to our FreeBSD 10 stable kernel the new changes that have been made to the Open Crypto Framework.
- Changes to the TLS library to use the new feature, setting in the keying material and not encrypting data that needed to be sent to the peer. When making this change we would only attach the changes to specific ciphers that we wished to support. Note that any TLS library could have been used and modified, but for our purposes we choose to work with OpenSSL since it is readily available in FreeBSD.

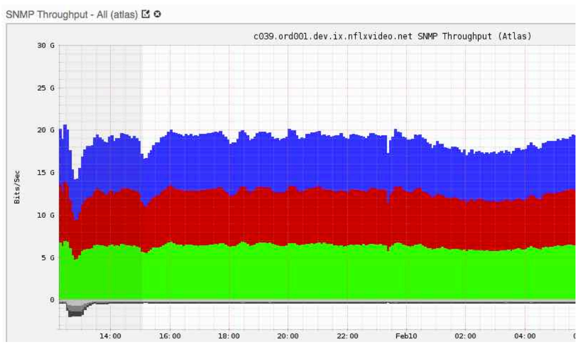

Fig. 6. OCA Rev D Performance under normal (no TLS) load

- Changes to the application so that if the OpenSSL library indicated it was acceptable to use sendfile, the application (nginx) would use the new SSLsendfile call instead of reading and writing the data to the OpenSSL library.

V. Results

First, let us look at a typical OCA that serves content. For our testing we will use a standard flash OCA called a ”Rev D”. A Rev D is an Intel AMD64 class machine with an E5-2650L processor, it is considered a ”Flash” box since it holds 40 Gigabytes of flash and has 64 Gigabytes of main memory.

We used three of these machines, serving real Netflix traffic, to gather results for our paper. First, lets look at how an OCA preforms that is not serving any TLS, and is instead handling current requests for service. We see in Fig 6 the result of a typical night of transfer of Netflix content to its subscribers (the same evening our TLS OCAs were running). Note that in general a Rev D will serve, during peak, around 19-20 Gbps of traffic as we see in the figure.

After making the changes described above to the FreeBSD kernel, OpenSSL library and nginx, we then put the new image into production on the same night to measure the impact of using TLS as well as using TLS with our enhancements.

To allow for precise testing, we added a sysctl so that we could use the exact same firmware image. The sysctl would either allow all hardware based ciphers that we support (chiefly AESNI based ciphers) to return that SSLsendfile was allowed or (when disabled) always return that SSLsendfile is not allowed.

For our tests (both with our enhancement and without) we configured nginx in such a way so that it prefers the AESNI ciphers and the most preferable cipher is AESNI with GCM (Galios Counter Mode). AES-GCM, as mentioned earlier, allows us to do only one pass over the data for both the encryption and the authentication.

The first results from our TLS enabled machines can be seen in Fig 7 this shows a plot of throughput of one of our OCAs with the feature completely disabled yet configured to serve

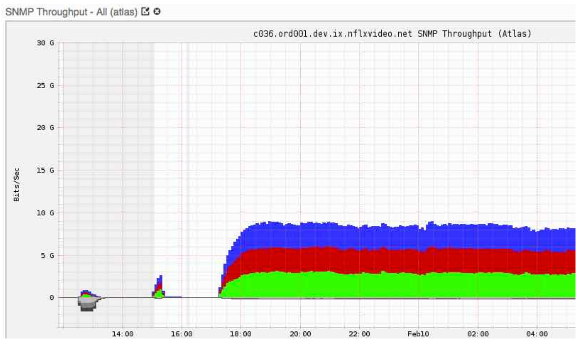

Fig. 7. OCA Rev D Performance using just OpenSSL

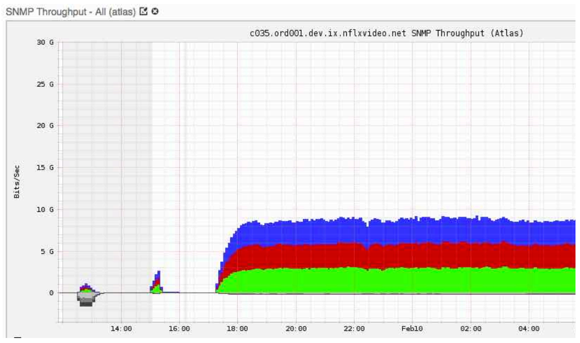

Fig. 8. OCA Rev D Performance using SSL sendfile

TLS traffic and shows our Rev D serving about a 8.5GPS. This is a drastic performance difference from our standard Rev D serving 19-20GPS of traffic. But this does give us a base line for how much performance we can expect from nginx when combined with OpenSSL. In Fig 8 we see the result of enabling the SSLsendfile feature, here we server about 9Gps of traffic instead of the 8.5Gps with OpenSSL. This gives us a slight performance improvement, but not nearly as much as we had hoped for. So why did we not see the performance improvements we had hoped for? We currently believe that there are several reasons our performance gains were less than we had expected:

- When we were looking at our initial performance change (from not using sendfile as compared to using sendfile) a large part of the work load that we observe with SSL is not in place (the encryption). This means that any gain seen without SSL is a smaller subset of the overall performance of the machine. Since SSL is such a big part of the new workload we will have a less beneficial gain from being able to keep the data in the kernel in comparison to the overall workload.
- Due to the current nature of the Open Crypto Framework, we must perform an extra copy of all data being encrypted. The Open Crypto Framework currently encrypts in place, any input. This means that data coming from the disk and used by sendfile will be ”read-only” mbufs that must be copied before passing into the Open Crypto Framework.
- Normally the kernel does not use floating point state, so during context switches into the kernel, often times, floating point state is not saved. This, of course, is not true for the AESNI code which uses some of the floating point registers. This then makes the kernel version require added state saving when it comes into context as compared to just plain Open SSL doing the encryption.

VI. Future directions and Improvements

Having completed our initial study and integrated the changes into the Netflix OCA code base, we are busy planning how we can further improve the performance we obtain by using sendfile. As noted above we have several things which we can improve that will help reduce the cost of encryption within the kernel. These include:

1) Enhancing the SHA256, 384 and 512 code so that AESNI with Cipher Block Chaining (CBC) type of encryption is faster. Currently the added HMAC has a huge performance penalty that the standard OpenSSL library does far more efficiently. Increasing the performance of the SHA algorithms may give us additional performance.
2) Elimnate the extra copy by expanding the Open Crypto Framework so we can pass in an additional pointer and flag to indicate to the Open Crypto Framework not to encrypt in place but to instead encrypt to the output pointer instead.
3) Enhance the way floating point saves occur, this will involve both optimizing FPU state saves as well as allowing for multiple encryption operations after the state has been saved, instead of the current operation where every request undergoes a FPU save and restore state.
4) Currently AES-CBC and SHA are not able to be fully pipelined. If multiple streams are processed at once, the operations can be pipelined, allowing additional performance increases.

We will also be exploring the addition of offload cards and other assist mechanisms using hardware acceleration. The advantage of continuing to use the Open Crypto Framework is that any work we do using it should be able to integrate nicely into what hardware support is currently (and becomes) available to FreeBSD.

Adding encryption and authentication no matter how you slice it will impact performance, as can be seen in our results. Most of our studies so far have shown a minimum of a 50 percent drop in performance when TLS is turned on (if not larger). Hopefully, our future enhancements will make some inroads against that number and lessen the overall cost of TLS. Since we, like many others, fully believe that TLS is the right way to move forward when it comes to securing communications between two internet applications.

Acknowledgment

The authors would like to Gleb Smirnoff for his help on nginx, Warner Losh for his support and reviews, Robert Watson for helping us narrow the problem, and Alex Gutarin for being supportive of our small team.

References

[1] Netflix, "Netflix Open Connect", http://openconnect.itp.netflix.com/openconnect/index.html, August 2014

[2] T Dierks, E Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", *RFC 5246*, August 2008

[3] E Snowden, "Edward Snowden: The ten biggest revelations", http://mashable.com/2014/06/05/edward-snowden-revelations, August 2014

[4] G Smirnoff, "New non-blocking on disk I/O implementation of sendfile() syscall for FreeBSD (in Russian)", http://people.freebsd.org/~glebius/articles/sendfile.pdf, December 2014

[5] J Salowey, A choudhury, D McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS", *RFC 5288*, August 2008

[6] S Gueron, "Intel Advanced Encryption Standard (AES) Instruction Set", April 2008

[7] S Kent, K Seo, "Security Architecture for the Internet Protocol", *RFC 4301*, December 2005

[8] S Gueron, M Kounavis, "Intel Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode", April 2008