Securing Traffic Tunnelled over TCP or UDP

Sowmini Varadhan,
Oracle Corporation,
Redwood City, CA
sowmini.varadhan@oracle.com

Abstract

The Linux kernel today has a number of tunnelling mechanisms for the cloud such as VXLAN, Geneve and GUE that typically tunnel packets over UDP. In addition we also have have socket types like PF_RDS [19] and the newly proposed PF_KCM [6] socket which tunnel over TCP. All of these technologies tunnel application and/or tenant payloads over a Layer 4 protocol in the Network stack.

In all these cases, the TCP or UDP socket is a kernel socket that is transparently created in the kernel. In many cases such as RDS and KCM, the data that is tunneled comes from HTTP or Database applications, and the tunneling solution needs to satisfy security goals for integrity protection, privacy and authentication.

In this paper we focus on the use-case of kernel-managed TCP and UDP sockets and evaluate two approaches for securing the application data sent or received on these sockets. One approach considered is by using TLS based keys for encrypting and decrypting packets at the kernel socket layer. The second approach is by using IPsec at the IP layer. Factors considered for the evaluation are the network layers that are secured by each approach, complexity of implementing each protocol in the kernel, and impact on network performance.

Keywords

TLS, Kernel, KSSL, IPsec, RDS, Layer 3 tunneling, AAA

Introduction

Recent advances in network virtualization have resulted in a number of tunnelling mechanisms for the Cloud, such as VXLAN, Geneve, GUE etc. The Linux kernel implementation of these protocols is typically achieved by having tenants in Virtual Machines send down frames to a virtual network device (e.g., the NVE in [17] or the VTEP in [13]) that then tunnels the frame over a TCP or UDP socket transparently created in the kernel.

In addition to this model, there are other modes for tunneling application data over TCP, such as those used by Reliable Datagram Sockets [19] or the Kernel Connection Multiplexor [6]. In these cases, the application creates a datagram socket with a new protocol family such as PF_RDS or PF_KCM, and data sent or received on this socket is tunnelled over TCP. The application is thus able to use datagram sockets semantics, while the TCP based tunneling provides guaranteed, reliable, ordered delivery.

In a Cloud topology where multiple tenant networks are involved, and where packet paths can traverse long distances over the Internet, the tenant and/or application payload needs to be encrypted (with appropriate authentication) for privacy. Typically this would be achieved by using TLS [2] or DTLS [18] at the TCP/UDP socket layer from user-space.

However, trying to apply TLS/DTLS to application data that is sent over kernel managed TCP and UDP sockets faces some challenges.

- The TCP or UDP socket is transparently created in the kernel and is not exposed to the application or tenant. As a result the application or tenant cannot control many of the parameters associated with TLS, or even whether TLS needs to be applied to its data.
- When the application is using something other than a TCP or UDP socket (e.g., RDS sockets with [19], or KCM sockets [6]), unmodified versions of commonly used user-space libraries for TLS such as `gnutls` or `openssl`, which only operate on TCP or UDP sockets, cannot be directly used by the application.
- The TLS control plane is complex, and there is no support for TLS/DTLS on kernel managed sockets in the Linux kernel.

In order to use TLS/DTLS with TCP/UDP sockets in the kernel, it would be necessary to implement some variant of the TLS protocol in the kernel itself. Attempts to implement subsets of TLS have been made in BSD and Solaris, and we evaluate the contents and conclusions from those attempts in this paper.

TLS/DTLS encrypts, and provides privacy for, application data, but it does not secure the TCP or UDP connection itself. TCP connections, in particular, are vulnerable to a number of attacks as documented in [23]. A multi-tenant Cloud/Cluster environment that has the potential to traverse long Internet paths needs to secure itself against such attacks.

IPsec [9], [14] provides an alternative solution for securing both the application data and the TCP/IP connection itself. IPsec provides encryption and authentication at the IP layer and is integrated into the Linux kernel. IPsec can be used in either the Transport Mode, where packet contents beyond the IP header are encapsulated, or in Tunnel Mode, where the IP

header itself is encapsulated and tunnelled as another IP or UDP packet.

Each of these solutions impacts throughput and latency, in different ways. This paper evaluates the applicability of each of these solutions for kernel-managed TCP and UDP sockets from the perspective of the type of security guarantees for each solution, the implementation complexity and the resulting performance profile.

We start by investigating the possibility of using a security solution based on TLS.

TLS based cryptography in the kernel

Transport Layer Security (TLS) [2] is a cryptographic protocol that sits on top of the Transport layer of the TCP/IP stack, and allows the participating applications to encrypt their data using a cipher and symmetric key that is negotiated as part of the initial TLS Handshake. The Handshake is initiated by the client which presents a list of cipher-suites to the server. The server selects a cipher-suite from that list, and sends back its public encryption key and a Certificate Authority (CA) to the client. The client can then confirm the identity of the server. It computes the Master Secret to be used for the exchange and sends this key back to the server after encrypting it with the server's public key. After this point, all data between the client and the server is encrypted using the session key which is generated by applying a pseudo-random function (PRF) and applying it to the Master Secret.

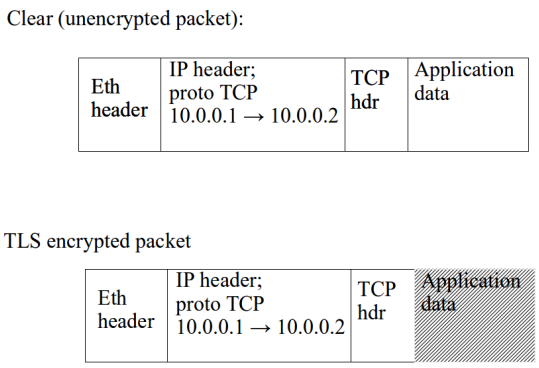

Figure 1: Wire format of unencrypted and TLS encrypted TCP/IP packet. Encrypted sections are shown in gray.

Figure 1 shows the format of a packet after encryption using TLS. An important point to note is that TLS (or DTLS) only secures the application data. The TCP (or UDP) and IP headers are sent in the clear, and are not protected by TLS (or DTLS).

TLS is a complex protocol with several provisions for mid-stream changes to the encryption parameters to provide robust security in case of key compromise. For example, the TLS protocol allows mid-session changes to the ciphering strategies via the ChangeCipherSpec protocol option. The ChangeCipherSpec (CCS) message is sent by both the client and the server to notify the peer that subsequent records will be protected under the newly negotiate CipherSpec and keys. Once the CCS has been sent, the TLS specification mandates that the new CipherSpec MUST be used. There are other similar requirements in the TLS specification that contribute to the complexity of TLS.

Due to its complexity, it would be preferable to avoid importing the TLS state machine and control-plane into the kernel, if that were possible.

We now describe efforts in BSD, Solaris and Linux to implement TLS based cryptography in the kernel. Each implementation takes a different approach to handling the complexity of TLS. The BSD and Linux proposals take the approach of splitting TLS into a user-space control plane, and a kernel data-plane, whereas the Solaris effort implements a stripped down version of the TLS state machine in the kernel.

BSD sendfile() acceleration An attempt to improve `sendfile()` throughput of encrypted data for the Netflix OpenConnect Appliance (OCA) is described in [21]. The OCA is a FreeBSD based web-server that handles and responds to incoming client-requests for objects stored on the local disk. Responses from the server to the client may need to be encrypted, and the objective in [21] is to try to accelerate the transaction by doing a zero-copy data flow from disk to socket by doing the encryption in the kernel. In order to achieve this, [21] investigates a proposal whereby TLS cryptographic parameters are negotiated in user-space and pushed to the kernel. These parameters are then used for encryption/decryption on kernel sockets used in the `sendfile()` path for the FreeBSD-based OCA server.

The Netflix OCA acceleration separates the user-space "control plane" which negotiates TLS session parameters, and the kernel "data plane" that uses these encryption parameters for data-transforms. The TLS control messages are no longer in-stream with the data-messages, which is a departure from a basic assumption of the TLS protocol, thus new types of asynchrony between the TLS state machine and the kernel encryption state may now be encountered. One example described in [21] is the case when encrypted messages arrive at the receiver before the keys needed to decrypt those messages are in place in the kernel. On the sender, the CCS transmission from the control-plane has to be coordinated with the usage of the new cipher for the next text message in the stream. An interesting comment in [21] with respect to the CCS handling in TLS is "when you consider [CCS processing] 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".

The scope of the Netflix/OCA investigation is limited to the evaluation of the performance acceleration for an encrypted `sendfile`. As a result, [21] is able to restrict itself to a subset of the client-side state-machine of the TLS protocol that avoids many of the more complex protocol features such as re-keying.

The Netflix/OCA investigation finds that the actual performance improvement from using the simplified BSD kernel implementation of client-side TLS to encrypt data for

`sendfile()` is only slightly better than the baseline solution of doing the TLS encryption in user-space. The actual bottlenecks that limit performance on BSD come from extra `bcopy`'s of the data being encrypted, and other factors such as context switching overhead resulting in loss of floating point state, and general system workload.

Even if the performance inhibiting factors were mitigated, the proposal in [21] does not constitute a complete security solution for the problem of providing AAA for kernel-managed TCP and UDP sockets such as those used by RDS, KCM and VXLAN. All of the asynchronicity issues arising from the split-TLS model would need to be resolved for both the sender and the receiver.

Kernel SSL proxy for Solaris An alternative "Kernel SSL" was attempted in the Solaris Operating System as part of the Kernel Secure Sockets Library (KSSL) feature [11]. The goal behind KSSL was to provide an in-kernel SSL proxy to accelerate web-server performance. In order to achieve this goal, Solaris implemented a simplified version of the server-side of the TLS state machine in the kernel.

As with the Netflix `sendfile()` study, the primary goal was to achieve improved performance for a specific micro-benchmark, rather than to provide a generic kernel infrastructure for TLS encryption of TCP payloads. One of the optimizing assumptions made by KSSL was to implement support for a small subset of cipher-suites in the KSSL TLS state machine. The subset was selected based on cipher-suites that were commonly encountered in web-server transactions at the time. Client requests for cipher-suites outside this subset, and other TLS features such as Mutual Authentication, would be punted to, and handled in, user-space.

While KSSL provided better performance for the specific SPECweb99 benchmark and platforms targeted at the time of its inception, its design and simplifying assumptions made it an easy candidate for obsoletion. The slight performance benefit gained by doing in-kernel SSL (and thus optimizing memory access) was quickly obsoleted when alternative solutions such as `stunnel` using AES-NI from user-space were able to offer competitive performance with better flexibility and support for a richer set of cryptographic algorithms. Solaris KSSL is EOF'ed in Solaris 12.

Linux kTLS proposal A proposal to implement TLS for kernel sockets is presented in [12]. This proposal is a hybrid of the BSD and Solaris approaches. It splits the TLS protocol into a user-space control plane that negotiates TLS parameters, and a kernel data-plane that does encryption/decryption with those parameters. Only the AES-NI crypto algorithms would be handled in the kernel, all other cipher-suite requests would be punted to user-space. The proposal does not clearly specify how CCS messages would be correctly handled in this split model. Additional deployment feedback and performance numbers are pending for this proposal.

Evaluation of kernel TLS implementations The Linux and BSD proposals split TLS processing into a control-plane and a data-plane, which is not what the TLS protocol intended. This split can give rise to various types of asynchrony where control and data plane messages arrive in different permutations, with the potential of inter-operability issues between different TLS implementations, and the risk of TLS specification violations.

For the BSD and Solaris case-studies, the primary motivation was to investigate performance improvements for a micro-benchmark, rather than to provide a complete security solution for TLS in the kernel.

In both Solaris and BSD, the complexity of the TLS control plane necessitated the need for simplifying assumptions. Each implementation reports some performance improvement to be gained by moving TLS into the kernel for the specific scope of its study, but that performance improvement is achieved by implementing a subset of the TLS protocol itself, with the assumption that there would be a fall-back mechanism to handle other aspects of TLS that are not covered by the optimization.

TLS/DTLS provides its security guarantees from the socket layer, atop the TCP/UDP layers of the Networking stack. One benefit in being able to do this from a kernel-managed TCP/UDP socket would be that the kernel socket would obtain the same performance profile from the underlying stack infrastructure as a user-space TCP or UDP socket that uses TLS or DTLS today.

However TLS/DTLS does not protect the TCP/UDP connection itself. Privacy, integrity protection and authentication for the L4 protocol is critical for some encapsulation protocols like RDS, where attacks to the TCP layer can jeopardize HA requirements for the Cluster. For example any of the attacks described in [23] would result in loss of RDS message reassembly state, and could trigger a reconnect and message retransmit storm.

Problems with the split-TLS approach The Netflix/BSD study recognizes that the separation of the TLS control-plane from the data-plane opens up new possibilities for asynchrony between the two planes and makes many simplifying assumptions to obtain its published results. While these assumptions may be valid for the scope of the proposed performance investigation in [21], these points of asynchrony between the control- and data-planes would have to be resolved to provide a complete security solution for the problem of securing traffic being tunnelled over kernel-managed TCP or UDP sockets in a multi-tenant Cloud/Cluster.

None of the three implementations addresses the additional complexity of High Availability and Service Migration/Failover that is a critical feature for Cloud and Cluster deployments where kernel managed TCP and UDP sockets are used. In the case of PF_RDS, a kernel TCP socket may receive a TCP RST for a number of reasons, such as address/service failover, module/node restart, or other errors at the peering node. It is possible to have a man-in-the-middle TCP attack such as a spoofed RST on either the control- or the data-plane TCP socket. In the case of RDS, the kernel TCP socket tracks liveness of its peer, and will attempt to reconnect to any new incarnation using parameters specified by the protocol. With a split-TLS model, this would imply that the data-plane has reconnected, but it may not always also be associated with a corresponding re-establishment of the TLS control plane. In general, reconnect at the data-plane should

be accompanied by a re-authentication and cryptographic re-negotiation for the new connection, which requires more coordination and kernel-user upcalls in the split-TLS model.

Authentication, privacy and integrity protection for L3 and L4 layers is provided by the IPsec suite of security protocols, which is implemented with an encrypting data-plane in the kernel that addresses all of the asynchronicity issues mentioned above. In the next section, we examine the feasibility of using IPsec to solve the security requirement for protocols that tunnel over TCP/UDP in the kernel.

IPsec

IP Security (IPsec) is a suite of security protocols that operates at the IP layer (Layer 3 of the network stack) and defines protocols for key negotiation, authentication and encryption. Authentication is provided by the optional addition of an Authentication Header defined RFC 4302 [8], and end-to-end payload encryption and decryption may be achieved by adding the ESP header RFC 4303 [9]. When used in combination with the Internet Key Exchange Protocol (IKE [5]), the Security Associations and Security Policies can be managed in a number of flexible ways through existing user space application.

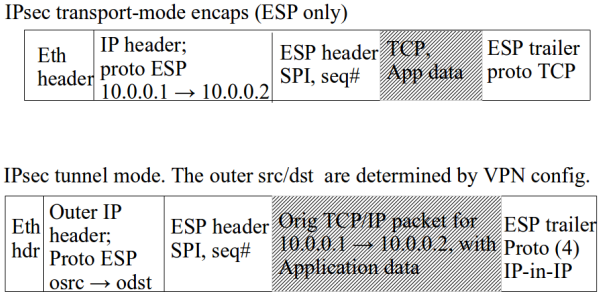

Figure 2: Wire format of a packet encrypted in IPsec transport and tunnel modes. Encrypted sections are shown in gray. When a VPN gateway is not used (typical in Cloud/cluster topologies), the outer IP source and destination addresses will be identical to the inner IP addresses

IPsec may be used in one of two modes:

- the "transport mode", where the IP payload is encrypted and/or authenticated without impacting the L3 routing information, or,
- "tunnel mode", where IP packet (header and payload) is itself encrypted/authenticated, and encapsulated into a new IP packet that is tunneled IPsec in the tunnel mode is typically used to create Virtual Private Networks (VPN).

Figure 2 shows the format of the encrypted packet for each of these two modes.

In the case of Cloud/NVO3 mechanisms, as well as Clustered database/server environments such as those that use RDS-TCP, the routing information at L3 for the packet is determined by external entities such as Cloud Controllers for the CLOS network, or by the Cluster topology for RDS. Although IPsec in tunnel mode can be used to secure the IP header itself, the use-case for Cloud/Cluster topologies is for securing host-host connections and a VPN gateway is not typically involved. IPsec transport mode, meets the anticipated security expectations of these use-cases by protecting the TCP or UDP header as well as payload.

IPsec is tightly integrated with the Linux TCP/IP stack. The benefit of the tight integration is that there is full and mature support for all parts of the IPsec in the control plane. The integration with IKEv2 provides flexibility for various key distribution modes. The maturity of IPsec architecture implies that the separation of the IKE control from the data-plane is well-understood. IKE negotiated parameters apply smoothly across all socket connects and re-connects, so that HA and failover are seamlessly handled. Thus IPsec would provide all the security assurances needed for kernel-managed TCP and UDP sockets.

IPsec meets all the security requirements of kernel TCP/UDP sockets, but we need to ensure that IPsec usage does not add unreasonable performance overhead. Application of cryptographic transforms will add some unavoidable latency to packet processing for any security solution. However, in addition to this cost, since IPsec modifies network stack headers, there is a risk that the performance profile with IPsec enabled may diverge significantly from the kernel fast-path performance profile. Any such divergence needs to be understood and minimized.

Three significant features available in the Linux network stack that contribute to performance today are

1. hardware offload features such as TSO
2. receive side flow hashing
3. software offload features such as GSO and GRO

We shall now briefly describe each of these features in detail, in order to understand their interaction with IPsec.

Hardware offload

In order to increase outbound throughput while also reducing CPU overhead, the TCP/IP stack leverages from NIC support for TCP Segmentation Offload (TSO) when it is available. TSO works by sending down large buffers to the NIC, and allows the hardware to split these buffers into TCP segments that are then sent on the wire.

IPsec transforms operate on the TCP header, thus IPsec processing must be applied after the segmentation operation. In order to offload this operation to the hardware, the network stack needs to convey the transform parameters (SA, keys etc) to the NIC, i.e., both TSO and IPsec offload would have to be enabled. The Linux stack today does not support IPsec offload, thus the insertion of the ESP header results in an implicit disabling of segmentation offload.

Note that this constraint also applies to other forms of TCP/IP header protection, including TCP-MD5 [3] and TCP-AO [22]

Many of the commonly deployed 10GbE NICs from Intel (Niantic, Twinville and Sageville) support IPsec offload for both transmit and receive paths today. However, the feature is only enabled for Windows [16] because the Linux stack does not have DDK interfaces needed for IPsec offload today. Providing equivalent APIs and DDKs to leverage from IPsec offload features supported by the NIC will help boost IPsec performance while reducing CPU utilization.

Flow hashing

TCP and UDP flows are identified by the 4-tuple for the connection, i.e., the flow is represented by some hash of the source and destination addresses and ports. This flow hash is used by the NIC to support multiple receive and transmit descriptor queues to allow efficient distributed processing across CPUs to achieve hardware based Receive Side Scaling (RSS) [20].

Application of ESP by IPsec [7] results in the encryption of the TCP header, so the port information is no longer available to the NIC. However, IPsec uses the Security Parameter Index (SPI) to specify a Security Association (SA) which is described in RFC 4301 (Section 4.1) as "The SA is a simplex connection that affords security services to the traffic carried by it". In the case of L3 traffic tunneled over TCP or UDP, the SA can be specified as the 4-tuple, thus the SPI uniquely identifies the flow for the receiver.

The Linux stack supports a software analog of RSS as part of Receive Packet Steering (RPS) [20] that can be used as a fall-back when the NIC does not support RPS. Another software feature critical to network performance is Receive Flow Steering (RFS), which also takes into account application locality, and attempts to steer packets where the application thread consuming the packet is running.

In our experiments with iPerf, the unavailability of RSS and RPS due to TCP header encryption proved to be a critical barrier to network throughput and manual placement of IRQs and `iPerf` process binding was needed to improve the performance.

Both RPS and RFS use the flow hash to identify flows, and in comparison to RSS, have the advantage of being software implementations that can be easily extended to use the SPI for hashing ESP flows.

Software offload

Just as RPS and RFS provide software analogs of hardware based RSS, the Generic Segmentation Offload (GSO) and Generic Receive Offload (GRO) provide software equivalents of TSO and its receive-side counterpart, Large Receive Offload (LRO).

GSO is based on the observation that the major savings behind TSO comes from the reduction of multiple network stack traversals for small segments to one traversal for a super-packet. The key concept behind GSO lies in postponing segmentation to the latest possible point in packet processing. Segmentation of the super-packet to MSS-sized TCP segments is done just before passing the packets to the drivers `xmit` routine.

GRO uses concepts similar to GSO on the receiver. Incoming packets that have the same MAC headers, and for which there is a match on the flow-identifying fields of the L4 and L3 headers are merged at reception time and passed up the stack.

IPsec offload in software can be hooked into the GSO framework by applying the IPsec transform after GSO breaks down the super-packet into MSS-sized TCP segments. Steffen Klassert describes one such proposal in [10] that also addresses GRO. Decapsulation and decryption are done at L2 using ESP GRO callbacks, and packets are re-injected into `napi_gro_receive()`. We describe the performance profile observed before and after the application of this concept in the next section where we describe results from iPerf experiments to measure throughput.

Results

We focussed our comparisons on single-stream iPerf [4] performance to get a fair comparison of throughput and latency between unencrypted and encrypted TCP flows in the absence NIC based Tx/Rx flow hashing based on port numbers. The platform used for this investigation was an Intel X5-4 with the 10 Gbe ixgbe driver on a point-to-point 10G link. The Linux kernel version used for these experiments was a 4.4.0 release-candidate and the ixgbe version/firmware reported by `ethtool` was 4.2.1-k/0x800004be. We measured throughput and peak CPU utilization for the scenarios listed below.

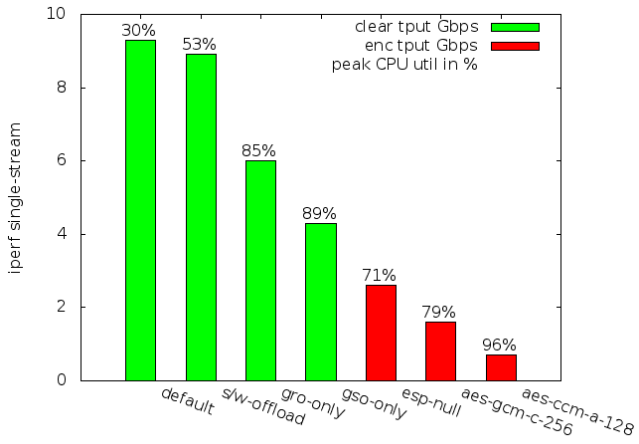

Figure 3: iPerf single-stream results for clear and encrypted traffic on an X5-4 using the ixgbe driver. An encryption algorithm of "none" implies that IPsec was not used, whereas "null" indicates Null encryption, i.e., an ESP header was added to the packet, but all traffic was sent in the clear.

1. Default settings, i.e., TSO, GSO, GRO, Checksum offload enabled, with clear (no encryption) traffic
2. s/w offload only: Using `ethtool`, explicitly disable TSO and checksum offload for clear traffic, but GSO is still enabled on sender, GRO on receiver
3. GRO-only: Using `ethtool` explicitly disable both TSO and GSO at the sender, for clear traffic, leaving only GRO at the receiver

4. GSO-only: Using `ethtool`, explicitly disable TSO and checksum offload on sender, disable GRO on receiver, for clear traffic. GSO is still enabled on sender
5. Default settings for hardware offload, using IPsec with NULL encryption. Thus even though no cryptography is actually done, and the packet contents are "clear" for packet sniffers like `tcpdump`, an ESP header is inserted between the IP and TCP header
6. Default settings for hardware offload, using IPsec AES-GCM (Galois Counter Mode) with a key length of 256, an an ICV length of 16
7. Default settings for hardware offload, using ipsec AES-CCM (Counter with CBC-MAC) with a key length of 128, an an ICV length of 8

The instrumentation of IPsec with NULL encryption was selected to profile the software latencies of the IPsec code-paths in the absence of any overhead from the cryptographic transforms. Of the encryption algorithms chosen, AES-GCM [15] is considered to be the most efficient NIST standard Authenticated Encryption scheme. The CCM mode [1] is similar to the GCM mode with some benefits that make it suitable for hardware implementation (smaller gate count), though it cannot be parallelized easily, thus implementations tend to be slower than GCM.

It can be seen from Figure 3 that the disabling of segmentation-offload, either explicitly in step 3 above, or implicitly through the insertion of an ESP header, results in an exponential drop in throughput. There is also an increase in CPU utilization with the loss of hardware offload, suggesting that at least two of the critical bottlenecks impacting performance are the loss of offloads in both software and hardware.

Our first approach to try to recover some of the lost performance was through software IPsec offload. We applied Steffen Klassert's patch described in [10] and found that the performance for the NULL-crypto case was vastly improved. The results reported in Figure 1 show that software offload of IPsec processing to GSO/GRO brings the throughput to 8 Gbps which is significantly closer to the maximum throughput for clear traffic when GSO and GRO are in effect. The GCM-256 case is also able to benefit from the IPsec software offload, but the computation is CPU bound in this case, due to the cost of crypto, suggesting that IPsec hardware offload may be needed for this case.

	throughput Gbps (peak CPU util)	
	esp-null	AES-GSM-256
Baseline	2.6 (71%)	2.17 (83%)
GSO/GRO offload	8 (100%)	4.2(100%)

Table 1: Effect of IPsec offload to GRO/GSO using early prototype of patches from Steffen Klassert. All numbers listed above were obtained using iPerf, with manual IRQ balancing to ensure that irqs and iPerf processes were pinned to discrete CPUS

Figure 3 also shows that there is a drop in throughput between the case of clear traffic with TSO/GSO disabled (case 3 above) and the case of NULL-crypto (case 5 above) for the baseline. Inspection of the IPsec code paths in the Linux kernel was done as part of this investigation, and some of the memory management operations in the ESP path were identified. A very expensive operation performed as part of each call to `esp_output` is an `skb_cow_data` invocation to copy the entire packet over to the tail of the `sk_buff`. Results reported in Table 1 include optimizations to the output path was modified to avoid the `skb_cow_data` when the `skb` is not cloned/shared.

Other areas where the software paths can be further optimized to reduce the number of memcpy/memmove of data are currently under investigation.

Conclusions and future work

We have examined two different approaches to encrypting TCP/UDP payloads, the first by using user-space implementations of TLS to compute the crypto key for the kernel, and the second by using IPsec in the conventional manner.

The TLS approach has the property that it does not secure or modify the TCP/IP header. This is both beneficial, as it does not deter any performance features in the stack, such as segmentation offload or receive-side flow steering, and problematic, as it does not address the security issue of providing AAA for the L4 headers.

A big drawback to TLS/DTLS is that it is not available in the kernel, and has a complex control-plane for negotiating session parameters that is based on the assumption that the control-plane and data-plane are all synchronized on the same TCP/UDP stream. Importing the TLS/DTLS control plane into the kernel is undesirable due the complexity of the TLS protocol, but separating control and data-planes is highly risky, as it splits the protocol in ways that the TLS design does not intend, and introduces new modes of asynchronicity that will have to be addressed/maintained in the network stack.

In comparison, IPsec is a well-established protocol that already supports control/data-plane separation, with a wide variety of tools and protocols to manage the control-plane state without compromising security. IPsec offers the flexibility of securing both L3 and L4 headers based on the usage mode.

However, network stack performance with IPsec may need to be improved in the Linux kernel. The software stack today relies heavily on segmentation offload for performance, and a first step for improving IPsec performance is to enhance the GSO/GRO infra-structure to support IPsec offload in software.

The performance of the networking stack with hardware offload of IPsec will always result in lower CPU utilization than the equivalent software implementation. Thus adding the ability to leverage from hardware IPsec offload in addition to TSO where possible will help manage the unavoidable overhead of cryptography for IPsec. We are currently investigating the feasibility of doing IPsec offload to hardware on Linux for some Intel NICs and for the IP over InfiniBand case using Mellanox NICS.

On the receive side, a few simple changes to the NICs are needed to support RX flow hashing based on SPI. i The problem of updating NICs to allow the usage of the SPI for Rx-side load-balancing is fairly simple to solve, and at the cur-

rent time, we are working with Intel and Mellanox architects to achieve this.

Securing the control plane protocols that tunnel over TCP/UDP should not happen at the expense of avoidable performance cost in the Linux Kernel, and this paper attempts to propose some feature enhancements to achieve that goal.

Acknowledgments

We would like to thank Steffen Klassert for sharing an early prototype of the GRO/GSO offload code used for this investigation, and James Morris for his help with reviewing this paper. Linden Corbett and Anjali Singhai from the Intel Fortville Driver team provided the information on NIC IPsec offload capabilities on Intel NICs. Boris Pismenny provided the equivalent information for Mellanox InfiniBand NICs.

References

[1] 2004. Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality. http://csrc.nist.gov/publications/nistpubs/800-38C/SP-800-38C.pdf.

[2] Dierks, T., and Rescorla, E. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard). Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685.

[3] Heffernan, A. 1998. Protection of BGP Sessions via the TCP MD5 Signature Option. RFC 2385 (Proposed Standard). Obsoleted by RFC 5925, updated by RFC 6691.

[4] iPerf - The network bandwidth measurement tool. https://iperf.fr/.

[5] Kaufman, C.; Hoffman, P.; Nir, Y.; Eronen, P.; and Kivinen, T. 2014. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296 (INTERNET STANDARD). Updated by RFC 7427.

[6] KCM: Kernel connection multiplexor . http://thread.gmane.org/gmane.linux.network/378365.

[7] Kent, S., and Seo, K. 2005. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard). Updated by RFCs 6040, 7619.

[8] Kent, S. 2005a. IP Authentication Header. RFC 4302 (Proposed Standard).

[9] Kent, S. 2005b. IP Encapsulating Security Payload (ESP). RFC 4303 (Proposed Standard).

[10] strongswan: the opensource ipsec-based vpn solution. http://lists.openwall.net/netdev/2015/12/02/56.

[11] kssl(5) kssl, KSSL - kernel SSL proxy . http://docs.oracle.com/cd/E23824_01/html/821-1474/kssl-5.html.

[12] [RFC PATCH 0/2] Crypto kernel TLS socket . https://lkml.org/lkml/2015/11/23/634.

[13] Mahalingam, M.; Dutt, D.; Duda, K.; Agarwal, P.; Kreeger, L.; Sridhar, T.; Bursell, M.; and Wright, C. 2014. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348 (Informational).

[14] McGrew, D., and Hoffman, P. 2014. Cryptographic Algorithm Implementation Requirements and Usage Guidance for Encapsulating Security Payload (ESP) and Authentication Header (AH). RFC 7321 (Proposed Standard).

[15] M.Dworkin. 2006. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) for Confidentiality and Authentication. http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf.

[16] IPsec Offload Version 2 . https://msdn.microsoft.com/en-us/library/windows/hardware/ff5569965

[17] Narten, T.; Gray, E.; Black, D.; Fang, L.; Kreeger, L.; and Napierala, M. 2014. Problem Statement: Overlays for Network Virtualization. RFC 7364 (Informational).

[18] Phelan, T. 2008. Datagram Transport Layer Security (DTLS) over the Datagram Congestion Control Protocol (DCCP). RFC 5238 (Proposed Standard).

[19] RDS Reliable Datagram Sockets . https://oss.oracle.com/projects/rds/dist/documentation/rds-3.1-spec.html.

[20] Scaling in the linux networking stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[21] Stewart, R. 2015. Optimizing TLS for High-bandwidth Applications In FreeBSD. Technical report, Netflix Inc., 100 Winchester Circle Los Gatos, CA 95032. https://people.freebsd.org/ rrs/asiabsd_2015_tls.pdf.

[22] Touch, J.; Mankin, A.; and Bonica, R. 2010. The TCP Authentication Option. RFC 5925 (Proposed Standard).

[23] Touch, J. 2007. Defending TCP Against Spoofing Attacks. RFC 4953 (Informational).

Author Biography

Sowmini Varadhan is a Consulting Software Engineer in the Mainline Linux Kernel Group at Oracle Corporation. where she works on projects spanning Kernel Networking, Distributed Computing, and Performance.