TLS Offload in the Kernel

BY JOHN BALDWIN

FreeBSD 13.0 adds support for Transport Layer Security (TLS) socket kernel offload. TLS offload permits the kernel to send and receive data over a socket using TLS where the TLS framing and encryption is performed in the kernel rather than userland. Kernel TLS (KTLS) requires changes to userland SSL libraries, the kernel's network stack, and, in some cases, device drivers.

Transport Layer Security

TLS is an application layer protocol designed to provide authentication and privacy to upper layer application protocols. TLS is structured as a stream of records, or frames, transmitted and received over a stream protocol. (There is a version of TLS for datagram protocols, but KTLS only supports TLS over TCP.) Each TLS record contains a header with a message type and length. This record layer is used to transport messages defined by the TLS protocol to manage the TLS connection as well as application data messages that carry data from an upper layer application protocol such as HTTP, SMTP, or IMAP.

A typical TLS connection begins with a few TLS protocol messages used to establish a TLS session. These messages are used to verify the identity of the other end of the connection, choose a set of encryption and authentication algorithms (known as a **cipher suite**), and establish a pair of ephemeral session keys used to encrypt and authenticate subsequent TLS records. Once the connection is established, the connection switches to passing the upper layer application data via application data messages. These application data messages constitute the bulk of the messages in a TLS connection.

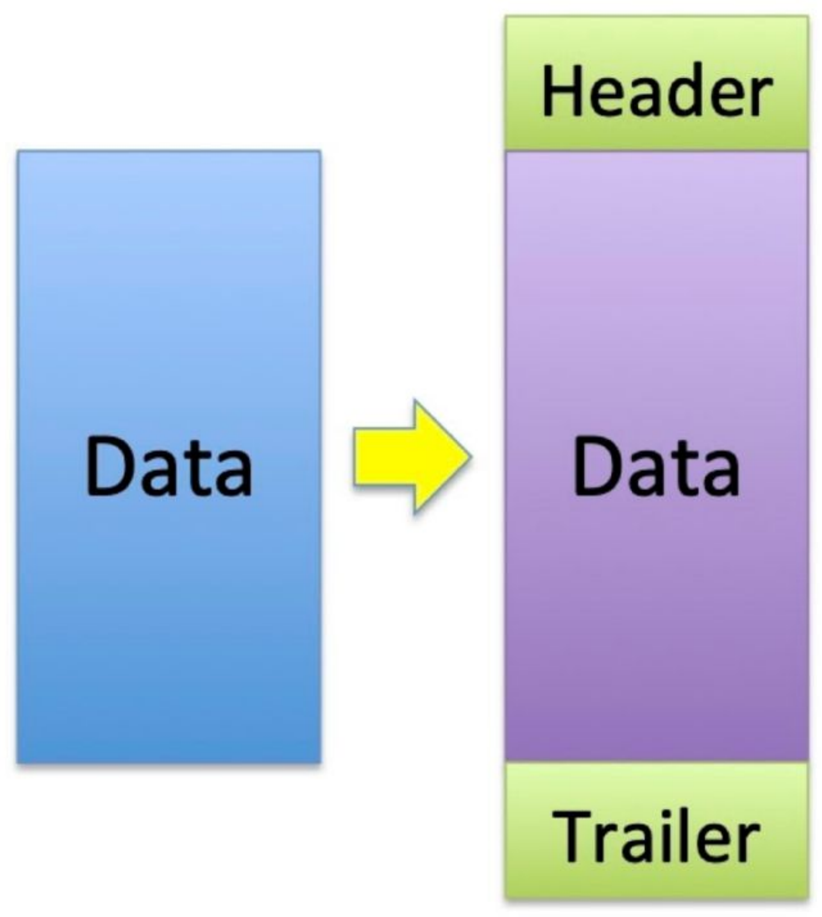
Application data messages are built from variable-sized blocks of application data. The application data is encrypted and headers and trailers are added to form a TLS record (see Figure 1). TLS protocol messages are constructed similarly except that the TLS header contains a different message type and the payload data of the TLS record comes from the TLS protocol. The format of the TLS header and trailer are determined by the specific cipher suite used by the session.

Why Move TLS into the Kernel?

Normally we avoid putting code into the kernel when possible. User code runs with less privilege and is isolated from other processes. A crash in a user process only affects the process, and vulnerabilities are constrained to a specific process. Kernel code runs with more privilege and is able to access all data in the system. Bugs in kernel code have more significant consequences. Given that, why add the additional complexity of TLS to the kernel? The reason, as with most other code that resides in the kernel, is performance.

There are two main justifications for KTLS. The first is to avoid extra copies of data in and out of the kernel. The second is to enable the use of TLS offload in network interface cards.

Figure 1: Constructing a TLS Record



Avoiding Data Copies

The initial work on KTLS was motivated by regaining the zero-copy performance of `sendfile()` when using HTTP over TLS. Prior to the advent of `sendfile()`, the typical workflow of FTP and HTTP servers required reading the contents of files into buffers in a user process via `read()` followed by a call to `write()` to send the data out through a socket (see Figure 2). This resulted in three copies of the data: one in the kernel's buffer cache (or VM page cache) associated with the file, a second in temporary buffers in userland, and a third in the socket buffer. The `sendfile()` system call instructs the kernel to send a portion of a file's contents over a socket. With this higher-level request, the kernel is able to reuse the copy of the file data in the buffer cache directly in the socket's send buffer eliminating the additional copies of the file data (see Figure 3).

Figure 2: Pre-sendfile() HTTP/FTP Workflow

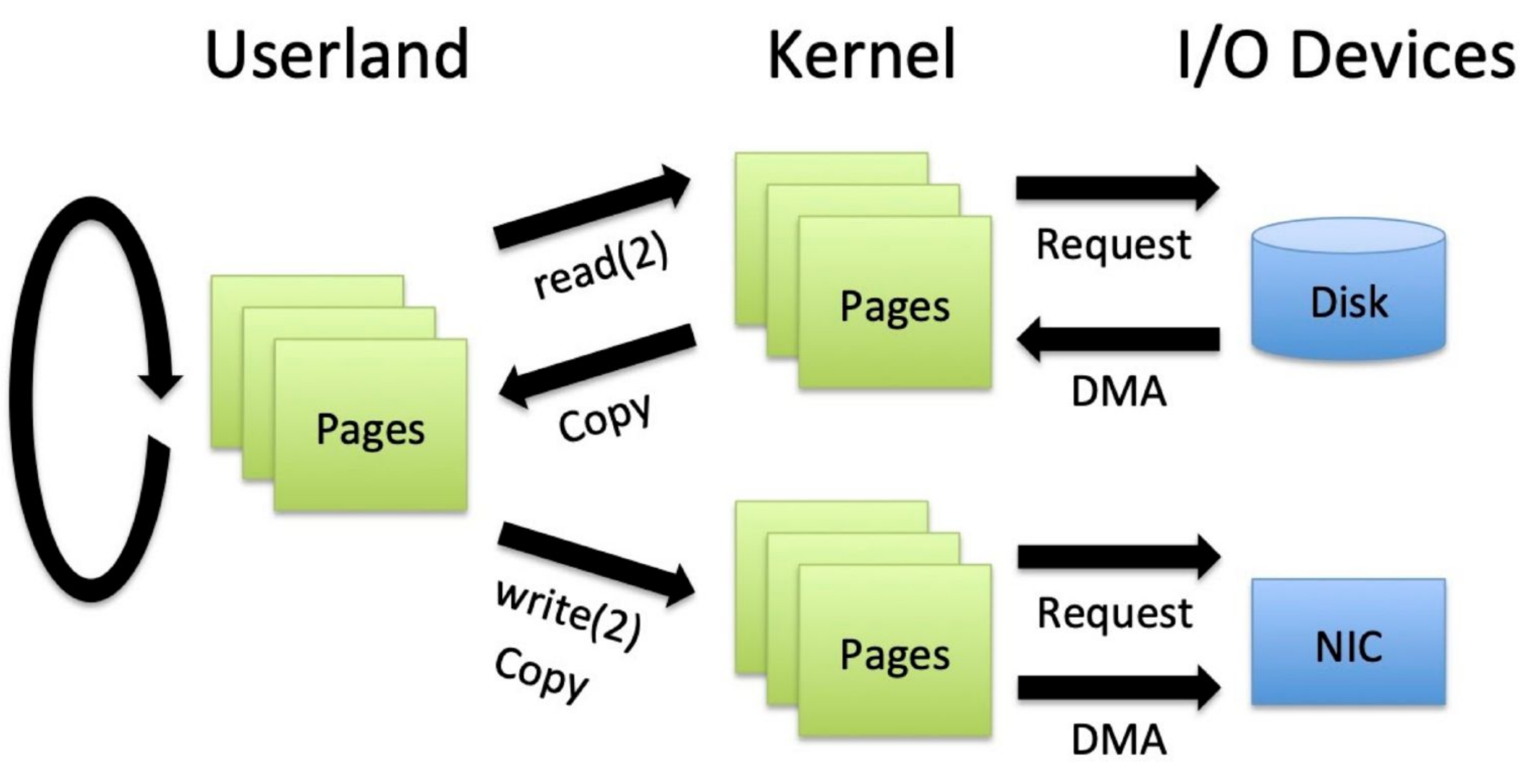
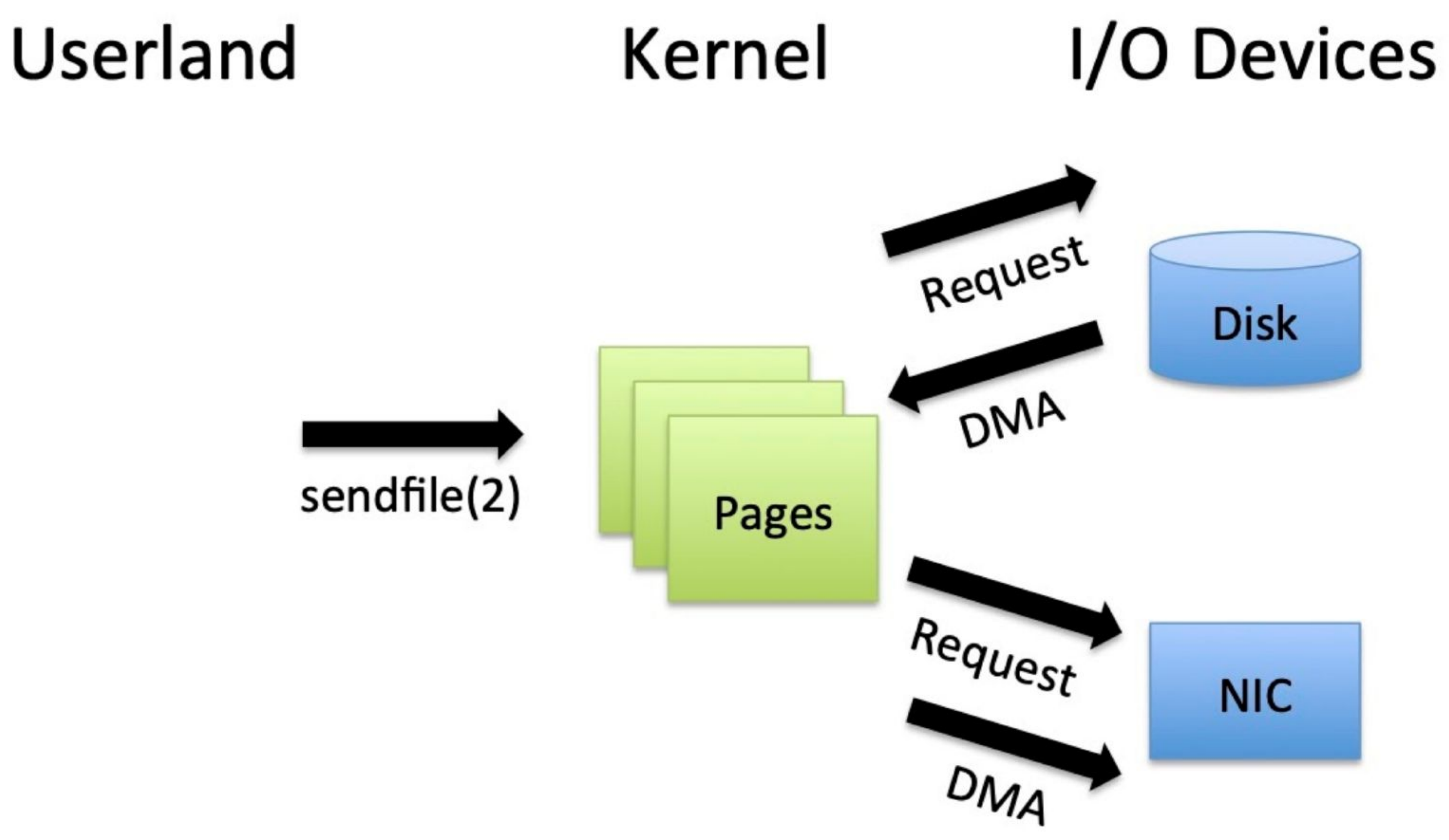
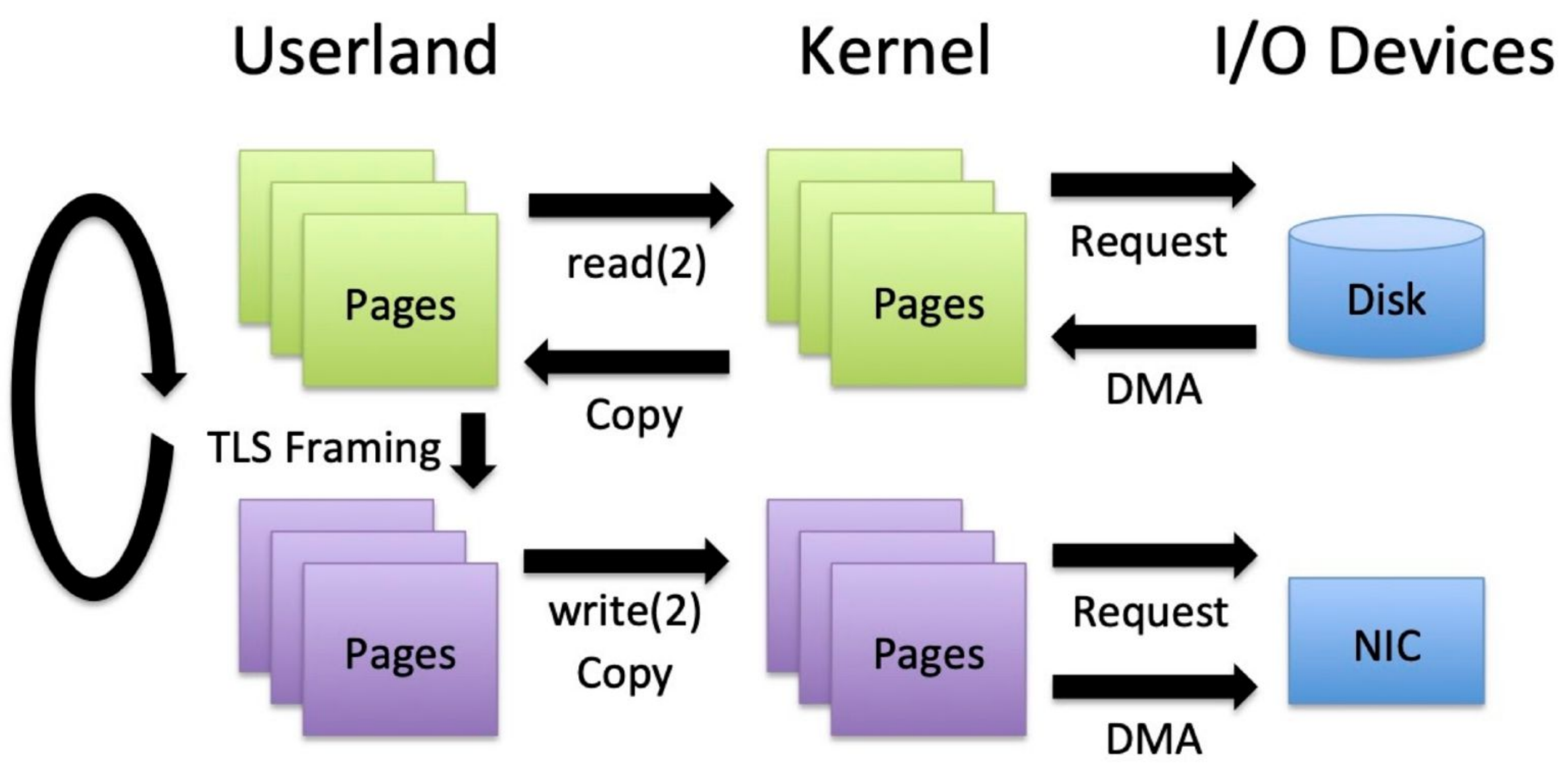


Figure 3: sendfile() HTTP/FTP Workflow



With HTTP over TLS, the `sendfile()` system call can no longer be used as-is. The data sent over the socket is not an exact copy of the file's contents. Instead, the file data must be encrypted and encapsulated in TLS frames. With TLS framing performed in userland, this requires a return to the pre-`sendfile()` workflow. Data is copied from the buffer cache into temporary buffers in userland where it is encrypted and framed. This modified data is then copied into the kernel's socket buffer (see Figure 4). By moving TLS awareness into the kernel, we can avoid one or both of these extra copies.

Figure 4: HTTPS Workflow with User TLS



A similar issue exists with a more recent workflow: NFS over TLS. The NFS client and server run in the kernel. They directly move data for files between the buffer cache and socket buffers. While the client and server do copy data between the buffer cache and socket buffers, performing TLS in userland would require additional copies into and out of userland.

TLS Offload in NICs

A couple of recent Ethernet cards including the Chelsio T6 and Mellanox ConnectX-6 Dx include support for inline encryption and decryption of TLS records. For transmit, this permits the host to send unencrypted data to the NIC, and the NIC will encrypt the data and split it into multiple TCP segments before transmitting the data on the wire. Similarly, for the receive case the NIC assembles multiple TCP segments into a TLS record and supplies the decrypted TLS record to the host. The driver is responsible for providing the session keys to the NIC to permit the inline encryption and decryption.

KTLS Requirements

These use cases have similar requirements. `sendfile()` requires the kernel to have control over TLS framing including sequence numbers. Since the sequence number is an input into the authentication algorithms used in TLS, this in turn requires the kernel to manage the encryption of all TLS frames on the socket for `sendfile()`. Encrypting all TLS frames requires userland to provide unencrypted data for all TLS records to the kernel. NFS over TLS similarly requires full control over TLS framing in the kernel. TLS offload in NICs requires access to unencrypted data for all TLS frames on the socket.

In short, these use cases require moving the TLS record layer into the kernel. However, none of these use cases require moving the handling of TLS protocol messages, such as those used to establish a TLS session and negotiate session keys, into the kernel. As a result, KTLS moves the handling of the TLS record layer into the kernel but leaves the management of TLS protocol messages in existing userland SSL libraries.

This is similar to the approach used with IPsec where key negotiation is performed by userland daemons, but individual packets are encrypted and decrypted in the kernel. For IPsec, the keys are negotiated over a separate, side-band connection and are shared across several connections. For TLS, the keys are unique to a connection and are negotiated within the connection via TLS protocol messages. In addition, TLS records and IPsec packets are encrypted and authenticated by many of the same algorithms.

KTLS Implementation

KTLS in the kernel consists of three main components. TLS sessions store information about one direction of a TLS connection including the cipher suite used, session keys, and a backend used to encrypt or decrypt TLS records. KTLS also hooks into the network stack transmit and receive paths. While the transmit and receive paths do share some properties, they are implemented differently.

TLS Sessions

TLS sessions manage the encryption and decryption of TLS records. The send and receive sides of a TLS connection are managed independently with separate TLS sessions for transmit and receive. Each session is described by a `struct tls_session` object and contains information about the session such as the version of TLS, the cipher suite, and the encryption and authentication keys. Sessions are also associated with a backend which performs the encryption and decryption.

Three different types of session backends are supported. Software backends encrypt and decrypt TLS records while they are present in the socket buffer. They require no knowledge of TLS outside of the socket layer. Specifically, protocols such as TCP and device drivers do not require any changes to support software backends. NIC TLS backends encrypt and decrypt TLS records in a network card as part of transmitting or receiving packets. This requires cooperation with the protocol layers as well as explicit support in device drivers. Finally, TOE TLS backends work similarly to NIC TLS except that they leverage TOE support to manage TCP state management such as retransmits.

KTLS generally treats the send and receive sides of a TLS connection independently. An individual connection may only offload one side instead of both. In addition, an individual connection may use different types of backends for each side. For example, connections may offload TLS transmit in the kernel while handling TLS receive in the userland SSL library, or a connection may use NIC TLS to offload TLS transmit and a software backend to handle TLS receive.

TLS Transmit

The initial work on KTLS focused on offloading encryption of transmitted TLS records. HTTPS server workloads of static content generally transmit significantly more data than they receive. As a result, the biggest performance gains for these workloads come from offloading encryption of transmitted data rather than decryption of received data.

When TLS transmit is offloaded to the kernel, userland applications always provide unencrypted data to the kernel. Data sent via system calls such as `write()` or `sendfile()` are split into separate TLS records by the kernel. These TLS records always use the application data message type. Userland can use the `sendmsg()` system call with a `TLS_SET_RECORD_TYPE` control message to send individual TLS records with a custom type and size. For these requests, the contents of the data described by scatter/gather list in the message header is sent in a single

TLS record with the message type given in the control message. This is used by the SSL library to send TLS protocol messages such as handshake messages and alerts.

A TLS transmit session is created by a `TCP_TXTLS_ENABLE` socket option. A userland SSL library invokes `setsockopt()` with this option supplying an instance of `struct tls_enable` as the option value. This structure includes pointers to the session keys and a description of the negotiated cipher suite and TLS protocol version. The socket option handler in the kernel creates a new TLS session object and probes for a backend. `TOE TLS` and `NIC TLS` backends are probed by calling down to the device driver for the NIC associated with this connection. If the NIC driver does not support TLS or is not able to offload the connection, the handler searches for a software backend. If no backend is found, the `setsockopt()` system call fails and the SSL library continues to perform TLS encryption in userland. If a backend is found, a reference to the new TLS session is saved in the socket's send buffer. Existing data in the socket send buffer is transmitted as-is, but all subsequently written data to the socket buffer is encapsulated in TLS records and encrypted.

TLS mbufs

Each transmitted TLS record is described by a single `struct mbuf`. TLS mbufs use a new **external pages** mbuf added in FreeBSD 13. External pages mbufs do not store payload data in a virtually contiguous buffer pointed to by `m_data`. Instead, these mbufs contain an array of physical address pointers to one or more pages in RAM. This does mean that the traditional way to access mbuf data, the `mtod()` macro, cannot be used with these mbufs. Kernel code paths using these mbufs were extensively audited for code using `mtod()`.

These mbufs were first added to improve the performance of `sendfile()`. Previously, `sendfile()` used a separate mbuf for each page in a file. With external pages mbufs, a single mbuf can describe multiple pages. This permits a smaller number of mbufs to describe the same amount of file data in a socket buffer. Fewer mbufs means fewer cache misses when walking the linked list of mbufs in the socket buffer which improves performance.

Network interface device drivers can choose to support transmitting these types of mbufs by advertising support for the NOMAP capability. For device drivers which use existing `bus_dma` routines to map mbufs without examining the contents of mbufs being transmitted, no further changes are needed apart from setting `IFCAP_NOMAP` in both `if_capabilities` and `if_capenable` during their attach routines. Note that a device driver must be able to offload checksums for packets containing these mbufs, but checksum offloading is a widely supported feature in NICs. If a device driver does not support external pages mbufs or it does not support necessary checksum offloading, then the network stack will convert external pages mbufs into a chain of conventional mbufs before passing them to the device driver.

TLS mbufs extend external pages mbufs to store TLS-specific information such as the TLS record header and trailer and a reference to a TLS session. Device drivers which support external pages mbufs must also support transmitting the data stored in the TLS record header and trailer fields. However, for most device drivers this does not require any specific changes since existing bus_dma routines already handle these fields.

The `ktls_frame()` function sets TLS-specific information for TLS mbufs. This function uses the TLS session associated with the socket's send buffer to construct the record. It stores a reference to this TLS session in the mbuf and populates the TLS header of the record including any explicit IV or nonce. Finally, this function computes the length of the TLS trailer accounting for any required padding. The TLS trailer's contents are not set until the TLS record is encrypted but setting the trailer length ensures that the mbuf reserves the correct amount of space in TCP sequence numbers.

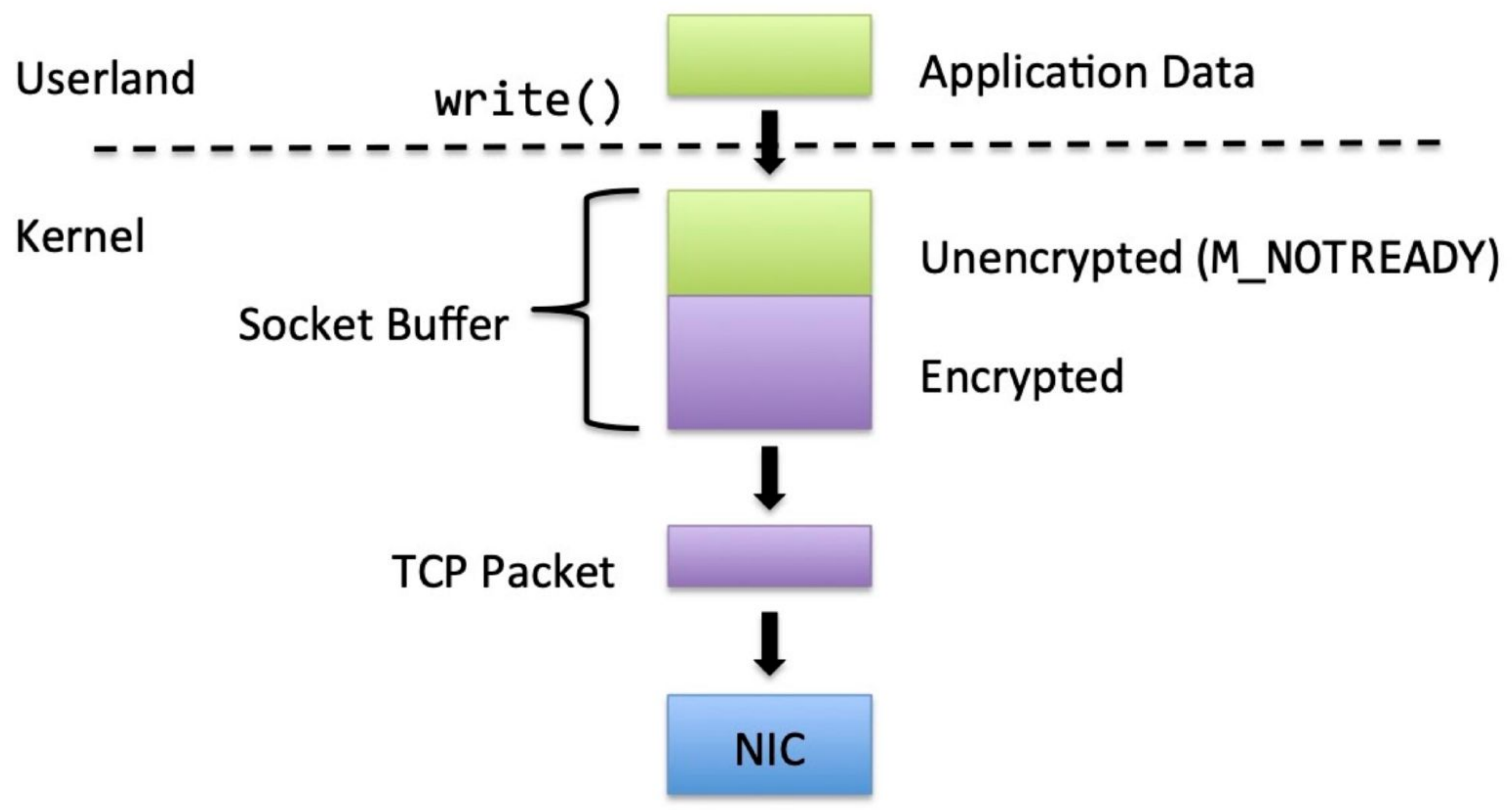
Software TLS Transmit

When software backends are used, TLS transmit encrypts TLS records in the socket buffer before they are released to the transport protocol. In the socket buffer this relies on the existing `M_NOTREADY` mbuf flag.

The `M_NOTREADY` mbuf flag is used to mark mbufs in a socket buffer that do not yet contain valid data. These mbufs still reserve space in the socket buffer to provide backpressure to whatever is producing data, but they cannot be used by the socket buffer consumer. The `sendfile()` system call uses these mbufs to reserve space in a send socket buffer for the regions of a file that are not already in memory at the time of the system call. Instead, disk I/O requests are scheduled to populate these missing pages. When the I/O requests complete, the mbufs are marked ready by clearing the `M_NOTREADY` flag, and the protocol is notified that new data is available to send from the socket buffer.

Software TLS transmit reuses this same framework to handle TLS records to differentiate unencrypted TLS records from encrypted TLS records. When an unencrypted TLS record is queued to the socket buffer by a system call such as `write()` or `sendfile()`, the `M_NOTREADY` flag is set on the mbuf describing the TLS record by `ktls_frame()`. In addition, the mbuf is placed on a queue of unencrypted TLS records. A worker pool of threads (one per CPU) services this queue. The worker threads invoke the software backend to encrypt each TLS record. Once the TLS record has been encrypted, the mbuf associated with the TLS record is marked ready (see Figure 5). At this point, the TLS mbuf is now a "regular" external pages mbuf. None of the network stack from the protocol layer down to the device driver has to perform any additional work to support TLS when using a software backend. The caveat about external pages mbufs being converted to chains of conventional mbufs from above still apply, but that can be trivially mediated in most device drivers by supporting external pages mbufs.

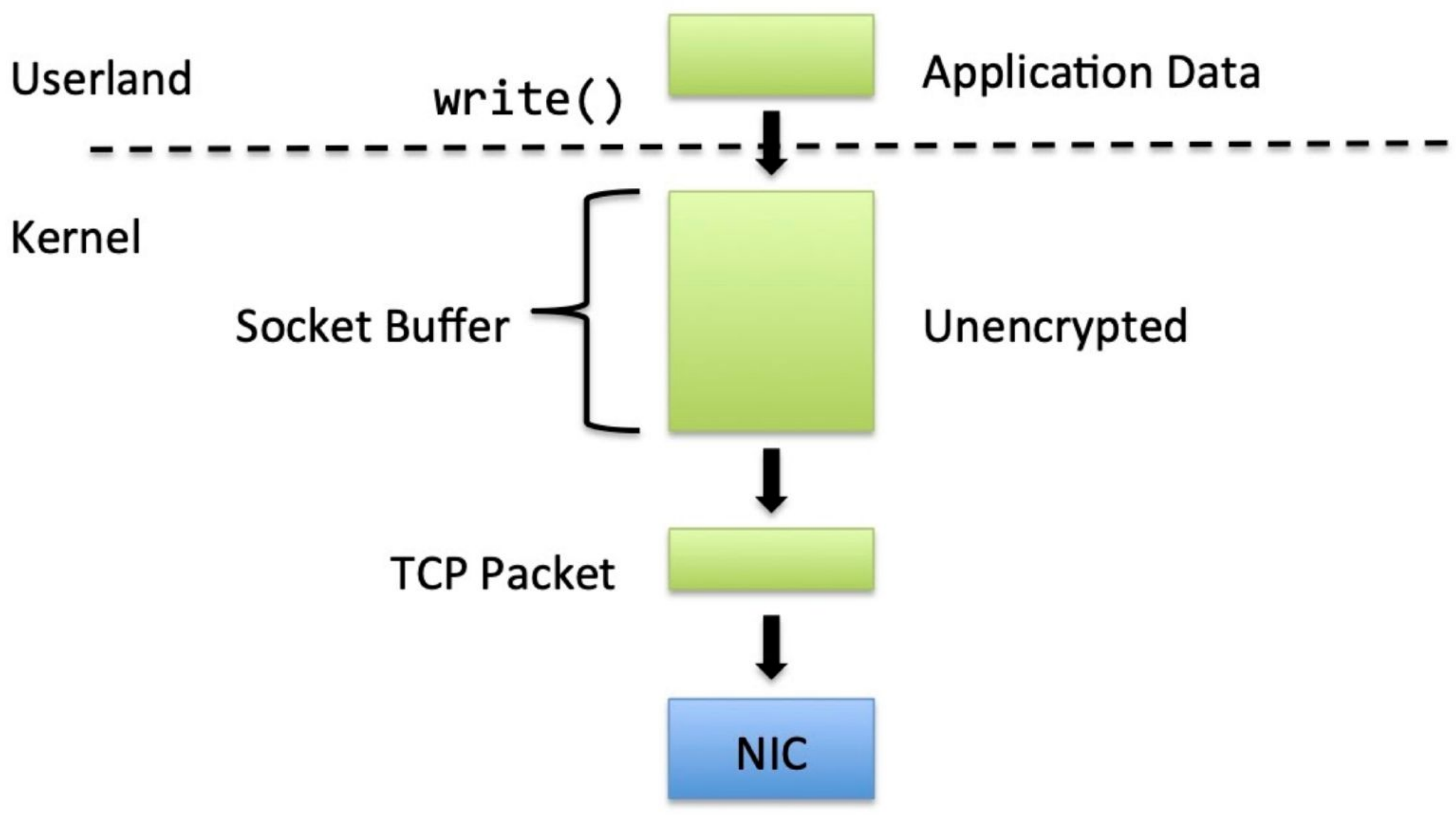
Figure 5: Software TLS Transmit Overview



NIC TLS Transmit

NIC TLS backends require additional changes in the network stack and device drivers beyond the changes needed for software TLS. With NIC TLS, unencrypted TLS records are passed down through the network stack to the device driver. The TLS record payload is encrypted by the NIC after it has been read from host memory via DMA but before it is transmitted on the wire (see Figure 6).

Figure 6: NIC TLS Transmit Overview



To probe for a NIC TLS backend, the handler for the `TCP_TXTLS_ENABLE` socket option uses the route associated with the socket to lookup the network interface servicing the socket. The handler attempts to allocate an `IF_SND_TAG_TLS` send tag from the interface. This send tag will be set on the first mbuf in an mbuf chain that contains one or more TLS records from this session.

To pass unencrypted TLS records down to the NIC, the `ktls_frame()` function does not set the `M_NOTREADY` flag on TLS mbufs when using a NIC TLS backend. NIC TLS mbufs are passed directly to the transport protocol at the time they are queued to the socket buffer.

NIC TLS does require a few changes in the TCP and IP protocols. First, TCP must ensure that it does not mix "plain" data with TLS records in a single packet. In general, TCP is permitted to construct packets which span multiple mbufs in a socket buffer. With KTLS, a connection initially contains several "plain" data mbufs. Once TLS transmit is enabled, all future mbufs in the socket buffer will contain TLS records. This means that there is a brief window after TLS transmit is enabled when the socket buffer contains both types of mbufs. To simplify support for NIC TLS farther down the stack, TCP does not send packets which contain both types of mbufs. This is handled in the `tcp_mcopym()` function. Second, IP and IP6 output must set the send tag on the packet header mbuf at the start of each packet before passing the packet down to the device driver. Since TCP does not mix mbufs of different types, IP output can check the first data mbuf in a packet to see if it has a TLS session. If so, it reads the send tag from the TLS session and sets it on the packet header mbuf. This is implemented in the `ip_output_send()` and `ip6_output_send()` functions.

Finally, NIC TLS requires support in device drivers. Device drivers must handle requests to create TLS send tags. The driver checks that it supports the requested TLS version and cipher suite. If so, it allocates a new send tag holding any device-specific state needed for the TLS session. In addition, the driver must recognize incoming packets which specify a TLS send tag and arrange for them to be segmented and encrypted. This is made more complicated by the fact that TCP may choose to send packets which only transmit a portion of a TLS record. Thus, a driver cannot rely on sending complete TLS records for each request. It may need to send the start of a TLS record, the end of a TLS record, or a region in the middle of a TLS record. In addition, since TLS implicitly supplies support for TCP Segmentation Offload (TSO), a single TCP "packet" may span multiple TLS records.

A key feature of TLS mbufs is that each TLS record is described by a single mbuf. This ensures that a NIC driver can always access the entire contents of a TLS record easily when any portion of a TLS record needs to be retransmitted. For example, a NIC may need a record's entire contents to compute the authentication code stored in the TLS record trailer when transiting the end of a TLS record. If a TLS record were split across multiple mbufs, TCP could potentially free mbufs holding ports of a TLS record that have been acknowledged by the remote end. In addition, NIC TLS drivers would need a reliable way to find other mbufs belonging to a TLS record such as holding extra mbuf references in the driver or examining the socket buffer to locate other mbufs belonging to a TLS record. Using a single mbuf for each TLS record removes the need for such complexity and simplifies NIC TLS drivers.

TOE TLS Transmit

TOE TLS uses a data flow similar to NIC TLS for transmitting TLS records. As with NIC TLS, unencrypted TLS records are passed in the socket buffer directly to the device driver. However, TOE drivers read socket data directly from the socket buffer without relying on software protocols. This simplifies the implementation of TOE TLS compared to NIC TLS.

To probe for TOE TLS, the handler for the `TCP_TXTLS_ENABLE` socket option checks if the current socket is using TOE. If the socket is offloaded, the handler invokes a method on the attached TOE driver to allocate a TLS session.

Similar to NIC TLS, `ktls_frame()` does not set the `M_NOTREADY` flag on TLS mbufs when using TOE TLS. TOE TLS mbufs are inserted into the socket buffer for immediate consumption. TCP invokes the TOE driver output method when data is available. This method reads data from the socket buffer and sends it to the associated NIC's TOE queue for transmit. Since the TOE engine on the NIC is responsible for TCP segmentation and header generation, the TOE method queues complete mbufs from the socket buffer to the NIC. This means that for TOE TLS the method is always able to send complete TLS records in one shot without having to handle edge cases for partial record transmit unlike NIC TLS. In addition, since TOE examines the mbufs in the socket buffer, TOE TLS does not require a separate send tag but uses the TLS session references in TLS mbufs.

TLS Receive

After TLS transmit, kernel TLS receive offload was a natural next step. While web server workloads benefit less from TLS receive offload relative to TLS transmit, other workloads using TLS with more balanced traffic can benefit from TLS receive offload. FreeBSD does not provide an analog to `sendfile()` (such as `recvfile()`) for reading data from a socket, so the main justifications for receive offload are NFS over TLS and supporting TLS receive offload in NICs.

Kernel TLS receive offload uses more invasive changes in both userland and the kernel's socket layer compared to TLS transmit. TLS transmit allows application data to be written directly as larger writes that are split into multiple TLS records by the kernel. TLS receive, on the other hand, returns a single TLS record for each system call. Userland must use `recvmsg()` to read from a socket using KTLS receive as each TLS record is prefixed with a new `TLS_GET_RECORD` control message containing the TLS record header. This permits the userland SSL library to intercept and handle received TLS protocol messages.

The use of control messages for `recvmsg()` requires the receive socket buffer to be treated as a datagram socket holding a linked list of records rather than as a stream socket holding a single linked list of mbufs representing a stream of data. TCP uses an optimized hook for returning received data to userland (`soreceive_stream()`) which assumes a stream socket layout in the socket buffer and does not support control messages. For TLS receive, `soreceive_stream()` was modified to invoke the generic hook (`soreceive_generic()`) if TLS has been enabled on the socket. Representing decrypted TLS records as datagrams also permits arbitrary mbufs to hold the payload data of TLS records rather than requiring the use of TLS mbufs.

A TLS receive session is created by a `TCP_RXTLS_ENABLE` socket option. This socket option uses the same `struct tls_enable` value used for enabling transmit. For receive this structure was extended to add an initial sequence number field. This is used to handle an edge case where the SSL client library might have read additional data from the socket after the TLS protocol message concluding a key exchange. In that case, the userland library will decrypt TLS records in that pending message in userland. The kernel will only decrypt subsequent TLS records. However, the kernel needs to know the TLS sequence number of the first message it decrypts since the sequence number is used as in input to the authentication phase of TLS record framing. For transmit, the SSL client library invokes the `TCP_TXTLS_ENABLE` socket option before sending any encrypted TLS records, so the initial sequence number for the transmit side is always zero.

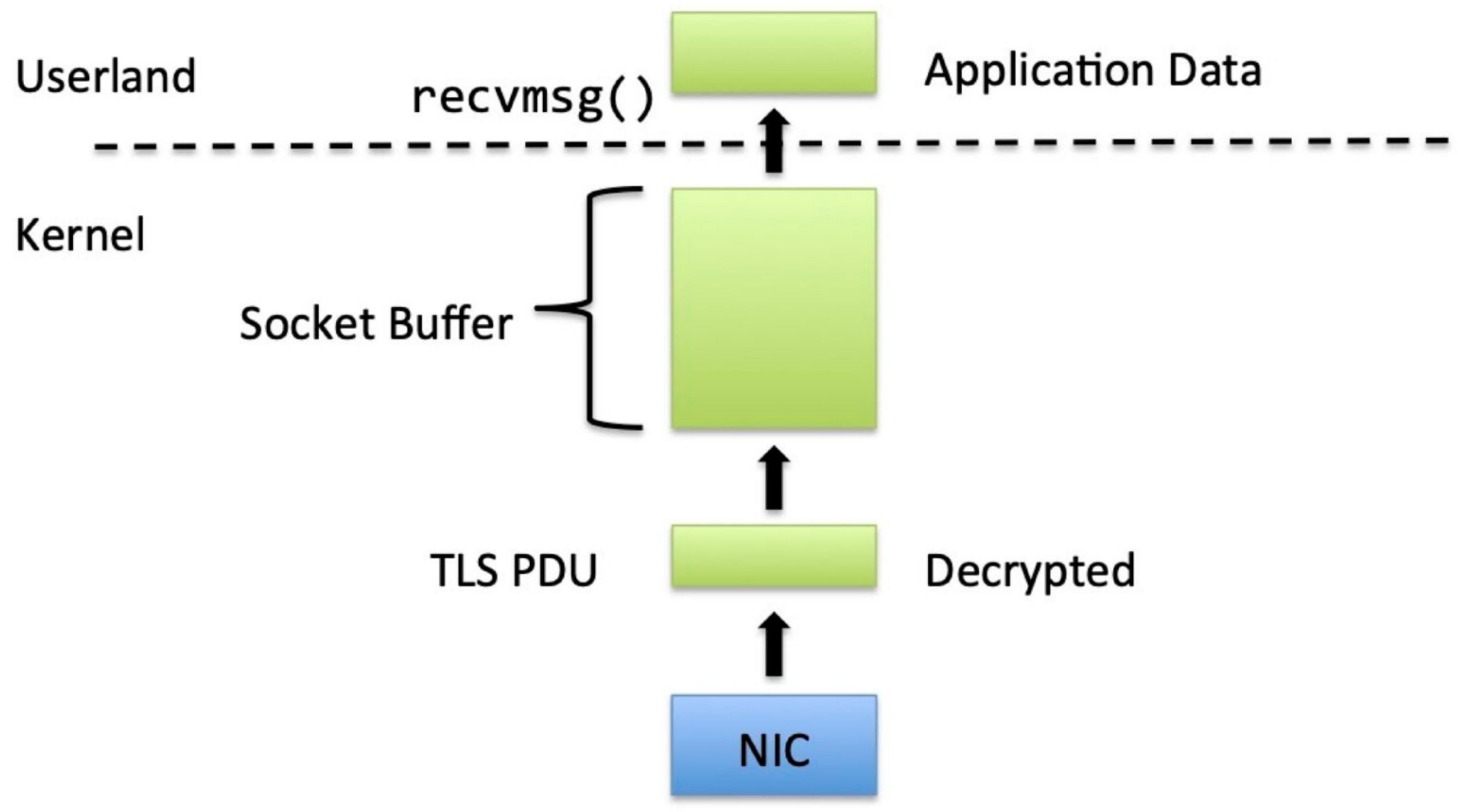
Similar to the transmit case, the handler for `TCP_RXTLS_ENABLE` probes the available backends. If a backend is found, a TLS session is created and associated with the socket's receive buffer. Any data currently present in the socket buffer is treated as encrypted TLS records. The mbufs holding this data are marked as not ready via the `M_NOTREADY` mbuf flag and scheduled for decryption before userland is permitted to read them. If a backend was not found, the `setsockopt()` system call fails and the userland SSL library decrypts TLS records in userland.

Kernel TLS receive currently supports TOE and software TLS backends. NIC TLS backends are not yet supported. For TLS transmit, software TLS was implemented first and NIC and TOE TLS were added later. For TLS receive, TOE TLS was the first backend implemented, and software TLS was implemented second.

TOE TLS Receive

TOE TLS receive uses a straightforward data flow (see Figure 7). The NIC delivers decrypted and verified TLS records to the TOE device driver. The TOE device driver appends these decrypted TLS records directly to the receive socket buffer.

Figure 7: TOE TLS Receive Overview



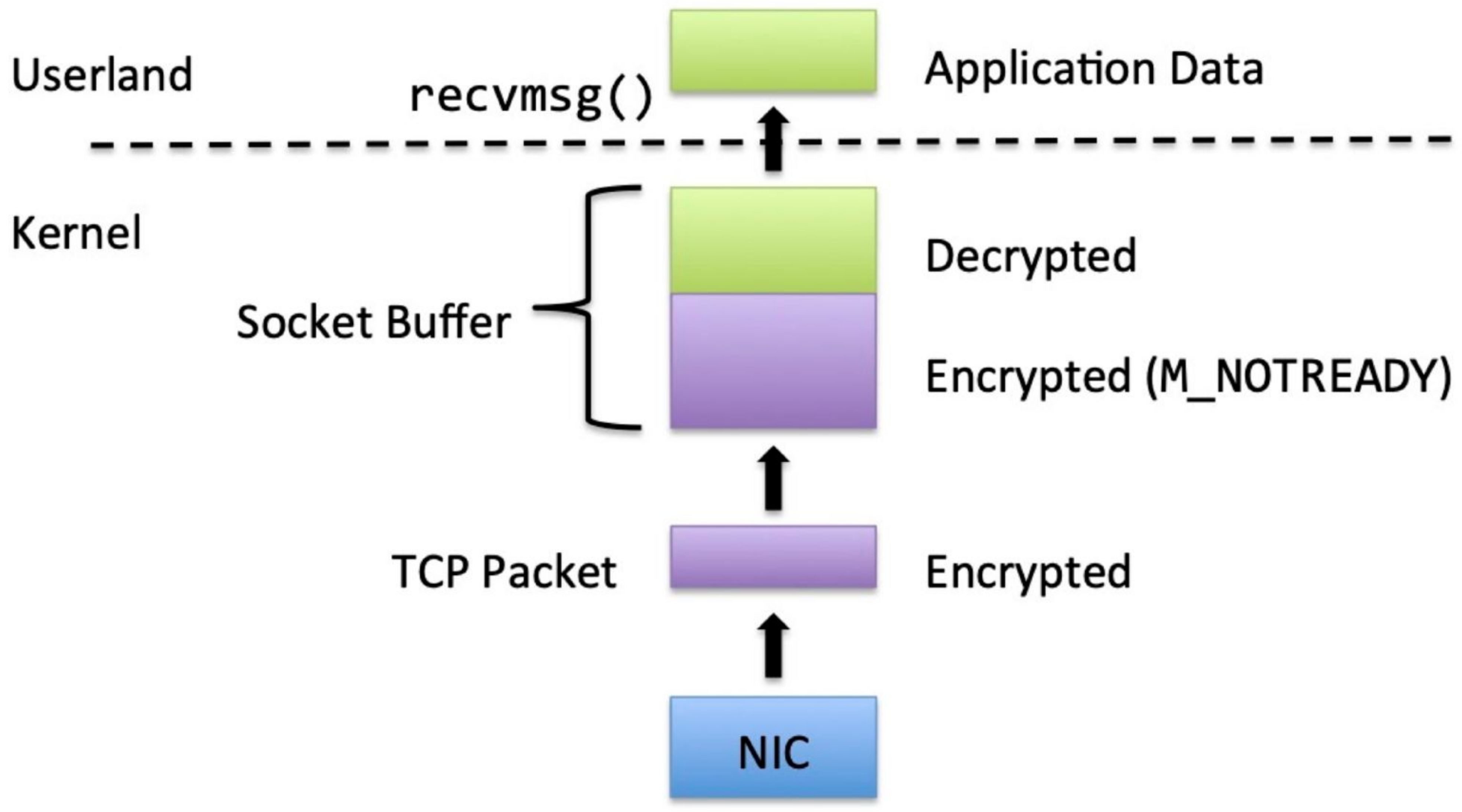
As with TOE TLS transmit, the handler for the `TCP_RXTLS_ENABLE` socket option checks if the current socket is using TOE. If the socket is offloaded, the handler invokes a method on the attached TOE driver to allocate a TLS session.

Appending decrypted TLS records to the socket buffer is straightforward for TOE TLS. The TOE device driver allocates a control message mbuf to hold the TLS header. The driver then passes this control message along with a chain of mbufs holding the TLS record payload to `sbappendcontrol()` to append the decrypted TLS record as a datagram in the socket buffer.

Software TLS Receive

Software TLS receive operates similarly to software TLS transmit, except that the data flows in the reverse direction. Protocols enqueue mbufs containing encrypted TLS records into the socket buffer. These mbufs are marked `M_NOTREADY` until they have been decrypted, at which point they are available for userland applications to read (see Figure 8). As with TLS transmit, decryption of TLS records is performed asynchronously by a pool of worker threads. However, software TLS receive has several differences compared to software TLS transmit.

Figure 8: Software TLS Receive Overview



For TLS transmit, the producer generating data to be stored in the socket buffer knows that it is generating TLS records. The producer in this case being a system call such as `write()` or `sendfile()`. As a result, each transmitted TLS record is stored as a single TLS mbuf. This permits the socket's send buffer to use the normal layout of a stream socket buffer even though it holds a list of TLS records. Software transmit TLS is also able to modify the contents of TLS mbufs in place when encrypting TLS records without modifying the linked list of mbufs in the socket buffer. Once encryption is complete, the `M_NOTREADY` flag is cleared from the mbuf making the TLS record available for transmission by the protocol layer.

For TLS receive, the producer receiving the data into mbufs that are eventually delivered to the socket buffer has no knowledge of TLS. Device drivers receive a stream of packets containing various protocols and belonging to multiple streams. The protocol eventually delivers these mbufs to the socket buffer as a stream of bytes. There is no defined alignment between mbuf boundaries and TLS record boundaries in this case. A single TLS record can span multiple mbufs, and a single mbuf may contain data from multiple TLS records. For example, a TLS record may begin in the middle of one mbuf and continue on through ten or more mbufs before ending. The most natural way to store this stream of mbufs is as a single stream of data as in a normal stream socket. However, the data stored in the receive socket buffer for TLS consists of a list of datagrams that can be read via `recvmsg()`.

Another key difference between software TLS transmit and receive is the lifetime of mbufs in the socket buffer. For both directions of software TLS, the encryption and decryption of TLS records is performed by worker threads. These worker threads must ensure that the mbufs being modified by a software backend are not freed while the software backend is performing the encryption or decryption. When the transmit side of a socket is closed by `close()` or `shutdown()`, the pending data in the send socket buffer is not immediately freed but is held in the socket buffer until it has been sent to the remote client. When the receive side of a socket is closed, however, the pending data in the receive socket buffer is immediately freed. If one of the worker threads is currently decrypting a TLS record when the receiver side of a socket is closed, TLS receive must ensure that the mbufs are not freed until the decryption has finished.

To handle these differences, TLS receive divides data in the receive socket buffer into three classes. While the socket buffer includes all three types of data in its accounting, each class is stored differently, and accounting details vary by class.

The first class of data stored in a TLS receive socket buffer are decrypted TLS records. These are stored as datagrams in the normal socket buffer chain pointed to by `sb_mb`. All mbufs associated with these TLS records are accounted for the same as in a regular socket buffer including counts of mbufs and mbuf clusters in `sb_ccnt`, `sb_mcnt`, and `sb_mbcnt`.

The second class of data stored in a TLS receive socket buffer are raw protocol data containing encrypted TLS records. These mbufs are stored in a single linked list pointed to by a new `sb_mtls` member. In addition to the regular socket buffer accounting, `sb_tlscc` counts the number of bytes of this class of data stored in the socket buffer.

The final class of data stored in a TLS receive socket buffer are "detached" TLS records. When a worker thread has assembled a chain of mbufs holding an encrypted TLS record, it detaches the mbufs from the `sb_mtls` chain. It then increments a new sb_dtlscc member by the count of bytes in the detached TLS record. After the record has been decrypted, the worker thread checks to see if `sb_dtlscc` has been cleared by a call to `sbflush()` or `sbcut()`. If so, the worker thread frees the mbuf chain. Otherwise, it appends the now-decrypted TLS record as a datagram to the `sb_mb` chain. While a TLS record is detached, the socket buffer accounting does not track the mbufs or clusters used by the TLS record. Only the bytes are accounted for in `sb_tlsdcc` and `sb_ccc`. This still provides an accurate view of the used space to TCP for the purposes of computing the receive window advertised to the remote end of the TCP connection.

Logically, these three classes of data are ordered in the socket buffer as the `sb_mb` chain, followed by any detached TLS record, followed by the `sb_mtls` chain. When `sbcut()` or `sbflush()` walk the socket buffer to free mbufs, they walk the classes in this order to free data. In practice, the only time these functions are invoked on a receive buffer for a TCP socket is to flush the entire buffer when the receive side of a socket is closed. The handling of detached records takes advantage of this by asserting that individual calls to `sbcut()` or `sbflush()` always free all of the detached TLS record or none of it.

When data is appended by the protocol layer to a TLS receive socket buffer, `sbappendstream()` calls a new `sbappend_ktls_rx()` function which appends the new mbufs to the `sb_mtls` chain. After the mbufs have been appended, `ktls_check_rx()` examines the head of the sb_mtls chain. This function reads the TLS header of the next TLS record to decrypt (if a full TLS header is available) and extracts the length field. It then checks if the length of the `sb_mtls` chain against the length from the TLS header. If the full TLS record has been received, `ktls_check_rx()` schedules the socket for decryption by a KTLS worker thread.

KTLS worker threads decrypt TLS records from the `sb_mtls` chain. The worker thread first detaches the TLS record from the receive socket buffer. This removes the mbufs holding the TLS record data from the `sb_mtls` chain but accounts for their data in `sb_tlsdcc`. It then invokes the software backend to decrypt the TLS record. Since the record has been detached, this is performed without holding any socket buffer locks. After the decryption finishes, the worker thread locks the socket buffer to check if `sbcut()` or `sbflush()` were invoked during decryption. If not, the worker thread allocates a control message to hold the TLS record header, trims the original TLS header and trailer from the original TLS record mbuf chain, and then calls `sbappendcontrol()` to append the TLS record to `sb_mb` as a datagram.

Final Thoughts

Kernel TLS is an exciting, yet complex new feature in FreeBSD 13.0. At the time of writing, support for kernel TLS transmit has been upstreamed to FreeBSD's head branch including support for software TLS, NIC TLS, and TOE TLS. A ktls_ocf kernel module provides software TLS support for AES-GCM cipher suites for connections using TLS protocols 1.2 and 1.3. Since this module uses the kernel's opencrypto framework, it permits TLS records to be encrypted by crypto co-processor drivers and not just via software run on the host CPU. Chelsio and Mellanox device drivers both include support for NIC TLS, and Chelsio's TOE device driver includes support for TOE TLS. On the receive side, the core TLS receive framework including support for TOE TLS has been merged to the head branch. Software TLS receive is still in review but should appear in head prior to the release of 13.0. Ongoing work continues to further optimize performance as well.

Kernel TLS also requires changes to userland SSL libraries. At the time of writing, OpenSSL's development branch includes support for TLS transmit for TLS 1.0-1.2 on FreeBSD. This functionality is available as a KTLS option in the security/openssl-devel port. In addition, the current KTLS patches have been backported to OpenSSL 1.1.1 as a KTLS option for the security/openssl port. Support for TLS transmit for TLS 1.3 and TLS receive for TLS 1.1-1.2 is currently in review and should land in OpenSSL prior to the release of 3.0.0.

Netflix also developed an extension to nginx to take advantage of sendfile() over TLS using the new SSL_sendfile() function in OpenSSL. This is currently available as a patch to the www/nginx-devel port under a KTLS port option.

Finally, kernel TLS is a large project that has been worked on by several members of the FreeBSD community. The current work is the product of iterations and refinements over multiple years. Scott Long first envisioned moving TLS into the kernel while working at Netflix. He worked with Randall Stewart (also at Netflix) to design and implement the first versions of software TLS transmit. Drew Gallatin introduced external pages mbufs and their later extension as TLS mbufs and converted Netflix's early KTLS to use `M_NOTREADY` mbufs for software TLS transmit. Drew also added a pluggable interface for software TLS backends. I worked with Drew to add the infrastructure for NIC TLS transmit as well as support for NIC TLS transmit on Chelsio T6 adapters. Hans Petter Selasky added support for NIC TLS on Mellanox ConnectX-6 Dx adapters. I subsequently worked on TOE TLS and TLS receive. Scott's, Randall's, and Drew's work was funded by Netflix. Hans Petter's work was funded by Mellanox. My work has been funded by both Chelsio and Netflix.

JOHN BALDWIN is a systems software developer. He has directly committed changes to the FreeBSD operating system for 20 years across various parts of the kernel (including x86 platform support, SMP, various device drivers, and the virtual memory subsystem) and userspace programs. In addition to writing code, John has served on the FreeBSD core and release engineering teams. He has also contributed to the GDB debugger and LLVM. John lives in Concord, California, with his wife, Kimberly, and three children: Janelle, Evan, and Bella.