Verified TCP Packet Reordering with Verifast

Joshua M. Cohen
Computer Science, Princeton University

Abhishek Panigrahi
Computer Science, Princeton University

January 6, 2022

Abstract

We have verified a real-world implementation of TCP packet reordering using Verifast, a semi-automated tool for proving C programs correct. We prove both that the underlying code is memory safe and that the processes of packet reordering and updating expected sequence numbers are correct according to the TCP protocol. In contrast to previous approaches to verifying network functions, which use more automation but are very restricted in the types of code they can handle, we evaluate the use of using Verifast on mostly-unmodified real-world code. We find that verifying network code is possible but expensive, and a large part of the difficulty came from modelling external packet-processing libraries and specific features of the TCP protocol.[1]

1 Introduction

1.1 Formal Verification

Writing correct code is hard. Bugs appear constantly in programs, causing crashes, data loss, and successful attacks by malicious actors. Accordingly, many approaches have been devised to find bugs and prevent them before they are exploited. These include type systems, software testing techniques, and static analysis tools, which scan code to find potential vulnerabilities. Yet these tools cannot guarantee correctness on all inputs. To do this, we need formal verification.

Formal verification uses the ideas of mathematical logic to rigorously prove a program correct against a specification. This problem is undecidable in general, and intractable in many cases. Thus, there are fundamental tradeoffs between automtaion and the degree of scalability and/or expressiveness (what can be proved). Fully automated tools are either very special-purpose or face scalability issues such as state space explosion on large or unbounded programs. Proof assistants, meanwhile, allow arbitrarily-complex properties to be proved manually, with limited automation. In the middle are semi-automated tools like Dafny [4] and Verifast [3], where the user annotates code with propositions (more limited in scope than proof assistants), which are automatically checked by an SMT solver. Many annotations may be needed for complex programs.

For networking and other systems code, formal verification is particularly valuable. These programs are often written in low-level languages like C, and are thus prone to bugs due to manual memory management, such as buffer overflows, null pointer dereferencing, and double frees.

1.2 Verifast

Verifast [3] is a program verification tool for C and Java that allows the user to annotate their code with preconditions (conditions that must hold when a function is called), postconditions (conditions guaranteed to hold when a function returns), loop invariants (conditions that are true before a loop begins and true after each iteration of the loop), and assertions (conditions that should be true at a given point in the code). Verifast then uses an SMT solver to determine if it can guarantee that the annotated predicates hold. If not, it may be the case that some are violated or simply that more information is needed. Verifast is based on *separation logic*, a logic for reasoning about data structures in the heap. With Verifast, we are able to prove both safety properties (no crashes, no invalid memory access, etc) and full functional correctness.

1.3 TCP Packet Reordering

TCP provides reliable delivery over (best-effort) IP. TCP relies on *sequence numbers* to identify the packet order. Each packet's sequence number marks the index of the

[1] The verified code is available at joscoh/Verified-Packet-Reorder

first byte sent in that packet. The receiver uses sequence numbers to identify missing packets and to reorder the packets if they arrive out of order. The different TCP packet types (SYN, ACK, etc) make the precise sequence number semantics are bit more complicated (§3.3).

In our project, we used a TCP packet reordering implementation written the WAND research group at the University of Waikato in New Zealand[2]. We used Verifast to write a TCP packet reordering specification and added annotations to the code to prove it correct against this specification. The code and external libraries were slightly modified to overcome Verifast limitations, see §4.2.

2 Related Work

Networks are a frequent target for formal verification. Most of this work falls under the category of *network verification*, or modelling the entire network and proving properties of the whole network like reachability. This is not particularly related to our work.

Separately, in recent years *network function verification*, verifying code running in the dataplane, has become feasible. There are 3 promiment such efforts. VigNAT [7] uses a mix of Verifast and automatic symbolic execution to formally verify a NAT written in C. Building on this, Vigor [6] extends these methods to handle a range of network functions, including a NAT, firewall, and load balancer, requiring no user annotations on the code. Separately, Gravel [8] uses symbolic execution and SMT to verify existing Click elements for a variety of network tasks. It also requires no user annotations.

These methods, which are mostly or fully automated, rely on specific code structure and specific ways of handling state. Vigor requires all state to be in a special, annotated data structure library, restricts the user of pointers, and requires the overall program to have only a single event loop. Gravel can only verify code using certain data structures that have been encoded in SMT (vectors, HashSets, and HashMaps), prohibits recursive pointer-based data structures, and relies on small, modular, and bounded code. TCP packet reordering runs on the end hosts, uses recursive data structures, and contains unbounded loops. Thus, it does not fit into these paradigms.

3 Verification Structure and Challenges

The code consists of 4 main functions: initialize a packet reorderer structure (which stores a linked list of packets and the expected sequence number), free a reorderer structure, insert a new packet onto the reorderer (ie: get information from the packet header, then call a core insertion function to insert into the sorted list), and pop the first packet from the reorderer, updating the expected sequence number.

To verify these functions in Verifast, we needed to define predicates for each data structure and to annotate the functions with pre and post-conditions. In the function bodies, we need various annotations to ensure there is enough information in the context for the solver. Most of the functions are quite lightweight to verify except for the core insertion code, which uses a loop and requires substantial work to annotate (§4.3). We now focus a few key network-related verification challenges.

3.1 Sequence Number Comparison

TCP sequence number comparison is more complicated than simply comparing the two numbers, because sequence numbers wrap around after 2^{32} [5, 3.3]. That is, a packet with sequence number $2^{32}-1$ should come before a packet with sequence number 1.

The TCP sequence number comparison function in the code, which incorporates this wraparound, is:

```
static int seq_cmp (uint32_t seq_a,
    uint32_t seq_b)
{
        if (seq_a == seq_b) return 0;
        if (seq_a > seq_b)
            return (int)(seq_a - seq_b);
        else
            return (int)(UINT32_MAX -
                ((seq_b - seq_a) - 1));
}
```

This code introduces two main problems for verification. First, it uses intentional integer overflow, which Verifast cannot handle (it either checks or assumes that all integer operations don't overflow). Second, incorporating wraparound means that this is not a valid ordering – it is neither transitive nor antisymmetric.[3]

To deal with these issues, we verified this function separately in the Coq proof assistant using the Verified Software Toolchain (VST) for verifying C programs. VST can handle overflow and has good support for integer arithmetic. We verified that the above function has the following properties:

- It is antisymmetric in all cases except when $|a-b| = 2^{31}$.
- It is transitive on a,b,c when $|a-b| \leq 2^{31}$, $|b-c| \leq 2^{31}$, and $|a-c| < 2^{31}$ (among other cases).

[2] Part of libflowmanager.

[3] Antisymmetry means that $cmp(a,b) > 0 \rightarrow cmp(b,a) < 0$ and vice versa.

This means that as long as all sequence numbers we are dealing with are within a 2^{31} range, the comparison function is valid. This is an acceptable assumption to make; the recieve window's maximum size is 1GB (2^{30} bytes), so we will never encounter sequence numbers which violate this range in the same sliding window [2, 2.2]. However, it is possible that the client does not pop any data off the reorderer structure for more than 2^{31} sequence numbers; in this case, the structure is no longer guaranteed to be sorted correctly (we make an assumption in the proofs that this does not happen).

For simplicity, we assume that all received packets have sequence numbers in the range of $[0, 2^{31} - 1]$ and in Verifast, we axiomatize seq_cmp (due to the lack of support for overflow). This is safe; we proved that these axioms are sound in VST.

3.2 Verifying Core Packet Insertion

The core packet insertion function (insert_packet) handles the insertion into the list in the reorderer. This list is always sorted (we formally stated this as an invariant of the data structure), so this amounts to insertion into a sorted list. This is conceptually simple but needs a good amount of bookkeeping for the loop invariants and predicates needed.

Linked lists are fairly simple to express in separation logic: we need a predicate that describes the contents of a list cell and says that the predicate recursively holds over the list starting from the next pointer. See Appendix A for our predicate definitions. Since insertion is also (essentially) recursive, we can verify the loop in a fairly straightforward way.

However, insert_packet first tests the sequence number against the last packet to see if the new packet should be inserted at the end. If not, it inserts from the beginning. Intuitively, assuming the network is mostly reliable, we would expect that most packets arrive in order and should be put at the end. Furthermore, since we are also popping packets from the front, we would expect the missing packets to form a bottleneck at the front of the list. In other words, most packets should be inserted at the end, and a few should be inserted close to the front, so this method of insertion is well-suited to the application.

This small, seemingly insignificant change, makes the verification significantly more complicated. The recursive linked list predicate does not allow us to reason about the end of a list directly, and due to Verifast's limited existentially-quantified variables and separation-logic lemmas, we need another predicate which directly exposes the end node of a list (and contains the first $n-1$ nodes separately). Defining this correctly and proving it equivalent to the natural predicate was quite complicated.

While this obstacle only required manipulating separation logic predicates, it illustrates some of the tradeoffs between code designed for verification and code designed for domain-specific performance. A simpler insert function would have been easier to verify, but it would not have been optimized for the specific networking context where it is used.

3.3 Formalizing TCP Semantics

In order to verify the packet insertion (tcp_reorder_packet) and pop (tcp_pop_packet) functions, we need to know what correctness entails. Therefore, we need to model some TCP semantics, based on our understanding of the TCP standard [5].

First, we need to model the types of TCP packets and determine the type of a packet based on which bits (SYN, ACK, FIN, RST) are set. We use an inductive datatype to express the types of TCP packets (SYN, ACK, FIN, RST, SYN-ACK, FIN-ACK, RST-ACK), then provide a pure function tcp_flags_to_type which takes in the values for the bits and gives a TCP type option, indicating whether the arrangement of bits is valid, and if so, which type it corresponds to.[4]

Next, we need to specify what the insert and pop functions should do. When inserting a packet, the C code stores a "reorder effect" with the packet based on the packet type, length, sequence number, and current expected sequence number (the expected sequence number represents the next byte, in order, that has not yet been popped). Then, when the packet is popped, the expected sequence number is updated based on the stored effect. Thus, we need to define, again based on the TCP standard [5, 3.4-3.5], functions that describe what the reorder effect should be when inserted, and how the expected sequence number should be updated (these are related; the effect is just a shorthand to denote how the expected sequence number should be updated later).

We summarize this as follows:[5]

1. In the pop function, we only pop if the top sequence number is $\leq$ the expected sequence number to ensure we don't skip any bytes. For all following cases, assume this is the case.

2. A SYN, SYN-ACK, FIN, or FIN-ACK packet should increase the sequence number by 1 [5, Glossary (SYN, FIN)].

3. An ACK packet with no data does not change the sequence number [5, Glossary (ACK)].

4. An RST or RST-ACK packet does not change the sequence number [5, Glossary (RST)].

[4] See file src/tcp_type.gh

[5] The full functions are available at src/tcp_semantics.gh

5. All remaining packets are data packets. When the packet arrived, if it had sequence number smaller than expected, it is a retransmission. We update exp_seq to $\max(exp\_seq, seq + plen)$, where $plen$ is the length of the packet data (and the max is taken with respect to the TCP sequence number comparison function).

6. Otherwise, the packet is regular data (arriving in order). When it arrived, it has sequence number $\geq$ expected. Thus, when we pop this packet, $seq = exp_s eq$ (since we cannot skip bytes). We update exp_seq to $seq + plen$.

Note that these specifications are independent of the implementation, and were derived from the TCP standard. Their specific structure is modelled after the implementation to make the verification simpler.

3.4 Modelling External Libraries

libflowmanager uses the libtrace packet processing library [1] to represent packets and get the needed IP/TCP fields out of the respective headers. The tcp_reorder_packet function takes in a libtrace packet, extracts some packet payload information (based on a user-inputted function pointer), and examines the IP and TCP header lengths, total length, and TCP SYN, ACK, FIN, and RST bits. It then determines the payload length and the TCP reorder effect (§3.3) based on this information. Therefore, we only need to model a small part of libtrace.

libtrace includes TCP and IP structs and functions trace_get_ip and trace_get_tcp to get these structs from packets. These structs have fields for each TCP and IP header, among others.

libtrace is very large, making it infeasible to modify, so we need to model this. On our first attempt, we declared separation-logic predicates for the IP and TCP structs. We then defined abstract predicates valid_tcp_packet and valid_ip_packet saying that the input packet is a well-formed TCP/IP packet (reordering may not work on invalid packets). Then, we assume that the trace_get_* functions produce the appropriate separation-logic predicate on valid inputs. Below shows a simplified version of this approach:

```
predicate valid_tcp_packet(
    libtrace_packet_t *packet, int seq);

predicate libtrace_tcp_p(libtrace_tcp_t
    *tcp, int seq) =
    //heap chunk tcp->seq
    tcp->seq |-> seq

//In libtrace.h
libtrace_tcp_t *trace_get_tcp
    (libtrace_packet_t *packet);
//@ requires valid_tcp_packet(packet, ?
    seq);
/*@ ensures result != 0 &*&
    valid_tcp_packet(packet, seq) &*&
    libtrace_tcp_p(result, seq); @*/
```

However, this approach does not work; we cannot verify the client function. This model assumes that trace_get_ip and trace_get_tcp allocate new objects on the heap for the TCP and IP structs. These are not freed, so Verifast notes that the insert function leaks memory.

Delving into libtrace further, we saw that the structs are simply pointers into the packet, which is arranged in such a way (and the structs are defined in such a way) that the appropriate offsets from this pointer correspond to each of the header fields. While the actual code is quite complicated, we can still model this. This time, we define pure functions which calculate the offset from a given packet pointer to one of the required fields, then in our separation-logic predicates for TCP and IP structs, declare pointer equality between the struct field and the offset. Below shows a simplified version of this approach:

```
fixpoint uint32_t *tcp_seq_ptr(
    libtrace_packet_t *packet);

predicate valid_tcp_packet(
    libtrace_packet_t *packet, int seq) =
        *(tcp_seq_ptr(packet)) |-> seq;

predicate libtrace_tcp_p (
    libtrace_packet_t *packet,
    libtrace_tcp_t *tcp, int seq) =
        &(tcp->seq) == tcp_seq_ptr(packet
            );

libtrace_tcp_t *trace_get_tcp(
    libtrace_packet_t *packet);
//@ requires valid_tcp_packet(packet, ?
    seq);
/*@ ensures result != 0 &*&
    valid_tcp_packet(packet, seq) &*&
    libtrace_tcp_p(packet, result, seq);
@*/
```

Therefore, we assume that there is only a single heap-allocated structure (the packet) and that in the IP and TCP structs, we simply have a pointer to a part of the packet structure. Since libtrace_tcp_p is now a pure predicate (no heap structures), there are no memory leaks. Furthermore, because we declare the pointer offsets as fixpoint functions, they must be pure; the libtrace function cannot allocate memory or rely on side effects to extract the fields. We believe that this assumption is justified based on our understanding of TCP/IP packets and examination of the libtrace code.

4 Results and Analysis

For the formal specifications of each function, see Appendix B. Below we detail some key metrics and overall observations about the verification effort.

4.1 Bugs Found

We found several memory-related bugs in the code:

1. When inserting a packet, a new structure is malloc'ed. If malloc fails,[6] the insert function ignores this, causing segfaults and memory leaks, since some previously-allocated structures (based on the packet data) are not freed.

2. The insert function calls a user-specified function pointer to extract packet data. It does not check that this pointer is nonzero before calling it.

3. When freeing the packet data structure, the code calls another user-specified function pointer. The code does check if the pointer is zero, but if it is, the function simply calls free on the data. This is dangerous, since the data may not have been malloc'ed (resulting in undefined behavior).

We fixed each of these bugs with small modifications to the code. Additionally, there were two "bugs" that did not result in an incorrect value, but could cause problems for other clients of the code:

1. RST-ACK packets with no data are treated as ACK packets, not RST packets. This is OK since neither affect the expected sequence number.

2. The code uses htons (host to network short) to convert the IP total length header field to host byte order. On most systems, such as x86, this is OK because htons and ntohs do the same thing (reverse byte order), but this may not always be the case.[7]

4.2 Code Modifications and Assumptions

Besides the small bug fixes, we needed to modify the code and external libraries in a few places to overcome Verifast limitations. These changes were quite small: adding explicit casting to some integer operations, splitting function pointer invocations into 2 lines, and changing a for loop to the equivalent while loop. In libtrace, we removed bitfields from the structs (because Verifast does not support them) and removed an unused (for us) struct field, because it used part of the C standard library unsupported by Verifast. We do not believe that any of these changes affect the functionality or performance of the reordering code.

We needed to make several assumptions about the code to verify it. Besides the assumptions made about the external library correctness (§3.4), we assume that the function pointers used to extract and destroy packet contents are consistent, in the sense that the extraction function produces an abstract memory predicate data_present(p) and the destroy function deallocates this memory (we cannot assume it uses malloc). We also assume that all sequence numbers stored in the the structure are in the range of $[0, 2^{31} - 1]$ (§3.1). Of course, this may not hold in general, and even the weaker assumption that the range never exceeds $2^{31} - 1$ may not always hold. This is not necessarily a problem; it is simply a more limited specification. Finally, we disabled Verifast's integer overflow checking, so we do not ensure that the structure's length stays within signed integer bounds.

4.3 Verification Effort

It took several dozen hours to verify the roughly 250-line program (comments excluded). In total, we used roughly 1221 lines of annotations (external libraries excluded), about 4-5 annotations per line of code. However, this conceals some differences: specifying the TCP semantics used 140 lines and defining and proving properties of sorting (which ideally would be in a standard library) used another 258. We used 1 annotation per line to define the struct predicates, and then needed another 391 for lemmas about these predicates. Within the actual code, the core insertion function needed 224 lines of annotations for the 60-line function, but the rest of the code needed fewer than 1 annotation per line-107 annotations compared with 166 lines of code. This shows that, even in more general settings than Vigor, a similar paradigm emerges; the core stateful (and looping) code is the difficult part, while the rest can be done quite automatically. Finally, we note that we did not make any effort to minimize the number of annotations; it may be possible to do better. However, the Vigor project needed a 10:1 ratio for annotations, so we are unlikely to improve significantly[8].

5 Conclusion

We have formally verified an implementation of TCP packet reordering with Verifast. We found that this verification is possible, though it quickly runs into network-specific issues about specifying packet libraries and formalizing TCP semantics, including sequence number comparisons and updates. However, we overcame these

[6] This is possible, although rare. See the man page for details.

[7] In [1], a similar libtrace example uses ntohs.

[8] Vigor's library of complex data structures unsurprisingly required more annotations than our code.

challenges to verify the code with few changes, finding several bugs along the way. While this kind of formal verification is feasible on real-world network components, it is expensive and difficult, and it is not surprising there has been a push for more automated methods. Ideally, a large formalized library of network protocol semantics, data structures, and packet-processing functions would ease the burden significantly. Nevertheless, even without any of that, verification of network components can increase our confidence in their correctness without sacrificing performance, at the cost of significant effort.

References

[1] ALCOCK, S., LORIER, P., AND NELSON, R. Libtrace: A Packet Capture and Analysis Library. *SIGCOMM Comput. Commun. Rev. 42*, 2 (mar 2012), 42–48.

[2] BORMAN, D., BRADEN, B., JACOBSON, V., AND SCHEFFENEGGER (ED.), R. TCP Extensions for High Performance. RFC 7323 (Proposed Standard), Sept. 2014.

[3] JACOBS, B., AND PIESSENS, F. The VeriFast Program Verifier, 2008.

[4] LEINO, K. R. M. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning* (Berlin, Heidelberg, 2010), E. M. Clarke and A. Voronkov, Eds., Springer Berlin Heidelberg, pp. 348–370.

[5] POSTEL, J. Transmission Control Protocol. RFC 793 (Internet Standard), Sept. 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[6] ZAOSTROVNYKH, A., PIRELLI, S., IYER, R., RIZZO, M., PEDROSA, L., ARGYRAKI, K., AND CANDEA, G. Verifying Software Network Functions with No Verification Expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 275–290.

[7] ZAOSTROVNYKH, A., PIRELLI, S., PEDROSA, L., ARGYRAKI, K., AND CANDEA, G. A Formally Verified NAT. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, Association for Computing Machinery, p. 141–154.

[8] ZHANG, K., ZHUO, D., AKELLA, A., KRISHNAMURTHY, A., AND WANG, X. Automated Verification of Customizable Middlebox Properties with Gravel. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 221–239.

A Structure Predicates

The packet data is stored in a tcp_packet_t struct defined as follows: [9]

```
/* An entry in the reordering list for a
    TCP packet */
typedef struct tcp_pkt {
        /* The type of TCP packet */
        tcp_reorder_t type;
        /* The sequence number of the
            packet */
        uint32_t seq;
        /* The size of the packet payload
             (i.e. post-TCP header) */
        uint32_t plen;
        /* The timestamp of the packet */
        double ts;
        /* Pointer to packet data
            extracted via a read callback
            */
        void *data;
        /* Pointer to the next packet in
            the reordering list */
        struct tcp_pkt *next;
} tcp_packet_t;
```

To specify this in Verifast, we first define a predicate for all of the non-list parts of the packet. We do this separately because these parts are not changing in any loop invariants. Note that any existentially quantified values are marked with a ?. data_present is an abstract predicate used to denote that the data was extracted with a valid read function pointer (§4.2).

```
predicate tcp_packet_single(tcp_packet_t
    *start, int seq, tcp_reorder_effect
    eff, int plen) =
// start is properly intialized
start != 0 &*& malloc_block_tcp_pkt(start
    ) &*&
//fields are initialized
start->type |-> effect_to_reorder_t(eff)
    &*& start->plen |-> plen &*& start->ts
     |-> ?ts &*&
// data is initialized
start->data |-> ?data &*& data_present(
    data) &*&
//sequence number
start->seq |-> seq &*& inrange(seq) ==
    true;
```

To express the entire list (which is specified with start and end pointers), we need a recursive predicate. We specify the contents of the list with the type list<pair<**int**, tcp_reorder_effect>>, which is a standard functional list over the tuple type for sequence numbers and reordering effects. Note that we require that the list is sorted (by sequence number) and actually corresponds to the sequence numbers and effects in each list cell; this ensures that all well-formed packet lists consist of correctly-reordered packets:

```
predicate tcp_packet_partial(tcp_packet_t
     *start, tcp_packet_t *end,
    tcp_packet_t *end_next, list<pair<int,
     tcp_reorder_effect>> contents, int
    seq, tcp_reorder_effect eff) =
tcp_packet_single(start, seq, eff, ?plen)
     &*& start->next |-> ?next &*&
// sortedness/contents
sorted(contents) == true &*& contents ==
    cons(pair(seq, eff), ?tl) &*&
// predicate recursively holds
(start == end ? tl == nil && next ==
    end_next: next != 0 &*&
    tcp_packet_partial(next, end, end_next
    , tl, ?seq1, ?eff1));
```

We made end_next a parameter because we want to deal with partial lists (hence the name). But a complete list should end with NULL as the last element:

```
predicate tcp_packet_full(tcp_packet_t *
    start, tcp_packet_t *end, list<pair<
    int, tcp_reorder_effect>> contents,
    int seq) =
        end != 0 &*& tcp_packet_partial(
            start, end, 0, contents, seq,
            ?eff);
```

Now, the full reorderer struct contains several other fields as well as the list, most notably the expected sequence number:

```
/* A TCP reorderer - one is required for
    each half of a TCP connection */
typedef struct tcp_reorder {
        /* Current expected sequence
            number */
        uint32_t expected_seq;
        /* Number of packets in the
            reordering list */
        uint32_t list_len;
        /* Read callback function for
            packets that are to be
            inserted into
         * the reordering list */
        read_packet_callback *read_packet
            ;
        /* Destroy callback function for
            packet data extracted using
            the
         * read callback */
        destroy_packet_callback *
            destroy_packet;
        /* The head of the reordering
            list */
```

[9] Some of these definitions are hard to read due to space constraints; all can be found at src/tcp_reorder.h.

```
        tcp_packet_t *list;
        /* The last element in the
            reordering list */
        tcp_packet_t *list_end;
} tcp_packet_list_t;
```

To define the predicate for this, we again split into 2 parts. First, we deal with all of the non contents or start related fields. These are not changing throughout the loop invariants, so it convenient to bundle them together. Note that the is_read_packet_callback and is_destroy_packet_callback ensure that the function pointers given conform to our specification.

```
predicate tcp_packet_list_wf(
    tcp_packet_list_t *reorder,
    tcp_packet_t *end, int length, int
    exp_seq) =
     malloc_block_tcp_reorder(reorder) &*&
     //fields initialized
     reorder->expected_seq |-> exp_seq &*&
          reorder->list_len |-> length &*&
         reorder->read_packet |-> ?rp &*&
         reorder->destroy_packet |-> ?dp
         &*&
     dp != 0 &*&
         is_destroy_packet_callback(dp) ==
         true &*& is_read_packet_callback(
         rp) == true &*&
     inrange(exp_seq) == true &*&
     reorder->list_end |-> end;
```

Finally, we have a complete predicate for the reorderer struct:

```
predicate tcp_packet_list_tp(
    tcp_packet_list_t *reorder, list<pair<
    int, tcp_reorder_effect>> contents,
    int exp_seq) =
    tcp_packet_list_wf(reorder, ?end,
         length(contents), exp_seq) &*&
         reorder->list |-> ?start &*&
         // either empty or well-formed
             packet
         start == 0 ? end == 0 && contents
              == nil:
         tcp_packet_full(start, end,
             contents, _);
```

B Function Specifications

With these predicates, we can specify each of the functions. The requires clause denotes the precondition (which must be true when a function is called) and the ensures clause denotes the postcondition (which must be true when a function finishes).

The creation function is simple: as long as the inputted callbacks are valid, if the function succeeds in allocating a new reorderer, then it is valid with no contents:

```
tcp_packet_list_t *tcp_create_reorderer(
    read_packet_callback *cb,
    destroy_packet_callback *destroy_cb)
//@ requires is_read_packet_callback(cb)
    == true &*& destroy_cb != 0 &*&
    is_destroy_packet_callback(destroy_cb)
     == true;
//@ ensures result == 0 ? true :
    tcp_packet_list_tp(result, nil, 0);
```

The destruction function is even simpler: it completely frees a valid reorderer. Note that Verifast ensures there are no memory leaks from the postcondition.

```
void tcp_destroy_reorderer(
    tcp_packet_list_t *ord)
//@ requires tcp_packet_list_tp(ord, ?
    seqs, ?exp_seq);
//@ ensures true;
```

The core insertion function requires a valid reorderer and for the other inputs to be valid. Then, if it succeeds (we added this case for when malloc fails, §4.1), it adds the given sequence number and corresponding effect to the list, using the insert function which inserts into a sorted list. Otherwise, it leaves the heap unchanged.

```
static int insert_packet(
    tcp_packet_list_t *ord, void *packet,
    uint32_t seq, uint32_t plen, double ts
    , tcp_reorder_t type)
//@requires tcp_packet_list_tp(ord, ?l, ?
    exp_seq) &*& data_present(packet) &*&
    inrange(seq) == true &*&
    valid_reorder_t(type) == true;
/*@ ensures result == 0 ?
        tcp_packet_list_tp(ord, l,
            exp_seq) &*& data_present(
            packet)
        : tcp_packet_list_tp(ord, insert(
            seq, reorder_t_to_effect(type)
            , l), exp_seq); @*/
```

The overall insertion function is a bit more complicated. It can fail (ie: not add the packet) for a few reasons: the core insertion function may fail or the packet may be an invalid TCP packet. We assume that the input is a valid TCP packet. However, due to the order of operations, if the packet is a SYN packet but insert fails, the expected sequence number of the reorderer is updated. Otherwise, the structure is unchanged (it is unclear if this is a bug; that depends on the desired behavior). In the usual case, we return the correct TCP reorder effect (based on the TCP semantics, §3.3), insert the packet, and, if the packet was a SYN packet, update the expected sequence number to be the packet's sequence number. This specification is longer and is much easier to read in the source code.

Finally, the pop function similarly has to deal with a few cases. It requires a valid reorderer and says that, if

the contents list is nil, nothing happens (and NULL is returned). Otherwise, if the sequence number of the top packet (seq) is larger than the expected sequence number, nothing happens and NULL is returned. Otherwise, we get a valid start packet (a pointer to which is returned) and a valid reorderer, which consists of all packets after the start and with the expected sequence number updated correctly (again according to the TCP semantics). See the source code for the formal spec.