To appear in VSTTE'23, *15th International Conference on Verified Software: Theories, Tools, and Experiments,* October 23, 2023

Specifying and Verifying a Real-World Packet Error-Correction System

Joshua M. Cohen and Andrew W. Appel

Princeton University, Princeton NJ 08544, USA

Abstract. Automated and semi-automated formal methods have been widely employed to verify properties of network models and *per-packet* network functions, which operate on single packets in the middle of a network (firewall, NAT, etc). But these methods do not extend to *end-to-end* network functions, those whose specification relates a stream of packets sent at one endpoint of the network with a stream received at the other end. Among other complications, such specifications must account for the network's behavior, including packet reordering, duplication, delay, and loss. We develop a methodology for formally specifying and verifying such code, demonstrating our techniques on a real-world packet error-correction system that encounters all of these challenges and whose specification had been highly unclear. We prove a close model of this system correct in the Coq proof assistant; along the way, we formalize more general networking constructs including IP/UDP packets, a metric for packet reordering, and sequence number comparison. Finally, through our specification, we develop an improved version of the error-correction system, giving a more predictable, provably correct program that recovers more packets. We show that formal specification and verification can be powerful tools to clarify assumptions, improve code quality, and find and fix bugs in complex, real-world systems.

1 Introduction

Formal verification has been widely applied to networks and network components, whose correctness is critical for higher-level internet applications. Broadly, these efforts take one of two forms: verifying properties of entire networks (reachability, routing protocol convergence, etc) by using automated solvers on a simple model of the network (e.g. a graph) or verifying network function implementations that operate per-packet and which are often implemented at the intermediate nodes of a network (NAT, firewall, etc). Neither of these methods extends to *end-to-end* network functions: those run at the end hosts and whose specification cannot be expressed solely as a function of the input packet but must relate multiple packets in the stream. These include critical transport-layer functions for reliable delivery including packet reordering, congestion control, flow control, error correction, and packet retransmission mechanisms.

Proving the correctness of per-packet functions generally involves reasoning about which header fields in a packet must change (based on the packet and stored data, e.g. a map of internal/external IPs for a

NAT) and proving some related higher-level properties (e.g. a firewall blocks packets from a certain IP range). But end-to-end functions require a completely different approach, as they have several distinctive challenges. First, though executed packet-by-packet, their behavior (and thus, a specification) depends on the often-complex interactions between the entire stream of packets sent and received, as well as state at both ends. Second, such implementations typically use timeout mechanisms and may depend on additional external state such as the system time; their behavior depends not only on the packet inputs but on the ability of the network to drop, delay, reorder, and duplicate packets. Finally, the intended guarantees provided by such a system may be unclear; for instance, the system may be permitted to drop packets (say, if it has not seen a missing packet for a long time) or it could assume properties about the underlying network which may or may not hold.

In this paper, we investigate how formal specification and verification can be used to effectively reason about these types of real-world end-to-end systems. We use as an extended case study an existing, real-world packet error-correction system written in C; we determine a specification for it and verify a close model against this spec in the Coq proof assistant. The core error-correction algorithm and its C implementation were verified by Cohen *et al.* [12]; here we reason about the larger system to group, store, send, and receive packets at the end hosts. The core algorithm's verification was complex – requiring reasoning about sophisticated mathematics and low-level C programming conditions – but its specification was not, arising from well-known mathematical properties of the underlying Reed-Solomon code. In contrast, for packet reordering, duplicating, and timeouts, formulating a specification is difficult, both because of the end-to-end challenges discussed above and because this implementation violates any reasonable specification. We detail our process and demonstrate how to reason about code whose specification is unknown, dependent on external conditions, and reliant on invariants maintained through different executions, and we show that even the act of specifying a "dusty deck" program can clarify assumptions, find bugs, and lead to cleaner, more performant, and more predictable code. Our contributions are as follows:

1. We present a methodology for formally specifying real-world, end-to-end network functions, especially transport-layer programs for reliable delivery.
2. We apply these techniques to specify and verify in the Coq proof assistant a real-world error-correction system, the first verification of such a program. We identify many bugs in the implementation and write a new version that is more predictable, provably correct, and recovers more packets.
3. Along the way, we formalize in Coq several more general networking concepts: IP and UDP packets, a well-studied metric for measuring packet reordering, and sequence number arithmetic; the latter two are particularly useful for reasoning about end-to-end network functions that use timeouts.

Our proofs are available at github.com/verified-network-toolchain/Verified-FEC/tree/end-to-end/proofs/FEC.

2 Background – Verification Techniques

Software verification often proceeds in layers; rather than proving desired properties directly about the source program, one writes a simpler *functional model* of the program, proves properties about this model, and separately proves that the implementation refines this model. This approach is modular: the two proofs, often requiring very different kinds of reasoning, are kept separate, and multiple low-level implementations can be proved correct against the same functional model without repetition of the high-level proofs. If the two proofs are performed in the same system, they can compose top-to-bottom to give a single theorem. This approach has been used to interactively verify distributed systems [14], an HTTP key-value server [30], a pseudorandom number generator [27], and an ODE solver [16], among others.

Alternatively, more automated verification efforts, including for networks, typically omit one of these steps: either they prove that the implementation correctly refines the model, but don't prove important properties of the model, or they prove that the model has the correct high-level properties but don't connect the abstract model to an implementation.

Most network function verification – the verification of individual network components like firewalls and load balancers – focuses on verifying that the program correctly implements a functional model and treats these models as the high-level specification [28, 29, 22]. They enable automation by heavily restricting the use of state, encapsulating specific data structures in custom libraries (see §9 for more detail). Verifying properties of entire networks [8, 13, 32] involves using simpler models tractable for fully automated verification tools like SMT solvers without verifying implementations.

Many end-to-end network functions make heavy use of state and require sophisticated higher-level reasoning, including both complex invariants describing the relationship between the state and the input/output packet streams, as well as mathematical reasoning in the domain of interest; for instance, the error-correction algorithm in the system we verify is based on linear algebra over finite fields. Proving the correctness of an implementation further involves reasoning about memory, integer overflow, and undefined behavior (we do not prove all this, but we describe how it could be done in §8.4). Therefore, we need a tool capable of reasoning at all of these levels; we use the Coq interactive theorem prover, a higher-order, dependently typed logic that is widely used in software verification, formalized mathematics, and programming language research. Coq has a large ecosystem of libraries in a variety of domains that makes proofs about functional models possible [4], as well as libraries to connect high-level proofs with low-level code, such as the Verified Software Toolchain (VST) [5] that enables sound reasoning in Coq about C programs.

3 Specifying End-to-End Network Functions

To further illustrate the challenges involved when specifying end-to-end network functions, we consider a hypothetical simplified packet reorderer

```
function addPacket(p)
    if p.seqNum ≥ e then
        insert(p, s);
        p.timeout ← currTime() + t
    end if
end function

function popPacket
    p ← s.head
    if p.timeout ≥ currTime() or
p.seqNum = e then
        e ← p.seqNum + 1;
        forward p; popPacket
    end if
end function

global e, s
function main
    while true do
        p ← receive(); addPacket(p); popPacket
    end while
end function
```

Fig. 1. A simple packet reorderer with intended invariants (1) s is sorted by sequence number and (2) all sequence numbers in s are at least e, but arithmetic mod 2^{32} results in a gap between intention and reality

(Figure 1). The reorderer keeps a list s of packets sorted by sequence number and the index e of the next expected sequence number; on arrival addPacket(p) adds the input packet p to s, and popPacket forwards in-order and timed-out packets, updating e.

Even this simple program is trickier to specify than it may appear. A natural specification is that all packets outputted by popPacket are from the sender and appear in sorted order. However, even this illustrates many of the challenges we will face in writing such a specification and proving that the program satisfies it:

1. This specification depends on the entire *stream* of sent and received packets. The output of a single call to popPacket cannot be specified except in reference to the larger stream.
2. To prove the program correct against this specification, we need to maintain invariants about the state (Figure 1), which may themselves depend on previously sent or received packets.
3. Such a reorderer may run for a long enough time for sequence numbers to wrap around (violating the intended invariants).
4. Even with all of the above, this is a very weak spec: an implementation that dropped every packet could satisfy it. If we additionally wanted a guarantee about packets being returned, we would need to reason both about packet loss and about how many packets arrive before timing out; this depends on the amount of delay, reordering, and duplication in the network.

We will address each of these issues in the context of a packet error-correction system; many of our methods and formalizations are applicable to more general end-to-end network functions.

4 A Packet Error-Correction System

Often, networks deal with loss by retransmitting lost data, but in many cases, this is expensive or impossible (e.g., due to latency requirements or limited storage at the sender). Instead, one can use an *error-correcting*

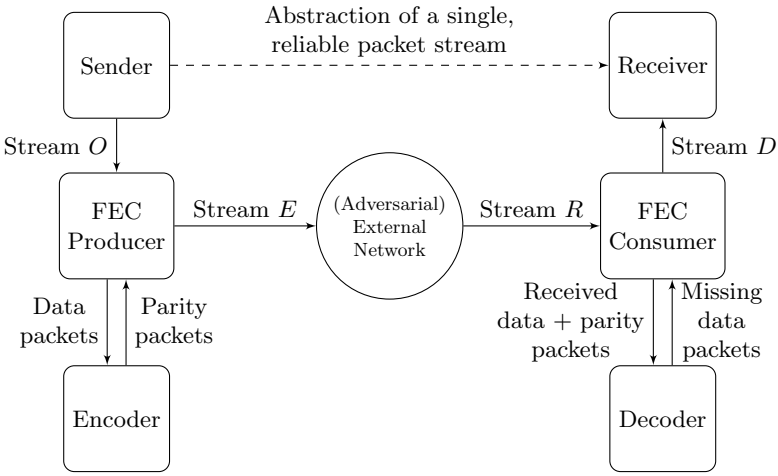

Fig. 2. FEC System Architecture

code, carefully encoding the data to introduce some redundancy, which allows the original data to be recovered even with some loss. In networks, this technique is known as *Forward Error Correction* (FEC).

We verify an FEC implementation developed by Bellcore (now Peraton Labs) based on Reed-Solomon coding [23]. The core encoding/decoding algorithm [18] was developed about 25 years ago;[1] the buffer- and packet-management system is about 8 years old. The program is written in C and has been used in various networking projects to support resilient communication, most recently in the DARPA EdgeCT program. We use an existing implementation rather than writing our own to show how our methods can be used to derive specifications from existing but unspecified code, to demonstrate that such analysis can find bugs in real code, and to explain how we can improve the code to create a new program that is simpler, more reliable, more efficient, and correct.

The program architecture is shown in Figure 2. We identify four streams of packets given as inputs and outputs to different parts of the system; our specification will reference these streams. The Sender (some higher-level application) sends a stream of packets (the original stream O) to the FEC Producer, which calls the Encoder to produce parity packets. The received data packets and the generated parity packets are both sent over the network as the encoded stream E. Here, there may be some adversary or other conditions in the network causing loss, delay, reordering, and/or duplication. This results in the received stream R (some subset of E with reordering and duplication) arriving at the FEC Consumer, which calls the Decoder to reconstruct missing packets. The resulting decoded stream D is sent to the Receiver. The Sender and Receiver (and any other higher-level applications) believe that they are communicating with each other over a mostly reliable connection.

[1] See Cohen, et al. [12] for a more detailed history of this algorithm.

The FEC system can be broken into two pieces: the Encoder/Decoder, which implements the Reed-Solomon erasure code to determine the parities and reconstruct missing packets, and the Producer/Consumer, which sends and receives packets and decides which packets to send to the Encoder/Decoder to recover missing data. The Reed-Solomon code is a *block code* – it encodes a batch of k packets, producing h parity packets such that if at least k of these $k + h$ total packets are received, the Decoder can recover all packets in the batch. Accordingly, the Producer receives packets and forwards them along, marking each with some metadata to identify their batch and their data/parity status and storing copies of these packets until k have been received, at which point it calls the Encoder to produce parities, outputting the result. The Consumer groups incoming packets into their batches and calls the Decoder when k packets in a batch have been received. It also periodically times out stored batches to prevent long search times and limit memory usage.

5 Developing a High-Level Specification

Any attempt to develop a specification for this system immediately encounters all of the challenges described in §3, though the stored state is much more complicated and the timeout mechanism much less predictable than for the simple reorderer (§5.1). Comments in the existing C code indicate that the program was supposed to handle loss, reordering, duplication, and delay; our spec must account for these as well.

5.1 Implementation-Specific Behavior

Beyond the considerations in §3, efforts to formulate a spec for this FEC implementation encounter two additional challenges. First, there is no "natural" specification; it is entirely unclear what such a system should guarantee. FEC cannot guarantee packet recovery if too many packets are dropped, so the program might be expected to provide some guarantees on recovery in a well-behaved network environment or it may operate in a best-effort fashion to attempt to recover as much data as possible.

Second, the program as written does not satisfy any reasonable spec. Formally, we cannot yet call this a bug, as the program makes no guarantees about its behavior; indeed, part of our motivation in giving a formal spec is to be able to concretely identify bugs. Yet in the course of attempting to derive a specification from the code, we discovered (and fixed, see §7) many problems, which we grouped into 3 categories:
The first consists of issues that should be considered bugs under any reasonable specification.

1. The code leaks memory. Some of the memory leaks are acknowledged by the code's comments; others are not. Due to the unpredictable timeout mechanism (see below), the data structure to store the batches in the Consumer can grow arbitrarily large even in "nice" cases (e.g. all the packet arrive in order).
2. The code implicitly (and seemingly unintentionally) casts between signed and unsigned ints.

The second category consists of behaviors that could cause a serious problem: hallucinatory "reconstruction" of packets that were never sent.

3. The program does not handle sequence number (and integer) wrap-around correctly; it uses ordinary integer comparison rather than serial number arithmetic [10]. If enough packets arrive, it can group packets into batches incorrectly, producing garbage packets that are "recovered" and sent to the Receiver. This is a problem if program handles packet streams with more than $2^{31} - 1$ packets,[2] which is a fairly small number of packets at current network speeds.

The third category consists of behaviors that cause the program to fail to recover all packets it could plausibly recover.

4. The implementation does not call the Decoder unless the kth packet received in a batch is a parity packet; thus, with even a small amount of reordering, a recoverable batch can be ignored.
5. With only small amounts of reordering, the Consumer ignores the received packet, forwarding it to the Receiver without storing it.
6. Timeouts are handled inconsistently and unpredictably. They not only prevent us from giving guarantees about the recovered packets, but violate any notion of *locality* about packet-batches – that is, whether a batch is recovered does not only depend on the packets in that batch. In particular, both of the following can occur:
   (a) An input packet can be forwarded without being stored if some *other* batch has timed out.
   (b) In other cases, batches that should time out do not, as long as no packet from a later batch arrives. This, tiny changes in reordering and/or delay can change whether a batch is recovered.

5.2 Layers of Specification

How then, should we formulate a specification so as to capture which of these behaviors should be regarded as bugs? We are interested in knowing both *what guarantees the program gives in good scenarios* (defining these appropriately) as well as *how bad things can be in bad/adversarial ones.* Our approach is to design different layers of specification based on various assumptions about the external environment. For the FEC system, we first want the following under all circumstances:

Property 1. The program does not crash, leak memory, access invalid memory, have signed integer overflow, or use undefined behavior.

Under this spec, items 1 and 2 in the above list are definitely bugs. We note that unsigned integer overflow (carry) is expected due to sequence numbers; we explicitly account for this below.

Next, we identify two properties which must hold to give the following principle: the higher-level Sender/Receiver should never be worse off for having used FEC (as opposed to just sending packets and accepting loss). The FEC system should not drop any packets that are correctly sent and received, and it should not create any invalid packets that were not originally sent:

[2] This implementation uses a custom sequence number that counts packets, not bytes.

Property 2. Suppose a data packet (i.e., a packet from stream O) is in the received stream R. Then it is in the decoded stream D.

Property 3. Every packet in the decoded stream D is in the original stream O.

Property 3 does *not* hold of the current implementation due to bug 3 above. To rule out sequence number wraparound, we need some assumption about the environment. To fix this, we change the program to use 64-bit sequence numbers and assume that no more than $2^{63}-1$ packets are sent.[3] Additionally, we need serial number arithmetic for other comparisons that could be affected by wraparound.

Thus, we have two levels of specification: the FEC system should *always* satisfy Properties 1 and 2, and if at most $2^{63}-1$ packets are sent, it satisfies Property 3 (of course, it will satisfy Property 3 in other settings as well, but we do not prove this). These properties claim that even in adversarial network conditions, the FEC system will not do anything too bad. But this spec is still quite weak; even a system that did nothing but forward data packets and ignore parities could satisfy it. We want to give a stronger spec – one that guarantees, under "normal" network conditions, that the FEC system actually ensures reliable delivery by recovering lost packets.

We expect the ith batch to be recovered under the following condition:

Condition 1. k and h are fixed for all packets. $0 \leq i \leq \frac{|O|}{k}$, and at least k packets of the $k+h$ packets between position $i(k+h)$ and $(i+1)(k+h)$ in stream E appear in stream R.[4]

We would like to say something like the following:

Property 4. Suppose that Condition 1 holds for i. Then packets ik to $(i+1)k$ from stream O appear in D.

In other words, if the FEC parameters are fixed and no more than k packets in the batch are lost, then all packets in this batch are received by the Receiver. It immediately follows from this property that if *all* batches are recoverable (no more than k packets in each batch are lost), then all packets are received. Combined with Property 3, this implies that streams O and D have exactly the same packets. But Property 4 does not hold unconditionally: a batch can timeout before it is recovered; furthermore, bugs 3-6 cause violations of this property even if a particular batch did not timeout. Our next step is therefore to determine assumptions under which Property 4 holds, which involves detailed reasoning about the external network environment.

[3] This is a safe assumption. At gigabit speeds, even if each packet were only 1 bit, wraparound would only occur after 250 years. Alternatively, we could assume weak bounds on reordering, duplication, etc to ensure that sequence numbers are never ambiguous. But we would like Property 3 to hold under *any* network behavior.

[4] This loss condition is not ideal: it reveals the batch structure of the FEC algorithm. However, other formulations (for example, that k out of every $k+h$ consecutive packets are received) are overly restrictive or do not correctly capture the condition.

$seq[i]$	1	2	3	6	4	5	7
$RI[i]$	1	2	3	4	5	6	7
$d[i]$	0	0	0	−2	1	1	0

$seq[i]$	1	4	3	5	3	8	7	6
$RI[i]$	1	3	4	5	x	6	7	8
$d[i]$	0	−1	1	0	x	−2	0	2

Fig. 3. Reorder Density (RD) (a) without and (b) with duplicates and drops

6 Formalizing Properties of Packet Streams

To prove that the FEC program actually recovers certain batches, we need a way to state that all packets in a batch arrive before timing out. In other words, we need the notion that packets sent at similar times from the Producer (close together in E) should arrive within some specified time interval at the Consumer (reasonably close together in R). This is not unique to FEC; any end-to-end program with timeouts and operating on groups of elements will need similar reasoning.

But reasoning about this is difficult: packets can be delayed, dropped, reordered, and duplicated, so we cannot assume direct relationships between a packet's position in E and its position in R. Instead, we will quantify and formalize well-studied, empirical metrics for each of these features and prove that under reasonable bounds on these metrics, batches will not be timed out before they are completed. Since timeouts serve the purpose of identifying exceptional circumstances under which we should *not* expect some packets to be received, this approach makes sense; if the external network behaves reasonably, timeouts should not prevent otherwise recoverable batches from being recovered. To ensure this, we will modify the program's timeout mechanism along the way to simplify it and ensure locality.

6.1 Reordering

Measuring packet reordering is a well-studied problem; metrics for doing so are summarized in RFCs 4737 [19] and 5236 [15]. Some metrics count the number of reordered packets; other quantify the extent of the reordering, which is more useful to us. One such metric is Reorder Density (RD) [7, 21], which measures the *displacement* of each packet, or the difference between the packet's arrival position and its position in the correctly ordered stream, ignoring duplicates. In a comparative study, RD compared favorably to a variety of other reordering metrics on its robustness to packet loss and duplication, ability to capture reordering, usefulness in evaluating network behavior, time and space complexity, and more [20]; the same study found that reordering events are frequent but small. This validates our approach; most reordering is quite small, so we can safely assume a bound on the maximum displacement that the vast majority of packet streams will satisfy.

Figure 3 shows how RD is computed: the input sequence is compared with the Receive Index (RI) sequence, indicating the in-order arrivals; these values are subtracted to get the displacement (d) of each packet. Duplicate packets are ignored; they have no d or RI value, and dropped

packets are skipped in the RI sequence. We choose RD as our reordering metric and we will assume a bound d on the displacement of each packet between the sent stream E and the received stream R. §8.3 discusses our formalization of RD in Coq.

6.2 Duplication and Timeouts

Bounds on reordering help us prove that packets in the same batch are received before the batch times out by quantifying how many packets can arrive in between packets in the same batch. However, duplicate packets cause problems: not only could arbitrarily many packets arrive in the middle of a batch (without a further bound), but reordering metrics like RD intentionally ignore duplication.

Metrics for duplication are sparse; RFC 5560 [24] defines a metric that simply counts the number of occurrences of each packet, but it is unclear if this metric has ever been empirically studied. We instead care about how spread out packets can be, so we use the following condition: there is a bound m such that any two duplicate packets in R have at most m packets between them. This is inspired by RD: if we imagine duplicate packets as two different packets sent from the sender in sequence, then this metric is very close to difference between their displacements.

To reason about timeouts, we must combine the assumed bounds on reordering and duplicates with an assumption about the arrival times of packets close together in R. But this is indirect and unsatisfactory: it depends on network speeds and congestion, and does not allow the Producer to pause between batches. Moreover, reasoning about duplication and reordering together is difficult; this approach results in only weak, multiplicative bounds. Instead, we argue that the timeout mechanism should be changed: instead of measuring in seconds, we should count *the number of unique packets received*. This choice improves the program, the spec, and the proofs in several ways. The sender is allowed to wait between packet arrivals or batches; the condition does not depend on network speeds or system time (making the Consumer a pure function of the packet inputs). Moreover, this improves efficiency: the program already checks for duplicate packets, but now the size of the data structures can be bounded exactly; they also do not depend on network speeds. Finally, reasoning is much simpler: RD is naturally expressed in terms of unique packets; aligning our timeout mechanism with this allows us to reason about reordering and duplication separately and give stronger, additive bounds (§8.3).

With all of these features in mind, we can write our final strong spec, first specifying the bounds on the external conditions and some parameters to prevent overflow.

Condition 2. The following bounds hold:

1. For all packets, the displacement between E and R is bounded by d.
2. Any two identical packets in R have at most m packets between them.
3. The timeout threshold is at least $k + h + 2d + m$.

Condition 3. The timeout threshold is smaller than 2^{31}, all sequence numbers are unique and less than 2^{63}, $0 < k \leq 127$, and $0 < h \leq 128$.[5]

Property 5. Suppose Conditions 2 and 3 hold and Condition 1 holds for i. Then all packets in batch i (packets ik to $(i+1)k$ in O) appear in D.

This specification tells us that the program guarantees recovery of certain packets under reasonable network conditions. Now, we have two tasks: correct the program's bugs so that it satisfies Properties 1, 2, 3, and 5 and then prove that this is the case.

7 A New Program

To fix the problems described in §5.1, we make some modifications to the source program:

- We fix the memory leaks resulting from the lack of free after malloc.
- We use 64-bit sequence numbers instead of 32-bit ones (§5.1).
- For all sequence number comparisons (including the batch ID numbers, which are based on the first packet's sequence number), we use serial number arithmetic (§8.3), which handles wraparound correctly.
- In the Consumer, we change the timeout mechanism to count unique packet arrivals rather than seconds (§6.2). In reality, this is an estimate (packets may have timed out, causing duplicates to be identified as unique); we account for this in our correctness proofs (§8).
- Finally, we completely change the timeout mechanism. Before, the Consumer only timed out batches if a packet from a previous batch (before the latest) arrived, it only ever timed out a single batch, and it violated locality as described in §5.1. The new implementation iterates through the entire list after each arrival, deleting all timed-out batches.

With these changes, the system is more reliable, more predictable, and recovers more packets. However, the new timeout mechanism seemingly reduces the performance, adding iteration through the batch list each time. But this is not necessarily the case. Before, the batch list could be arbitrarily long, and thus a single iteration could take much longer (even in typical cases, such as a packet arriving in the batch immediately preceding the latest). More importantly, the lack of iteration in the previous implementation was really a bug, both because it led to space leaks and because the unpredictable timeout mechanism made the system recover fewer packets than it otherwise could have. This would degrade the performance further if retransmissions were required to recover these missing packets. Finally, we note that our new approach could be implemented efficiently with the right data structures: a hash table to identify the batch and a priority queue to remove old batches would reduce the time per arrival to logarithmic in the number of batches, which itself is kept quite small with the new timeout mechanism.

[5] The k and h bounds arise from the FEC algorithm.

8 Proving the Program Correct

8.1 Functional Models and Data Structures

As §2 describes, we create a functional model of the program, which we prove correct according to the various high-level specs (Properties 1, 2, 3, and 5; we mainly focus here on Property 5). Our models are functional programs in Coq, one Coq function closely matching each C function. We use machine-length integers and data structures closely mirroring those in the C program. The models of the Producer and Consumer take as input some internal state, the current packet, and some external state, returning the updated internal state and a list of output packets; from this, we define iterated versions that input and output packet streams, updating the internal state with each arrival.

Representing the state of each function is a crucial ingredient in our proofs. The Producer and Consumer operate over streams of packets, passing appropriate batches to the core Encoder/Decoder; to link these streams together, we introduce a Block data type, which bundles together the data and parity packets in a batch (using our separate formalization of IP and UDP packets) along with additional metadata (batch ID, FEC k and h values, etc). This abstraction serves two purposes: we can reason about the state of each function in a similar way (the Producer stores an option Block, and the Consumer stores a list Block, each representing the batch(es) in progress) and we can view the E and R packet streams alternatively as a stream of Blocks satisfying particular invariants. This allows us to reason about the Producer and Consumer separately and provides a common view of the inputs, outputs, and internal state.

8.2 Linking the Producer and Consumer

The Block abstraction helps us in the following way: the Producer maintains a currently-in-progress Block; when a batch is complete, it clears its internal state and begins the next batch. Thus, the Blocks comprising the stream E consist of those formed by the Producer. Meanwhile, the Blocks in stream R are *subblocks* of those from E – since packets may be dropped but no new packets can be created. The internal state of the Consumer stores the currently-in-progress batches as Blocks, each of which is a subblock of a Block from R (some packets may be dropped due to timeouts). Therefore, our basic proof strategy is to prove properties of the Blocks in E by proving invariants about the internal state of the Producer, then proving that these properties hold of subblocks and therefore hold of the Consumer's state as well.

For example, the Consumer calls the Reed-Solomon Decoder on a batch when it receives k packets. For the operation to be valid, all packets from k to $k+h$ in the batch must be parity packets produced by the Reed-Solomon Encoder from some input data consistent with the received data packets (packets 0 to $k-1$ in the batch). Then, the Decoder will recover the original input (proved in Coq in [12]). We prove that the Blocks stored in the Consumer have this structure (which we call *well-formed*) by showing that the Producer creates a batch by appropriately calling the Reed-Solomon Encoder and that subblocks remain well-formed.

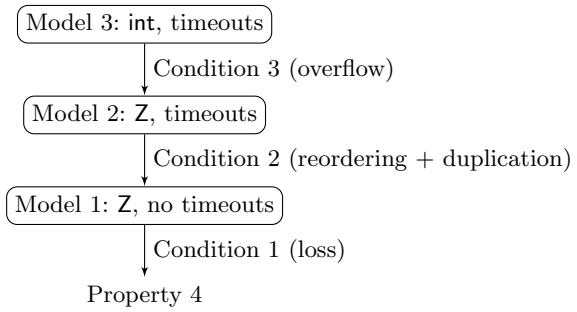

Fig. 4. Consumer Refinement Proofs

8.3 The Consumer

The Consumer is complicated, and its proofs must reason about the Decoder correctness, sequence number wraparound, and timeouts. To handle this cleanly, we use a refinement-based approach, writing 3 layers of functional models (Figure 4). Model 1 uses mathematical (unbounded) integers and has no timeouts; under the loss assumptions, we prove that it satisfies Property 4. Model 2 adds timeouts; we prove that under the reordering and duplication bounds, it is equivalent to Model 1. Model 3 uses machine-length integers and serial number arithmetic; we prove that under the bounds assumptions for sequence numbers and the timeout threshold, it is equivalent to Model 2. We compose all of these proofs to show that Property 5 holds of Model 3.

The Consumer with no timeouts Proving Model 1 correct follows from the observation that, with no timeouts, the Blocks stored in the Consumer's state are *exactly* the blocks of the received stream R. By the loss assumption, the ith batch has at least k packets in the stream, so when the kth packet arrives, the Consumer will call the Reed-Solomon Decoder and (by the well-formedness result above and [12]), output the missing data packets. There is some subtlety; the Block abstraction is useful for avoiding reasoning about reordering and duplication when not needed, as it only notes the presence of a packet, not its location in the stream. Thus, to lift our location-based loss condition (Condition 1) to the Block level, we show that packets $i(k+h)$ to $(i+1)(k+h)$ in E comprise exactly the packets of some block in E.

The Consumer with Timeouts With timeouts, it is no longer the case that the Consumer's blocks are those of R; instead, each is a subblock of a block from R (and thus of a block from E). To prove Models 1 and 2 equivalent, we show that the bounds on reordering and duplication imply that all packets in a batch arrive before the batch times out. We first formalize the Reorder Density metric (§6.1) in Coq. Since this metric ignores duplicates, we can express it naturally via the sent and received packet streams with duplicates removed. This fits very

nicely with our packet-based timeout mechanism; a packet's arrival time is its index in deduplicated received packet stream. Thus, we can relate the RI and diplacement values directly with the timing mechanism and show the following:

Theorem 1. *Consider sent and received packet lists E and R, with no duplicates in E, and suppose that the displacements of all arriving packets are bounded by d. Then if packets p_1 and p_2 are separated by at most n packets in E, they are separated by at most $n + 2d$ unique packets in R.*

This theorem is intuitive – the worst case is that each packet moves d spaces in the "opposite" direction – but is nontrivial to show; it requires reasoning about the precise relationship between RI, displacement, and sequence numbers. However, the duplication case is independent and much simpler:

Theorem 2. *Suppose that at most m packets appear between any pair of duplicates in R. Suppose that p_1 and p_2 are separated by at most n packets in E and that p_1 arrives before p_2 in R. Then, p_2 arrives at most $n + m$ timesteps after p_1.*

These theorems imply a bound on our timeout threshold: any value at least $k+h+2d+m$ suffices, where k and h are the FEC parameters, d is the displacement bound, and m is the duplication bound. This bound is additive rather than multiplicative and thus much tighter than those achievable with the time-based timeout mechanism. In practice, k and h are often smaller than 10, d has rarely exceeded 50 in real-world tests [7, 20] (though these studies are 15-20 years old – higher network speeds may result in higher displacements), and m is not measured but seems likely to be no larger than d (we would expect more reordering than duplication in practice). Thus, a threshold of a few hundred packets is likely sufficient to handle normal network conditions; our proofs are all parametric in the choice of threshold.

With these results, we prove by a series of invariants that Model 2 never times out blocks that have packets yet to arrive; this implies that its output is the same as Model 1.

Machine and Sequence Number Arithmetic Finally, for our machine-length-integer version, we formalize Serial Number Arithmetic from RFC 1982 [10] and prove it correct. We can define serial number comparison efficiently with the following C function:[6]

```
int seq_cmp(unsigned int i1, unsigned int i2) { return ((int)(i1−i2)); }
```

a is considered smaller than b if `seq_cmp(a, b)` is negative, equal if zero, and larger otherwise. In this definition, integers close to each other (within 2^{31}) are comparable, with the expected behavior (e.g., $2^{32} - 1 < 0$). To formalize this in Coq, we define functions `seq_lt`, `seq_eq`, `seq_gt` that compute the appropriate comparisons, and we prove the theorem:

[6] We use 32-bit as an example; our proofs are generic and we also need the 64-bit case.

Theorem 3. *Let* repr *be the function that gives the 32-bit representation of an integer (i.e.,* $z \mod 2^{32}$*). Let* z_1 *and* z_2 *be unbounded integers such that* $|z_1 - z_2| < 2^{31}$*. Then,* seq_lt(repr($z1$),repr($z2$)) $= true \iff z1 < z2$.

In other words, given two integers close to each other, sequence number comparison correctly decides which is smaller, even with wraparound (and likewise for the other two functions). With these results, formalized using CompCert's [17] machine-length integer Coq library, we show that all sequence number comparisons are performed between unsigned integers whose values are within 2^{31} (this results in the 2^{30} upper bound on the timeout threshold); hence, by Theorem 3, all uses of serial number arithmetic exactly correspond to unbounded integer comparison and therefore Models 2 and 3 are equivalent.

Finally, we compose all 3 layers and the Producer proofs to prove a single theorem about the machine-length functional model;[7] this model should exactly correspond to the C program's behavior:

Theorem 4. *Suppose Conditions 2 and 3 hold and Condition 1 holds for* i*. Then, packets* ik *to* $(i+1)k$ *in* O *appear in* D*, the decoded stream formed by the Producer and Model 3 of the Consumer.*

This theorem provides guarantees even if particular batches are unrecoverable. If all batches are recoverable, we have the following:

Corollary 1. *Suppose Conditions 2 and 3 hold, Condition 1 holds for all* $0 \leq i \leq \frac{|O|}{k}$*, and* k *divides* $|O|$*. Then all packets in* O *appear in* D*.*

8.4 From Coq to C code

From here, we could use the Verified Software Toolchain (VST) [5] to prove that the C code refines our machine-length functional model, composing the proofs with our functional-model proofs above. VST is proved sound with respect to the CompCert verified C compiler [17], so our proofs would hold down to the assembly-language level. Additionally, this would also show that the program is free of memory leaks, undefined behavior, or I/O beyond what is written in the specification. This approach is feasible for verifying real C code [12, 27, 3], and we designed our functional model to adhere closely to the C code and be verifiable with VST, for instance using CompCert's machine-length integers.

We did not carry on our verification down to the C code, as our primary focus was in developing tools for specifying and reasoning about end-to-end network functions at a higher level. We note that even without a VST proof, we identified and fixed many bugs in our target program, leading to simpler, more predictable, and correct code, demonstrating that formal specification and verification techniques are useful even partially applied.

[7] The Producer only compares integers between 0 and 256; wraparound is impossible.

9 Related Work

As discussed in §2, most previous network verification efforts take one of two approaches, both of which are orthogonal to our work. *Network verification* models a network as a simpler abstraction, proving properties with SMT solvers and other automated methods. Some recent work on control plane verification includes Minesweeper [9], Tiramisu [1], Hoyan [26], SRE [32] and Timepiece [2]. In the data plane, verification tools include Katra [8] and Flash [13]. *Network function verification* develops verified per-packet network function implementations (NAT, firewall, load balancer, etc); examples include VigNAT [29], Vigor [28], Klint [22], and Gravel [31]. While these network functions can maintain state updated on packet arrival, this use is restricted to particular data structures separately (interactively) verified against a functional specification or axiomatized using SMT formulas. These restrictions, and others on the presensece of loops, pointers, and the overall program structure, enable automation. Crucially, the specifications treat these data structures as abstract, enabling reasoning about how an individual packet interacts with them but not about how this state relates to previous packets.

Other work has focused on verifying transport-layer components. Cluzel, et. al. [11] verify a TCP implementation by translating to SPARK., verifying against the protocol expressed as a state machine – the reasoning is still per-packet, since they do not prove higher-level reliability guarantees. Arun, et. al. [6] verify congestion control algorithms against first-order logic specifications using SMT solvers; they prove higher-level properties, and the SMT-outputted packet traces may include duplicates and timeouts. Beyond network components, Verdi [25] is a Coq library to verify distributed systems; it allows the user to define the assumed network environment, including reordering, duplication, loss, and timeouts; however, this controls the presence, not the extent, of these effects.

10 Conclusion and Future Work

Formally specifying and verifying end-to-end network functions involves numerous challenges, many of which are widely applicable to other real-world programs. Specifications may be unclear, dependent on external conditions, and more complicated to express than via simple inputs and outputs (as the behavior may depend on previously processed packets). We show that a layered, interactive approach can clarify assumptions, identify intended guarantees, and lead to cleaner, more predictable, provably correct code. We would like to demonstrate our methods on other end-to-end transport-layer network functions and even more general "dusty deck" programs that are underspecified and encounter similar issues. We would also like to automate some of our verification and to develop more general libraries for reasoning about external conditions like timeouts and packet reordering. We believe that our techniques are applicable to a wide variety of complex software, and we hope that formal specification and verification can become standard ways to analyze and improve real-world systems.

Acknowledgment.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001120C0160.

References

1. Abhashkumar, A., Gember-Jacobson, A., Akella, A.: Tiramisu: Fast multilayer network verification. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20). pp. 201–219. USENIX Association, Santa Clara, CA (Feb 2020)
2. Alberdingk Thijm, T., Beckett, R., Gupta, A., Walker, D.: Modular control plane verification via temporal invariants. Proc. ACM Program. Lang. **7**(PLDI) (June 2023). https://doi.org/10.1145/3591222
3. Appel, A.W.: Verification of a cryptographic primitive: SHA-256. ACM Trans. Program. Lang. Syst. **37**(2) (Apr 2015). https://doi.org/10.1145/2701415
4. Appel, A.W.: Coq's vibrant ecosystem for verification engineering (invited talk). In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 2–11. CPP 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3497775.3503951
5. Appel, A.W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., Leroy, X.: Program Logics for Certified Compilers. Cambridge University Press, USA (2014)
6. Arun, V., Arashloo, M.T., Saeed, A., Alizadeh, M., Balakrishnan, H.: Toward formally verifying congestion control behavior. In: Proceedings of the 2021 ACM SIGCOMM 2021 Conference. p. 1–16. SIGCOMM '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3452296.3472912
7. Bare, A.A., Jayasumana, A.P., Banka, T.: Metrics for degree of reordering in packet sequences. In: Proceedings LCN 2002. 27th Annual IEEE Conference on Local Computer Networks. p. 0333. IEEE Computer Society, Los Alamitos, CA, USA (nov 2002). https://doi.org/10.1109/LCN.2002.1181802
8. Beckett, R., Gupta, A.: Katra: Realtime verification for multilayer networks. In: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22). pp. 617–634. USENIX Association, Renton, WA (Apr 2022)
9. Beckett, R., Gupta, A., Mahajan, R., Walker, D.: A general approach to network configuration verification. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication. p. 155–168. SIGCOMM '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3098822.3098834
10. Bush, R., Elz, R.: Serial Number Arithmetic. RFC 1982 (Aug 1996). https://doi.org/10.17487/RFC1982
11. Cluzel, G., Georgiou, K., Moy, Y., Zeller, C.: Layered formal verification of a TCP stack. In: 2021 IEEE Secure Development Conference (SecDev). pp. 86–93 (2021). https://doi.org/10.1109/SecDev51306.2021.00028

12. Cohen, J.M., Wang, Q., Appel, A.W.: Verified erasure correction in Coq with MathComp and VST. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. pp. 272–292. Springer International Publishing, Cham (2022)
13. Guo, D., Chen, S., Gao, K., Xiang, Q., Zhang, Y., Yang, Y.R.: Flash: Fast, consistent data plane verification for large-scale network settings. In: Proceedings of the ACM SIGCOMM 2022 Conference. p. 314–335 (2022). https://doi.org/10.1145/3544216.3544246
14. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J., Parno, B., Stephenson, J., Setty, S., Zill, B.: Ironfleet: Proving practical distributed systems correct. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP). ACM - Association for Computing Machinery (October 2015)
15. Jayasumana, A., Piratla, N., Banka, T., Bare, A., Whitner, R.: Improved packet reordering metrics. RFC 5236, RFC Editor (June 2008)
16. Kellison, A.E., Appel, A.W.: Verified numerical methods for ordinary differential equations. In: Isac, O., Ivanov, R., Katz, G., Narodytska, N., Nenzi, L. (eds.) Software Verification and Formal Methods for ML-Enabled Autonomous Systems. pp. 147–163. Springer International Publishing, Cham (2022)
17. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (Jul 2009). https://doi.org/10.1145/1538788.1538814
18. McAuley, A.J.: Reliable broadband communication using a burst erasure correcting code. In: Proceedings of the ACM Symposium on Communications Architectures & Protocols. p. 297–306. SIGCOMM '90, New York, NY, USA (1990). https://doi.org/10.1145/99508.99566
19. Morton, A., Ciavattone, L., Ramachandran, G., Shalunov, S., Perser, J.: Packet reordering metrics. RFC 4737, RFC Editor (November 2006)
20. Piratla, N.M., Jayasumana, A.P.: Metrics for packet reordering—a comparative analysis. International Journal of Communication Systems **21**(1), 99–113 (2008). https://doi.org/10.1002/dac.884
21. Piratla, N.M., Jayasumana, A.P., Bare, A.A.: Reorder density (RD): A formal, comprehensive metric for packet reordering. In: Boutaba, R., Almeroth, K., Puigjaner, R., Shen, S., Black, J.P. (eds.) NETWORKING 2005. Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications Systems. pp. 78–89. Springer (2005)
22. Pirelli, S., Valentukonytė, A., Argyraki, K., Candea, G.: Automated verification of network function binaries. In: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22). pp. 585–600. USENIX Association, Renton, WA (Apr 2022)
23. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. Journal of The Society for Industrial and Applied Mathematics **8**, 300–304 (1960)
24. Uijterwaal, D.H.A.: A One-Way Packet Duplication Metric. RFC 5560 (May 2009). https://doi.org/10.17487/RFC5560

25. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: A framework for implementing and formally verifying distributed systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 357–368. PLDI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737958
26. Ye, F., Yu, D., Zhai, E., Liu, H.H., Tian, B., Ye, Q., Wang, C., Wu, X., Guo, T., Jin, C., She, D., Ma, Q., Cheng, B., Xu, H., Zhang, M., Wang, Z., Fonseca, R.: Accuracy, scalability, coverage: A practical configuration verifier on a global wan. In: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication. p. 599–614. SIGCOMM '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3387514.3406217
27. Ye, K.Q., Green, M., Sanguansin, N., Beringer, L., Petcher, A., Appel, A.W.: Verified correctness and security of MbedTLS HMAC-DRBG. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. p. 2007–2020. CCS '17, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3133974
28. Zaostrovnykh, A., Pirelli, S., Iyer, R., Rizzo, M., Pedrosa, L., Argyraki, K., Candea, G.: Verifying software network functions with no verification expertise. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. p. 275–290. SOSP '19, New York, NY, USA (2019). https://doi.org/10.1145/3341301.3359647
29. Zaostrovnykh, A., Pirelli, S., Pedrosa, L., Argyraki, K., Candea, G.: A formally verified NAT. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication. p. 141–154. SIGCOMM '17, New York, NY, USA (2017). https://doi.org/10.1145/3098822.3098833
30. Zhang, H., Honoré, W., Koh, N., Li, Y., Li, Y., Xia, L.Y., Beringer, L., Mansky, W., Pierce, B., Zdancewic, S.: Verifying an HTTP Key-Value Server with Interaction Trees and VST. In: Cohen, L., Kaliszyk, C. (eds.) 12th International Conference on Interactive Theorem Proving (ITP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 193, pp. 32:1–32:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.32
31. Zhang, K., Zhuo, D., Akella, A., Krishnamurthy, A., Wang, X.: Automated verification of customizable middlebox properties with Gravel. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20). pp. 221–239. USENIX Association, Santa Clara, CA (Feb 2020)
32. Zhang, P., Wang, D., Gember-Jacobson, A.: Symbolic router execution. In: Proceedings of the ACM SIGCOMM 2022 Conference. p. 336–349. SIGCOMM '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3544216.3544264