Layered Formal Verification of a TCP Stack

Guillaume Cluzel
AdaCore & ENS de Lyon

Kyriakos Georgiou
AdaCore & University of Bristol

Yannick Moy
AdaCore

Clément Zeller
Oryx Embedded

***Abstract*—The Transmission Control Protocol (TCP) at the heart of TCP/IP protocol stacks is a critical part of our current digital infrastructure. In this article, we show how an existing professional-grade open source embedded TCP/IP library can benefit from a formally verified TCP reimplementation. Our approach is to apply formal verification to the TCP layer only, relying on validated models of the lower layers on which it depends.**

***Index Terms*—Network protocols, deductive verification, symbolic execution.**

I. Introduction

By 2030 it is estimated that one trillion Internet of Things (IoT) devices will be deployed and be part of every aspect of our daily life [1]. At the same time, it is evident that security for such systems is a major challenge [2], [3]. Any vulnerability in software libraries that are widely used by embedded systems can potentially open the door for compromising the security of billions of deployed IoT devices. Researchers have shown that such vulnerabilities exist: in 2019, critical security vulnerabilities were discovered in a TCP implementation that has been used by billions of IoT devices for more than 13 years [4], some of them allowing remote code execution.

Networking communication models define the processes of transferring information from one network component to another. Currently, the two most popular networking communication models are the TCP/IP (Transmission Control Protocol/Internet Protocol) [5] and the OSI (Open System Interconnection) [6] models. Layering is an essential design requirement for both models to achieve modularity, flexibility, and abstraction. Applications that only need to use the lower levels' functionality can ignore all the unnecessary upper levels. Figure 1 shows the various layers of the OSI model. Each layer supports a wide range of protocols that are compatible with the layer's specifications. The lowest layers, bottom layers in Figure 1, are more hardware-oriented, while the top layers are more software-specific. While the *application* level has a wide range of protocols that can be used depending on the application specification, the *transport* layer is heavily dependent on the two dominant protocols: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). Today, almost every networking application, including IoT devices, is using either the TCP or the UDP protocols to traffic data between the Network and the Application layers.

Existing research has dealt with formally verifying several protocols of the TCP/IP stack. Early work formalized and proved some key properties of the TCP protocol [7]. In later work, researchers have formally specified TCP, UDP and the Sockets API in HOL and tested existing libraries against the formal specification [8], [9]. More recently, others have formally specified and implemented the SSL/TLS protocol in F* and integrated the resulting implementation in existing libraries [10]. To the best of our knowledge, no prior work attempted to formally verify an implementation of TCP, although the authors of [9] suggest that it would possible to extract an actual implementation in Haskell from their specification.

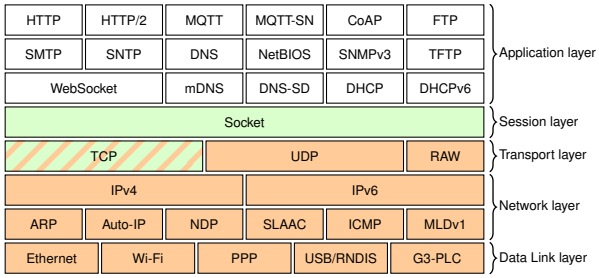

Fig. 1. Overview of the TCP/IP CycloneTCP stack (OSI model). Green parts have been translated into SPARK. They depend on orange parts written in C.

In this paper, we take a practical approach, where the security of an existing TCP/IP stack implementation written in the C programming language is incrementally enhanced by replacing parts of its code with formally verified code in SPARK [11], a subset of the Ada programming language supported by formal verification tools. We chose CycloneTCP, a professional-grade open source embedded TCP/IP library developed by the Oryx Embedded company [12]. Figure 1 shows the protocols supported by the library. This work focuses on the TCP and Socket components of the stack. The library's TCP implementation is meant to conform to the RFC 793 protocol specifications [13]. The quality of CycloneTCP is acknowledged by the AMNESIA:33 report [14], which classifies it as one of the most resilient TCP/IP stacks.

The contributions of this article are the following. First, we show that formal verification can be applied selectively to a layer of an existing TCP/IP library, by translating the TCP layer to SPARK and modelling the lower layers on which it depends. The benefits are three-fold: we found and corrected two bugs in the TCP layer, we proved the correction of that layer, and we defined formal contracts for the upper layers to call the TCP layer. Second, we showed how the models of the lower layers can be validated by symbolic execution with

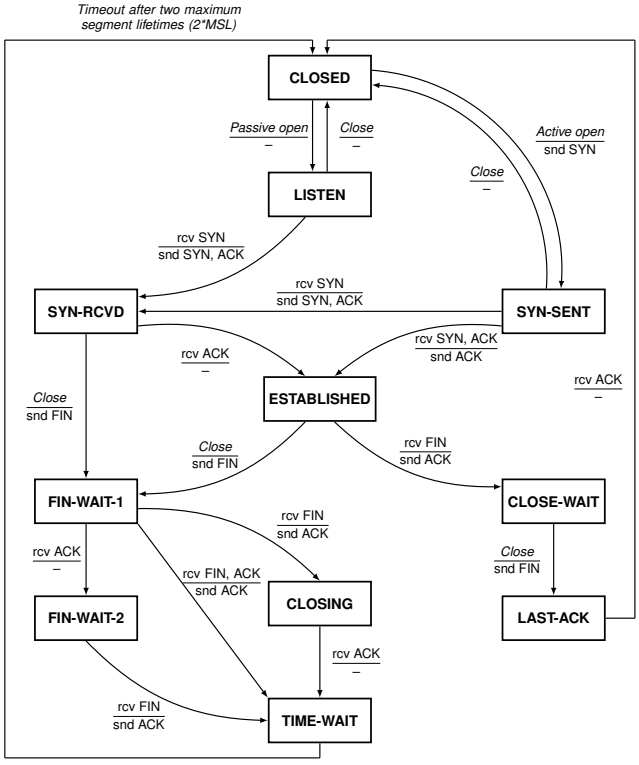

Fig. 2. TCP state machine.

KLEE on the C code of those layers. The tools[1] and code[2] are freely available online.

We present the TCP protocol in section II and the specification techniques used in section III. We explain how we formally implemented in SPARK the TCP user functions in section IV and how we hardened the user's API in section V. Finally we conclude with a summary of the results of our work in section VI.

II. Overview of the TCP protocol specification

TCP is a connection-oriented protocol; a connection between the sender and the receiver must be established before transmitting any data. Furthermore, TCP is a reliable protocol since it guarantees the delivery of all the messages in absence of a network outage and that they are delivered in the order they were sent. It also provides error-checking mechanisms to discard and recover from corrupted data.

Generally, every TCP communication session will go through the following three phases (if no error occurs): 1) opening the connection, 2) sending and receiving the data, and 3) closing the connection. The behavior of the three-phase communication can be described by the state machine given in Figure 2. An edge represents a state transition. An edge-label in the format of $\frac{x}{y}$ represents the actions associated with the state transition, where x is triggering the transition, and y is produced by the transition. The trigger x is either an explicit action like *Close*, or the arrival of specific flags, while y always represents the flags transmitted in response when present. An explicit action is either a user-triggered action (through the user's API) or an action automatically triggered by a timeout event. The flags are embedded in the header section of a transmitted segment and can be one of the following:

- ACK: the last message received by the sender is acknowledged.
- SYN: this flag is sent to establish a connection.
- FIN: this flag is sent to close the connection.
- RST: this flag is sent to reset the connection, often when an error occurred on one or the other side.

The state machine in Figure 2 does not represent the complete protocol specification; for example, it does not reflect error conditions or any actions which are not connected to the state changes. Instead, it gives an overview of all the possible states a TCP connection could reach over its lifetime. Each state has a clear meaning. LISTEN represents waiting for a connection request from any remote TCP and port. SYN-SENT and SYN-RCVD represent the initialization of the connection that ends when ESTABLISHED is reached. FIN-WAIT-1, FIN-WAIT-2, CLOSING, and TIME-WAIT are a group of states that represent waiting for a connection termination request from the remote TCP. CLOSE-WAIT and LAST-ACK represent waiting for a connection termination request from the local user. Finally, CLOSED represents no connection state at all.

TCP is based on a multitasking model. Different tasks can interact and update the socket data structure, used to retain the status of a TCP connection, to handle the various events that can occur within a TCP session. RFC 793, under Section "3.9. Event Processing" page 52, describes a possible implementation of how to handle these events based on three tasks: one for the *user calls*, one for the *arriving segments* and one for the *timers*. The role of each of the three tasks can be summarized as follows. *User calls* refer to functions, namely `open`, `close`, `abort`, `send` and `receive`, that can be called by the user to control the connection, send, or receive data. These functions can trigger transitions between the connection's states since they are intended to control the connection. In the *arriving segments* task, the received segments are being processed, and the corresponding messages are sent back. Transitions between states can be triggered on the reception of a segment. Finally, *timers* control the timeouts, for example, the retransmission timeout to retransmit a message or the time-wait timeout to close the connection after a specified amount of time elapsed. Thus, transitions corresponding to the timeout events can also be triggered by this task. A complete description of TCP specification can be found in the RFC 793 document [13].

The CycloneTCP implementation adopts the three-task-based model and the socket data structure defined by the RFC 793. Thus, it has tasks dedicated to the user's code, to the timers, and to the processing of incoming segments. These three tasks communicate together, either synchronously:

[1] https://www.adacore.com/download
[2] https://github.com/Adacore/Http_Cyclone

a task requests for an action to be done by another task; or asynchronously: some actions are performed on a socket by a task between two uses of the socket by another task. All the uses of a socket are protected by a global mutex, common to all the sockets. It ensures that only one task can modify a socket at the same time.

III. Specification techniques used

A. The SPARK programming language

Ada is a general-purpose, procedural and Object-Oriented programming language that puts great emphasis on the safety and correctness of the program. Some of the safety characteristics of the language are its strong typing system and its extensive compile-time and runtime checks. Common C programming vulnerabilities like using unsafe pointers or improper null termination of strings cannot exist in an Ada program, while issues like buffer overflow or integer overflow can be dynamically captured by Ada's runtime checks and dealt with via exception handlers. Furthermore, Ada provides contracts, such as preconditions and postconditions, as part of the language's standard syntax. Such contracts are vital for explicitly expressing software and verification requirements in the source code to allow for static analyzers or runtime checks to verify that the stated requirements are met.

By subsetting Ada, the SPARK programming language [11] provides the largest possible subset of Ada that is suitable for functional specification and static verification. For instance, in SPARK, expressions are free from side effects, and aliasing is forbidden (no two variables can share the exact memory location or overlap in memory), including when using pointers thanks to the use of an ownership policy [15].

GNATprove [16] is the tool that provides formal verification for SPARK through two kinds of analyses. The first enables flow analysis to check the initialization of variables, verify data dependencies between inputs and outputs of subprograms, and detect useless code. The second uses deductive verification to generate verification conditions for SMT solvers via a weakest-precondition calculus to detect possible runtime errors and violations of functional contracts.

B. Interfacing SPARK and C code

The verification of the entire functional specification of the CycloneTCP library is beyond the scope of this work. Instead, the aim is to provide significant security hardening to mostly-C libraries by selective use of SPARK. Thus, this work focuses on the hardening of the TCP library areas that its original authors designated as the most vulnerable or crucial to conform to their functional specifications, namely, the API. This mainly falls under two categories: a) proving the conformance to the protocol's functional specification for the translated code, discussed in Section IV, and b) hardening the user's API to enforce its correct usage by users, discussed in Section V.

Ada and SPARK offer an easy-to-use mechanism for interfacing with C code. The mechanism allows C-written functions to be called by Ada/SPARK code and vice versa and for representing shared data objects. The memory representations of a shared data object must be the same in both the C and the SPARK code. For instance, if a C function is called with a 32-bits integer argument, the SPARK interface of this function should also be called with a 32-bits integer argument. It is also true for record types. It is the responsibility of the user to ensure this property. Nevertheless, the compiler GCC offers an option `-fdump-ada-spec` that helps the user in writing correct Ada specifications by generating them directly from the C code. The layout of the record types are then the same by construction in C and SPARK.

Although formal verification can only be applied to the SPARK translated code, preconditions and postconditions can be attached to C functions interfaced in SPARK to allow reasoning about the side effects of calling these functions by the SPARK code. Since the functional specification of these contracts cannot be proved, it is the programmer's role to ensure their correctness. Interfacing C and SPARK code can be error-prone if it is not done carefully. Section IV discusses how to mitigate this issue effectively.

C. Dealing with pointers

In the CycloneTCP library, pointers are used both as the type of variables passed to functions and as the type of structure fields; for instance, in the socket structure to store other complex data structures and recursive pointer-based data structures. SPARK uses a pointer ownership model to prevent bugs involving pointers such as memory leak or double free, which imposes some restrictions on how pointers are being manipulated. When a memory area is allocated, a pointer points to this memory area, and thus, it is considered "owned." The ownership of the memory can be transferred, but the memory must be deallocated before the end of the execution of the program.

D. Specifying the frame condition

The socket structure is shared between different functions and contains a large number of fields. Most of the fields are used in library parts that are not considered by this work, and thus, are not relevant for our verification. Thus we use *ghost code* in SPARK (that is, special code only meant for verification and not included in the final executable) to focus the verification effort on the relevant subset of the fields, which also makes the specifications easier to write and understand. This is a solution to the classical problem of specifying the *frame condition* for a function, that is, the set of objects possibly modified by the function. *Ghost code* is used by GNATprove for analysis but is not compiled into the final executable. More specifically, we define a *model* of the actual socket as a ghost function that can be used in contracts: `Model(Sock)` extracts the fields of interest from socket `Sock`. A common usage is within a function postcondition:

```
Model(Sock) = (Model(Sock)'Old with delta
                        State => TCP_STATE_CLOSED)
```

This states that the fields of `Sock` used by `Model` are not changed by the function call, except for the field `State` that

is equal to TCP_STATE_CLOSED after the call. Nothing is stated on the fields that are not selected by Model(Sock). In particular they can have changed during the execution of the function or not.

IV. Conformance to the TCP protocol

A. Extracting a specification from RFC 793

Extracting a specification is mandatory for providing functional contracts to the user's functions. This can be achieved by interpreting the text of RFC 793. However, the text is underspecified, and thus, only one of the possible implementations is used by this work. The following choices were made in our formalization of the TCP specification:

- The transitions between states must respect the order given by the state machine described in Figure 2. In particular, a transition from one state to another is valid only if the state machine allows a transition between these two states.
- RFC 793 describes the conditions for the state of the socket which need to hold when a user's function is called [13, p. 52], to avoid errors. These conditions were turned into preconditions in our formalization.

B. Rewriting the TCP user functions

While being extensively tested against industry standard regression test suites, the original CycloneTCP library provides no formal guarantees that the implementation respects the TCP functional specification. Thus, there is no assurance that the implementation will only allow valid transitions between the different states that a TCP session can exhibit. We aim to use the SPARK technology to verify that the implementation respects the TCP automaton, given in Figure 2. The work is focused on hardening the API of the user task, mainly the high-level user functions, to verify the state transitions. To achieve this, the rest of the TCP protocol, not related to the user task, has to be taken into account. The reason for this is two-fold. First, although TCP user functions can trigger state transitions, the actual transitions are done by other library functions called from TCP user functions. Second, other parts of the library can trigger state transitions during and between TCP user function calls, which can affect the intermediate and final states that a TCP user function can exhibit. The TCP user functions were fully translated to SPARK. SPARK bindings were also provided for most of the rest of the library's C functions to allow their invocation from the SPARK code.

RFC 793 provides all the information needed for the allowed transitions between the different TCP possible states. For any transitions directly triggered by calling the TCP user functions, SPARK contracts, representing transitions allowed by the functional specification, are embedded within the SPARK translated code. The helper function TCP_Change_State is called explicitly every time a state transition needs to be made to update the socket's state. Any incorrect transition allowed by the code will be detected by GNATprove.

The concurrent implementation of the TCP protocol allows for multiple interactions between the tasks, both synchronous (with a call to TCP_Wait_For_Events) and asynchronous (between two user function calls), and thus, numerous possible state changes are possible at any given moment. This makes the functional specification verification significantly challenging, as SPARK does not have a native mode to deal precisely with interactions related to concurrency. To address this, we avoid using concurrency in SPARK and instead we introduced sequential SPARK functions with contracts that model how the different possible interactions can modify the state of a socket. Finding a correct and precise contract for TCP_Wait_For_Events was challenging but needed for the verification of the TCP user functions. The solution is elaborated in the following section.

Algorithm 1 Function to compute the possible state after the completion of a particular event that is requested by a TCP user function.

```
1:  function TCP_Wait_For_Events_Proof(Socket, Event_Mask)
2:      S_last := Socket
3:      E := TCP_Update_Events(S_last)
4:      if (E & Event_Mask) ≠ 0 then
5:          return S_last
6:      end if
7:      for i = 1 to 3 do
8:          S_last := TCP_Process_One_Segment(S_last)
9:          E := TCP_Update_Events(S_last)
10:         if (E & Event_Mask) ≠ 0 then
11:             return S_last
12:         end if
13:     end for
14:     return ∅
15: end function
```

C. Dealing with synchronous exchanges

In the synchronous scenario of interaction between the TCP tasks, the resulting state of a user function can be affected by the rest of the TCP tasks when the TCP user function releases the mutex to allow for other events to occur. In this case, the function TCP_Wait_For_Events is called from the TCP user function to check if the event requested is completed. For example, the TCP_Connect user function calls TCP_Wait_For_Events to wait for the completion of the socket connection event, namely SOCKET_EVENT_CONNECTED. The call to TCP_Wait_For_Events releases the mutex, and when the relevant task receives the appropriate segment, in this case, a segment with the SYN flag, the connection will be established. This will move the socket to the ESTABLISHED state. In the meantime, the TCP_Update_Events function is monitoring for any state changes, and when it notices that the connection is established, it will trigger the SOCKET_EVENT_CONNECTED event, using the OS event mechanism. This will allow the TCP_Connect user function to resume execution.

Essentially, the function TCP_Wait_For_Events releases the mutex to allow for required events to happen. The function TCP_Update_Events detects that the event's expected outcome is completed by monitoring the changes to the TCP states. Then it updates the specified event to be true

in the socket structure, and it raises the desired event to allow resuming the TCP user function that requested the event in the first place. Thus, the `TCP_Update_Events` function is called when a segment is being received and a state change took place. This makes the `TCP_Update_Events` function a perfect candidate to introduce contracts in SPARK and model the possible states after the completion of each event.

To model any possible state changes upon the reception of a segment in SPARK, the function `TCP_Process_One_Segment` is introduced. The behavior of `TCP_Wait_For_Events` is modeled by the ghost function `TCP_Wait_For_Events_Proof` shown in Algorithm 1. It computes the set of possible final states after the completion of an event by calling repeatedly the function `TCP_Process_One_Segment`. It is enough to perform only three iterations in the loop of this algorithm because the maximum path between two states from the TCP automaton, given in Figure 2, that does not require a user interaction has a size of three. Both functions have the same behavior, and thus, the contract of the function `TCP_Wait_For_Events_Proof` is also attached to `TCP_Wait_For_Events`. This ghost function is proved automatically without even requiring a loop invariant, thanks to loop unrolling in GNATprove.

D. Symbolic execution to extract contracts

The soundness of our verification requires all the contracts to be complete and correct; all the possible state transitions of a TCP connection must be modeled and proved. In particular, this is true for the contract of `TCP_Process_One_Segment` that is used to prove the function `TCP_Wait_For_Events`, as it models calls to parts of the library in C that are not proved. To overcome this issue, symbolic execution was used on the original CycloneTCP C code to exhaustively verify any contracts related to transitions triggered from the C parts of the library.

Symbolic execution is a means of executing programs with symbolic values rather than concrete values [17]. Exhaustive symbolic execution can be considered sound and complete (it prevents false negatives and false positives), and thus, it is on par with deductive verification. In [18], the authors present how symbolic execution can be used to improve deductive verification, and in [19] the authors demonstrated the technique's value in inferring contracts. For our work, the KLEE v2.1 symbolic execution engine is used [20]. KLEE is built upon the LLVM optimizer's bytecode, and it offers a very simple yet powerful C interface.

Symbolic execution is used to verify that all the possible paths when executing the `tcpProcessSegment` function will result in a state that respects the RFC 793 extracted specifications. This process requires three steps shown in Figure 3. Firstly, a symbolic incoming segment is created, allowing to account for all possible incoming segments (see lines 1-4). Then at line 11, function `tcpProcessSegment` is executed symbolically by KLEE. Finally, the assertion, `klee_assert`, at line 14 checks if the code leads to a valid state, one that is within the TCP specification. If there is no raised assertion, then the contract represented by the assertion is deemed proved, and the relevant C code respects the TCP specification regarding that contract. The assertion is then manually converted to a SPARK contract and added to the function `TCP_Process_One_Segment`. This enables the SPARK prover to reason about the possible side-effects caused by the C code on the state of a TCP connection and thus, enables the functional correctness verification of the TCP user functions.

```
1  // Create a fake incoming segment
2  TCPheader *segment = malloc(sizeof(TCPheader));
3  klee_make_symbolic(segment, sizeof(segment), "seg");
4  klee_assume(segment->flag <= 31);
5  // Create the socket
6  Socket *sock = malloc(sizeof(Socket)), oldSock;
7  klee_make_symbolic(sock, sizeof(sock), "sock");
8  memcpy(&oldSock, sock, sizeof(Socket));
9
10 // Call the function to process the segment
11 tcpProcessSegment(sock, segment);
12
13 // Check the expected postcondition
14 klee_assert(
15     (oldSock.state == TCP_STATE_ESTABLISHED) ?
16         sock->state == TCP_STATE_ESTABLISHED ||
17         sock->state == TCP_STATE_CLOSE_WAIT ||
18         sock->state == TCP_STATE_CLOSED :
19     (oldSock.state == ...) ? ... )
20 )
```

Fig. 3. Driver for the verification of `tcpProcessSegment` with KLEE

Our choice of combining deductive verification with SPARK and symbolic execution with KLEE was driven by practical considerations which are typical of industrial applications. Using KLEE on user functions would be difficult due to the presence of loops with a possible high number of iterations, which symbolic execution will need to unroll, leading to combinatorial explosion. Using SPARK on the function that processes incoming segments would be desirable, and may be done in the future, but was not doable in the timeframe of this project. Note that KLEE has other limitations compared to SPARK in terms of the verification scope. For example it is lacking a detection of memory leaks, which turned out to be useful in SPARK, see section VI.

V. Hardening the user's API

One of the most significant problems pointed to by the primary author of the *CycloneTCP* library is the incorrect usage of the library's API. More specifically, users of the library might call the TCP user functions in the wrong order and forget to check the return code after some calls. This causes their TCP implementation to behave outside of the functional specification of the protocol. The following sections demonstrate how SPARK can be used to enforce a correct usage of the library.

```
procedure Socket_Connect
   (Sock           : in out Not_Null_Socket;
    Remote_Ip_Addr : in     IpAddr;
    Remote_Port    : in     Port;
    Error          :    out Error_T)
   with
   Pre => Is_Initialized_Ip (Remote_Ip_Addr),
   Post =>
    (if Sock.S_Type = SOCKET_TYPE_STREAM then
      (if Error = NO_ERROR then
        Sock.S_Remote_Ip_Addr = Remote_Ip_Addr)
    else
      Sock.S_Remote_Ip_Addr = Remote_Ip_Addr)

procedure Socket_Send
   (Sock    : in out Not_Null_Socket;
    Data    : in     Send_Buffer;
    Written :    out Natural;
    Flags   :        Socket_Flags;
    Error   :    out Error_T)
   with
   Pre => Is_Initialized_Ip (Sock.S_Remote_Ip_Addr)
```

Fig. 4. An example of how function calls can be ordered by preconditions and postconditions

A. Enforcing the correct order of TCP user functions

The TCP protocol implies a specific order such that the user can call the TCP user functions without breaking the functional specification of the protocol. This order is also conveyed by the protocol's state machine given in Figure 2. Ada's preconditions and postconditions are a powerful tool to express such inter-function dependencies, while SPARK technology can be used to guarantee that the assertions always hold. Thus, preconditions and postconditions were introduced to model a partial order on the calls to the TCP user functions. The preconditions in Ada can be enabled at runtime to protect against a violation when called from code that was not proved, for example from C code.

Simplified contracts extracted from our SPARK implementation are reproduced in Figure 4, to demonstrate how SPARK can be used to enforce the correct function calls' ordering. If no error occurs, the function `Socket_Connect` sets `Sock.S_Remote_Ip_Addr` to `Remote_Ip_Addr` which is supposed to be initialized when the function is called. `Socket_Send` requires being called with a not null socket such that its field `Remote_Ip_Addr` is initialized. The only way to ensure that the precondition holds is to call `Socket_Connect` before `Socket_Send`. By analyzing the user's code, GNATprove will statically detect if a call to `Socket_Send` is not preceded by a call to `Socket_Connect`.

B. Checking the correctness of return codes

The following code demonstrates how SPARK will enforce that return codes of TCP user functions are checked, and allow execution to proceed only if no error code is returned:

```
Socket_Connect(Sock, Ip_Addr, Port, Error);
Socket_Send(Sock, Data, Written, Flags, Error);
```

In the above scenario, the postcondition of `Socket_Connect` specifies that `Sock.S_Remote_Ip_Addr` is initialized only if the function successfully executes, and thus, `Error` is equal to `NO_ERROR`. In the case where an error occurs, nothing is specified about the value of `Sock.S_Remote_Ip_Addr`, and thus, the function `Socket_Send` cannot be called because its precondition won't be satisfied.

VI. Bugs Captured

As a result of this work, two bugs were detected and reported to the fourth author of this article, also main developer of CycloneTCP, and were fixed in the C implementation.

1) Memory leak: A memory leak was detected thanks to the pointer ownership policy of SPARK detailed in Section III-C. When closing a socket, the associated memory allocated for the buffer that stores incoming segments was only partially freed, due to the use of the wrong library function to free the memory.

2) Violation of the TCP protocol: A subtle bug violating TCP specification is reproduced below:

```
1 case Sock.State is
2   when TCP_STATE_SYN_RECEIVED
3      | TCP_STATE_ESTABLISHED =>
4   -- Flush the send buffer
5   TCP_Send (Sock, Buf, Ignore_Written,
6             SOCKET_FLAG_NO_DELAY, Error);
7   if Error /= NO_ERROR then
8      return;
9   end if;
10
11  -- Make sure all the data has been sent out
12  TCP_Wait_For_Events
13     (Sock       => Sock,
14      Event_Mask => SOCKET_EVENT_TX_DONE,
15      Timeout    => Sock.S_Timeout,
16      Event      => Event);
17
18  -- Timeout error?
19  if Event /= SOCKET_EVENT_TX_DONE then
20     Error := ERROR_TIMEOUT;
21     return;
22  end if;
23
24  -- Send a FIN segment
25  TCP_Send_Segment
26     (Sock         => Sock,
27      Flags        => TCP_FLAG_FIN or TCP_FLAG_ACK,
28      Seq_Num      => Sock.sndNxt,
29      Ack_Num      => Sock.rcvNxt,
30      Length       => 0,
31      Add_To_Queue => True,
32      Error        => Error);
33
34  -- Failed to send FIN segment?
35  if Error /= NO_ERROR then
36     return;
37  end if;
38
39  -- Switch to the FIN-WAIT-1 state
40  TCP_Change_State (Sock, TCP_STATE_FIN_WAIT_1);
```

In the original implementation, after the call to `Tcp_Send`, if no error occurs, the state of the socket can either be ESTABLISHED or CLOSE-WAIT. After the call at line 12,

TABLE I
COMPARISON OF THE NUMBER OF ASM INSTRUCTIONS BETWEEN THE C AND THE SPARK IMPLEMENTATION

Function	Number of ASM instructions		Δ num. of instr.
	C	SPARK	
TCP_Listen	18	11	−39%
TCP_Accept	153	167	+9%
TCP_Connect	103	114	+10%
TCP_Send	94	124	+31%
TCP_Receive	113	153	+35%
TCP_Shutdown	107	122	+14%
TCP_Abort	42	50	+19%

the state of the socket can still be ESTABLISHED or CLOSE-WAIT, or CLOSED if a RST flag has been received when the mutex was released. At line 25, a FIN segment is sent, which does not modify the state of the socket. Finally, at line 40 the code tries to move the connection to the FIN-WAIT-1 state, and thus, the transitions CLOSE-WAIT → FIN-WAIT-1 and CLOSED → FIN-WAIT-1 are possible. These transitions are not allowed by RFC 793, and thus, not allowed by the precondition of TCP_Change_State, which models all the transitions allowed by TCP specification. Thus GNATprove reported an unproved precondition check here.

VII. RESULTS

In total, for this work, 50 functions have been translated from C to SPARK. This accounts for 2266 lines of code, of which 1165 are SPARK logic code (e.g. contracts) used to represent and prove the TCP specification. All the socket functions and the user functions of TCP have been translated and proved in SPARK, in particular TCP_Init, TCP_Connect, TCP_Listen, TCP_Send, TCP_Receive, TCP_Shutdown, TCP_Accept and TCP_Abort. The corresponding socket functions from the Socket interface have also been fully translated to SPARK. At the end, 25% of the functions that implement the TCP protocol have been rewritten in SPARK.

The overall proving time with GNATprove from the Community 2021 release is 10 minutes on a 12-processor Intel(R) Core(TM) i7-9750H CPU with 2.60GHz speed and 16GB of RAM.

Comparing the performance of C and SPARK functions can be very volatile and might not reflect the real time taken by the functions due to network latency. Instead, we have reproduced in Table I the number of ARM assembly instructions of the TCP functions written in C and in SPARK. Both C and SPARK code have been compiled with GCC by enabling the option -Os that performs many optimizations similar to -O2 that do not increase the size of the generated code. We observe that the C functions contain fewer instructions than their SPARK counterparts, hence it is expected that their runtime performance will be better. However, we don't expect this to be of practical significance, given the modest amount of time that an application will spend running these functions.

VIII. CONCLUSION AND FUTURE WORK

We selectively applied formal verification to a TCP/IP library layer by translating TCP user functions into SPARK and modeling asynchronous events with ghost code. This enabled the detection of two bugs in the original C implementation related to memory management and concurrency. The work is a step towards a more secure implementation. The translated code has been proven to be free of run-time errors and to respect the transitions of the TCP state machine, specified in RFC 793 [13].

These results depend upon other layers written in the C programming language, which were not formally verified to the same level. Instead, we used symbolic execution with KLEE to formally verify the contracts used in SPARK to model the corresponding behaviors. The principal function of those layers is to format packets before they are sent, or parse incoming packets, check their integrity, and transmit the resulting payload to the corresponding upper-level layer. This processing part can be a source of errors and bugs as explained in [21]. As an alternative to manual translation into SPARK, we envision using the RecordFlux DSL [21] to specify the parser and printer of the different protocols' packets and generate provable SPARK code from this specification. This would complete our current verification of the CycloneTCP stack towards the lower layers.

ACKNOWLEDGMENTS

This research is part of the "High-Integrity Complex Large Software and Electronic Systems" (HICLASS) project that is supported by the Aerospace Technology Institute (ATI) Programme, a joint Government and industry investment to maintain and grow the UK's competitive position in civil aerospace design and manufacture, under grant agreement No. 113213. We also thank the reviewers for their valuable feedback.

REFERENCES

[1] P. Sparks. (2017, 6) The route to a trillion devices – the outlook for IoT investment to 2035. ARM – White Paper. [Online]. Available: https://community.arm.com/iot/b/internet-of-things/posts/white-paper-the-route-to-a-trillion-devices

[2] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani, "Demystifying IoT security: An exhaustive survey on IoT vulnerabilities and a first empirical look on internet-scale IoT exploitations," *IEEE Communications Surveys Tutorials*, vol. 21, no. 3, pp. 2702–2733, 2019.

[3] R. Alguliyev, Y. Imamverdiyev, and L. Sukhostat, "Cyber-physical systems and their security issues," *Computers in Industry*, vol. 100, pp. 212 – 223, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166361517304244

[4] Ben Seri, Gregory Vishnepolsky, Dor Zusman. (2019) Urgent/11 – Critical vulnerabilities to remotely compromise VxWorks, the most popular RTOS. ARMIS – White Paper. [Online]. Available: https://info.armis.com/rs/645-PDC-047/images/Urgent11%20Technical%20White%20Paper.pdf

[5] (1991, January) A TCP/IP tutorial. [Online]. Available: https://tools.ietf.org/html/rfc1180

[6] 35.100 – Open Systems Interconnection (OSI). [Online]. Available: https://www.iso.org/ics/35.100/x/

[7] M. A. Smith, "Formal verification of communication protocols," in *Formal Description Techniques IX*. Springer, 1996, pp. 129–144.

[8] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets," in *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 265–276. [Online]. Available: https://doi.org/10.1145/1080091.1080123

[9] T. Ridge, M. Norrish, and P. Sewell, "A rigorous approach to networking: TCP, from implementation to protocol to service," in *International Symposium on Formal Methods*. Springer, 2008, pp. 294–309.

[10] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub, "Implementing tls with verified cryptographic security," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 445–459.

[11] J. W. McCormick and P. C. Chapin, *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.

[12] CycloneTCP embedded IPv4/IPv6 stack. [Online]. Available: https://www.oryx-embedded.com/products/CycloneTCP.html

[13] "Transmission Control Protocol," RFC 793, Sep. 1981.

[14] "AMNESIA:33 – how TCP/IP stacks breed critical vulnerabilities in IoT, OT and IT devices," Research Report, Forescout, 2020.

[15] C. Dross and J. Kanig, "Recursive data structures in SPARK," in *International Conference on Computer Aided Verification*. Springer, 2020, pp. 178–189.

[16] AdaCore. Formal verification with GNATprove. [Online]. Available: https://docs.adacore.com/spark2014-docs/html/ug/en/gnatprove.html

[17] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, May 2018. [Online]. Available: https://doi.org/10.1145/3182657

[18] D. Vanoverberghe, N. Bjørner, J. de Halleux, W. Schulte, and N. Tillmann, "Using dynamic symbolic execution to improve deductive verification," in *International SPIN Workshop on Model Checking of Software*. Springer, 2008, pp. 9–25.

[19] I. T. Kassios, P. Müller, and M. Schwerhoff, "Comparing verification condition generation with symbolic execution: an experience report," in *International Conference on Verified Software: Tools, Theories, Experiments*. Springer, 2012, pp. 196–208.

[20] KLEE Symbolic Execution, version 2.1. [Online]. Available: https://klee.github.io

[21] T. Reiher, A. Senier, J. Castrillón, and T. Strufe, "RecordFlux: Formal message specification and generation of verifiable binary parsers," in *International Conference on Formal Aspects of Component Software*. Springer International Publishing, 2020, p. 170–190.