arXiv:2109.10347v1 [cs.CR] 2 Sep 2021

AdaCore

Security-Hardening Software Libraries with Ada and SPARK

A TCP Stack Use Case

White Paper

Kyriakos Georgiou, Guillaume Cluzel, Paul Butcher, Yannick Moy

September 23, 2021

https://t.me/learningnets

Abstract

The work is part of a series of white papers to demonstrate how the SPARK technology, a subset of the Ada programming language supported by formal verification tools, can be applied for the security-hardening of Software libraries. The first white paper of this series, [8], introduced the SPARK technology through the conversion of a *C* benchmark suite to SPARK. The work demonstrated how the use of the SPARK technology could guarantee the absence of runtime errors without significantly impacting performance when compared to the *C* version of the code. Furthermore, best practices and guidelines on achieving the absence of runtime errors with SPARK were reported. This white paper builds on the previous work and demonstrates how an existing professional-grade, open-source embedded TCP/IP stack implementation written in the C programming language can be hardened using the SPARK technology to increase its assurance, reliability, and security. More specifically, the work demonstrates the ability of SPARK to enforce the correct usage of the library and to verify the conformance to its functional specifications. A multifaceted approach achieves this. Firstly, the TCP layer's C code is being replaced with formally verified SPARK. Then the lower layers, still written in C and on which the TCP layer depends, are modeled using SPARK contracts and validated using symbolic execution with KLEE. Finally, formal contracts for the upper layers are defined to call the TCP layer. The work allowed the detection and correction of two bugs in the TCP layer. The powerful approach detailed in this work can be applied to any existing critical C library.

https://t.me/learningnets

Contents

https://t.me/learningnets

1 Introduction

TCP is the most widely used network protocol to communicate on the Internet. Thus, ensuring the TCP/IP stack's safety is an essential step towards safer cyber-physical systems. Existing research deals with formally verifying protocols of other TCP/IP stack levels. For example, the work in *miTLS* [3] formally verifies an SSL/TLS protocol implementation, and the work in [7] uses a technology called RecordFlux to safely parse data segments. At this point of writing, we are not aware of any formally verified TCP protocol implementation. Such an implementation would provide strong assurances on the security and the safety of a TCP implementation. Furthermore, the security of higher-level protocols in the TCP/IP stack, such as the TLS protocol, can only be ensured if the underlying TCP implementation is bug-free and conforms with its functional specifications.

SPARK 2014 is a programming language designed as a subset of Ada to produce highly reliable software through formal verification. SPARK can detect uninitialized variables with control flow analysis, ensure the absence of run-time errors, and, based on SMT-solver technology, provides mechanisms to specify and mathematically prove the functional behavior of a program.

This work aims to enhance the safety and security of an embedded industrial-grade implementation of the TCP protocol by using the SPARK technology.

To fully grasp the technical parts of this work, we urge the reader to first have a read at the first white paper on hardening software libraries with SPARK [8]. The SPARK syntax, usage, assurance levels, and bottom-up approach of introducing it into a C-based project are described in great detail.

2 The TCP protocol

This section is not meant to define the full functional specification of the TCP protocol. Instead, the intention is to give enough context to enable the reader to follow the work conducted in this deliverable. For a full definition of the protocol, please refer to the *RFC 793* document [1], the defining standard for TCP.

https://t.me/learningnets

2.1 Networking Communication Models

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) [2] and the OSI (Open System Interconnection) [9] models, with the first one being the most widely used. The TCP/IP model was developed by the US Department of Defence (DoD) [16] and the OSI model was introduced by ISO (International Standard Organization) [12]. Their purpose is to provide hardware vendors with a common base for the development of networking products that can communicate with each other.

Figure 1 shows an overview of the two models, TCP/IP and OSI. TCP/IP has four layers, while the OSI is a 7-layer model. Layering is an essential design requirement for both models to achieve modularity, flexibility, and abstraction. Applications that only need to use the functionality of the lower levels can drop all the unnecessary upper levels. Furthermore, layering enables easier troubleshooting as the network's functionality is well-defined and split between the different layers. 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. Software developers producing networking applications have more responsibility for the implementation and usage of the TCP/IP *Transport* and *Application* layers, or their corresponding OSI model's layers. While the *Application* level has a wide range of protocols that can be used depending on the application specification, the *Transport* layer is heavily depended on the two dominant protocols:

1. Transmission Control Protocol (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 and that they are delivered in the order they were sent. It also provides error-checking mechanisms to discard and recover any corrupted data.

2. User Datagram Protocol (UDP) is a connectionless protocol; no connection establishment is required between the sender and the receiver. UDP is more performance-oriented rather than reliability oriented; thus, it does not provide any error-checking mechanism, and it does not guarantee the delivery of datagrams sent.

Today, almost any networking application is using either the TCP or the UDP protocols to traffic data between the Network and the Application lay-

https://t.me/learningnets

OSI Model Layers	TCP/IP Model Layers	Example Protocols for Each Layer		Protocol Data Format
Application	Application	HTTP, FTP, POP3, Telnet, SSH		Data
Presentation		SSL, AFP, TLS		
Session		Sockets, NetBIOS, ZIP		
Transport	Transport	TCP	UDP	Segments
Network	Internet	IPv4/IPv6, ICPM, IGMP, ARP		Packets
Data Link	Network Interface	Ethernet, Wi-Fi		Frames
Physical		DSL, ISDN, USB		Bits

Figure 1: Overview of the TCP/IP and the OSI networking models for communication

ers. Thus, the reliability, assurance and security of these protocols are of paramount importance for critical applications. A security vulnerability in the implementation of these protocols can lead to a severe compromise of the security of any application using them [5]. Since for critical cyber-physical systems reliability is an important factor, the focus of this work will be on the TCP protocol.

2.2 *CycloneTCP* TCP/IP library

CycloneTCP is a professional-grade embedded TCP/IP library developed by the *Oryx Embedded* company. The implementation is meant to conform with the Request for Comments (RFC) Internet standards, namely the *RFC 793* TCP protocol specifications [1], provided by the Internet Engineering Task Force (IETF) [13]. The library is written in ANSI C, and it supports a large number of 32-bit embedded processors and a large number of Real-time Operating Systems (RTOS). It can also run on bare metal environments. The library offers implementations for a wide range of TCP/IP protocols. These includes the IPv4 and IPv6 of the network layer, the TCP and UDP of the transport layer, and the DHCP and HTTP of the application layer. A quick overview of the library can be found here [11] while an open-source version can be downloaded at [10]. The development of this work is based on the *CycloneTCP*'s Transport layer implementation of the TCP and UDP protocols, with a focus on the TCP protocol. While the TCP protocol is meant to conform with the *RFC 793* TCP Specification [1], no formal guarantees are being provided on the conformance. Thus, this work takes a step further than the original protocol's implementation to provide some assurances on the specification conformance and the overall security hardening of the library.

https://t.me/learningnets

```
   type Socket_Struct is record
   S_Descriptor : Sock_Descriptor;
   S_Type       : Socket_Type;
   S_Protocol   : Socket_Protocol;
   S_Net_Interface : System.Address;
   S_localIpAddr : IpAddr;
   S_Local_Port : Port;
   S_Remote_Ip_Addr : IpAddr;
   S_Remote_Port : Port;
   S_Timeout    : Systime;
   State        : Tcp_State;
   -- Other fields
end record;
```

Figure 2: The *socket* Ada implementation.

2.3 Overview of the TCP protocol Specification

This section gives an overview of the TCP functional specification without going into unnecessary depth. A complete description of the protocol's specification can be found at the *RFC 793* document [1]. TCP is a reliable, ordered, and error-checked connection-oriented protocol; connection must be established before the transmission of any data, and the connection must be closed once all the data has been transmitted.

In the *CycloneTCP* implementation, the *socket* data structure is used to retain the status of a TCP connection. Figure 2 shows part of its equivalent implementation in the Ada/SPARK code, demonstrating some of the fields that store vital information for the status of a connection. Furthermore, a socket can be manipulated by the user through the TCP's library Application Programming Interface (API) to perform various TCP operations, such as the transmission of 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
3. Closing the connection

The behaviour of the three-phase communication can be described by the state machine given in Figure 3. An edge represents a state transition. A transition is either triggered by an explicit action, stated in *italics* format

https://t.me/learningnets

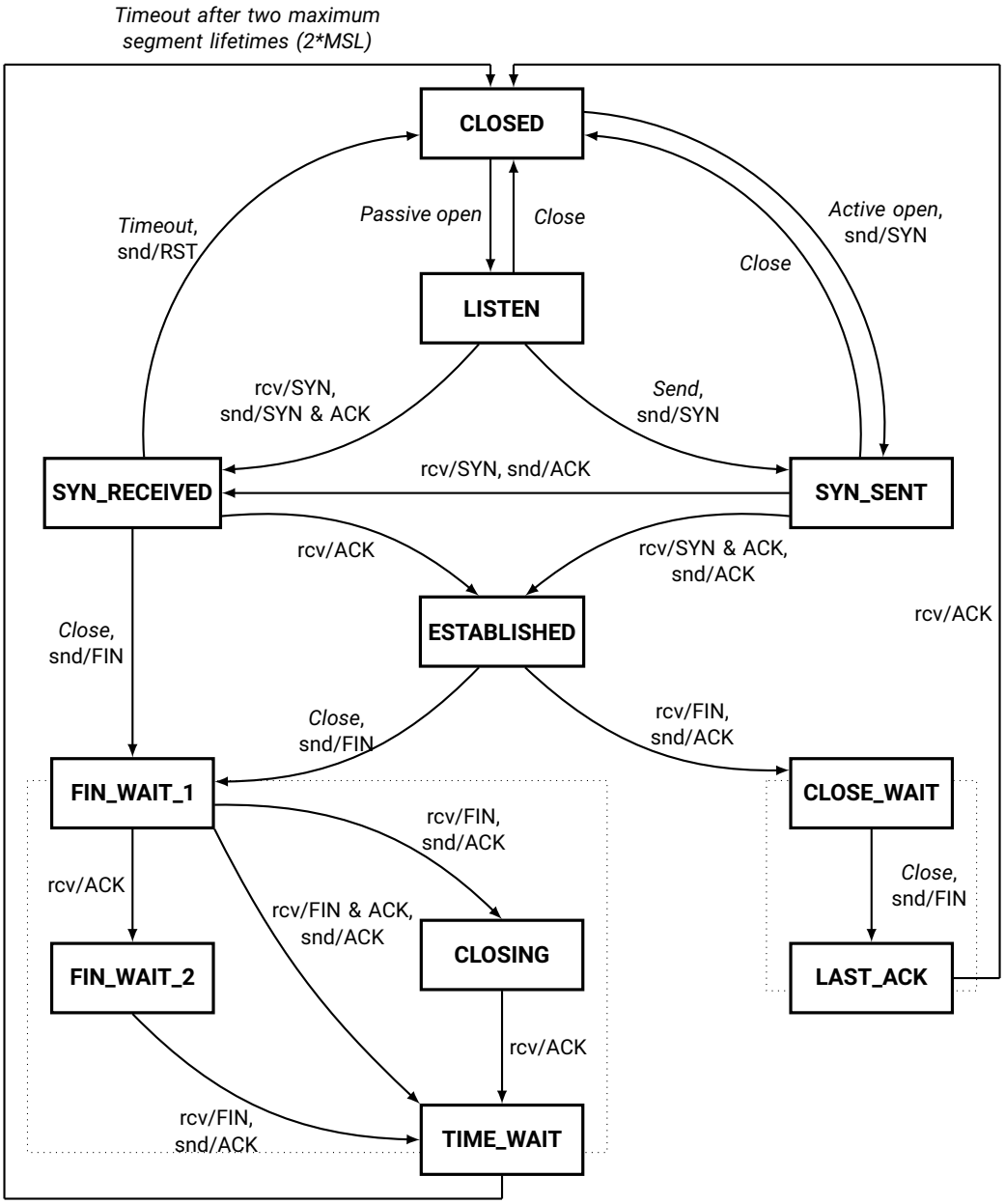

Figure 3: The TCP connection automaton demonstrates the different states a TCP session can exhibit. The figure is recreated from [14].

https://t.me/learningnets

on the label's edge, for example, *Close*, or is triggered by the arrival of a specific flag/s. An edge-label in the format of A/B represents the associated flags transmitted with each transition, where A is either send or receive and B refers to the actual flag/s transmitted in response to the action that caused the state-transition. An explicit action is either a user-triggered action (through the library's API) or an automatically performed action 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** – Acknowledgement field significant. The last message received by the sender is acknowledged.
- **SYN** – Synchronize sequence number. This flag is sent to establish a connection.
- **FIN** – No more data from sender. This flag is sent to close the connection.

The state machine in Figure 3 does not represent the complete protocol specification; for example, it does not reflect error-conditions or any actions which are not connected with the state changes. Rather, it gives an overview of all the possible states a TCP connection could reach over its lifetime. The *socket* data structure given in Figure 2 can represent these states.

Finally, although the state machine of Figure 3 is a recreation of the TCP state machine found in the TCP specification document on page 22 [1, p. 23] the two are not entirely identical. The state machine of Figure 3 is updated to include an extra transition between the states FIN-WAIT-1 and TIME-WAIT. The extra transition reflects the case where the ACK and FIN flags are received in the same segment. It is an allowed behaviour by the TCP specification, which the *CycloneTCP* library also exhibits.

The meaning of the states, as taken from the TCP specification document [1], are:

- **LISTEN** represents waiting for a connection request from any remote TCP and port.
- **SYN-SENT** represents waiting for a matching connection request after having sent a connection request.
- **SYN-RECEIVED** represents waiting for a confirming connection request acknowledgement after having both received and sent a connection request.

https://t.me/learningnets

- **ESTABLISHED** represents an open connection, data received can be delivered to the user. The normal state for the data transfer phase of the connection.
- **FIN-WAIT-1** represents waiting for a connection termination request from the remote TCP, or an acknowledgement of the connection termination request previously sent.
- **FIN-WAIT-2** represents waiting for a connection termination request from the remote TCP.
- **CLOSE-WAIT** represents waiting for a connection termination request from the local user.
- **CLOSING** represents waiting for a connection termination request acknowledgement from the remote TCP.
- **LAST-ACK** represents waiting for an acknowledgement of the connection termination request previously sent to the remote TCP (which includes an acknowledgement of its connection termination request).
- **TIME-WAIT** represents waiting for enough time to pass to be sure the remote TCP received the acknowledgement of its connection termination request.
- **CLOSED** represents no connection state at all.

Using the state machine of Figure 3, an example of how a TCP communication can be established between two machines that support the TCP protocol is presented in Figure 4. When one of the TCP machines, TCP-α, is in the CLOSED state and the other TCP machine, (TCP-β), is in the state LISTEN, the following steps are needed to establish their TCP connection:

1. TCP-α wants to initiate a connection with TCP-β, thus, it sends a SYN segment to TCP-β and moves to the SYN-SENT state.
2. TCP-β receives the SYN segment. It has to respond back to TCP-α with a segment containing the SYN and ACK flags, to acknowledge the received segment. Then, it changes its state to SYN-RECEIVED.
3. When TCP-α receives the segment sent by TCP-β containing the SYN and ACK flags, it only has to send back an ACK of successfully receiving this segment, and then move to the ESTABLISHED state.
4. TCP-β receives the ACK send by TCP-α and thus also moves to the ESTABLISHED state.

https://t.me/learningnets

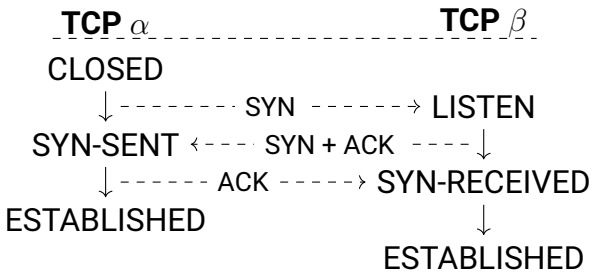

Figure 4: Three-way handshake procedure. The solid lines represent the transitions between the states and the dashed lines represent the messages sent.

At the end of this procedure, both of the TCP machines are in the ESTABLISHED state and thus they can begin transmitting data. This procedure to open a connection is known as the "three-way handshake" in the TCP norm.

For more scenarios on the TCP functionality, such as how two TCP machines can close a connection, the reader can refer to the TCP norm [1].

2.3.1 TCP Multitasking

Different tasks can interact and update the socket data structure to handle the various events that can occur within a TCP session. The TCP norm, 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*. This design has also been adopted in *CycloneTCP*. The role of each of the three tasks can be summarized as follow:

- **User calls** – User calls refer to functions, namely OPEN, CLOSE, ABORT, SEND and RECEIVED, 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.
- **Arriving segments** – In this 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. For example, when a segment containing the SYN flag is received while the socket is in the LISTEN state, a segment with the the flags SYN + ACK will be auto-triggered in response, and the current state will be changed to SYN-RECEIVED.

https://t.me/learningnets

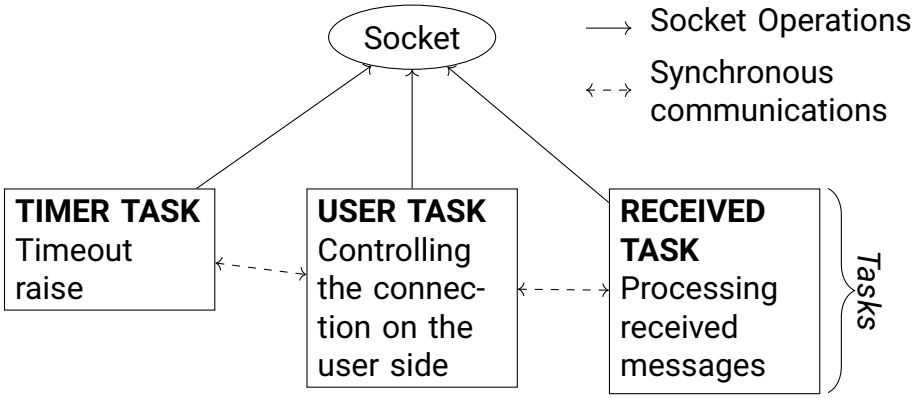

Figure 5: Concurrency in the TCP protocol, using the 3-tasks based model described in the protocol's specification [1].

- **Timers** – 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, corresponding transitions to the timeout events can also be triggered by this task.

Figure 5 illustrates the interactions between the socket's data structure and the three tasks. The implementation allows for a single socket to act as a shared resource between the multiple tasks. All tasks can manipulate the socket and change its status concurrently. Only one task can operate on a socket at a given time. The access to the socket's fields is protected by a mutex. Two tasks can communicate synchronously or asynchronously. The synchronous communications are based on the interface provided by the OS, in particular, by the use of events.

3 The TCP library hardening

A large part of the TCP behaviour defined by its functional specification in [1] is amenable to automatic proof. For example, formal proof can be used to achieve the:

- Verification of the transitions between the different allowed states.
- Validation of the integrity of sent messages, *i.e.* we could check that the sent message contains the correct flags and the correct sequence and acknowledgement numbers.

https://t.me/learningnets

- Validation of the integrity of received messages, *i.e.*, we could check that the messages received are correctly processed in regard to the flags they contain.
- Functional correctness of the user-task based on the protocols functional specification.

The verification of the full functional specification of the *CycloneTCP* library is beyond the scope of this work, as it will require the extensive involvement of the original developers of the library. Instead, our aim is the hardening of the TCP library in the areas that its original authors designated as the most vulnerable or crucial to conform to their functional specifications. Thus, this work mainly focuses on hardening the API of the user task. This mainly falls under two categories:

- **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, the library's users tend to call the API functions in the wrong order, and to forget to check the return code of a function call. Thus, they are subsequently allowing their TCP implementation to behave outside of the functional specification of the protocol.
- **Conformance to the protocol's functional specification** – The safety of the library largely depends on the assumption that the user's functions are indeed implementing the protocol's specified functionality. Thus, first, we want to verify that the transitions allowed by the current implementation of the user-task related functions respect the state machine of Figure 3, which defines the permitted transitions between the different TCP states. Second, we want to ensure that the user's related functions are always updating a socket's state within the functional specifications of the protocol.

So far in the first white paper on the hardening of software libraries with Ada and SPARK, [8], we demonstrated how the SPARK technology could be used to achieve AoRTE, the Silver level of SPARK assurance. This was done in a bottom-up approach, starting from the lowest level of SPARK assurance and moving to the upper ones, only when the previous level in the hierarchy is completely achieved. As a reminder to the reader, the SPARK level of assurance are:

1. **Stone level** – valid SPARK.

https://t.me/learningnets

2. **Bronze level** – initialization and correct data flow.
3. **Silver level** – Absence of Run-Time Errors (AoRTE).
4. **Gold level** – proof of key integrity properties.
5. **Platinum level** – full functional proof of requirements.

Although a large part of this work was devoted to achieving AoRTE for the code of interest, this process will not be covered in this deliverable. D3.5.7AD offers a wealth of information, examples, and guidelines on how to achieve the Silver level of SPARK assurance, and thus, this will not be repeated in this document. The focus of this deliverable is to step further from proving AoRTE and demonstrate how the SPARK technology can be used to prove key integrity properties and conformance to functional specifications.

The following sections demonstrate the key areas that SPARK hardening was applied and the techniques used to achieve the relevant goals within the two categories of improvements described earlier.

3.1 Hardening the user's API

3.1.1 Enforcing the correct order of API functions

As described earlier in Section 2.3.1, the user-task implementation offers an API where a user can perform various operations on the network. To achieve this, the API offers several high-level user functions that can alter the state of a socket, which represents the state of the network. The Ada/SPARK functions, translated from their *C CycloneTCP* equivalent functions, are located in the files `socket_interface.ad(b|s)`. AoRTE was first achieved on these functions, before proceeding with the enforcement of the correct order of calling them.

The TCP protocol implies a specific order such that the user can call the API functions without breaking the functional specification of the protocol. This order also is being conveyed by the protocol's state machine given in Figure 3. Ada's pre- and post-conditions are a powerful tool to express such inter-function dependencies, while SPARK technology can be used to guarantee that the assertions always hold. Thus, post- and pre-conditions were introduced to model a partial order on the calls to the API's functions.

If two functions f_1 and f_2 are ordered such that $f_1 \preceq f_2$, where $\preceq$ is a relation over the order in which functions have to be called that specifies that f_1 must be called before f_2, then the post-condition of f_1 should be respected by the precondition of f_2.

https://t.me/learningnets

An example of how to enforce such a call-dependency using pre- and post-conditions is given in Figure 6 with the procedures `Socket_Connect` and `Socket_Send`. The procedure `Socket_Connect` tries to connect to a distant TCP. If the connection succeeds, the remote IP address of the distant TCP is set in the field `S_Remote_Ip_Addr` of the `Sock` structure. When a user calls the procedure `Socket_Send`, we want to ensure that the connection has already been established. Thus, the precondition `Is_Initialized_Ip(Sock.S_Remote_Ip_Addr)` of `Socket_Send` at line 30 of Figure 6, can only be proved if `Socket_Connect` has been called prior to the call of `Socket_Send`.

Such proof is enabled by the *Contract_Cases* aspect at line 8 of the procedure `Socket_Connect` in Figure 6. The *Contract_Cases* aspect will generate a post-condition for the procedure based on the aspect's case that is true upon the entry of the procedure. At each call of the subprogram, one and only one case of the aspect is permitted to evaluate to *true*. The consequence of that case is a contract that is required to be satisfied upon the subprogram's return. Thus, this generated contract acts as a post-condition for the subprogram. In our example, the first case at line 9 checks if the socket is of TCP type, namely `SOCKET_TYPE_STREAM`, at the entry of the procedure. If this is true the consequence of the valid case, namely the *if* statement at line 10–20 will be generated as a post-condition, which is required to be satisfied when the subprogram returns. Thus, this post-condition will provide the automatic provers with the information that the `S_Remote_Ip_Addr` is correctly set when there is no `NO_ERROR`. This information will be used later on to prove the precondition of `Socket_Send`, for all calls in the user's code.

Table 1 demonstrates all the socket-related user procedures, located in the `socket_interface.ad(b|s)` file, that we had to enforce the right call-dependencies using SPARK. The call-dependencies can be determined by the second column which represents the socket's data structure fields that are needed to be set prior to the call of each subroutine, and the third column, which shows which procedures can set these socket's fields. Beyond the dependencies on the socket's fields there are also dependencies on the creation of the socket structure by the *Socket_Open* procedure. Based on these conditions, the following call-dependencies were enforced using SPARK, similarly to the Figure 6 example:

https://t.me/learningnets

```
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),
   Contract_Cases => (
    Sock.S_Type = SOCKET_TYPE_STREAM =>
      (if Error = NO_ERROR then
        Sock.S_Type = Sock.S_Type'Old and then
        Sock.S_Protocol = Sock.S_Protocol'Old and then
        Is_Initialized_Ip (Sock.S_localIpAddr) and then
        Sock.S_Local_Port = Sock.S_Local_Port'Old and then
        Sock.S_Remote_Ip_Addr = Remote_Ip_Addr and then
        Sock.S_Remote_Port = Remote_Port and then
        Sock.State = TCP_STATE_ESTABLISHED
      else
        Sock.S_Type = Sock.S_Type'Old and then
        Sock.S_Protocol = Sock.S_Protocol'Old)
    others => True)

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)
```

Figure 6: An example of how function calls can be ordered by Pre- and Post-conditions.

https://t.me/learningnets

Procedures	Pre-requirements	Socket fields set by the procedure or action
Socket_Open	None	Allocation of new socket. No field set.
Socket_Connect	Valid socket. Remote_Ip_Addr not set.	Local_Ip_Addr, Local_Port, Remote_Ip_Addr, Remote_Port.
Socket_Send	Valid socket. Remote_Ip_Addr set	None.
Socket_Receive		
Socket_Shutdown		
Socket_Close	Valid socket.	Free the socket.

Table 1: The socket-related user procedures, located in the `socket_interface.ad(b|s)` file, that their call-dependencies must be met. The call-dependencies can be determined by the second column which represents the socket's data-structure fields that are needed to be set prior to the call of each subroutine, and the third column, which shows which procedures can set these socket's fields. Beyond the dependencies on the socket's fields there are also dependencies on the creation of the socket structure by the *Socket_Open* procedure.

1. $Socket\_Open \preceq Socket\_Connect$
2. $Socket\_Open \preceq Socket\_Send$
3. $Socket\_Open \preceq Socket\_Receive$
4. $Socket\_Open \preceq Socket\_Shutdown$
5. $Socket\_Open \preceq Socket\_Close$
6. $Socket\_Connect \preceq Socket\_Send$
7. $Socket\_Connect \preceq Socket\_Receive$
8. $Socket\_Connect \preceq Socket\_Shutdown$

The transitivity property ensures that when the first dependency is enforced with SPARK then the dependencies 2,3,4 are also enforced by default, due to the dependencies 6,7 and 8.

3.1.2 Checking the correctness of return codes

An observation made by the development team of the *CycloneTCP* library is that their customers often neglect to check the return code of the socket-related user functions. Thus, they do not deal with the possibility of function-calls failing to achieve their goal and returning an error code. If the return code is not checked, no assumption can be made on the validity of the execution that follows a function call which has the potential of producing an error.

https://t.me/learningnets

Post-conditions can be used to enforce checks on the return code of a procedure. Such checks will be required after the call of this procedure by GNATprove to enable proving of the program. To understand the mechanisms of this approach, one must understand GNATprove's goal. GNATprove aims to compute all possible values of variables to guarantee the AoRTE. Thus, the user has to either provide extra information to GNATprove or write the code in such a way that eliminates any ambiguity about a variable's state. GNATprove will generate a warning if it can not deduct the possible states of a variable at any place in the code. In Section 3.1.1, we explained how the *Contract_Cases* aspect used in Figure 6 at line 8 will generate a post-condition for the `Socket_Connect` procedure. This generated post-condition will convey the following information after each call of the `Socket_Connect`:

1. if the `ERROR` is equal to `NO_ERROR` the TCP session will be in the `TCP_STATE_ESTABLISHED` state, and all the relevant socket's data structure fields will be set, including `S_Remote_Ip_Addr`, otherwise
2. the field `S_Remote_Ip_Addr` can have any value.

Thus, if the user calls the two procedures of Figure 6 in the following manner:

```
1 Socket_Connect (Sock, Remote_Ip_Addr, Port, Error);
2 Socket_Send (Sock, Data, Written, Flags, Error);
```

GNATprove will warn that:

```
medium: precondition might fail.
```

This is because, depending on the `ERROR` value, the introduced post-condition will provide the information to GNATprove that `Sock.S_Remote_Ip_Addr` is either set or not. Thus, when a call to the `Socket_Send` procedure follows, its pre-condition, given at line 30 of Figure 6, will fail because of the ambiguity on the value of `Sock.S_Remote_Ip_Addr`. The user is now enforced to check the return code of the `Socket_Connect` after the function returns to resolve the ambiguity on the TCP's session state:

```
1 Socket_Connect (Sock, Remote_Ip_Addr, Port, Error);
2 if Error /= NO_ERROR then
3   -- TCP session state -> no connection established
4   Socket_Close(Sock);
5 else
6   -- TCP session state -> connection established
```


https://t.me/learningnets

```
7   Socket_Send (Sock, Data, Written, Flags, Error);
8   -- continue processing
9 end if;
```

Now the prover knows that `Socket_Send` will only be called when `Socket_Connect` has returned without an error code, and thus, the `Socket_Send`'s pre-condition can be proved. In the case of `Socket_Connect` returning an error code, we decided to close the socket and bring the TCP session to the `CLOSED` state. Users can handle this case more precisely depending on the exact error-code returned, for example, `Error = ERROR_PORT_UNREACHABLE`, and provide their implementation of how to proceed in each error case; always respecting the TCP specification.

3.2 Conformance to the protocol's functional specification

The original *CycloneTCP* library provides no 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 3. Although, this work is focused on hardening the API of the user task, mainly the high-level user functions located in the file `tcp_interface.ad(b|s)`, to verify the state transitions the rest of the TCP protocol has to be taken into account. The reason for this is two-fold. First, although user-functions can trigger state transitions, the actual transitions are done by library functions that do not belong to the user-task set of functions. Second, other parts of the library can trigger state transitions during and between the user-task function calls, which can affect the intermediate and final states that a user related function can exhibit.

The TCP user API related functions were fully translated to SPARK. SPARK bindings were also provided to the majority of the rest of the library's C functions to allow their invocation from the SPARK code. Furthermore, SPARK helper functions were introduced to assist with the state-transition validation. Table 2 gives a list of all the files we needed to introduce or alter, to complete this work. The table also provides a short description of their purpose.

3.2.1 Technical challenges encountered and overview of solutions

The validation of functional specifications with SPARK can be a quite challenging task depending on the complexity, the size, and the implementation

https://t.me/learningnets

File	Description	Translation or binding
ada_main.adb/s	Customer SPARK code that tests the socket API.	SPARK code.
socket_types.ads	Types and structure of a socket.	SPARK code
socket_interface.adb/s	Socket API.	SPARK code.
socket_helper.ads	Helper function for proofs.	Helper functions for proofs.
tcp_type.ads	Types used for TCP.	SPARK code.
tcp_interface.adb/s	TCP user functions.	SPARK code.
tcp_misc_binding.adb/s	Helper functions for TCP.	SPARK code / binding to C code.
tcp_fsm_binding.ads	TCP finite state machine. Functions to process incoming segments.	Binding to C code.
tcp_timer_interface.ads	Simulate a timer tick.	SPARK code. Helper functions for proofs.
udp_binding.adb/s	UDP functions.	Binding to C code.
net_mem_interface.adb/s	Memory management.	Binding to C code.
ip_binding.adb/s	Underlaying IP layer functions.	Binding to C code.

Table 2

of the project. In the case of the *CycloneTCP* TCP implementation, the following two issues required innovative solutions:

3.2.1.1 Modelling concurrent behaviour with SPARK

The concurrent implementation of the TCP protocol allows for multiple interactions between the tasks, synchronous or asynchronous, and thus, numerous possible state changes 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 introduced SPARK functions with contracts that model how the different possible interactions can modify the state of a socket. The solution will be elaborated in Section 3.2.2.

3.2.1.2 Verifying non-user functions

The TCP/IP specification document provides all the information needed for the allowed transitions between the different TCP possible states. For any transitions that are directly triggered through the call of the user-tasks functions, SPARK contracts, that represent transitions allowed by the functional specification, are embedded within the SPARK translated code. Then *GNATprove* can be used to prove that the code will always respect those transitions. The SPARK contracts were placed into the `Tcp_Change_State` procedure, found in the SPARK translated file `tcp_misc_binding.ads`. The `Tcp_Change_State` is a helper function that is called every time a state transition needs to be made, to update the socket's state. An incorrect transition allowed by the code will be detected by the prover.

The above approach can not be used in the case of transitions that are not related to the user-task. This is because the code of these functions is not translated to SPARK, but rather only SPARK bindings to the C code are

https://t.me/learningnets

introduced. Thus the prover can not be used to evaluate the validity of the state transitions allowed by the C code.

The first step of verifying the non-user related state transitions is to identify their source. As shown in Figure 3, the majority of these state transitions are triggered through segment transmissions. The `tcp_fsm.c` file includes all the functions that are responsible for processing received segments. The main function of this file `tcpProcessSegment` looks for the socket corresponding to the received segment, and then according to the current TCP state of this socket, it calls one of the `tcpState<StateName>` functions to process the segment. This is shown the the following code taken from the `tcpProcessSegment` function:

```
1  //Check current state
2  switch(socket->state)
3  {
4  //Process CLOSED state
5  case TCP_STATE_CLOSED:
6    //This is the default state that each connection starts in
       before
7    //the process of establishing it begins
8    tcpStateClosed(interface, pseudoHeader, segment, length);
9    break;
10 //Process LISTEN state
11 case TCP_STATE_LISTEN:
12   //A device (normally a server) is waiting to receive a
       synchronize
13   //(SYN) message from a client. It has not yet sent its own
       SYN message
14   tcpStateListen(socket, interface, pseudoHeader, segment,
       length);
15   break;
16 //Process SYN_SENT state
17 case TCP_STATE_SYN_SENT:
18   //The device (normally a client) has sent a synchronize
       (SYN) message
19   //and is waiting for a matching SYN from the other device
       (usually
20   //a server)
21   tcpStateSynSent(socket, segment, length);
22   break;
23 //Process SYN_RECEIVED state
24 case TCP_STATE_SYN_RECEIVED:
25   //The device has both received a SYN from its partner and
       sent its own
```


https://t.me/learningnets

```
26   //SYN. It is now waiting for an ACK to its SYN to finish
        connection
27   //setup
28   tcpStateSynReceived(socket, segment, buffer, offset,
        length);
29   break;
30 //Process ESTABLISHED state
31 case TCP_STATE_ESTABLISHED:
32   //Data can be exchanged freely once both devices in the
        connection
33   //enter this state. This will continue until the connection
        is closed
34   tcpStateEstablished(socket, segment, buffer, offset,
        length);
35   break;
36 //Process CLOSE_WAIT state
37 case TCP_STATE_CLOSE_WAIT:
38   //The device has received a close request (FIN) from the
        other device.
39   //It must now wait for the application to acknowledge this
        request and
40   //generate a matching request
41   tcpStateCloseWait(socket, segment, length);
42   break;
43 //Process LAST_ACK state
44 case TCP_STATE_LAST_ACK:
45   //A device that has already received a close request and
        acknowledged
46   //it, has sent its own FIN and is waiting for an ACK to
        this request
47   tcpStateLastAck(socket, segment, length);
48   break;
49 //Process FIN_WAIT_1 state
50 case TCP_STATE_FIN_WAIT_1:
51   //A device in this state is waiting for an ACK for a FIN it
        has sent,
52   //or is waiting for a connection termination request from
        the
53   //other device
54   tcpStateFinWait1(socket, segment, buffer, offset, length);
55   break;
56 //Process FIN_WAIT_2 state
57 case TCP_STATE_FIN_WAIT_2:
58   //A device in this state has received an ACK for its
```


https://t.me/learningnets

```
      request to
59   //terminate the connection and is now waiting for a
      matching FIN
60   //from the other device
61   tcpStateFinWait2(socket, segment, buffer, offset, length);
62   break;
63 //Process CLOSING state
64 case TCP_STATE_CLOSING:
65   //The device has received a FIN from the other device and
      sent an ACK
66   //for it, but not yet received an ACK for its own FIN
      message
67   tcpStateClosing(socket, segment, length);
68   break;
69 //Process TIME_WAIT state
70 case TCP_STATE_TIME_WAIT:
71   //The device has now received a FIN from the other device
      and
72   //acknowledged it, and sent its own FIN and received an ACK
      for
73   //it. We are done, except for waiting to ensure the ACK is
74   //received and prevent potential overlap with new
      connections
75   tcpStateTimeWait(socket, segment, length);
76   break;
77 //Invalid state...
78 default:
79   //Back to the CLOSED state
80   tcpChangeState(socket, TCP_STATE_CLOSED);
81   //Silently discard incoming packet
82   break;
83 }
```

The family of `tcpState<StateName>` functions, also located in the `tcp_fsm.c` file, check the information contained in the segment and can perform a change of state in the socket structure depending on the segment's flags. Any of these functions that have to perform a state transition has a call to the `Tcp_Change_State` procedure which will perform the transition.

Trying to extract all the possible state transitions allowed by the *CycloneTCP* TCP implementation by manual code inspection is an expensive and error-prone process. One way to approach this is to locate all the calls to the `Tcp_Change_State` procedure and try to understand the state

https://t.me/learningnets

transitions performed at each one of these cases. Instead, we decided to deploy automatic techniques to extract the possible state transitions triggered by non-user-task related functions. This is where SPARK technology excels. Even though the C part of the code can't be proved, it allows for a hybrid-verification approach, where other traditional testing approaches can be used to test that the behaviour of the code matches the functional specifications. SPARK procedures that model the state transitions of the C code can then be enhanced with contracts that reflect this verified behaviour. These new contracts allow GNATprove to verify any parts of the SPARK code that have dependencies on those C bindings. This allows for the hardening of software libraries that are not completely translated to SPARK. The hybrid approach used for this work is based on symbolic execution and is being explained in Section 3.2.3.

3.2.2 Dealing with concurrency

The TCP concurrent implementation of the *CycloneTCP* library is based on the *mutex* synchronization mechanism. The socket of a TCP connection is protected by a *mutex*, namely the `netMutex`, and only one function at any given time can lock this mutex and perform changes to the socket's state. Interactions must be considered at two locations: between the function calls, such as the `Open` or `Close` user functions, or during a function call, when the program waits for an event, such as `rcv` (see Figure 3). In both cases, the mutex on the socket is being released. Sections 3.2.2.1 and 3.2.2.2 deal with these two cases, respectively.

3.2.2.1 Concurrency: asynchronous changes of state

Between the user-task related function calls, the mutex that protects the socket structure is released, and segments can be received. The reception of a segment can lead to socket state changes, which represents TCP state transitions, and are considered asynchronous to the user-task related functions. Between two function calls, an infinite number of segments can be potentially received. To model any possible state-changes upon the reception of a segment in SPARK, the function `Tcp_Process_One_Segment` is introduced. The iteration over the `Tcp_Process_One_Segment`, when multiple segments are received, must also be considered to compute the resulting state after the iteration completes.

Let $\rightarrow$ be the function modelling the transitions of the TCP automaton, restricted to the possible transitions that can be performed by the reception of a message and its automatic response mechanism. Thus in our case,

https://t.me/learningnets

Algorithm 1: Algorithm to compute the reflexive and transitive closure of `Tcp_Process_One_Segment`

```
function Tcp_Process_Segment(Socket)
    S_last := Socket;
    S := S_last;
    for i = 1 to 3 do
        S_last := Tcp_Process_One_Segment(S_last) ;
        S := S ∪ S_last;
    end
    return S;
end
```

$\rightarrow$ represents the `Tcp_Process_One_Segment` function. We also need to consider the reflexive transitive closure [15] $\rightarrow^*$ of $\rightarrow$, with

$$\rightarrow^* = \bigcup_{n \in \mathbb{N}} \rightarrow^n$$

where $\bigcup$ represents the final result over calling the `Tcp_Process_One_Segment` repeatedly. By examining the TCP automaton in Figure 3, the maximum number of automatic state transitions triggered by received messages is three. This represents the longest path between transitions where its edges are correlated to a *rcv* action; namely the maximum path is between the SYN-SENT and CLOSE-WAIT, going through the SYN-RECEIVED and ESTABLISHED states without any user-related action. Thus, since we only consider the transitive closure in terms of states, and not in the number of continuously received segments, we can significantly reduce the number of iterations of $\rightarrow$ to compute $\rightarrow^*$:

$$\rightarrow^* = \bigcup_{n=1}^{3} \rightarrow^n$$

The algorithm given in 1 is sufficient to compute the transitive closure of the function `Tcp_Process_One_Segment` by taking advantage of the fact that SPARK can unroll small loops to enable the proving.

A call to the `Tcp_Process_Segment` function is added everywhere there is a call to `Os_Acquire_Mutex (Net_Mutex)` to consider all the possible states that the TCP connection can be in at the point of calling a user function. The choice of the `Tcp_Process_Segment` function-calls placement is based on the fact that the `Os_Acquire_Mutex (Net_Mutex)` is called at the beginning of the user functions, and after its call, the mutex is held by the

https://t.me/learningnets

user function. Thus, no other thread will be able to interfere and change the TCP state after the user function has acquired the mutex.

3.2.2.2 Concurrency: synchronous exchange

The second case in which the resulting state of a user function can be affected by the rest of the TCP tasks is when the user's functions release the mutex to allow for other events to take place. In this case, the C function `Tcp_Wait_For_Events` is called from the user function to check if the event requested is completed. For example, the `Tcp_Connect` user function, located in the `tcp_interface.adb`, calls the `Tcp_Wait_For_Events` to wait for the completion of the socket connection event, namely the `SOCKET_EVENT_CONNECTED`. The call to the `Tcp_Wait_For_Events` releases the mutex, and when the *arriving* task receives the appropriate segments, 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`, located in the `tcp_misc.c` file releases the mutex to allow for required events to happen. The function `Tcp_Update_Events`, located in the same file, detects that the event 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 the resuming of the 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.

Algorithm 2, which reuses `Tcp_Process_One_Segment`, computes the set of possible final states after the completion of an event. This is implemented by the `Tcp_Wait_For_Events_Proof` function, located in the `tcp_misc_binding.adb` file. This function is dedicated to proving the conformance to the functional specifications of the protocol in regards to the possible states that the TCP session can exhibit after each event completion.

We can compute precisely the states reached for each expected event thanks to the fact that SPARK unrolls loops.

https://t.me/learningnets

Algorithm 2: Function to compute the possible state after the completion of a particular event that is requested by a user-task related function.

```
function Tcp_Wait_For_Events_Proof(Socket, Event, Event_Mask)
    S_last := Socket;
    S := S_last;
    E := Tcp_Update_Events(S_last);
    if (E & Event_Mask) ≠ 0 then
        return S;
    end
    for i = 1 to 3 do
        S_last := Tcp_Process_One_Segment(S_last) ;
        S := S ∪ S_last;
        E := Tcp_Update_Events(S_last);
        if (E & Event_Mask) ≠ 0 then
            return S;
        end
    end
    return ∅;
end
```

3.2.3 Enhancing the library's security with symbolic execution and SPARK

Although SPARK does not have a native-mechanism to deal with concurrency, in Section 3.2.2 we demonstrated how such a mechanism could be improvised to allow the modelling of concurrency and all the possible interactions that can affect the state of a TCP connection using SPARK. More specifically, the three introduced functions, `Tcp_Process_One_Segment`, `Tcp_Process_Segment`, and `Tcp_Wait_For_Events_Proof`, are able to model the state changes that happen in functions that are not related to the user-task. Thus, these functions can be used to prove that the allowed state transitions of the *CycloneTCP* implementation conform to the functional specifications of the TCP protocol. The next step to enable this is to add appropriate contracts to the introduced functions. These contracts will reflect only the state transitions documented by the protocol's specification.

In the case of the user-task related functions, introducing and proving such contracts is straightforward as the whole code was translated to SPARK. In the case of non-user-related tasks, any introduced contracts that are needed to prove the validity of state transitions can not be proved by

https://t.me/learningnets

SPARK since the code is only written in C. To overcome this issue, a hybrid approach can be deployed, where symbolic execution is used to test the conformance of the transitions allowed by the implementation using assertions and symbolic execution. The assertions introduced in the C code for testing are based on the protocol's functional specification in [1]. Then, when assurances are gained through symbolic execution that the C code is conforming to the protocols functional specifications, the tested assertions are translated to SPARK contracts and added to the introduced functions that model the state transitions, namely, `Tcp_Process_One_Segment`, `Tcp_Process_Segment`, and `Tcp_Wait_For_Events_Proof`. The new assertions are then used to enable the proving of the user API code.

3.2.3.1 Symbolic execution brief introduction

Symbolic execution is a technique introduced to overcome the shortcomings of traditional testing approaches. In most realistic applications, the input space is too large to be exhaustively tested by any traditional testing method. Let's consider the case of testing a program using randomly generated inputs to identify if the program's implementation is violating any of the functional specifications and leading to security violations. Since random testing usually covers only a fraction of the possible input space, it is highly possible for the testing to miss important input cases that reveal such security vulnerabilities. Symbolic execution provides an alternative solution to this problem. Rather than using concrete inputs to test a program, it represents the program's inputs abstractly as symbols [6]. Then it progresses the execution symbolically constructing: a) expressions based on the symbols and taking into account the expressions and variables found on the symbolically executed path, and b) constraints in terms of those symbols and by accounting for each branch statement's possible outcomes. Solving these expressions and constraints using constraint solvers allows for the construction of test cases that could cause the violation of the property under investigation (if such cases exist).

Although exhaustive symbolic execution can be considered sound and complete (it prevents false negatives and false positives [4]), and thus, all the possible unsafe inputs can be captured, in reality, the technique suffers from scalability issues related mainly to state space explosion. Therefore, such an approach can not, in practice, replace the use of formal verification. Nevertheless, in this work, we trade soundness for performance to gain further software assurances. Although this hybrid-approach can not be considered complete because the C code has not been tested exhaustively with symbolic execution, it can still significantly contribute to the

https://t.me/learningnets

hardening of libraries that are not fully converted to SPARK. The approach provided confidence that state transitions, triggered by non-user-task related functions but capable of affecting the user task's state, are valid.

For this work the *KLEE*[1] symbolic execution engine was used. To better understand the work done with *KLEE*, an example of its usage within the scope of this work is given in the following section.

3.2.3.2 Example of gaining assurance on the conformance to the protocol's functional specification using *KLEE*

As explained in Section 4.2.1.2, the function `tcpProcessSegment`, (located in the file `tcp_fsm.c`) is responsible for processing the incoming segments depending on the state of the socket that received the segment. As shown in the code given in Section 4.2.1.2, for each of the possible cases of states, one of the `tcpState<StateName>` functions will be invoked from `tcpProcessSegment` to handle the segment. A call to one of the functions can cause a state of change, which will be reflected in the socket structure. Thus the *KLEE* symbolic execution can be utilized to test if each one of these functions respects the states transitions specified by the protocol's functional specification.

Let's take for example the `tcpProcessCloseWait`, function from the `tcpState <StateName>` family of functions. This function is called from the `tcpProcessSegment` function when a segment is received and the socket is in the state `CLOSEWAIT`. The TCP protocol's functional specification describes the expected behaviour as follows:

- If the segment contains the `RST` bit (Reset flag) then the TCP connection state becomes `CLOSED`.
- No other transition can be done, since the only transition to `LASTACK` requires an action by the user.

Thus, we want to prove that the C code implementation respects these rules extracted from the TCP specifications. To achieve this, symbolic execution is used to verify that all the possible paths when executing the `tcpProcessCloseWait` function will result in a state that respects the above rules. This process requires four steps:

1. Using the *KLEE* symbolic execution framework, incoming segments are represented symbolically. This representation allow us to account for all possible incoming segments.

[1] `https://klee.github.io`

https://t.me/learningnets

2. The state of the socket is changed to `CLOSEWAIT` to force the execution to go through the `tcpProcessCloseWait` function.
3. The function `tcpProcessCloseWait` is executed symbolically with KLEE.
4. After the call to the `tcpProcessCloseWait` function, a `klee_assert` is introduced that checks if the code leads to a valid state identified in the protocol's specification; in this case the two specifications given above for the possible outcome of calling the `tcpProcessCloseWait` function. The following C code demonstrates the implementation of this four steps:

```
int main() {
   // Initialisation
   socketInit();
   Socket* socket;
   TcpHeader *segment;
   size_t length;
   uint32_t ackNum, seqNum;
   uint8_t flags;
   uint16_t checksum;
   struct socketModel* sModel;
   segment = malloc(sizeof(TcpHeader));
   // creation of a TCP socket
   socket = socketOpen(SOCKET_TYPE_STREAM,
      SOCKET_IP_PROTO_TCP);

   // Step 1 : Representing the incoming segment symbolically
   klee_make_symbolic(&ackNum, sizeof(ackNum), "ackNum");
   klee_make_symbolic(&seqNum, sizeof(seqNum), "seqNum");
   klee_make_symbolic(&flags, sizeof(flags), "flags");
   klee_assume(flags >= 0 && flags <= 31);
   klee_make_symbolic(&checksum, sizeof(checksum),
      "checksum");
   segment->srcPort = 80;
   segment->destPort = socket->localPort;
   segment->seqNum = seqNum;
   segment->ackNum = ackNum;
   segment->reserved1 = 0;
   segment->dataOffset = 6;
   segment->flags = flags;
   segment->reserved2 = 0;
   segment->window = 26883;
```


https://t.me/learningnets

```
30    segment->checksum = checksum;
31    segment->urgentPointer = 0;
32    klee_make_symbolic(&length, sizeof(length), "length");
33
34    // Step 2 : Change of the socket state.
35    tcpChangeState(socket, TCP_STATE_CLOSE_WAIT);
36    klee_assert(socket->state == TCP_STATE_CLOSE_WAIT);
37    // We make a hard copy of the struct to check if the fields
38    // representing the Model of the socket have changed after
39    // the call to tcpStateCloseWait - a change would represent
40    // a potential bug in this case.
41    sModel = toSockModel(socket);
42
43    // Step 3 : Execution of the function
44    tcpStateCloseWait(socket, segment, length);
45    }
46    // Step 4 : Verification of the assertion that represents
         the allowed
47    // by the protocol's specification states to be reached,
         after the
48    // execution of the function in question; in this case the
49    // tcpStateCloseWait function
50    klee_assert(equalSocketModel(socket, sModel) &&
51            (socket->state == TCP_STATE_CLOSE_WAIT ||
52            (socket->state == TCP_STATE_CLOSED &&
53             socket->resetFlag)));
```

Using the above code, *KLEE* runs exhaustive symbolic execution on the `tcpStateCloseWait` function without raising an error on the introduced assertion. This proves that the function respects the TCP functional specification. Thus, the assertion used in step four of the above example can be translated into a SPARK post-condition for the `Tcp_Process_One_Segment` function. As explained earlier in Section 3.2.2, this function is essential for the modelling and the verification of concurrency and is used to prove that the concurrent interactions of the user-task with the rest of the tasks lead to valid TCP states; states that respect the protocol's functional specification. The code's security relies on the confidence we have on the `Tcp_Process_One_Segment` function. Thus, the symbolic execution approach followed for the `tcpStateCloseWait` example is used for all the functions of the same family, namely functions with the format `tcpState<StateName>`. In each case, if the protocol's specification is respected, then there is no error raised by the symbolic execution and the relevant assertion is also introduced as a post-condition to `Tcp_Process_One_Segment`. In the case that

https://t.me/learningnets

the symbolic execution raises an error, the code does not respect the protocol's specification, and this is considered a bug to be fixed. An example, of such detected bug, is given in the following section.

3.3 Bug found and resolved

Thanks to the SPARK work done for proving the conformance of the implementation to the TCP protocol's functional specifications, a bug was found in the original C implementation. In general, when a direct translation from C to SPARK source code is applied, there is always the risk that the SPARK code will inherit any diversions from the functional specifications that the C code already has. Any such diversions from the functional specification can be captured by expressing the program's functional specifications with SPARK contracts and by trying to prove them.

In our case, the initial translated SPARK user-task functions were also based on the C code, and thus, their functional correctness could not be taken for granted. Therefore, we introduced the correct contracts that represent the TCP's functional specifications on the appropriate procedures, such as `Tcp_Change_State` (see Section 3.2), and we used the SPARK technology to prove them. As explained in Section 3.2, such SPARK contracts represent all the valid transitions between TCP states described by the TCP automaton in Figure 3. Thus any attempt to prove these contracts while the source code does not conform to them will result in the SPARK provers raising a warning. Such a warning means that the TCP protocol's current implementation has a deviation from the protocol's functional specification, and thus, a bug exists in the implementation.

Indeed, the contract placed on the `Tcp_Change_State` procedure to represent the permitted by the protocol's functional specifications state transitions was able to capture a bug of an unauthorized state transition. The bug was found in the `Tcp_shutdown` procedure of the `Tcp_interface.adb` file. The part of the `Tcp_shutdown` procedure's code that contained the bug is listed 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;
```


https://t.me/learningnets

```
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  -- Failed to send FIN segment?
34  if Error /= NO_ERROR then
35    return;
36  end if;
37  -- Switch to the FIN-WAIT-1 state
38  Tcp_Change_State (Sock, TCP_STATE_FIN_WAIT_1);
```

The function `Tcp_Send` called at line 5 in the above code snippet changes the state of the socket to either the ESTABLISHED or CLOSE-WAIT if no error exists. The procedure `Tcp_Wait_For_Events` is then called at line 12. During this call, the mutex that guards the socket structure can be released, and at this point, there is a possibility that the remote can reset the connection. In this scenario, the socket's state is changed to either the ESTABLISHED, CLOSE-WAIT, or CLOSED state. This was the C code's original behavior, which was also translated to SPARK and encoded to the post-condition contract that was originally introduced to the `Tcp_Wait_For_Events` procedure. The relevant part of this contract is given below:

```
1 (if (Event_Mask and SOCKET_EVENT_TX_DONE) /= 0 then
2 (if Event = SOCKET_EVENT_TX_DONE then
3  -- RST segment received
```


https://t.me/learningnets

```
4  Model(Sock) = (Model(Sock)'Old with delta
5    S_State => TCP_STATE_CLOSED,
6    S_Reset_Flag => True) or else
7  (if Sock.S_State'Old = TCP_STATE_CLOSE_WAIT then
8    Model(Sock) = Model(Sock)'Old or else
9  elsif Sock.S_State'Old = TCP_STATE_ESTABLISHED then
10   Model(Sock) = Model(Sock)'Old or else
11   Model(Sock) = (Model(Sock)'Old with delta
12     S_State => TCP_STATE_CLOSE_WAIT))))
```

Thus, in case the above scenario is executed after the call of the `Tcp_Wait_For_Events` procedure at line 12, the execution continues to line 25 where the `Tcp_Send_Segment` is called to send a `FIN` segment, and thus, the state transitions CLOSE-WAIT → FIN-WAIT-1 and CLOSED → FIN-WAIT-1 are possible. Then at line 38 the procedure `Tcp_Change_State` is called to perform the possible transition of states. As explained in Section 3.2, the `Tcp_Change_State` procedure is the one called to implement all possible state transitions. Thus, it was selected as the procedure where we introduced a SPARK precondition that captures all the valid by the protocol's functional specification state transitions. Continuing our scenario of execution, when the procedure `Tcp_Change_State` is called at line 38, its precondition warns that the states CLOSE-WAIT → FIN-WAIT-1 and CLOSED → FIN-WAIT-1 are not allowed. Looking at the TCP automaton in Figure 3, we can confirm that the warning is correct since no such transitions exist on the automaton, and thus, both the C code and the SPARK translated code had this bug of allowing incorrect state transitions. The bug was also encoded on the original SPARK post-condition introduced to the `Tcp_Wait_For_Events` procedure. This contract was hand-written and imprecise as, initially, not enough information was known about the possible state transitions that could be triggered by the C code. After gaining confidence about such transitions with the use of the *KLEE* symbolic execution engine, as described in Section 3.2.3, the contracts that captured the bug were introduced to the `Tcp_Change_State`. The incorrect contracts introduced initially to the `Tcp_Wait_For_Events` procedure were also corrected. The affected code was then fixed to remove the erroneous transitions, both from the C and SPARK implementations. Finally, the SPARK prove was executed again to confirm that no more incorrect-transition warnings exist.

https://t.me/learningnets

4 Conclusion

The work is building upon the previous white paper of hardening software libraries using the SPARK technology, [8], where we demonstrated how to achieve AoRTE when translating C code to Ada and SPARK. In this deliverable, we go a step further than AoRTE to illustrate how the SPARK technology can be used to prove the conformance of a software library's implementation to the library's predefined functional specifications.

Acknowledging the wide use of the TCP communication protocol in cyber-physical systems, we selected a professional-grade embedded TCP/IP library, namely the *CycloneTCP* TCP, and we focused on the hardening of its TCP protocol implementation. We aimed to harden the TCP library in the areas that its original authors designated as the most vulnerable or crucial to conform to their functional specifications. Thus, the work was focused on hardening the user task's API, which allows the user to control the state of a TCP connection. More specifically, the work achieved:

- **Hardening the user's API** – Using SPARK contracts, we enforced the correct usage of the *CycloneTCP* TCP library's API by the user. More specifically, we enforced the library's user calls to the API functions to be in an order only allowed by the protocol's functional specification. We also ensure that the return codes of function-calls are checked for errors. Thus, in case of an error code is being returned, the application will not result in behaviour that is unspecified by the protocol specification.
- **Conformance to the protocol's functional specification** – We verified that the transitions allowed by the current implementation of the user-task related functions respect the state machine of Figure 3, which defines the permitted transitions between the different TCP states. We also ensured that the user's related functions are always updating a socket's state within the protocol's functional specifications.

Another significant contribution of this work is using a hybrid-approach involving the SPARK technology and a symbolic execution engine, KLEE, in our case. This hybrid approach allows proving the conformance of functional specifications related to parts of the code that are not fully translated to SPARK, but that can affect the behavior of the SPARK translated code. More specifically, we used KLEE to gain confidence that TCP state transitions enabled by code written in C respect the state transitions allowed by the TCP's functional specification. Then we introduced SPARK contracts related to those transitions that allowed the SPARK code to be proved.

https://t.me/learningnets

Any critical application that operates outside of its functional specifications is susceptible to safety and security vulnerabilities. Thus, the approach taken in this work can be used as a guideline on how to enhance the security of software libraries written in C by using the SPARK technology to prove the conformance of the library's implementation to its functional specifications. This is perfectly aligned with the need to conform to the emerging aviation standards related to aviation cyber-security airworthiness certification, such as the ED-202A, ED-203A, DO-326A, and the DO-356A standards.

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. The programme, delivered through a partnership between the ATI, Department for Business, Energy & Industrial Strategy (BEIS), and Innovate UK, addresses technology, capability, and supply chain challenges.

References

[1] *Transmission Control Protocol*. RFC 793. Sept. 1981. URL: https://tools.ietf.org/html/rfc793. (accessed: 04.07.2020).

[2] *A TCP/IP Tutorial*. Jan. 1991. URL: https://tools.ietf.org/html/rfc1180. (accessed: 13.08.2020).

[3] Karthikeyan Bhargavan et al. "Implementing TLS with verified cryptographic security". In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 445–459.

[4] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (May 2018). ISSN: 0360-0300. DOI: 10.1145/3182657. URL: https://doi.org/10.1145/3182657.

[5] Dor Zusman Ben Seri Gregory Vishnepolsky. *Urgent/11 – Critical vulnerabilities to remotely compromise VxWorks, the most popular RTOS*. ARMIS – White Paper. 2019. URL: https://info.armis.com/rs/645-PDC-047/images/Urgent11%5C%20Technical%5C%20White%5C%20Paper.pdf. (accessed: 24.06.2020).

https://t.me/learningnets

[6] C.S. Păsăreanu. *Symbolic Execution and Quantitative Reasoning: Applications to Software Safety and Security*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2020. URL: https://books.google.co.uk/books?id=UXLnDwAAQBAJ.

[7] Tobias Reiher et al. "RecordFlux: Formal Message Specification and Generation of Verifiable Binary Parsers". In: *Lecture Notes in Computer Science* (2020), pp. 170–190. ISSN: 1611-3349.

[8] Kyriakos Georgiou, Paul Butcher, and Yannick Moy. "Security-Hardening Software Libraries with Ada and SPARK". In: (June 2021), pp. 1–41. URL: https://www.adacore.com/papers/security-hardening-software-libraries-ada-spark.

[9] *35.100 – OPEN SYSTEMS INTERCONNECTION (OSI)*. URL: https://www.iso.org/ics/35.100/x/. (accessed: 13.08.2020).

[10] *CycloneTCP Download*. URL: https://www.oryx-embedded.com/download.html. (accessed: 13.08.2020).

[11] *CycloneTCP Embedded IPv4 / IPv6 Stack*. URL: https://www.oryx-embedded.com/products/CycloneTCP.html. (accessed: 13.08.2020).

[12] *International Organization for Standardization*. URL: https://www.iso.org/home.html. (accessed: 10.06.2020).

[13] *Internet Engineering Task Force (IETF) Tools*. URL: https://tools.ietf.org/. (accessed: 13.08.2020).

[14] *TCP automaton source*. URL: http://www.texample.net. (accessed: 17.06.2020).

[15] *Transitive Closure*. URL: https://en.wikipedia.org/wiki/Transitive_closure. (accessed: 10.09.2020).

[16] *U.S. Department Defense*. URL: https://dod.defense.gov/. (accessed: 13.08.2020).

https://t.me/learningnets