2017 IEEE Symposium on Security and Privacy Workshops

A Framework for Validating Session Protocols

Kit Sum Tse
Middlebury College
Email: kitsumtse@gmail.com

Peter C. Johnson
Department of Computer Science
Middlebury College
Email: pjohnson@middlebury.edu

***Abstract*—Communication protocols are complex; their implementations are difficult, causing many unintended (and severe) vulnerabilities in protocol parsing. While the problem of packet parsing is solved, session parsing remains challenging. Building on existing systems that reliably parse individual messages, we present our four-component framework for implementing protocol session parsers with the goal to improve security of protocol parsing: specification of a protocol message, description of a protocol state machine, testing routines to validate implementations against fake and real data, and graph generation to visualize implementations. This framework enables the creation of a session parser, which validates individual protocol messages in the context of other messages in the same conversation. This is helpful because more secure parsers lead to more secure communication.**

***Keywords*—Language-theoretic security, protocol state machine, protocol parsing, session parsing**

I. Introduction

Parsers are an important focus of the language-theoretic security research community, and for good reason: they usually comprise the initial line of defense against malformed input, whether malicious or otherwise. These defenses can be both explicit, in the form of, e.g., firewalls and intrusion detection systems (IDS), or implicit, in the code deep inside software that consumes the input. One of the central tenets of language-theoretic security is that parsing and verification of input should be performed both explicitly and completely before the data is operated upon—failure to do so results in well-documented "shotgun parsers" [3], which tend to be fragile and vulnerability-prone. As a result, much attention has been paid to the task of parsing data formats such as USB [6], DNP3 [4], and Zip [1].

The task of parsing the zipfile format is exemplary of most of the parsing work presented recently in that it consumes the entire input all at once and determines whether or not it is valid. But a great deal of data that needs to be parsed is not available all at once: indeed, the data often takes the form of sequences of messages in a *session* that follows the rules of a protocol. Consider the vast array of network communication protocols that feature messages passing back and forth between endpoints: TCP, HTTP, IMAP, SMB, Tor, POP3, SMTP, BitTorrent, not to mention myriad ad-hoc protocols for, e.g., online games and other proprietary software.

Checking the well-formedness of an individual message within these protocols is necessary but not sufficient: systems must also check whether the message makes sense in the context of the protocol session. For example, in TCP, a packet with the SYN flag set must be followed by a packet with the SYN and ACK flags set; any other packet, even if well-formed, is invalid given the context established by the first packet. Likewise, in SMTP, the MAIL command must precede the RCPT command. Despite the fact that the individual packets or commands may be correct in and of themselves, the conversation as a whole is not correct due to discrepancies between messages.

A huge number of protocols that support different message types use state machines to specify which messages are permissible given the sequence of previously-received messages. These state machines are often specified graphically (as in the case of TCP—see Figure 13) but also textually (also in the case of TCP—see RFC 793 [11]). Depending on the complexity of the protocol involved, these state machines can become horribly convoluted [10]. In any case, however, the task of implementing a given protocol often reduces to that of translating the graphical or text description of the state machine to executable code.

The protocol parsing efforts cited above can determine the well-formedness of a single message within the protocol but do not consider whether even a syntatically valid message makes sense within the context of the protocol conversation. In this paper, we present our methodology for creating *session parsers*: parsers that ensure a sequence of well-formed protocol messages follows the rules laid out by the protocol. In doing so, we build upon all the previous work that reliably parses individual messages; indeed, we assume such a component has already successfully parsed each and every message that is handed to the session parser.

Our work takes the form of a embedded domain-specific language (eDSL) for describing the protocol state machine, a system for passing sequences of messages through that state machine, routines to generate a graphical representation of the state machine, and a proof-of-concept implemention for TCP.

A domain-specific language is exactly as advertised: a language devised for a specific purpose; in our case, to specify a protocol state machine. (DSLs have long been used to describe protocol *messages*: for a brief history, see Section IV.) An embedded DSL is one that builds its constructs upon an existing language, an approach which has a number of benefits. First, rather than create a language toolchain from scratch, we can re-use the existing Haskell infrastructure. Second, because Haskell is a functional programming language, it includes a wide selection of ways to compose functionality, thus allowing for easy extensibility of our DSL. Third, we

DOI 10.1109/SPW.2017.35

intend to follow in the footsteps of previous work on parsing protocol messages [6] and automatically generate session-parsing C code that can be included in operating system kernels. Fourth, and in the same vein, the Haskell protocol-session model and its generated, C-language counterpart can be formally verified using the methodology pioneered in the seL4 project [7].

We imagine our framework being used by those designing protocols, implementing protocols, and testing protocol implementations. Protocol designers would use our approach to define their protocol, use our framework to test sample communication sessions, and automatically generate a graphical state-machine representation for documentation. In the fullness of time, we hope that implementers will take the protocol model created by the designers and automatically translate that model into whichever language the implementer desires. Testers would manually verify (though, also in the fullness of time, we would like to see this testing to be both formal and automated) that the generated state machine diagram matches hand-written diagrams provided with protocol documentation.

Our contributions are summarized as follows:

- a formalization of protocol state machines using a domain-specific language (DSL) allowing for the discrete definition of states, tests, and effects;
- a testing framework whereby sequences of crafted or captured messages can be passed through the session parser to verify its correctness;
- the ability to generate a graph from the protocol definition that can be used to informally verify state machine correctness, either independently or in conjunction with state machine diagrams given in protocol specifications; and
- a proof-of-concept session parser for TCP.

As mentioned above, our DSL is embedded in Haskell, which allows for extensibility using the broad selection of pre-existing infrastructure and the potential for easily generating a C implementation of a session parser that can be formally verified.

We proceed with a description of our session-parser formalization (Section II), followed by our our implementation of TCP (Section III), in which we demonstrate how our implementation can be empirically tested with crafted inputs, with captured inputs, and by visually comparing the state machine described in the code to the state machine presented in the spec. We review related work (Section IV) and finally close by musing on opportunities for future effort (Section V).

II. Implementation of protocol state machines

Given the fact that many stateful protocols are described as protocol state machines, it follows logically that their session parsers should be written as such on the code level. Thus, we need representations of states and transitions, and mechanisms to handle transitions between states based on an incoming packet and to run a sequence of messages through the implemented state machine. In addition, for the purpose of ensuring the parser's correctness, we have developed a testing framework that allows both micro- and macro-testing using crafted and captured data. Since many protocol specifications provide state machines describing protocols, we will also generate a state diagram with annotations of transitions from the implemented parser, enabling the visual comparison between the implemented version and the one provided by specification. This allows one to informally convince oneself that a particular implementation of a protocol state machine is correct.

A. Specifying a protocol state machine

1) States: In a protocol, a state is a collection of information that describes a given communication. Such information is not limited to the simple connection status, e.g., Closed, Established; it also contains other information, like port numbers of the two communicating endpoints, that helps determine the validity of a conversation as other packets arrive. This data structure in code should contain fields that mirror as closely as possible the various pieces of information needed to semantically parse a packet in a sequence, as specified in the protocol specification.

Since all stateful protocols have some notion of a connection or operational status such as Open and Closed, we created a simple data structure named `ProtocolStatus` for it. This helps narrow down legal transitions when handling packets. The possible values of `ProtocolStatus` varies based on the protocol being implemented. Other necessary, usually host-specific information for parsing a communication is also organized and encapsulated in a data strcture called `HostInfo`. Similar to `ProtocolStatus`, the fields of `HostInfo` depend on the protocol. If these two data structures are sufficiently expressive, i.e. encapsulate all necessary information for parsing a conversation, they can be used together as one data strcture to represent any state of a protocol state machine. However, if they are not, developers can create new data structures to capture unincorporated information as they see fit.

Note that in the upcoming discussion about specification of a state machine, we will be using `State` to denote the data structure that contains all necessary data for parsing.

2) Transitions: Transitions are integral to a protocol state machine because they are, in essence, the processing logic on packet semantics. A transition occurs when an input message is deemed well-typed in the context of the conversation. In other words, if the input message satisfies a set of conditions set forth by the protocol specification, then the automaton changes its state to reflect the new state of conversation. A transition, then, is a clean cause-effect relationship where condition satisfaction is the cause and state change is the effect. Thus, a transition can be broken down into two components: tests and effects. Tests represent transitional conditions an input has to meet, and effects represent transitional updates applied to a state.

Test: There are many semantic conditions that a single message must satisfy to maintain the validity of a conversation. Each condition can be cleanly represented as a `Test` data type in our framework. `Test` contains two fields: name and a

```
data Test = Test {
    name :: String
  , runTest :: (State, Message) -> Bool
}
```

Fig. 1: A Haskell data type representing a condition needed to be satisfied by a syntactically valid packet.

```
data Effect = Effect {
    name :: String
  , runEffect :: State -> State -> Message -> State
}
```

Fig. 2: A Haskell data type representing an update to the State of the protocol machine.

function. The name field allows protocol developers to identify a particular test, easing the process of debugging. The function is the test that will be applied to a message in a given communication context. It takes in a tuple of a `State` and a `Message` and evaluates to a `Bool` (see Figure 1). This structure can be utilized to represent all conditions that the TCP specification has outlined for state transitions. Then, to ensure semantic correctness, the session parser can simply execute a list of `Test`s against a target packet.

Effect: If a packet passes all the `Test`s, then the associated transition can take place. There are many moving parts, many fields in `State` that require updates, during the active process of transitioning the protocol state machine from one state to another. Luckily, these updates are generally explicit and discrete. Thus, the results of a single transition can be decomposed into individual, explicit `Effect`s.

The `Effect` data type, similar to `Test`, has two fields: name and a function. The name field serves the same purpose as that in `Test`. The function, on the other hand, is an update applied to the `State` of a conversation once the parser determines that the incoming message is semantically valid. This function takes in two `State`s and a `Message` and evaluates to another `State` (see Figure 2). Unlike `runTest`, `runEffect` requires two `State`s argument; using one `State` is insufficient in determining how to apply a specific effect and reflecting the new protocol state after application of all effects. For the same transition, already-administered updates to a `State` may affect subsequent updates since the manner in which any given update is applied primarily depends on the protocol `State` at the start of the processing of the target packet. Thus, `runEffect` requires two `State`s, where the first `State` argument is the protocol state after fully executing the transition based on the last packet, and the second `State` argument is for chaining effects of the current transition.

Given the implementations of `Test` and `Effect`, a transition can be fully described as a three-tuple of a string that identifies the transition, a list of `Test`s, and a list of `Effect`s, and it is declared as such with a type synonym named `Transition` in our framework. We can develop a

```
type Transition = ([Test], [Effect])

type DFA = [(ProtocolStatus, [Transition])]
```

Fig. 3: Haskell definition of a transition; Haskell definition of a protocol state machine.

data structure, with `ProtocolStatus` and `Transition`, that fully encapsulates the notion of a protocol DFA (see Figure 3). This data structure is a list of two-tuples where the first element of each tuple is of `ProtocolStatus` and the second element is a list of `Transition`. In essence, each tuple in the list is a state and all of its outgoing transitions.

With a DFA in place, we can now run packet(s) through the protocol state machine with a function that takes in a `DFA`, an initial `State`, and a list of `Message`s and evaluates to a `Maybe State`. The resulting `State` is wrapped in the `Maybe` type to reflect the notion that there may be semantically invalid packets in a sequence of messages. If the sequence is indeed invalid, the function evaluates to `Nothing`.

B. Micro- and macro-testing framework

All code is guilty of causing bugs until proven innocent.

Regardless of the difficulty of implementation, it is best practice to test code to prevent vulnerabilities. Thus, we have devised a framework for identifying bugs in a given protocol implementation. In particular, it examines whether the defined `Transition`s evaluates to the right `Maybe State` that reflects the specification and the programmers expectations.

Figure 5 shows the fundamental building block of this testing mechanism: `execTest`, a function that takes in a three-tuple of `State`, a list of `Messages`, and the `Maybe State`. It evaluates to a `Bool` that indicates whether the result of running the protocol state machine on the sequence of `Messages` from the initial `State`—first argument—matches the expected `Maybe State`, which is the third argument. With this encapsulation, we can perform micro-tests and macro-tests with various sequences of packets to test both single transitions and sequential transitions respectively. It is important to support both types of testing in this framework because the former allows programmers to pinpoint specific and isolated errors, while the latter reveals that (independently working) transitions do not work together as expected.

`execTests` supplements `execTest` by allowing developers to execute many tests at once and see which tests, if any, have failed. The function accepts a list of two-tuples where each two-tuple associates a name with a particular test and evaluates to a list of names of tests that have failed.

C. Graph generation

There is a gap between derivation and implementation. In other words, there is no guarantee that the developer's implementation of a protocol fully reflects the state diagram and the protocol specification. Since these session protocols

```
runDFA :: DFA -> State -> [Message] -> Maybe State
runDFA _ s [] = Just s
runDFA dfa s (m:ms) | not (null effects) = runDFA dfa newState ms
                    | otherwise = Nothing
    where getTransitions :: DFA -> State -> Maybe [Transition]
          getTransitions dfa (State s _ _) = lookup s dfa

          canChangeState :: [Transition] -> State -> Message -> [Effect]
          canChangeState [] _ _ = []
          canChangeState ((_,t,e):ts) s m
              | validate t s m = e
              | otherwise      = canChangeState ts s m

          validate :: [Test] -> State -> Message -> Bool
          validate ts s m = all (== True) tests
              where tests = map ($ (s, m)) . map runTest $ ts

          changeState :: State -> State -> Message -> [Effect] -> State
          changeState _ s _ [] = s
          changeState os s m (e:es) = changeState os (runEffect e os s m) m es
              where Just transitions = getTransitions dfa s
                    effects          = canChangeState transitions s m
                    newState         = changeState s s m effects
```

Fig. 4: Mechanism for running a list of Messages through a protocol state machine.

```
execTest :: (State, [Message], Maybe State) -> Bool
execTest (i, ms, e) = runDFA machine i ms == e

execTests :: [(String, (State, [Message], Maybe State))] -> [String]
execTests ts = map fst . filter (\(name,result) -> not result) $ tests
    where testNames = map fst ts
          conds     = map snd ts
          tests     = zip testNames $ execTests' conds
```

Fig. 5: Functions in Haskell for running tests on protocol state machine implementation.

can be described as a state machine, we have devised a mechanism to generate a directed graph from implementation. The nodes on this graph are the states of the protocol, its edges the transitions. With a state diagram descibing the processing logic of the parser, one can informally verify the parser's correctness by comparing the generated diagram to the state diagram provided by the specification.

We utilize the open-source software graphviz [5], specifically its *dot* tool to generate a directed graph. The dot program requires a description of the graph marked up in the DOT language. Thus, we have implemented code that produces and outputs to two different files a list of state transitions of the protocol state machine and a list of `Tests` and `Effects` associated with each and every transitions. To obtain the first list, multiple functions together take in a `DFA`, iterate over the list to retrieve names of the transitions, form a string containing all transitions where each is denoted with its originating state, target state, and a unique label for annotating the edges in a graph, and write the string to a file at a given file path. The second list is acquired in a similar fashion: multiple functions are used to produce a file with such list by taking in a list of `Transitions` and retrieving the names of the `Tests` and `Effects` associated with each `Transitions`. Each set of `Tests` and `Effects` associated with a particular

```
State 1,State 1,edge label 1
State 1,State 2,edge label 2
...
```

(a) Format of the file that describes state transitions.

```
Test 1,Test 2,...|Effect 1,Effect 2,...|1
Test 1,Test 5,...|Effect 2,Effect 3,...|2
...
```

(b) Format of the file that associates Tests and Effects with state transitions

Fig. 6: Formats of files describing a protocol state machine.

state transition is labeled accordingly to the labels denoted in the first file. The formats of these two files are displayed in Figure 6.

Once our program produces these two lists in two separate files, we can use them to generate a directed graph and a table annotating the transitions. The file responsible for generating the state diagram requires some parsing because it is not marked up in the DOT language. Therefore, we wrote a Bash script that outputs a file with equivalent content in the DOT language. On the other hand, the file for nnotating transitions requires no parsing. Thus, we wrote a simple Bash script that

```
----------------------------------------
| Label  | Tests        | Effects       |
|--------------------------------------|
| 1      | Test1        | Effect1       |
|        | Test2        | Effect2       |
|        | Test3        | Effect3       |
|        | Test4        | Effect4       |
|        | Test5        |               |
|--------------------------------------|
| 2      | Test2        | Effect3       |
|        | Test3        | Effect5       |
|        | Test6        | Effect6       |
|        |              | Effect7       |
----------------------------------------
```

Fig. 7: Format of a generated ASCII-art table of transition annotations.

```
data Message = Message {
    sPort :: Port
  , dPort :: Port
  , seqNum :: Number
  , ackNum :: Number
  , urg :: Flag
  , ack :: Flag
  , psh :: Flag
  , rst :: Flag
  , syn :: Flag
  , fin :: Flag
  , windowSize :: WindowSize
  , urgPtr :: UrgPtr
  , options :: Options
  , dataLen :: DataLen
}
```

Fig. 8: A Haskell data type representing a TCP packet.

outputs an ASCII-art table associating transitions with their respective `Tests` and `Effects` (see Figure 7). The two Bash scripts used to generate the diagram and table are then wrapped in a Bash wrapper script for ease of use.

III. Our TCP session parser

After discussion of our protocol implementation model, we now present its application using TCP (Transmission Control Protocol) as an example. This implemenation follows RFC 793 [11].

A. A TCP message

A TCP packet is defined as a `Message` using lang-sec principles and foundings from existing packet parsing work (see Figure 8). This data type is slightly different from the packet format specified in RFC 793: some defined fields in the specification carry no semantic significance, and are therefore meaningless for our session parser. `Message`, however, has a derived field, i.e., `DataLen` to help the session parser determine whether packets are acknowledged and in window. This field is derived from information in the headers of the IP and TCP packet.

```
data HostInfo = HostInfo {
    lPort :: Maybe Port
  , rPort :: Maybe Port
  , sndUnack :: Maybe Number
  , sndNxt :: Maybe Number
  , sndWnd :: Maybe WindowSize
  , sndWl1 :: Maybe Number
  , sndWl2 :: Maybe Number
  , iss :: Maybe Number
  , rcvNxt :: Maybe Number
  , rcvWnd :: Maybe WindowSize
  , irs :: Maybe Number
  , finStatus :: Maybe FinStatus
}
```

Fig. 9: A Haskell data structure that contains information needed to determine the state of a TCP communication session.

B. TCP state machine

Before discussing the implementation of our TCP session parser, we would like to clarify that our parser is written as an omniscient observer for the purpose of "objectively" validating a conversation. This means that the processing logic of packets is in essence the same as the explicit logic documented in RFC 793, but its implementation and operation do not parse packets from the perspective of a single host. Additionally, due to time and complexity constraints, there are limitations on our implementation of TCP. We will discuss these shortcomings at the end of this section.

1) ProtocolStatus and State: Our TCP session parsing logic is comprised of nine states—the different connection statuses. In our code, the connection states are of data type `ProtocolStatus` and can take of one of the following nine values: Listen, Syn, SynAck, Established, HalfClosedUnack, HalfClosed, SimultaneousClosed, WaitingToClose, and Closed. As mentioned in the previous section, a protocol state must contain all information needed to correctly parse a packet in a communication. Thus we have defined another structure `State` to represent the comprehensive state that is used to parse messages in a session. This data structure has three fields: `ProtocolStatus` and two fields of `HostInfo`. `HostInfo` is a data structure that contains host-specific information that can be deduced from incoming packets (see Figure 9). It almost has one-to-one correspondence to TCB (Transmission Control Block): it has an extra field called `finStatus` that keeps track of the status of the FIN packet—`Sent`, `Confirmed`, or `Nothing` (not sent yet). Each field in this data structure is updated per protocol specification.

2) Transition: To reiterate, a `Transition` is defined as a three-tuple of a name, a list of `Tests`, and a list of `Effects`. Our TCP protocol machine has 17 transitions (see Table I), all of which denote legal state changes. In the case of parsing a semantically invalid packet, `runDFA` will evaluate to `Nothing`, which can be thought of as an implicit outgoing transition for each `ProtocolStatus`.

We wrote 18 `Tests` and 15 `Effects` in total, and they are listed in Table II. Each and every `Test` and `Effect`

From	To
Listen	Syn
Syn	SynAck
SynAck	Established
Established	Established
Established	HalfClosedUnack
Established	Closed
HalfClosedUnack	HalfClosedUnack
HalfClosedUnack	HalfClosed
HalfClosedUnack	SimultaneousClosed
HalfClosedUnack	WaitingToClose
HalfClosedUnack	Closed
HalfClosed	HalfClosed
HalfClosed	WaitingToClse
HalfClosed	Closed
SimultaneousClosed	WaitingToClose
SimultaneousClosed	Closed
WaitingToClose	Closed

TABLE I: Transitions of a TCP state machine.

Test	Effect
hasSynFlag	updateProtocolStatus
hasNoSynFlag	updateAPorts
hasAckFlag	updateBPorts
hasNoAckFlag	updateSenderInitSeqNum
hasFinFlag	updateSenderNxtSeq
hasNoFinFlag	updateSenderRcvWnd
hasRstFlag	updateSenderRcvNxt
hasNoRstFlag	updateSenderUnack
hasData	updateSenderFinStatus
hasNoData	updateSenderSndNxt
isAtoB	updateReceiverSndUnack
isBtoA	updateReceiverFinStatus
isExpectedAckNum	updateReceiverRcvNxt
isInWindow	updateReceiverIRS
isClosedNotSendingData	updateReceiverSndWnd
isFromOpenHost	
isAckingFin	
isNotAckingFin	

TABLE II: All Tests and Effects used in the TCP protocol state machine.

concerns themselves with one specific aspect of either the given `Message` or `State` at which a packet is being procesed. For instance, `isInWindow` checks whether all data in a packet are in the receiving host's window while `isExpectedAckNum` examines whether a packet's acknowledgement number is in the acceptable range. You may notice that for some attributes such as the SYN flag, I have `Tests` for both their existence and non-existence, e.g., `hasSynFlag` and `hasNoSynFlag`. This is because using the NOT operator and somehow combining that notion with a list of `Tests` for a transition are messy and non-conforming to the structure of the framework.

We provide, as an example, the transition from `WaitingToClose` to `Closed` in code in Figure 10. All 17 transitions are coded in such format. If a `ProtocolStatus` has more than one outgoing transition, these transitions will have their respective lists of `Tests` and `Effects` and will be organized together as a list of `Transitions`.

```
waitingToCloseToClosedTests :: [Test]
waitingToCloseToClosedTests = [ isInWindow
                              , hasNoRstFlag
                              , hasNoSynFlag
                              , hasAckFlag
                              , isExpectedAckNum
                              , hasNoFinFlag
                              , hasNoData
                              , isAckingFin
                              ]

waitingToCloseToClosedEffects :: [Effect]
waitingToCloseToClosedEffects = [ updateSenderSndNxt
                                , updateSenderRcvNxt
                                , updateReceiverSndUnack
                                , updateReceiverFinStatus
                                , updateProtocolStatus
                                ]

waitingToCloseTransitions :: [Transition]
waitingToCloseTransitions = [ ( "WaitingToClose,Closed"
                              , waitingToCloseToClosedTests
                              , waitingToCloseToClosedEffects )
                            ]
```

Fig. 10: Transition from WaitingToClose to Closed in code.

Protocol	Number of packets	Results
SSH	51	Failure
SMB #1	26	Failure
SMB #2	24	Failure
IMAP	38	Success
HTTP #1	10	Success
HTTP #2	45	Success
HTTP #3	32	Success
SMTP	48	Success

TABLE III: Results of running captured sessions through session-parsing framework.

C. Testing and evaluation

1) Using crafted data: To test whether the implemented transitions are independently correct, we have written 43 micro-tests and three macro-tests where the micro-tests focus on individual transitions and the macro-tests examine the sequential ones. These tests cover most valid transitional scenarios of TCP and use carefully crafted (both valid and invalid) packets to test whether our parser can handle unexpected sequences of messages. All of the functional tests evaluate to the expected `State`.

2) Using real data: Testing using only crafted inputs is insufficient in "proving" the functional correctness of the session parser. Thus, we have also captured real TCP data from network traffic and running them through our session parser. Real data exposes functional shortcoming and programming mistakes caused by either buggy code or incomplete understanding of the protocol. We used data captured from tcpdump, parsed the resulting .pcap files to produce interpreted packets, i.e., `Message`, and fed them to our parser. Below is a table summarizing the results of testing using various sequences of captured packets from different applications built on top of TCP, followed by brief explanations and observations.

a) SSH: : We captured a sequence of 51 valid packets and passed it to our session parser. Unfortunately, our parser decides that the sequence is invalid. It deems the three-step handshake portion of the sequence valid, but it transitions to Nothing after processing the 8th packet. This is because both hosts of the connection use Options, one of which is the window scale option. The data length of this packet exceeds the unscaled window size indicated during the three-step handshake. Our parser is currently not equipped with data structures and processing logic for such scenario. With the assumption that all packets in such a conversation do not send data out of the sliding window, we removed `isInWindow`, a `Test` for checking in-window delivery, from all sets of Tests so that the session parser can be examined for other potential functional errors. Disregarding the window, the protocol state machine succesfully parses the entire sequence of messages and evaluates to a `Maybe State` at the end.

Successfully parsing this sequence without any impromptu changes to the processing logic requires changes to `Options`, `HostInfo`, and a `Test` named `isInWindow`: `Options` needs to encapsulate all possible Options specified by various TCP-related RFC's; `HostInfo` should encompass a notion of which options are used; `isInWindow` must include some calculations to appropriately scale the window size when determining whether the sent data are in range.

b) SMB: : By remotely accessing a shared network node, we were able to capture SMB data. In the captured sequence, there were two distinct connections, thus we demultiplexed the packets and parsed the two sequences of packets separately. Since both connections used the Window Scale options, the result is the same as that of SSH—our parser halts somewhere in the middle of each conversation. Removing `isInWindow` test from all conditions of transitions led to successful parsing of both sequences in their entirety.

c) IMAP: : We captured a sequence of 38 packets that resemble a successful mail delivery from a host application to the remote IMAP server. Unlike the SSH packets, these IMAP packets do not employ Options, and therefore our session parsing logic is not affected. Our parser successfully parses this sequence and deems it valid.

d) HTTP: : We captured HTTP as well by navigating to one webpage. Similar to SMB, there are multiple distinct connections to the default HTTP port among the captured packets, nine of which carried HTTP data. We demulitplexed these packets and analyzed three of them. All of them contain a single GET request for various resources needed for a web page. Despite the variance in number of packetsthey have 10, 45, and 32 packets respectively, they have the same parsing results: our TCP session parser successfully parses all sequences from three-step handshake to four-step teardown, meaning all sequences are valid.

e) SMTP: : We captured 48 SMPT packets. While this sequence is successfully parsed, it is different from the other sequences in that the connection did not get terminated with four-step teardown. One host sent a FIN packet, which was acknowledged by the other host. However, the other host sent a RST packet instead of a FIN to terminate the connection.

3) Observations: Testing our session parser against real data reveals that our parser is limited: in its current state, it cannot correctly handle packets that use Options. This means that the `Option` data type and `Test`s such as `isInWindow` need to be modified so that the parser's data structures properly reflect what kinds of options are used by the hosts in a communication session.

Additionally, more rigorous testing must be carried out. First, it is because both protocol specification and real data capture only the happy flow, which describes the valid sequences of messages. However, just handling the semantically valid sequence of packets is insufficient; a secure parser must be able to handle all inputs, or else it can be exploited. Thus, the parser must be tested against more handcrafted packets that can invalidate a conversation in all sorts of manners. Second, testing against real data reveals functional weaknesses. If this parser were to be used in practice, or integrated with the kernel, it must be able to handle various flavors of connections. For instance, it must be able to handle connections that dont use Options and those that use all available Options. We can capture more real data from applications we have examined thus far or those that we have not explored such as Border Gateway Protocol (BGP) and Simple Mail Transfer Protocol (SMTP).

D. Graph generation

One last step we took to persuade ourselves that our session parser is correct is generating a graph of the implemented state machine and an ASCII-table annotating the transitions. Figure 11 shows the graph generated by Dot, and Figure 12 lists out annotations for a few transitions. For comparsion purposes, we have also included the state machine supplied by the protocol specification (see Figure 13).

Since our session parser is written as an objective entity that parses packets omnisciently, the state diagram of our implementation is different from that provided by the protocol specification. First, there are fewer states in our implementation than those in the specification. This is because the nature of an omniscient parser requires us to collapse a few states specified in the RFC. For instance, SYN-SENT and SYN-RCVD in the RFC become `Syn` in our implementation, while FIN-WAIT-1 and CLOSE-WAIT-1 are replaced by `HalfClosedUnack`. As mentioned previously, since our parser is "objective," it does not need to use a `ProtocolStatus` to explicitly keep track of which host has sent or received a SYN or FIN packet. Second, there are no transitions between `Closed` and `Syn`, and `Closed` and `Listen` in our implementation because those transitions require non-packet parsing logic. Lastly, there are more transitions in our implementation, especially going from a given state to `Closed`. While the state diagram in the RFC does not display those transitions, the document does specify that for those originating states, if the incoming packet is in window and has its RST flag set, the machine should transition to `Closed`.

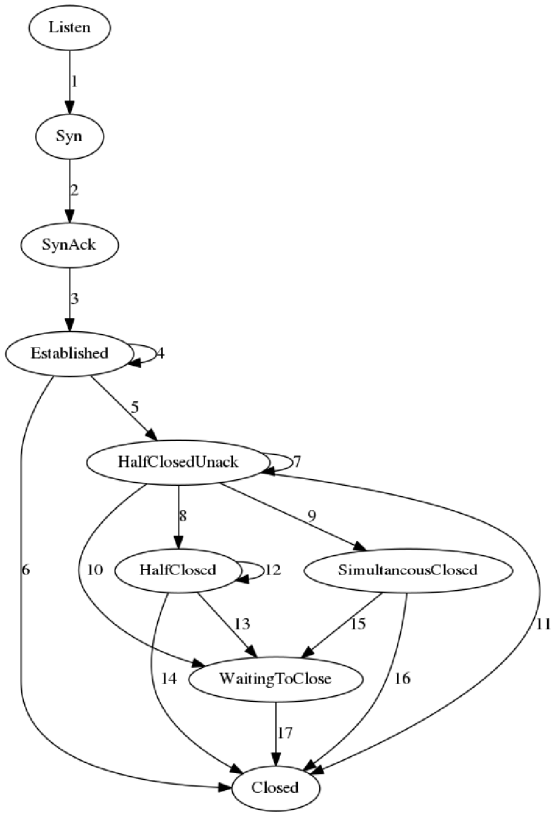

Fig. 11: TCP state diagram generated by the dot program.

```
|-------------------------------------------------|
| 4  | isInWindow         | updateSenderSndNxt    |
|    | hasNoRstFlag       | updateSenderRcvWnd    |
|    | hasNoSynFlag       | updateSenderRcvNxt    |
|    | hasAckFlag         | updateReceiverSndUnack|
|    | isExpectedAckNum   | updateReceiverSndWnd  |
|    | hasNoFinFlag       |                       |
|-------------------------------------------------|
| 5  | isInWindow         | updateSenderSndNxt    |
|    | hasNoRstFlag       | updateSenderRcvWnd    |
|    | hasNoSynFlag       | updateSenderRcvNxt    |
|    | hasAckFlag         | updateSenderFinStatus |
|    | isExpectedAckNum   | updateReceiverSndUnack|
|    | hasFinFlag         | updateReceiverSndWnd  |
|    |                    | updateProtocolStatus  |
|-------------------------------------------------|
| 6  | isInWindow         | updateProtocolStatus  |
|    | hasRstFlag         |                       |
|-------------------------------------------------|
```

Fig. 12: Part of the ASCII-art table that annotates the labeled state transitions.

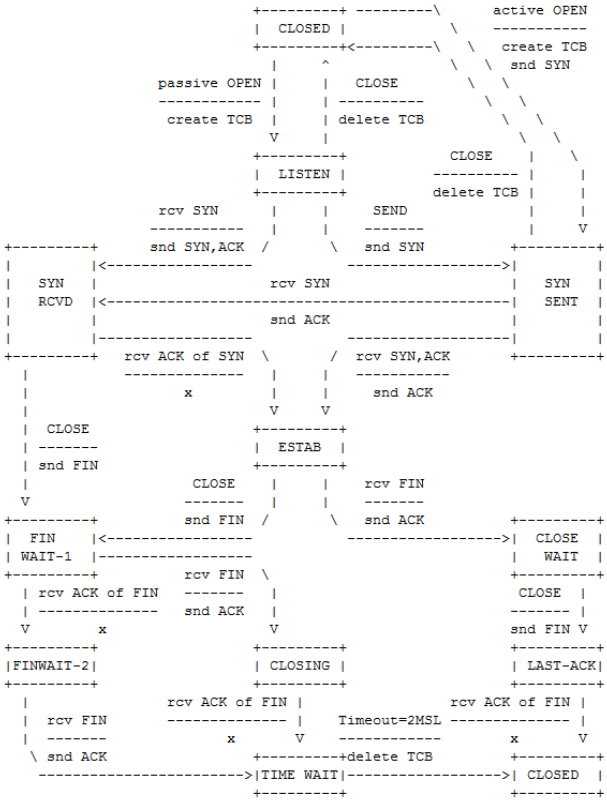

Fig. 13: TCP state machine from RFC 793.

E. Limtations

As briefly mentioned in the beginning of this section, there exist limitations on our TCP parser. They are enumerated in the following list.

1) Retransmission and duplicate packets are not handled correctly.
2) Out-of-order delivery is considered to be invalid.
3) URG and PSH flags are not considered in the processing logic.
4) The session parser packets contain no options.
5) Overflows of sequence and acknowledgment numbers are not taken care of.

These constraints exist because we decided to prioritize other features and functionalities of TCP, not because they are complicated to implement; in fact, our framework makes it easy to include them. Expressing a single `Transition` as a list of a two-tuple where its first element is a list of `Tests` and its second element a list of `Effects` enables easy modification. If more conditions need to be satisfied, we can simply compose more `Tests` and add them to the existing list of `Tests`; the same goes for updates. For instance, one of the things needed to be implemented to handle retransmitted packets is creating a new `Test` to check whether all of a given packets data have been acknowledged. It is also relatively

effortless to add possible transitions for a given state. New `Transitions` have to be constructed; however, once they are implemented, one simply has to add them to the list of `Transitions` associated with a `ProtocolStatus` in the `DFA`.

F. Summary

With the implementation of TCP parser, we have demonstrated the effectiveness of our abstractions and model. Using the abstractions of `State`s, `Test`s, and `Effect`s, our parser functions as specified by RFC 793.

Our testing framework provides a reliable mechanism for exposing functional weaknesses of an implementation against variations of protocol stack implementations and usage. For instance, TCP Options are used by some but not all applications. Using Options as an example, one can deduce that there are less utilized or popular features out there that our session parser must be able to handle. Without testing our system against real data, implementing additional features specified in other RFC's would be a tedious process.

Second, and perhaps more importantly, this model of writing session parsers makes it easy to write and debug parsing logic. Each condition an input has to satisfy and each update an accepted input caused, per protocol specification, is explicitly written and associated with a transition. Thus, to add another transition—which is what we had to do to accommodate the occurrence of FIN-ACK, FIN-ACK, ACK instead of a regular four-step teardown—is to simply write new `Test`s and `Effect`s and modify existing ones. If a condition (or update) were amiss for a transition, then adding it to the list of `Test`s for the transition would be sufficient.

To summarize, the data structures we have developed enable easy creation, development, and modification of a protocol session parser. Supplementing these data structures is our testing framework that makes it easy to discover bugs. Lastly, the generated graph helps to informally and visually verify the implementation.

IV. Related Work

Use of domain-specific languages for parsing protocol messages likely originated in the Bro [9] intrusion-detection system, which used a custom language to specify the format for dissection and examination. Other efforts include PacketTypes [8], intended for ad-hoc data formats; GAPA [2], intended mostly for application-level protocols; and, more recently, Nail [1].

In the realm of DSLs and eDSLs for parsing protocol sessions, work by Wang and Gaspes [13] is perhaps most relevant. Their Haskell-based DSL also allows specification of not just individual messages but also sequences of messages. Its intent is different, however, in that it aims to be used for embedded (e.g., hardware) systems, whereas we plan to integrate with operating system kernels and protocol analysis tools. Of the latter two, operating systems especially deviate from the behavior of embedded system: we expect to encounter issues of concurrency and scale that are not handled by existing work.

The pattern of creating a specification using a Haskell DSL, with the intention of generating a C implementation that can be formally verified, extends work by Johnson [6]. His proof-of-concept was for USB rather than a traditional networking protocol like TCP and did not include support for sessions. Using a Haskell model to formally verify a C implementation was prioneered (at least in the systems context) by the seL4 project [7].

Ragel [12] addresses the generic problem of producing C or C++ code that implements a finite state machine. Previous versions of our work attempted to use Ragel, in fact, though we found that building on top of Haskell was faster and more intuitive: embedding our DSL in Haskell gave us access to better tools and more convenient compositions.

V. Conclusion

We have presented our methodology and framework for parsing protocol sessions, where the semantics of individual messages are verified according to the communication context in which the message is received. Our domain-specific language, embedded in Haskell, allows one to specify the protocol state machine as a set of states and transitions, where the transitions are specified as sets of tests (if all pass, the transition is taken) and effects (ways in which the state is modified should all tests pass). We have shown how TCP can be specified using our DSL and demonstrated its efficacy by testing it with live network traffic. Despite the current lack of complete coverage of TCP features (e.g., Options), we are able to check the validity of the TCP layer of widely-used application-level protocols such as IMAP, HTTP, and SMTP. Additionally, our framework provides the structure by which additional tests can be conveniently added.

We see significant opportunities to expand this work. First and foremost, we would like to expand our proof-of-concept to support the full range of TCP features, most especially TCP options, which we expect to be the least simple aspect. Next, we would like to implement other session-based protocols than TCP, such as USB. We would also like to look at protocols such as HTTP that do not explicitly support sessions but which have been jury-rigged to do so (i.e., in the case of HTTP, using cookies).

With these models in hand, we would like to follow the pattern of previous work [6] and use them to generate C code to validate entire sessions, with the intention of integrating with operating system kernels. Finally, we would like to formally verify the generated code and thus provably harden the corner of the kernel that deals with session-based communication protocols.

References

[1] Julian Bangert and Nickolai Zeldovich. "Nail: A Practical Tool for Parsing and Generating Data Formats". In: *11th USENIX Symposium on Operating Systems Design*

and Implementation (OSDI 14). USENIX Association, 2014, pp. 615–628.

[2] Nikita Borisov et al. “Generic Application-Level Protocol Analyzer and its Language.” In: *Proceedings of the 14th Annual Network & Distributed System Security Symposium*. 2007.

[3] Sergey Bratus, Meredith L Patterson, and Dan Hirsch. “From shotgun parsers to more secure stacks”. In: *Shmoocon, Nov* (2013).

[4] Sergey Bratus et al. “Implementing a Vertically-Hardened DNP3 Control Stack for Power Applications”. In: *Proceedings of the 2nd Annual Industrial Control System Security Workshop*. 2016.

[5] *Graphviz — Graphviz - Graph Visualization Software*. http://www.graphviz.org/.

[6] Peter C. Johnson. *Towards A Verified Complex Protocol Stack in a Production Kernel: Methodology and Demonstration*. Tech. rep. TR2016-803. Hanover, NH: Dartmouth College, Computer Science, 2016.

[7] Gerwin Klein et al. “seL4: formal verification of an OS kernel”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009.

[8] Peter J. McCann and Satish Chandra. “Packet Types: Abstract Specification of Network Protocol Messages”. In: *SIGCOMM Comput. Commun. Rev.* 30.4 (Aug. 2000), pp. 321–333. ISSN: 0146-4833. DOI: 10.1145/347057.347563.

[9] Vern Paxson. “Bro: a system for detecting network intruders in real-time”. In: *Computer networks* 31.23 (1999), pp. 2435–2463.

[10] Erik Poll, Joeri De Ruiter, and Aleksy Schubert. “Protocol State Machines and Session Languages: Specification, Implementation, and Security Flaws”. In: *Proceedings of the 2015 IEEE Security and Privacy Workshops*. SPW ’15. IEEE Computer Society, 2015, pp. 125–133.

[11] Jon Postel. *Transmission Control Protocol*. RFC 793. RFC Editor, 1981, pp. 1–85.

[12] Adrian D. Thurston. *Ragel State Machine Compiler*. http://www.colm.net/open-source/ragel/.

[13] Yan Wang and Verónica Gaspes. “An Embedded Language for Programming Protocol Stacks in Embedded Systems”. In: *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM ’11. Austin, Texas, USA: ACM, 2011, pp. 63–72.