Programming Network Stack for Middleboxes with Rubik

Hao Li[1], Changhao Wu[1,2], Guangda Sun[1], Peng Zhang[1], Danfeng Shan[1], Tian Pan[3], Chengchen Hu[4]

[1]*Xi'an Jiaotong University* [2]*Brown University*
[3]*Beijing University of Posts and Telecommunications* [4]*Xilinx Labs Asia Pacific*

Abstract

Middleboxes are becoming indispensable in modern networks. However, programming the network stack of middleboxes to support emerging transport protocols and flexible stack hierarchy is still a daunting task. To this end, we propose Rubik, a language that greatly facilitates the task of middlebox stack programming. Different from existing hand-written approaches, Rubik offers various high-level constructs for relieving the operators from dealing with massive native code, so that they can focus on specifying their processing intents. We show that using Rubik one can program the middlebox stack with minor effort, *e.g.*, 250 lines of code for a complete TCP/IP stack, which is a reduction of 2 orders of magnitude compared to the hand-written versions. To maintain a high performance, we conduct extensive optimizations at the middle- and back-end of the compiler. Experiments show that the stacks generated by Rubik outperform the mature hand-written stacks by at least 30% in throughput.

1 Introduction

Middleboxes are pervasively deployed in modern networks. In the middlebox, a low-level *network stack* (*e.g.*, TCP/IP) is responsible for parsing raw packets, and a set of high-level hooks (*e.g.*, HTTP dissector) process the parsed data for various purposes. There is a constant need for programming the network stack of middleboxes in order to accommodate different networks (*e.g.*, IEEE 802.11 [9] in WLAN), support new protocols (*e.g.*, QUIC [50]), realize customized functions (*e.g.*, P4 INT [5]), and capture new events (*e.g.*, the IP fragmentation [11]), *etc*.

By programming a middlebox stack, an operator Alice is mainly concerned with the following tasks. (1) *Writing new parsers.* It is common that a network needs to support a new protocol. Then, Alice needs to write a new parser to parse such traffic. (2) *Customizing stack hierarchy.* Another common need is to change the protocol layering, say to support encapsulation methods like IP-in-IP [1]. Then, Alice needs to re-organize the parsers. (3) *Adding new functions.* Finally, new functions may be requested from the network stack to meet diverse needs. For example, Alice may need to know when IP fragmentation happens, and modify an existing network stack to capture this event. In the following, we will show the difficulty of the above programming tasks.

The difficulty of writing new parsers. Currently, protocol parsers are written in low-level native code to ensure the high efficiency, which leads to a large number of lines of code (LOC) even for a single protocol, *e.g.*, ∼7K C LOC for TCP protocol parser in mOS [14]. Someone may argue that the TCP/IP is the de facto narrow waist for middlebox processing, so a general-purpose TCP/IP substrate is sufficient for extended programmability. However, many networks have their own customized transport layers (*e.g.*, QUIC [50]), and in those cases, Alice still needs to manually write new parsers.

The difficulty of building stack hierarchy. The data structures in current middlebox stacks are monolithic and closely coupled with standard stacks, making it difficult to reuse the existing protocol parsers for upgrading the stacks. For example, `libnids` [11] can parse Ethernet, IP, UDP and TCP protocols, but due to its TCP-specific data structure, it can hardly support the IP-in-IP stack, *i.e.*, ETH→IP→GRE→IP→TCP, although there is only one thin GRE parser need to be added. As a result, we have to modify 1022 LOC of `libnids` in our preliminary work to support the IP-in-IP stack, where most effort (815 LOC) is devoted to stack refactoring.

The difficulty of adding new functions. Since the network stack is closely coupled, adding a new function often needs a deep understanding of a huge code base. For example, if one wants to capture a low-level IP fragmentation event that is not supported in Zeek [24], she has to first read all the native code related to the IP protocol and the event callback module, which involves ∼2K LOC. Another scenario would be the feature pruning: *e.g.*, for writing a stateful firewall without the need to buffer the TCP segments, the operator has to go through considerable code to ensure the code deletion in a full-functional TCP stack will not produce other side effects.

Table 1: Existing approaches to program middlebox stack and their support of the three programming tasks.

Approaches	Protocol Parser	Stack Hierarchy	Stack Functions
Packet Parser (*e.g.*, P4 [18], VPP [23])	◐	◐	○
TCP-Specific Stacks (*e.g.*, mOS [44])	○	○	◐
NFV Frameworks (*e.g.*, NetBricks [58])	◐	●	◐

● : *Can be fully programmed with high-level abstractions, i.e., minor LOC*
◐ : *Partially supported or can be programmed with moderate LOC*
○ : *Not supported or can only be programmed with large amount of LOC*

Apart from the mature and fixed TCP stack libraries like mOS and `libnids`, many attempts have been made to fulfill the above three tasks in order to make the middlebox stack programmable. However, none of them can fully facilitate those onerous tasks, as shown in Table 1: the packet parsers like P4 [18] cannot efficiently buffer the packets; the NFV frameworks like NetBricks [58] and ClickNF [37] often rely on TCP-specific modules that are pre-implemented with many native LOC. We discuss these related work in detail in §2.2.

In fact, there exists a dilemma between the abstraction level and code performance when enabling programmability on the performance-demanding middlebox stack. On the one hand, many exceptions like out-of-order packets can arise in L2-L4, so higher-level abstractions are desired to relieve the developers from handling those corner cases. On the other hand, optimizing a stack at wire speed also relies on tuning underlying processing details, which becomes much more challenging if those details are transparent to the developers. As a result, previous works tend to trade off the programmability for the performance, offering limited programmability over specific stacks, *e.g.*, TCP (see §2.3).

In this paper, we propose Rubik, a domain-specific language (DSL) for addressing the above dilemma, which can *fully* program the middlebox stack while assuring *wire-speed* processing capability. For facilitating the stack writing, Rubik offers a set of handy abstractions at the language level, *e.g.*, packet sequence and virtual ordered packets, which handle the exceptions in an elegant fashion. Using these declarative abstractions, operators can compose a more robust middlebox stack with much fewer lines of code, and retain the possibility of flexible extension for future customization needs. For maintaining high performance, Rubik translates its program into an intermediate representation (IR), and uses domain-specific knowledge to automatically optimize its control flow, *i.e.*, eliminating the redundant operations. The optimized IR is then translated into native C code as the performant runtime.

In sum, we make the following contributions in this paper.

- We propose Rubik, a Python-based DSL to program the network stack with minor coding effort, *e.g.*, 250 LOC for a complete TCP/IP stack (§3 and §4).
- We design and implement a compiler for Rubik, where a set of domain-specific optimizations are applied at the IR layer, so that all stacks written in Rubik can benefit from those common wisdom, without caring about how to integrate them into the large code base (§5).
- We prototype Rubik, and build various real cases on it, including 12 reusable protocol parsers, 5 network stacks, and 2 open-source middleboxes (§6). Experiments show that Rubik is at least 30% faster than state of the art (§7).

2 Motivation and Challenges

In this section, we demonstrate that programming middlebox stack is a necessity in modern networks (§2.1), while no existing tool can really enable such programmability (§2.2). We pose the challenges of designing a DSL for middlebox stack, and summarize how our approach addresses them (§2.3).

2.1 Programming Middlebox Stack Matters

As presented in §1, programming a middlebox stack requires huge human effort. However, some argue that it might not be a problem: most middleboxes work with standard TCP/IP protocols, thus a well-written TCP/IP stack should be sufficient. In contrast, we believe there are plenty of scenarios where a deeply customized middlebox stack is desired.

First, the middlebox stacks need to be customized for serving diverse networks with different protocols [13,32,34,42,43, 63,70]. Apart from the existing ones, we note that the emerging programmable data plane may cause an upsurge of new protocols, each of which requires an upgrade of the middlebox stack, or its traffic cannot traverse the network [16,55].

Second, even for a fixed stack, the operators may still manipulate the packets in arbitrary ways, and the implementation of middlebox stacks varies to satisfy those user-specified strategies. For example, if a TCP packet is lost in the mirrored traffic [22], `libnids` will view this as a broken flow and directly drop it for higher performance [11], while mOS will keep the flow and offer an interface to access the fragmented sequence for maximumly collecting the data [44].

Third, middleboxes are constantly evolving for providing value-added functions, *e.g.*, adding a new layer [12,35], measuring performance [20], inspecting encrypted data [40,49,67] and migrating/accelerating NFV [38,47,57,68,72]. These extensions heavily rely on a highly customized stack.

The above facts prove that programming middlebox stack is a necessity in modern networks, which however demands massive human effort. In practice, such overhead has begun to hinder the birth of new protocols: the middlebox vendors tend to be negative to support new protocols, as the huge code modification can cost large human labor and introduce bugs or security vulnerabilities. For example, some middlebox vendors suggest blocking the standard port of QUIC (UDP 443) to force it falling back to TCP, so that their products can analyze such connections [10]. This heavily impacts the user experience [4,19], and will finally result in the ossification of underlying networks [60].

2.2 Related Work

Plenty of attempts have been made to facilitate the middlebox development, as shown in Table 1. In the following, we show why they are not sufficient to program the middlebox stack.

Programmable packet parsers like P4 [18] and VPP [23] can dissect arbitrary-defined protocols in an amiable way. However, since they target at implementing a switch/router, they cannot efficiently buffer and/or reassemble the packets.

There are also DSLs, *e.g.*, Binpac [59], Ultrapac [52], FlowSifter [56] and COPY [51], that can automatically generate L7 protocol parsers. The parsers they generate focus on the L7 protocols like HTTP, hence can only work on the already reassembled segments. In other words, they can facilitate the development of high-level functions of a middlebox, *e.g.*, HTTP proxy, deep packet inspection, but have to cooperate with a low-level stack, instead of serving as one.

TCP stack libraries include those for end-host stacks and those for middlebox stacks. The end-host stack libraries, *e.g.*, mTCP [45], Modnet [61], Seastar [27], F-Stack [25], only maintain the unidirectional protocol state for a certain end host, while middlebox stacks must track the bidirectional behaviors of both sides. As a result, the middlebox developers cannot build their applications on end-host stack libraries.

On the other hand, the major feature a middlebox stack library provides is the bidirectional TCP flow management. Previously, such libraries are closely embedded in IDS frameworks like Snort [21] and Zeek [24], therefore cannot be reused when developing new applications. `libnids` [11] decouples the TCP middlebox stack from the high-level functions, making the stack reusable. Recent works like mOS [44] and Microboxes [53] implement a more comprehensive TCP stack with fast packet I/O, and more importantly, provide the flexible user-defined event (UDE) programming schemes, *e.g.*, dynamic UDE registration, parallel UDE execution. Besides, they provide limited programmability over TCP stack, *e.g.*, unidirectional buffer management. However, all above approaches are hard-coded, hence cannot support non-TCP stacks without massive native code understanding and writing.

NFV frameworks offer a packaged programming solution from L2 to L7. However, none of them provide complete middlebox stack programmability. MiddleClick [30] and ClickNF [37] can manipulate the stack hierarchy using Click model [48], but they rely on pre-implemented elements, *e.g.*, ClickNF implements the TCP-related elements with 156 source files (12K LOC in C++) [7]. NetBricks [58] supports the customization of the header parser, the scheduled events, *etc*, which is sufficient for programming a connectionless protocol. However, the abstractions for programming a connection-oriented protocol, *e.g.*, transmission window, connection handshake, are still TCP-specific. OpenBox [33] and Metron [46] abstract and optimize a set of L2-L7 elements for middlebox applications. However, the flow management element still has to be pre-implemented using native code.

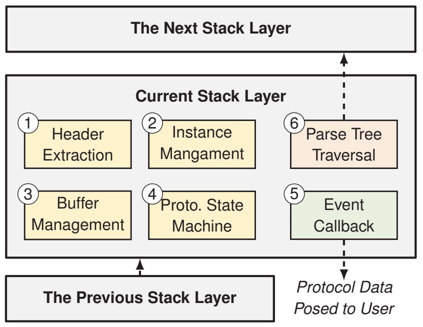


Figure 1: The three key modules in a middlebox stack layer: protocol parser (yellow boxes), event callback (green box), and parse tree traversal (red box).

2.3 Challenges and Our Approach

A DSL that fully captures the L2-L4 abstractions can be a cure to above problems. In the next, we revisit the high-level pipeline of middlebox stack, and pose the challenges of designing and implementing a DSL corresponding to it.

Middlebox stacks follow a layer-based processing pipeline, which is largely the same with the end-host stacks, as shown in Figure 1. Specifically, each stack layer starts by *parsing the protocol* (①–④), which extracts the header, manages the instance, buffers the segments, updates the protocol state machine (PSM), *etc*. Next, the *event callback* module (⑤) will raise the events with the protocol data fed to the users, *e.g.*, the reassembled or retransmitted data. Finally, the *parse tree* (⑥) will decide the next protocol to be parsed. However, even the above pipeline is seemingly natural and generalized, designing and implementing a DSL corresponding to it can still be a challenging task. The reason is two-fold.

First, working at L2-L4, the middlebox stacks run more complex logic than it appears in the pipeline. For example, the out-of-order packets can mess around the PSM, *e.g.*, an early-arrived FIN packet may mislead the stack to tear down the TCP connection. Each of these exceptions is handled with native code in fixed stacks, and it is extremely difficult to provide a neat DSL that covers all such cases. *Rubik addresses this challenge by offering a set of high-level constructs to hide such exceptions from programmers, e.g., "packet sequence" that hides the retransmission exception and "virtual ordered packet" that hides the out-of-order exception, which can maximumly correspond to the intuitive pipeline (§4).*

Second, the middlebox stacks must realize wire-speed processing to serve high-level functions. However, due to the complexities presented above, the optimizations on the stack can only be achieved by carefully tuning the native code. That is, the program written in DSL that hides the processing details will likely produce low-performance native code. *Rubik addresses this challenge by employing an IR and a set of domain-specific optimizations on it before producing the native code, which automatically optimize the DSL program to avoid potential performance traps (§5).*

3 Rubik Overview

In this section, we use a walk-through example to overview how Rubik can facilitate the middlebox stack writing. Our example is an ETH→IP/ARP stack, where we raise a typical event, IP fragmentation, for each IP fragment.

Figure 2 shows the real (and almost complete) Rubik code of realizing our example. We start from declaring the IP layer (Line 2), which initializes internal structures of a connectionless protocol, including the header parser, the packet sequence, PSM, *etc*. These components are specialized as follows.

Parsing the header fields (Line 5–18). One has to first define the header format before she references the headers. In Rubik, a header format is a Python class that inherits `layout`, and each header field is a member of this class, which specifies its length measured by `Bit()`. The order of the members indicates the layout of the fields. Line 5–15 show the IP header structure, `ip_hdr`. We can then use this structure to compose the header parser with one LOC in Line 18. After that, Rubik can reference the fields by their names, *e.g.*, `ihl` can be referenced by `ip.header.ihl`.

Managing the instance table (Line 20). Having the headers, the stack layer then finds the instance that the packet correlates to, *e.g.*, the TCP flow, and processes it by the previous state and data of the same instance. The instances are stored in an instance table, *e.g.*, TCP flow table.

To achieve this, Rubik forms a key to index the instance table, which consists of bi-directional protocol contexts. For IP protocol, the instance key is a list that contains the source and destination IP addresses (Line 20). Note that for connection-oriented protocols, the instance key should contain two lists, each of which indexes the packets of one direction.

Preprocessing the instance (Line 23–27). Before getting into buffer and PSM processing, operators can update some permanent contexts for each individual instance (`perm`), or use some temporary variables for facilitating the programming (`temp`). This part of logic will be executed each time after the instance is found/created. Line 23–27 define a temporary data structure that stores the fragmentation offset for each IP packet, which can then be referenced as `ip.temp.offset`.

Managing the packet buffers (Line 30–31). Many protocols buffer the packets to ensure the correct order of incoming packets. Rubik offers a *packet sequence* abstraction to handle this task. In our example, the IP protocol has to buffer the fragmented packets according to their fragmentation offset. Line 30–31 define a sequence block filled with the IP payload and indexed by the fragmentation offset. This block will be inserted into the packet sequence associated with the instance, which is automatically sorted in ascending order by the meta.

A connection-oriented instance will maintain two sequences for two sides, respectively. The packets payload will be automatically inserted into the corresponding sequence according to the direction indicated by the instance key.

```
1   # Declare IP layer
2   ip = Connectionless()
3
4   # Define the header layout
5   class ip_hdr(layout):
6       version = Bit(4)
7       ihl = Bit(4)
8       ...
9       dont_frag = Bit(1)
10      more_frag = Bit(1)
11      f1 = Bit(5)
12      f2 = Bit(8)
13      ...
14      saddr = Bit(32)
15      daddr = Bit(32)
16
17  # Build header parser
18  ip.header = ip_hdr
19  # Specify instance key
20  ip.selector = [ip.header.src_addr, ip.header.dst_addr]
21
22  # Preprocess the instance using 'temp'
23  class ip_temp(layout):
24      offset = Bit(16)
25  ip.temp = ip_temp
26  ip.prep = Assign(ip.temp.offset,
27                   ((ip.header.f1<<8)+ip.header.f2)<<3)
28
29  # Manage the packet sequence
30  ip.seq = Sequence(meta=ip.temp.offset,
31                    data=ip.payload[:ip.payload_len])
32  # Define the PSM transitions shown in Figure 3
33  ip.psm.last = (FRAG >> DUMP) + Pred(~ip.header.more_frag)
34  ip.psm.frag = ...
35
36  # Buffering event
37  ip.event.asm = If(ip.psm.last | ip.psm.dump) >> Assemble()
38  # Callback each IP fragment using 'ipc'
39  class ipc(layout):
40      sip = Bit(32)
41      dip = Bit(32)
42  ip.event.ip_frag = If(~ip.psm.dump) >> \
43                      Assign(ipc.sip, ip.header.saddr) + \
44                      Assign(ipc.dip, ip.header.daddr) + \
45                      Callback(ipc)
```

Figure 2: IP layer and fragmentation event written in Rubik.

Updating the PSM (Line 33–34). PSM tracks the protocol states, which are useful in most connection-oriented protocols (*e.g.*, TCP handshake), and also in some connectionless protocols that buffer the packets (*e.g.*, IP fragmentation). Consider the IP PSM shown in Figure 3. If an IP packet unsets the `dont_frag` flag, the parser will take a transition from the `DUMP` state to the `FRAG` state that waits for more fragments. The instance will be destroyed if the PSM jumps into an accept state, *e.g.*, `DUMP` in IP PSM. Line 33 defines the `last` transition, *i.e.*, `FRAG`→`DUMP`.

Assembling data and hooking IP fragments (Line 37–45). Unlike the UDEs that are raised by the high-level functions, *e.g.*, HTTP request event, the built-in events (BIEs) reveal the inherent behaviors in the stack, *e.g.*, buffer assembling, connection setup. Previous works only pose fixed and TCP-specific BIEs [11, 44], while Rubik can program two types of BIEs for arbitrary stacks.

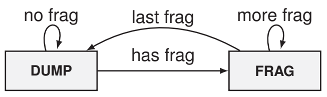


Figure 3: Simplified PSM for IP fragmentation.

The first is for the packet sequence operations, *i.e.*, buffer assembling. Rubik uses `If()` to specify the conditions of raising the events and `Assemble()` to assemble the continuous sequence blocks. This function will form a service data unit (SDU) for the next layer parsing. Line 37 defines the events for assembling the fragments in IP layer.

The second type of action is for posing the user-required data, which is achieved by a `Callback()` function that indicates what content should be posed. Line 39–45 define an event on the condition that fragmented packets arrive. The back-end compiler will declare an empty function in the native C code, *i.e.*, `ip_frag(struct ipc*)`, and invoke it each time the condition is satisfied.

Parse tree for ETH→IP/ARP. Each time after processing a layer, the network stack decides the next layer to be proceeded, until it reaches the end of the stack. All such parsing sequences form a parse tree [39].

The parse tree of our example consists of two layers, *i.e.*, Ethernet, and IP/ARP. The stack executes from the root node, which triggers the Ethernet protocol parser. This parser will extract the headers of Ethernet, *e.g.*, `dmac`, `type`. Next, the parse tree checks the predicates carried by the two transitions, and decides which one could be further parsed. In this case, the `type` field is used to distinguish the IP and ARP protocol. Rubik offers a simple syntax similar to PSM transition to define the parse tree, as shown below.

```
st = Stack()
st.eth, st.ip, st.arp = ethernet, ip, arp
st += (st.eth>>st.ip)  + Pred(st.eth.header.type==0x0800)
st += (st.eth>>st.arp) + Pred(st.eth.header.type==0x0806)
```

where `ethernet`, `ip` and `arp` are protocol parsers. We note that the parsers can be reused in the stack. For example, we can define another IP layer in this stack with `st.other_ip = ip`. This will largely facilitate the customization of encapsulation stacks (see Appendix C.3 for a GTP example).

Summary. We omit the implementation of Ethernet layer and ARP layer, which are quite simple compared to IP layer. In sum, we use ∼50 LOC to define the IP protocol parser (see Appendix C.1 for the complete code), 7 LOC to hook the expected event, and 4 LOC to build the parse tree. As a comparison, `libnids` consumes ∼1000 C LOC to implement the similar stack [11].

4 Rubik Programming Abstractions

§3 shows the potential of reducing coding effort with Rubik. However, as discussed in §2.3, there exists lots of complex programming needs that call for more sophisticated programming abstractions. In this section, we dive into the language internals to present how Rubik conquers those complexities.

4.1 Context-Aware Header Parsing

We consider the following two context-aware header parsing needs in middlebox stack, and address them using Rubik.

Conditional layout. The L2-L4 protocols can have conditional header layout. For example, QUIC uses its first bit to indicate the following format, *i.e.*, long header or short header. To this end, we can first parse the fixed layout, using which to determine the next layout to be parsed, as shown below.

```
quic.header =  quic_type
quic.header += If(quic.header.type == 0) >> long_header
                Else() >> short_header
```

Type-length-value (TLV) parsing. Rubik extends its header parsing component in two ways to express the TLV fields: (1) the value of a field can be assigned before parsing, which can be used to define a `type` field, *e.g.*, `type = Bit(32,const=128)` defines a 32-bit field that must be 128; (2) the length of a field can refer to a pre-defined field with arithmetic expressions, which can be used to define the length of `value` field, *e.g.*, `value = Bit(length << 3)`.

Besides, TLV headers are often used in a sequence with non-deterministic order, *e.g.*, TCP options. Rubik offers a syntax sugar for parsing those headers, as shown below.

```
tcp.header += AnyUntil((opt1, opt2, opt3), cond)
```

where `opt1`–`opt3` are TLV header layouts, and `AnyUntil()` will continuously parse the packet according to their first fields, *i.e.*, `type`, until `cond` turns to be false.

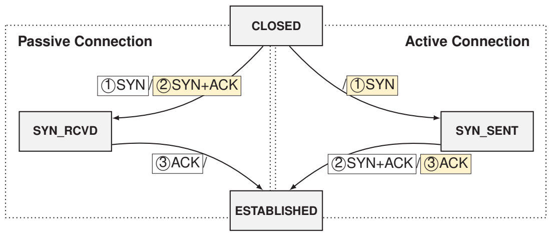
4.2 Flexible Buffer Management

The transport protocols can buffer the packets in flexible ways other than simply concatenating them in order. Specifically, we consider the following three exceptions in buffer management, *i.e.*, retransmission, conditional buffering and out-of-window packets, and use the sequence abstraction in Rubik to address them.

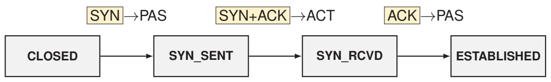
Each time a sequence block is inserted, `Sequence()` in Rubik will compare its meta (*e.g.*, sequence number in TCP) and length with existing buffered blocks, in order to identify the fully and partial retransmission. Operators can decide whether the retransmitted parts should be overwritten by passing a `overwrite_rexmit` flag to `Sequence()`. Once the retransmission is detected, Rubik will automatically raise an event, which can be referenced by `event.rexmit`.

In some cases operators would disable the sequence buffering. For example, a TCP stateful firewall relies on the meta of sequence block to track the TCP states, but does not need the content in the block. Operators can disable the content buffering by simply writing `tcp.buffer_data=False`.

Transport protocols use window to control the transmitting rate. Operators can pass a `window=(wnd,wnd_size)` parameter to `Sequence()`, which specifies the valid range of meta. The out-of-window packets will not be inserted into the sequence, but raise an `out_of_wnd` event.



(a) The end host stack PSM for TCP handshake.



(b) The sender-side PSM for TCP handshake.

Figure 4: The middlebox stack PSM (b) only models the sending behaviors of the end host stack PSM (a).

4.3 Virtual Ordered Packets

The sequence abstraction will sort the out-of-order packets, which, however, still mess around the stack processing. Consider the IP PSM shown in Figure 3. In the possible out-of-order cases, the "last frag" packet can arrive earlier than a "more frag" packet. With the transitions defined in Figure 2, such packet will trigger `ip.psm.last` and form an incomplete SDU for the next layer. To avoid such mistake, the transition has to track two states (instead of the fragmentation flag only), *i.e.*, whether the "last frag" packet *has* arrived and whether the sequence *is* continuous. This counter-intuitive expression makes `Pred` in PSM transitions quite complex.

Rubik addresses such problem by offering an abstraction of *virtual ordered packets*, which gives an illusion to the operators that they are accessing the ordered packets. For example, to handle the early-arrived "last frag" exception, the transition `ip.psm.last` can be rewritten as follows.

```
ip.psm.last = (FRAG >> DUMP) + Pred(~ip.v.header.more_frag)
```

where `ip.v` indicates the virtual ordered packet. The compiler of Rubik will take care of tracking the real arriving order and ensuring the sequence continuity (see §5.4).

Note that the virtual ordered packets are for facilitating the inconsistent condition checking, while no real packet will be buffered and re-accessed. In other words, operators can only use this abstraction in the conditions of `If()` or `Pred()`.

4.4 Sender-Side PSM

Directly emulating the PSM of the end host stack in the middlebox is not a trivial task. Consider a simplified PSM for TCP handshake, whose end host version is shown in Figure 4a. Each transition in the PSM is triggered by two packets: the received packet in the white frame and the sent packet in the gray frame. For example, the passive host (*i.e.*, server) can jump into `SYN_RCVD` state only after it received the SYN packet *and* sent the SYN+ACK packet. This transition is natural for the end hosts, since the receiving and sending behaviors are synchronized.

However, the middlebox cannot capture those two behaviors at the same time. Instead, it has to use two states to respectively capture them. For example, for the passive side, the PSM of a middlebox will jump to a new state, say `SYN_HALF_RCVD`, when processing an SYN packet sent from the client, and will further jump to `SYN_RCVD` only after it sees an SYN+ACK packet sent reversely. That is, the middlebox stack has to maintain two PSMs for two sides, each of which introduces many more states and transitions.

Rubik proposes a new PSM abstraction to reduce the number of states and transitions, *i.e.*, the sender-side PSM, which combines the two-side behaviors and is triggered by a single packet. Figure 4b shows the sender-side PSM of TCP handshake, which consists of only three transitions. The key of this PSM is that it proceeds only by the sent packets (yellow ones), but ignores whether they have been received (white ones). The following defines the first transition in Figure 4b.

```
tcp.psm.syn = (CLOSED >> SYN_SENT) +
                    Pred(tcp.v.header.syn & tcp.to_passive)
```

where `to_passive` indicates the packet is being sent to the passive side in a connection-oriented session.

Note that the sender-side PSM is not a unidirectional PSM. Instead, it tracks *all* the bi-directional packets, but removes the redundancy in the end-host PSMs. For example, in Figure 4a, $\boxed{\text{SYN}}$ is the same packet with $\boxed{\text{SYN}}$. In fact, the sender-side PSM assumes the sent packets must be received. This is reasonable, because the stack cannot detect the packets lost downstream the middlebox. In practice, the middlebox stack will eventually be in the correct state after seeing the retransmitted packets, and before that, a retransmission event will alert that the current state may be inconsistent.

4.5 Event Ordering

By default, all the events will be raised after proceeding PSM and before parsing the next layer (see §5.2). However, operators have to further clarify two kinds of relationships between the events to avoid the potential ambiguity.

First, operators may need to define the "happen-before" relationships of two events, if they have the same or overlapped raising conditions. Consider the aforementioned two events in IP layer, `ip.event.asm` and `ip.event.ip_frag`, both of which will be raised when `ip.psm.last` is triggered. As a result, `ip_frag` might lose the last fragment if `asm` happens first, since the reassemble operation will clear the sequence.

Second, operators may want to raise an event if the other is happening, *i.e.*, the "happen-with" relationship. For example in TCP, an event `rdata` that poses the retransmitted data should be raised only when the retransmission event occurs.

Rubik offers an event relationship abstraction to address the above requirements. The following code indicates that `ip_frag` should happen before `asm`, and `rdata` will be checked and raised each time `rexmit` is happening.

```
ip.event_relation  += ip.event.ip_frag, ip.event.asm
tcp.event_relation += tcp.event.rexmit >> tcp.event.rdata
```

5 Compiling Rubik Programs

In this section, we introduce the compiler of Rubik, which translates the Rubik program into native C code. We first reveal the difficulties of handling the performance issues in the middlebox stack (§5.1). To this end, we translate the Rubik program into an intermediate representation (IR) to reveal the factual control flow of the stack (§5.2). Then the middle-end of the compiler performs domain-specific optimizations on the IR to avoid the performance traps (§5.3), and finally the back-end translates the IR into performant C code (§5.4).

5.1 Avoiding Performance Issues is Hard

As mentioned in §2.3, the generalized execution model can cause severe performance issues. For example, the simple pipeline will insert every IP packet into the sequence, while this is redundant for the non-fragmented IP packets, as their blocks will be assembled right after being inserted. And since the normal IP packets dominate the traffic, this redundant copy will heavily degrade the stack performance.

Previously in hand-written stacks, developers handle each of those performance issues using native code. However, due to the function diversity in the stack, identifying and avoiding *all* such traps for *all* stacks is too harsh for the developers. Moreover, even the developers are aware of those traps, sometimes they have to trade off performance for the code modularity or generality, since the proper handling of those issues will heavily increase the size of the codebase, making the program more bug-prone.

As a DSL, Rubik has better chance to address those performance issues, if it can capture and handle them through its automatic compilation process. This task, however, is still challenging. First, Rubik is designed as a declarative language, which means although the developers can easily write a "correct" program without caring about the inner logic, they also can do little for providing more hints for a "better" program. Second, it is also an impossible mission for the native code compiler due to the lack of domain-specific knowledge, *e.g.*, the fact that the aligned IP packets can be directly passed cannot be obtained from the view of the native compiler.

5.2 Intermediate Representation in Rubik

Rubik addresses above challenges by introducing an IR into the compilation, which brings the following merits. First, the IR code is much smaller, making it possible to do effective optimizations that are unaffordable in native code (see §5.3). Second, the IR code still holds the high-level intent to perform the domain-specific optimizations. Third, the IR layer is a common ground for all Rubik programs, which means the optimizations applied on IR work for *all* stacks.

Specifically, we adopt the Control-Flow Graph (CFG) as the IR, which can clearly reveal the control flow of the stack.

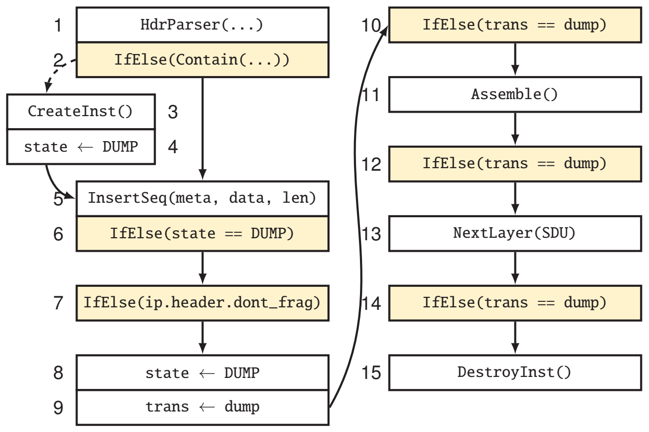


Figure 5: Partial CFG of the IP program. The yellow boxes are the branching blocks. Only one PSM transition (`ip.psm.dump`) is shown. Most of the `false` branches (dashed edge) are also omitted.

Note that each protocol layer works independently in the stack, so we view a protocol parser along with the events defined in its layer as an individual Rubik program.

Composing the control flow. The compiler composes the real control flow of the stack with the next four parts.

First, the compiler constructs the CFG for the protocol parser following the pipeline depicted in Figure 1, *i.e.*, header parsing, instance table, packet sequence and PSM transition, and elaborates it with more information that is relevant to the optimization, *e.g.*, the operations on the instance table and the sequence. The conditional statements, *e.g.*, PSM transitions and event callback, will be translated into the branching blocks with their `Pred`/`If` as the branching conditions.

Second, the compiler decides when to raise the events. It is possible to raise an event just after all its conditions have been met, *i.e.*, the triggering conditions are satisfied and the data in the callback structures are ready. As such, an event that only requires header information can be raised just after the header parser. However, the high-level functions hooking this event may modify the packets, which could impact the correct execution of the PSM. Hence, the compiler puts all events after proceeding the PSM, and decides their order by the explicitly defined happen-before and happen-with relationship in `event_relation`. The only exceptions are the built-in sequence events, *i.e.*, `rexmit` and `out_of_wnd`, which will be triggered by sequence operations before proceeding the PSM, as well as the events happening with them.

In the third and fourth parts, the compiler checks the conditions for parsing the next layer and destroys the instance if it jumps into the accepted PSM states.

Figure 5 shows a partial CFG for the IP program presented in §3. The white boxes are the basic blocks, and the yellow ones are branching blocks, where the solid/dashed edges indicate the `true`/`false` branches. In these blocks, IR uses instructions to reveal the operations on the real data structure. For example, we use `CreateInst()` to create and insert an

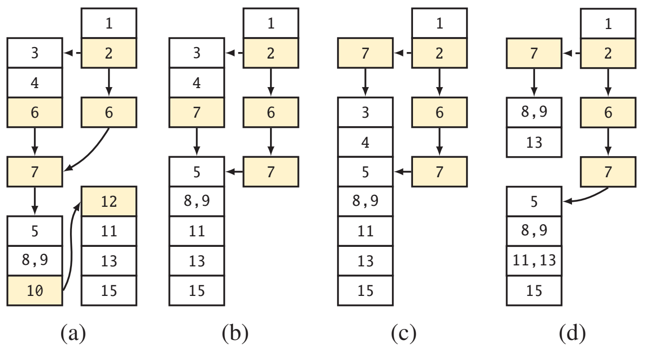


Figure 6: (a) First-round branch lifting, where $\boxed{6}$ is bounded by $\boxed{2}$ and $\boxed{4}$. (b) Constant analysis, where $\boxed{6}$, $\boxed{10}$ and $\boxed{12}$ are eliminated because they are always `true` guarded by $\boxed{4}$ and $\boxed{9}$. (c) Second-round branch lifting, where $\boxed{7}$ is further lifted above $\boxed{3}$, due to the absence of $\boxed{6}$. (d) Peephole optimization, where $\boxed{3}$-$\boxed{5}$, $\boxed{11}$ and $\boxed{15}$ are eliminated.

instance into the instance table, and `InsertSeq()` to insert the payload into its sequence. Blocks $\boxed{1}$–$\boxed{9}$ are for the IP protocol parser (only one PSM transition, *i.e.*, `ip.psm.dump`, is shown); blocks $\boxed{10}$ and $\boxed{11}$ are for the sequence assemble event; blocks $\boxed{12}$ and $\boxed{13}$ are for the next layer parsing, and blocks $\boxed{14}$ and $\boxed{15}$ handle the accept PSM state.

Revealing the dependency. Given the real control flow with CFG, the compiler can then reveal the dependency relationships between each instruction. This is achieved by checking the read/write operations in the instructions. For example, the instructions below a branching block can only be executed after reading the objects in that branching conditions. Hence, these instructions all depend on those objects.

We note that some read/write relationships are not explicit in the CFG. For example, `InsertSeq()` writes the sequence in current instance, and `Assemble()` reads the same sequence. Hence, `Assemble()` depends on `InsertSeq()`. We pre-define the implicit dependencies for all instructions.

5.3 Middle-End: Optimizing Control Flow

The middle-end first transforms the CFG to expose the complete processing logic on a same set of packets. Then, it applies domain-specific optimizations targeting on the "heavy" instructions to produce an optimal CFG. Specifically, the middle-end iteratively takes the following three steps until the CFG converges to a stable form.

Step 1: Lifting the branches. We lift all the branching blocks to the top of the CFG, as long as they do not depend on an upper block. Figure 6a shows the CFG with branching blocks lifted. We take $\boxed{6}$ as an example: it is lifted to top of the `true` branch of $\boxed{2}$, because in this branch, it only depends on the conditions in $\boxed{2}$; in contrast, in the `false` branch it can only be lifted below block $\boxed{4}$, because it reads the variable `state`, which is written by block $\boxed{4}$. Through this process, $\boxed{6}$ is duplicated, as both branches should traverse it. Note that some `false` branches are omitted in Figure 6a, *e.g.*, the `false` branches of $\boxed{6}$ and $\boxed{7}$ that contain the duplicated $\boxed{5}$.

The branch lifting process merges the basic blocks, which helps to expose a complete processing logic on an individual set of packets. This process maps to the "code sinking" transformation in conventional compilers, which however usually is not performed, since the codes would explode due to the duplication. In contrast, Rubik's IR code is small and with a neat pipeline, making this expensive transformation affordable.

Step 2: Constant analysis. This step replaces or removes the instructions if they are evaluated to be a constant. For example, consider $\boxed{6}$ in the `false` branch in Figure 6a. We can easily assert that its condition (`state==DUMP`) is always `true`, because $\boxed{4}$ has just assigned `state` with `DUMP`. Similar analysis takes place in block $\boxed{9}$, $\boxed{10}$, $\boxed{12}$, where `trans==dump` in the latter two must be `true`. Those always-`true` branch blocks can be removed, as shown in Figure 6b. Note that this elimination may create new opportunities to iteratively lift the branch, *i.e.*, Step 1. For example, $\boxed{7}$ can be further lifted above $\boxed{3}$, due to the absence of $\boxed{6}$, as shown in Figure 6c.

Step 3: Peephole optimizations. After the first two steps, the complete processing logic on each packet set is revealed. For example, $\boxed{3}$–$\boxed{15}$ in Figure 6c illustrates the processing logic on the first packet of an IP instance which is with `dont_frag` flag. In this step, we engage a series of peephole optimizations [36] to identify and eliminate performance traps.

Considering $\boxed{3}$–$\boxed{15}$, we have the following easy optimizations. (1) For $\boxed{3}$, $\boxed{5}$, $\boxed{11}$, $\boxed{13}$, it is obvious that the first and only inserted block is directly assembled. Hence, $\boxed{5}$ and $\boxed{11}$ can be eliminated, and $\boxed{13}$ can be rewritten into `NextLayer(Payload)`. (2) $\boxed{3}$ and $\boxed{15}$ is another pair of redundant operations, where the inserted instance is directly removed from the instance table. These two blocks can also be eliminated. Figure 6d shows the CFG that removes all redundant operations, most of which are very expensive, *e.g.*, instance creation and sequence insertion. We can therefore expect a much higher performance with this optimized CFG.

We emphasize that the patterns of the optimizations are not newly designed, but the common wisdoms borrowed from the mature stack implementations. The key is that manually realizing those optimizations for each stack would mess around the processing pipeline, and significantly increase the complexity of the code. In contrast, Rubik's middle-end are stack- and implementation-oblivious, *i.e.*, operators can focus on the logic of the optimizations without caring about how to integrate them with the stack logic. That is, the new patterns can be easily extended in the future, and the developers can obtain a fully optimized pipeline for all stacks. Appendix D shows the peephole optimizations that are currently employed.

5.4 Back-End: Producing Efficient Code

The back-end of compiler translates and assembles the optimized CFGs into native C code, and ensures its efficiency in two ways: (1) maximize the code efficiency without considering the code readability, *e.g.*, composing a single large

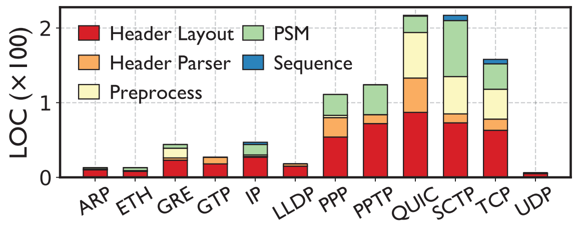


Figure 7: LOC breakdown of protocol parsers.

function for the whole stack to force the optimization in the C compiler; (2) borrow the best practice from existing middlebox stacks, *e.g.*, a fast hashing library. Except for the above general methods, we highlight two designs in the back-end.

Handling the header. The back-end translates the header fields and their references using the following principles.

- Each `layout` will be translated into a C `struct`, and the header parser is a `struct` pointer to the starting address of the header, so that each field can be directly accessed as a `struct` member. For the composite headers, *e.g.*, `ip.header=ip_hdr+ip_opt`, the back-end will generate multiple pointers pointing to different locations.
- Conditions and predicates of virtual ordered packets, *e.g.*, `If(ip.v.header.more_frag)`, will be implemented as tracking two states, *i.e.*, `if(seen_frag && no_hole)`, where `seen_frag` will be set if "more_frag" packet has arrived, and `no_hole` is assigned by checking the sequence each time a sequence block is inserted.

Threading model. We adopt the shared-nothing model with the run-to-complete workflow when generating native code. That is, each core runs an independent stack, which eliminates the inter-core communication [46]. Specifically, the back-end leverages the symmetric receive-side scaling (S-RSS) technique [71], so bi-directional packets from the same connection can be correlated to the same thread. Since modern NICs support hardware-based S-RSS, this usually linearly boosts the stack performance with the number of cores (see §7.1).

The back-end also takes care of other cases that require considerable human effort, *e.g.*, buffer outrun and timeout.

6 Rubik in Action

Rubik builds upon Python while offering domain-specific syntaxes and functions. In total, our prototype amounts to 3K Python LOC for Rubik internals, and 2K C LOC for hashing, packet I/O and sequence operations. The source code of Rubik is available at `https://github.com/ants-xjtu/rubik`.

In this section, we demonstrate the practicality of Rubik by implementing numbers of mainstream L2-L4 protocols and stacks (§6.1), and developing typical high-level middlebox functions (§6.2).

6.1 Collected Protocols and Stacks

We collect and implement 12 L2-L4 protocol parsers using Rubik. Here we focus on how many LOC used for the implementation, which reflects the complexity and robustness of the program. From the LOC breakdown shown in Figure 7, we have the following observations.

Table 2: Rubik and generated LOC for composing stacks.

Stack	Parse Tree	Addi.	Total	Gen.
TCP/IP	ETH→IP→UDP/TCP	14	245	11061
GTP	ETH→IP→UDP→GTP→IP→TCP	18	304	11384
PPTP	ETH→IP→TCP→PPTP ETH→IP→GRE→PPP→IP→TCP	37	586	46546
QUIC	loopback→IP→UDP→QUIC	23	361	14007
SCTP	ETH→IP→SCTP	9	233	23863

Addi.: *additional Rubik LOC apart from the individual protocol parsers*
Total: *total Rubik LOC* **Gen.:** *generated native LOC*

First, Rubik can express the mainstream L2-L4 protocols with minor LOC. Most connectionless protocols only take tens of LOC. The connection-oriented ones take more, but within hundreds of LOC. Second, most LOC are for defining the header layout (46% in average), since one field takes one LOC in Rubik. This task is quite straightforward if given the protocol specification, so the factual effort of writing a protocol parser is even less than it appears in the figure.

Reducing the effort of implementing above parsers is very valuable. For example, the stream control transmission protocol (SCTP) [2] provides many useful transmission features like message boundary preservation and multi-homing. However, this requires a significant change for middleboxes, *e.g.*, 4400 C LOC in Wireshark [3], making SCTP much less deployed [41]. Using Rubik, it only takes 210 LOC for implementing the SCTP layer. Another example is QUIC [6]: although its multiplexing feature improves the transmission efficiency, existing middleboxes cannot support it without a fundamental upgrade. As a reference, Wireshark takes ∼3100 C LOC to realize the QUIC protocol parser [26]. Using Rubik, merely 216 LOC is enough for prototyping a QUIC parser (without the decryption feature, see §8).

We finally implement 5 typical stacks, as shown in Table 2. We highlight that with the reusable protocol parsers, composing a middlebox stack requires minor additional Rubik LOC, although the native LOC generated is massive.

6.2 Developing Applications with Rubik

`Callback()` in defining BIEs will generate empty callback functions in the native code, which will be invoked each time the BIEs are triggered. The programmers can then develop their applications by implementing those functions.

The most typical example can be a DPI application that inspects the L7 data. In this case, the developer can pose an event happening with the assemble event (`tcp.event.asm`).

```
tcp.event.sdu = Assign(sdu_layout.sdu, tcp.sdu) + \
                Callback(sdu_layout)
tcp.event_relation += tcp.event.asm >> tcp.event.sdu
```

Other examples include the detection of SYN-flood and fake-reset, which can be implemented through the BIEs triggered by the first SYN packet and the reset transitions, respectively.

We present how we port Snort [21] as a more comprehensive example. Snort may scan the traffic multiple times against the rules, *e.g.*, on the fragmented IP packets or on the reassembled L7 data. With Rubik, the programmers can implement these scanning behaviors through the corresponding BIEs posed in the stack. Specifically, we implement 25 rule options, *e.g.*, `content`, `pcre`, `http_header`, and translate the rules into event-based callback functions. We replace Snort's `stream` and `http-inspect` modules with Rubik-generated stacks and events, and reuse the high-level matching modules like Aho-Corasick algorithm for string matching and HyperScan [69] for regular expression matching.

Note that, unlike mOS and Microboxes, Rubik currently does not support programming UDEs. As a result, for HTTP-related rules, we need to manually parse the L7 protocols in the callback function (instead of using a set of inherited UDEs), then the L7 rules can be matched against those parsed HTTP headers. §8 discusses the UDE programming in detail.

7 Evaluation

In this section, we evaluate the performance of Rubik. Specifically, our experiments aim to answer the following questions:

(1) Do Rubik-generated stacks provide comparable or even better performance than the hand-written stacks? (§7.1)

(2) Do Rubik-ported applications work correctly and efficiently on various stacks? (§7.2)

(3) Do the middle-end optimizations help improve the performance of Rubik? (§7.3)

7.1 Microbenchmarks

To measure Rubik's performance under certain traffic load, we build real end-host applications and set a bump-in-the-wire testbed as the middlebox stack. Due to the lack of high-performance non-TCP applications (*e.g.*, QUIC, SCTP), the microbenchmarks are mostly about the TCP/IP stack.

Experimental settings. We build the testbed on an x86 machine (20$\times$Intel Xeon 2.2Ghz, 192GB memory) with three dual-port 40G NICs (Intel XL710). We use another six machines (8$\times$Intel Xeon 2.2Ghz, 16GB memory) to build three server/client pairs. Each server/client has a single-port 40G NIC, and is connected through one NIC in the testbed server.

The clients and servers generate 96K concurrent connections in total (32K from each pair). Each connection fetches a file from the server (1KB by default), and will immediately restart when it terminates. Note that the three pairs cannot drain the 120Gbps link, so we indicate the upper bounds of the throughput for each setting in the experiments, *i.e.*, the throughputs when directly wiring the clients up to the servers.

We synthesize three high-level functions to simulate different workloads of the middlebox: (1) a flow tracker (FT) that tracks the L4 states but ignores the payload, (2) a data assembler (DA) that dumps bidirectional L7 data to `/dev/null`, and

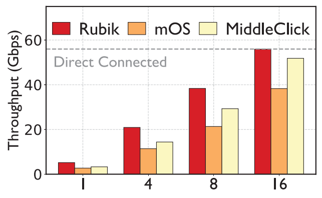


Figure 8: The multi-core scalability (DA, 1KB file).

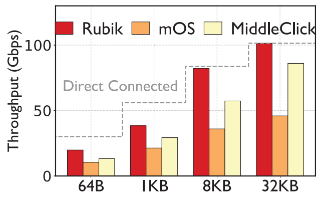


Figure 9: The file size scalability (DA, 8 cores).

(3) a string finder (SF) that matches 50 regular expressions against L7 data. We run DA as the default function in the experiments, as it reflects the intrinsic performance of a complete middlebox stack, *i.e.*, with bidirectional data reassembly and without heavy operations on that data. When running FT, we disable the data buffering for all involved approaches.

TCP stack. From the existing approaches shown in Table 1, we choose to compare Rubik with the TCP-specific stack and the NFV framework, since the packet parser cannot implement a full-functional stack. Specifically, we involve mOS [14] and MiddleClick [28] in our experiments. The former is the state-of-the-art TCP middlebox stack, and the latter with Click model is reported to be more efficient. All approaches have the same packet I/O capability with DPDK [8] and S-RSS. The clients and servers are implemented using mTCP [45].

Figure 8 shows the multi-core scalability of the involved approaches. Thanks to S-RSS, the performance of all approaches can almost linearly scale with the number of CPU cores. Note that Rubik's TCP stack achieves 5.2Gbps, 20.9Gbps, 38.4Gbps when using 1, 4, 8 cores, respectively, and can reach the upper bound (55.9Gbps) with 16 cores. Such throughput outperforms other approaches by 30%–90%.

Figure 9 shows the scalability with different file sizes. With 8 cores, Rubik's TCP stack can reach the upper bound with the file larger than 8KB (82.1Gbps in 8KB, 101.4Gbps in 32KB). We also report that Rubik can reach the upper bound for all file sizes if using 16 cores. Note that given the flow size (which can be inferred from the file size) and the throughput, we can estimate the connection arrival rates, *i.e.*, how many new connections can be handled per second. We report that with 8 cores, the connection arrival rates of Rubik's TCP stack are 4.5M/s and 1.1M/s for 64B and 8KB files, respectively.

Figure 10 shows the throughput with different functions. Rubik's stack can realize 44.1Gbps and 25.5Gbps for FT and SF with 8 cores, which maintain the lead to other approaches (34% and 90% faster than MiddleClick and mOS). We also report that Rubik's stack adds reasonable transferring latency (not shown in the figure), *e.g.*, running DA for a single flow adds 29μs and 97μs to the flow completion time when transferring 64B and 8KB files (62μs and 119μs without middlebox).

Why Rubik outperforms other approaches. First, the peephole optimizations applied in the middle-end let Rubik handles each packet class in the most efficient way, which guarantees a comparable performance to the mature hand-written ones. Second, the hand-written stacks have to trade off perfor-

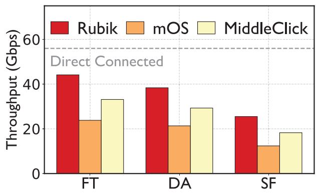


Figure 10: Throughput vs. functions (1KB file, 8 cores).

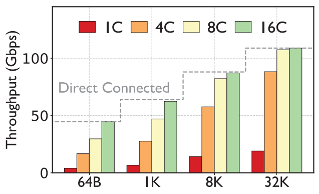


Figure 11: Performance of the GTP stack (DA).

mance for the code maintainability. For example, for maintaining the 8K C LOC of TCP stack, mOS spans more than 100 non-inline functions, dozens of which would be invoked for processing each packet. In contrast, Rubik puts 11K C LOC in a single function for composing the same stack, which incurs much fewer function calls, hence the higher throughput. Third, the one-big-function also forces the optimizations of native C compiler. Specifically, we re-compile MiddleClick and Rubik's generated code with `-O0` instead of `-O3`, and observe that MiddleClick's performance downgrades by 40%, while Rubik suffers 50% degradation. This partially confirms that deeper optimizations can be applied in the one-big-function.

We emphasize that both mOS and MiddleClick are with high-quality code. Even though, the performance trade-offs for maintainability are still inevitable when handling so many LOC with human oracle. Such risk would only be higher for more complex stacks. In contrast, Rubik avoids the performance traps for *all* stacks by automatically applying the domain-specific optimizations in its middle-end, while ensuring the maintainability with its neat syntaxes in the front-end.

GTP stack. Besides the TCP stack, we also run the GTP stack in end hosts for evaluation. Specifically, we modify mTCP to encapsulate/strip IP, UDP, and GTP layers for each TCP packet, where the new IP and UDP layers have the same IP addresses and ports with the original TCP packet. Note that this operation adds ~50 bytes to each packet, which will lift the throughput upper bound when transferring small files.

Figure 11 shows the performance of Rubik's GTP stack with different CPU cores and file sizes. With only one core, Rubik's GTP stack can realize 4.0Gbps and 14.2Gbps for 64B and 8KB file, respectively, and can reach their performance upper bounds (44.6Gbps and 88.0Gbps) with 16 cores. We highlight that even the GTP stack has much more layers than the TCP stack, Rubik can still catch up with the throughput, as its back-end introduces minor overhead between each layer.

7.2 Performance on Various Stacks

We collect real and synthetic traces to evaluate the performance of Rubik on various stacks with real applications.

Traces. We prepare traces for five stacks, as shown in Table 3. The TCP trace is captured in a campus network. The GTP trace is captured in an ISP's base station. For PPTP stack, we set a PPTP server with MPPE and PPP compression disabled, and capture the trace by accessing random websites. For QUIC stack, we set a pair of client and server using `ngtcp2` [17], and capture the trace by querying random resources. Rubik currently does not support the online decryption, so we replace the encrypted data in the trace with a deciphered one using the local SSL key (see §8). For SCTP stack, we set the server and client using `usrsctp` [29], and capture the trace by fetching random files. We filter out the incomplete connections (flows without handshake or teardown) for all the traces, so we can properly replay them on a loop.

Table 3: The collected traces.

Trace	#Pkts.	#Flows	Avg. Flow Size	Avg. Pkt. Length	L7 Data Size	Total Size
TCP	25.9M	558K	32KB	652.13B	8.4GB	16.8GB
GTP	18M	630K	14KB	484.03B	0.6GB	8.7GB
PPTP	6.7M	9K	665KB	892.98B	4.2GB	6.0GB
QUIC	12.7M	3K	637KB	643.25B	5.4GB	8.1GB
SCTP	7.6M	600K	5KB	374.35B	2.3GB	2.9GB

Table 4: The throughput (Gbps) on the traces (16 cores).

	Snort	Rubik+Snort	nDPI	Rubik+nDPI	DA
TCP	20.41	26.86	25.94	25.26	117.76
GTP	15.36	22.79	18.87	18.37	113.42
PPTP	13.91	20.01	18.79	18.22	118.41
QUIC	-	-	-	-	116.29
SCTP	-	-	-	-	101.27

Applications. We port two well-known middlebox applications, Snort [21] and nDPIReader [15], to Rubik. For Snort, we port it as presented in §6.2, and load 2800 TCP- or HTTP-related rules from its community rule set. We note that nDPIReader does not implement a complete flow reassembly feature. To this end, we pose the TCP and UDP assemble events, and invoke the core detecting functions provided by nDPI in the callback functions. The Rubik-ported version can then detect protocols on reassembled data.

We equip the original Snort and nDPI with DPDK/S-RSS and involve them into the comparison. Note that neither of original and Rubik-ported versions can inspect QUIC or SCTP trace, because the rules and detecting applications are TCP-specific, *i.e.*, they assume the transport layer must be TCP/UDP. However, we argue that with the help of Rubik, it would be quite simple to port such rules to new stacks, *e.g.*, inspecting the HTTP content carried by an SCTP connection.

Performance. We split the traces into three pieces by their IP addresses and use three machines to inject them into the testbed (120Gbps line rate). The testbed runs on 16 cores.

Table 4 shows the throughput with different applications, from which we have two observations. First, the Rubik-ported Snort is faster than the original and the boost is more significant on the stacks with more layers (+31.6% for TCP vs. +48.3% for GTP), because "heavy" stacks would amplify the efficiency of Rubik's generated one-big-function. Second, the Rubik-ported nDPI is slightly slower than the original (−2.8% in average), because the latter does not reassemble the data at all. Specifically for TCP stack, the Rubik-ported versions perform better than the mOS-ported versions (+31.6% vs.

Table 5: Throughput boost (Gbps) and compilation slow down (seconds) from the middle-end optimizations (1 core, DA). Shadowed cells show the number with optimizations.

	TCP		GTP		PPTP		QUIC		SCTP	
Throughput	8.83	22.4	5.47	14.4	8.16	18.9	7.03	13.7	4.38	6.61
Rubik→C	0.06	0.28	0.07	0.31	0.10	1.19	0.07	0.42	0.06	1.22
C→Binary	3.32	3.47	3.47	5.46	3.73	13.1	3.47	4.02	3.27	6.86

+16.8% for Snort, −2.6% vs.−3.7% for nDPI) [44]. We finally highlight that when running DA, all Rubik's stacks can achieve more than 100Gbps throughput for their traces.

Correctness. We respectively select 100 flows from all traces. By manually verifying the results of protocol parsers and event callbacks, we confirm the correctness of Rubik.

7.3 Middle-End Optimizations

We inject the same traces used in §7.2 into the testbed and measure the performance and overhead of corresponding stacks, by enabling/disabling the middle-end optimizations.

Performance. The top part of Table 5 shows that all stacks can significantly benefit from the optimizations, with a boost rate of 51%–163%. The effect of the optimizations depends on two factors. First, since each layer is independently optimized, more layers lead to more improvements. Second, the major optimization is the elimination of the sequence operations, so more boosts can be gained when handling large flows. For example, PPTP and QUIC traces have similar flow size, but the PPTP stack with more layers gains more from the optimizations; TCP and SCTP stacks have the same number of layers, but the SCTP stack does not boost as much as TCP due to the smaller flow size of the trace (5KB vs. 32KB).

Overhead. The branch lifting in the middle-end leads to much larger CFG, which increases the time of compilation, *i.e.*, from Rubik program to C code, and from C code to binary. The bottom part of Table 5 shows that such overhead is minor in practice, *i.e.*, all stacks are compiled within 15 seconds.

8 Limitations and Discussion

Semantics completeness. There are generally two types of middleboxes: the flow-monitoring ones that parse the protocols, check the reassembled data, and forward/drop the original packets (*e.g.*, IDS), and the flow-modifying ones that intercept the connection and modify L7 content (*e.g.*, HTTP proxy). To the best of our knowledge, Rubik can well support programming the former type. For the latter, Rubik should extend its sequence abstraction for inline-reordering, and event abstraction for modifying the packet content. These extensions are realizable and will be explored in our future work.

Encrypted layers. Rubik can cooperate with the encrypted layers in the following two ways: (1) the stacks can directly work on the raw packets, if the middleboxes are placed inside the secure district, where the encrypted content has already been resolved by the gateway; (2) the stacks can inspect the encrypted content, given the proper decipher keys. We simulate the first scenario with QUIC protocol in our evaluation. For the second, we can offer an extra decryption function to modify SDU. We leave this feature to our future work.

UDE programming. Prior to Rubik, literatures focus on how to facilitate the middlebox development by offering friendly UDE interfaces. mOS [44] unifies the TCP stack and provides a BSD-style socket interface, so the developers can dynamically register/deregister their UDEs. Microboxes [53] further optimizes the UDE model by parallelizing their executions, making the performance scalable with the number of network functions. Since the UDE programming is oblivious from the stack programming, we believe providing such feature should be an easy task for Rubik's back-end, by borrowing the best practice from mOS and Microboxes.

Boosting S-RSS. We discuss two possible techniques that can further boost the packet dispatching. First, unlike S-RSS that dispatches correlated packets to fixed CPU cores, RSS++ [31] and eRSS [64] can collect the flow statistics and dynamically dispatch packets to different cores, which can further balance the CPU load. Second, the hardware-based S-RSS can only classify fixed protocols. For example, for PPTP stack, it only dispatches the packets by the lower-layer IP addresses, which are almost fixed as a tunnel. The dispatching can be much more balanced if higher-layer IP and TCP protocols can be considered. A smart NIC [54] or a programmable switch associated with the middlebox [46] that can dispatch the packets by arbitrary headers could be a cure to this problem.

Middlebox deployments. While Rubik facilitates the development of a single middlebox, there are literatures that deploy middleboxes in a distributed way [62, 65] or as a cloud service [66]. These works can be complementary to Rubik.

Ethics statement. The TCP and GTP traces used in the experiments are anonymized before given to us. We claim that our work does not raise any ethics issue.

9 Conclusion

This paper proposed Rubik, a language for programming the middlebox stack, which offers a set of high-level constructs as efficient building blocks, and an optimizing compiler to produce high-performance native code. We demonstrated the minor effort for implementing 12 protocol parsers and 5 popular stacks using Rubik. We evaluated Rubik with real applications and traces, and showed that the generated stacks outperform existing approaches by at least 30% in throughput.

Acknowledgments. We thank the anonymous NSDI reviewers and our shepherd Dejan Kostić for their valuable feedback. We thank Patrick P. C. Lee and Aaron Gember-Jacobson for their comments on the early version of this paper, and Tom Barbette for his help on the codebase of MiddleClick. This work is supported by NSFC (61702407, 61772412, 61902307 and 61702049), and Fundamental Research Funds for the Central Universities. Peng Zhang is the corresponding author.

References

[1] Ip in ip tunneling. https://tools.ietf.org/html/rfc1853, 1995.

[2] Stream control transmission protocol. https://tools.ietf.org/html/rfc4960, 2007.

[3] packet-sctp.c. https://github.com/boundary/wireshark/blob/master/epan/dissectors/packet-sctp.c, 2013.

[4] Do you guys block udp port 443 for your proxy/web filtering? https://bit.ly/2OBM51l, 2015.

[5] In-band network telemetry (int). https://p4.org/assets/INT-current-spec.pdf, 2016.

[6] Quic: A udp-based secure and reliable transport for http/2. https://tools.ietf.org/html/draft-ietf-quic-transport-11, 2017.

[7] Clicknf. https://github.com/nokia/ClickNF, 2018.

[8] Dpdk. http://www.dpdk.org/, 2018.

[9] IEEE 802.11 wireless local area networks. http://grouper.ieee.org/groups/802/11/, 2018.

[10] The impact on network security through encrypted protocols - quic. https://bit.ly/2xw7z8y, 2018.

[11] Libnids. http://libnids.sourceforge.net/, 2018.

[12] Middlebox cooperation protocol specification and analysis. https://mami-project.eu/wp-content/uploads/2015/10/d32.pdf, 2018.

[13] Middleboxes: Taxonomy and issues. https://tools.ietf.org/html/rfc3234, 2018.

[14] mos-networking-stack. https://github.com/ndsl-kaist/mos-networking-stack, 2018.

[15] ndpi. https://www.ntop.org/products/deep-packet-inspection/ndpi/, 2018.

[16] The new waist of the hourglass. http://tools.ietf.org/html/draft-tschofenig-hourglass-00, 2018.

[17] ngtcp2. https://github.com/ngtcp2/ngtcp2, 2018.

[18] P4 language. https://p4.org/, 2018.

[19] Quic protocol | cisco community. https://community.cisco.com/t5/switching/quic-protocol/td-p/3402269, 2018.

[20] Report from the iab workshop on stack evolution in a middlebox internet (semi). https://www.rfc-editor.org/rfc/rfc7663.txt, 2018.

[21] Snort - network intrusion detection and prevention system. https://www.snort.org/, 2018.

[22] Snort: Re: Is there a snort/libnids alternative. http://seclists.org/snort/2012/q4/396., 2018.

[23] Vector packet processing. https://fd.io/, 2018.

[24] The zeek network security monitor. https://www.zeek.org/, 2018.

[25] F-Stack. http://www.f-stack.org/, 2019.

[26] packet-quic.c. https://bit.ly/32ED3YK, 2019.

[27] Seastar. http://seastar.io/, 2019.

[28] Middleclick. https://github.com/tbarbette/fastclick/tree/middleclick, 2020.

[29] usrsctp: A portable sctp userland stack. https://github.com/sctplab/usrsctp, 2020.

[30] T. Barbette, C. Soldani, R. Gaillard, and L. Mathy. Building a chain of high-speed vnfs in no time: Invited paper. In *IEEE HPSR*, 2018.

[31] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. Rss++: Load and state-aware receive side scaling. In *ACM CoNEXT*, 2019.

[32] Apurv Bhartia, Bo Chen, Feng Wang, Derrick Pallas, Raluca Musaloiu-E, Ted Tsung-Te Lai, and Hao Ma. Measurement-based, practical techniques to improve 802.11ac performance. In *ACM IMC*, 2017.

[33] Anat Bremler-Barr, Yotam Harchol, and David Hay. Openbox: A software-defined framework for developing, deploying, and managing network functions. In *ACM SIGCOMM*, 2016.

[34] Ryan Craven, Robert Beverly, and Mark Allman. A middlebox-cooperative tcp for a non end-to-end internet. In *ACM SIGCOMM*, 2014.

[35] K. Edeline and B. Donnet. Towards a middlebox policy taxonomy: Path impairments. In *IEEE INFOCOM Workshops*, 2015.

[36] Charles N. Fischer, Ron K. Cytron, and Richard J. LeBlanc Jr. *Crafting A Compiler*. Pearson, 2009.

[37] Massimo Gallo and Rafael Laufer. Clicknf: a modular stack for custom network functions. In *USENIX ATC*, 2018.

[38] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *ACM SIGCOMM*, 2014.

[39] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *IEEE/ACM ANCS*, 2013.

[40] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. Sgx-box: Enabling visibility on encrypted traffic using a secure middlebox module. In *ACM APNet*, 2017.

[41] David A. Hayes, Jason But, and Grenville Armitage. Issues with network address translation for sctp. *ACM SIGCOMM Comput. Commun. Rev.*, 39(1):23–33, 2009.

[42] Benjamin Hesmans, Fabien Duchene, Christoph Paasch, Gregory Detal, and Olivier Bonaventure. Are tcp extensions middlebox-proof? In *ACM HotMiddlebox*, 2013.

[43] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. Is it still possible to extend tcp? In *ACM IMC*, 2011.

[44] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mos: A reusable networking stack for flow monitoring middleboxes. In *USENIX NSDI*, 2017.

[45] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level TCP stack for multicore systems. In *USENIX NSDI*, 2014.

[46] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *USENIX NSDI*, 2018.

[47] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. Paving the way for NFV: Simplifying middlebox modifications using statealyzr. In *USENIX NSDI*, 2016.

[48] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[49] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *USENIX NSDI*, 2016.

[50] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. In *ACM SIGCOMM*, 2017.

[51] Hao Li, Chengchen Hu, Junkai Hong, Xiyu Chen, and Yuming Jiang. Parsing application layer protocol with commodity hardware for sdn. In *IEEE/ACM ANCS*, 2015.

[52] Zhichun Li, Gao Xia, Hongyu Gao, Yi Tang, Yan Chen, Bin Liu, Junchen Jiang, and Yuezhou Lv. Netshield: massive semantics-based vulnerability signature matching for high-speed networks. In *ACM SIGCOMM*, 2010.

[53] Guyue Liu, Yuxin Ren, Mykola Yurchenko, K.K Ramakrishnan, and Timothy Wood. Microboxes: High performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions. In *ACM SIGCOMM*, 2018.

[54] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *ACM SIGCOMM*, 2019.

[55] Alberto Medina, Mark Allman, and Sally Floyd. Measuring the evolution of transport protocols in the internet. *ACM SIGCOMM Computer Communication Review*, 35(2):37–52, 2005.

[56] C Meiners, E Norige, A. X Liu, and E Torng. Flowsifter: A counting automata approach to layer 7 field extraction for deep flow inspection. In *IEEE INFOCOM*, 2012.

[57] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for nfv applications. In *ACM SOSP*, 2015.

[58] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of NFV. In *USENIX OSDI*, 2016.

[59] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. binpac: A yacc for writing application protocol parsers. In *ACM IMC*, 2006.

[60] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, and S. Mangiante. De-ossifying the internet transport layer: A survey and future perspectives. *IEEE Communications Surveys Tutorials*, 19(1):619–639, 2017.

[61] Sharvanath Pathak and Vivek S. Pai. Modnet: A modular approach to network stack extension. In *USENIX NSDI*, 2015.

[62] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. Simple-fying middlebox policy enforcement using sdn. *ACM SIGCOMM*, 2013.

[63] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath TCP. In *USENIX NSDI*, 2012.

[64] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. Elastic rss: Co-scheduling packets and cores using programmable nics. In *APNet*, 2019.

[65] Vyas Sekar, Sylvia Ratnasamy, Michael K. Reiter, Norbert Egi, and Guangyu Shi. The middlebox manifesto: Enabling innovation in middlebox deployment. In *ACM HotNets*, 2011.

[66] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *ACM SIGCOMM*, 2012.

[67] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *ACM SIGCOMM*, 2015.

[68] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. Resq: Enabling slos in network function virtualization. In *USENIX NSDI*, 2018.

[69] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern cpus. In *USENIX NSDI*, 2019.

[70] Zhaoguang Wang, Zhiyun Qian, Qiang Xu, Zhuoqing Mao, and Ming Zhang. An untold story of middleboxes in cellular networks. In *ACM SIGCOMM*, 2011.

[71] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. Comparison of caching strategies in modern cellular backhaul networks. In *ACM MobiSys*, 2013.

[72] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *USENIX NSDI*, 2018.

Appendix A Rubik Built-in Abstractions

	Abstractions	Initialized Syntax and Parameters	Functionality and Semantics
Packet Data	`p.header`	`layout+layout` `AnyUntil((layout), cond)`	generate a header layout
	`p.temp, p.perm`	`layout`	the permanent/temporary data bound to the instance
	`p.seq`	`Sequence(meta,data,data_len)`	the sequence indexed by *meta* and filled with *data*
	`p.payload`	initialized by `p.header`	the payload of layer *p*
	`p.sdu`	initialized by `Assemble()`	the SDU passing to the upper layer
Inst. Table	`p.selector`	`[inst_key]` `([active_key],[passive_key])`	maintain the instance table by the key
Proto. State Machine	*s*	`PSMState(start, accept)`	define a PSM state *s*
	`p.psm`	`PSM(state set)`	build the PSM with the *state set*
	`p.psm.t`	$(s_1$ `>>` $s_2)$ `+ Pred(cond)`	define a transition *t* from s_1 to s_2 with condition *cond*
	`p.to_active/passive`	initialized by `p`	sending to active/passive side (connection-oriented)
Event Callback	`p.event.e`	`Assemble()`, `Callback(lay)`	an event *e* that assembles the sequence or poses *lay*
	`p.event_relation`	`(p.event.a, p.event.b)` `p.event.c >> p.event.d`	*a* happens before *b*, while *d* happens when *c* happens
Parse Tree	`st`	`Stack()`	define a stack *st*
	`st.`*lay*	*p*	define layer *lay* with parser *p*
	`st`	`(st.`*lay*1 `>> st.`*lay*2`) + Pred(`*pred*`)`	define *lay*2 as the next layer of *lay*1 with condition *pred*
Global Parameters	`p.cursor`	initialized by `p`	the current position (in byte) through the header parsing
	`p.cur_state`	initialized by `p`	the current PSM state of the instance
	`p.timeout`	a float number	the global keep-live threshold for the instance of `p`
	`p.psm.t.timeout`	a float number	the keep-live threshold of the instance in transition *t*

Appendix B Instructions and Expressions in Rubik's CFG

Instructions	Read	Write	Description
Assign(*reg*, *expr*)	*expr*	*reg*	modify *reg* with the result of *expr*
AssignSDU(*expr*)	*expr*	`SEQUENCE`	modify sequence with the result of *expr*
CreateInst()	-	`INSTTABLE`	create and insert the instance into instance table
DestroyInst()	-	`INSTTABLE`	remove the instance from the instance table
InsertSeq(*m*,*d*,*l*)	-	`SEQUENCE`	insert a sequence block (`meta=`*m*`,data=`*d*`,len=`*l*) into the sequence
Assemble()	-	`SEQUENCE`	assemble the continuous blocks in the sequence and pop it
Call(*expr*)	*expr*	-	invoke a callback function with the data *expr*
NextLayer()	-	-	jump into the next layer according to the parse tree
Expressions	**Read**	**Write**	**Description**
HeaderContain(*f*)	`HEADER`	-	check whether the parsed header contains field *f*
Payload()	`HEADER`	-	the content besides the parsed header from the packet
SDU()	`SEQUENCE`	-	the SDU assembled from the sequence
Contain(*key*)	`INSTTABLE`	-	check whether the instance with *key* is in the instance table

Appendix C   Example Rubik Programs

C.1   IP Protocol Parser

The following code shows a complete IP protocol parser.

```python
class ip_hdr(layout):
    version = Bit(4)
    ihl = Bit(4)
    tos = Bit(8)
    tot_len = UInt(16)
    id = Bit(16)
    blank = Bit(1)
    dont_frag = Bit(1)
    more_frag = Bit(1)
    f1 = Bit(5)
    f2 = Bit(8)
    ttl = Bit(8)
    protocol = Bit(8)
    check = Bit(16)
    saddr = Bit(32)
    daddr = Bit(32)


class ip_temp(layout):
    offset = Bit(16)
    length = Bit(16)


def ip_parser():
    ip = Connectionless()

    ip.header = ip_hdr
    ip.selector = [ip.header.saddr, ip.header.daddr]

    ip.temp = ip_temp
    ip.prep = Assign(
        ip.temp.offset, ((ip.header.f1 << 8) + ip.header.f2) << 3
    ) + Assign(ip.temp.length, ip.header.tot_len - (ip.header.ihl << 2))

    ip.seq = Sequence(meta=ip.temp.offset, data=ip.payload[: ip.temp.length])

    DUMP = PSMState(start=True, accept=True)
    FRAG = PSMState()
    ip.psm = PSM(DUMP, FRAG)
    ip.psm.dump = (DUMP >> DUMP) + Pred(
        ((ip.header.dont_frag == 1) & (ip.temp.offset == 0))
        | ((ip.header.more_frag == 0) & (ip.temp.offset == 0))
    )
    ip.psm.frag = (DUMP >> FRAG) + Pred(
        (ip.header.more_frag == 1) | (ip.temp.offset != 0)
    )
    ip.psm.more = (FRAG >> FRAG) + Pred(ip.header.more_frag == 1)
    ip.psm.last = (FRAG >> DUMP) + Pred(ip.v.header.more_frag == 0)

    ip.event.asm = If(ip.psm.dump | ip.psm.last) >> Assemble()

    return ip
```

C.2   TCP Protocol Parser

The following shows the complete TCP protocol parser written in Rubik, including the header option parsing, the bi-directional buffering, the out-of-window exception handling, *etc*. We note that the code is formatted according to PEP8, which largely increases the number of LOC. Moreover, the definitions of header layout and auxiliary structures take about 40% of the LOC, which means the factual effort of writing this parser is even less than the number of LOC shows.

```
class tcp_hdr(layout):
    sport = UInt(16)
    dport = UInt(16)
    seq_num = UInt(32)
    ack_num = UInt(32)
    hdr_len = Bit(4)
    blank = Bit(4)
    cwr = Bit(1)
    ece = Bit(1)
    urg = Bit(1)
    ack = Bit(1)
    psh = Bit(1)
    rst = Bit(1)
    syn = Bit(1)
    fin = Bit(1)
    window_size = UInt(16)
    checksum = Bit(16)
    urgent_pointer = Bit(16)


class tcp_nop(layout):
    nop_type = Bit(8, const=1)

class tcp_mss(layout):
    mss_type = Bit(8, const=2)
    mss_len = Bit(8)
    mss_value = Bit(16)


class tcp_ws(layout):
    ws_type = Bit(8, const=3)
    ws_len = Bit(8)
    ws_value = Bit(8)


class tcp_SACK_permitted(layout):
    SCAK_permitted_type = Bit(8, const=4)
    SCAK_permitted_len = Bit(8)


class tcp_SACK(layout):
    SACK_type = Bit(8, const=5)
    SACK_len = Bit(8)
    SACK_value = Bit((SACK_len - 2) << 3)


class tcp_TS(layout):
    TS_type = Bit(8, const=8)
    TS_len = Bit(8)
    TS_value = Bit(32)
    TS_echo_reply = Bit(32)


class tcp_cc_new(layout):
    cc_new_type = Bit(8, const=12)
    cc_new_len = Bit(8)
    cc_new_value = Bit(32)


class tcp_eol(layout):
    eol_type = Bit(8, const=0)


class tcp_blank(layout):
    blank_type = Bit(8)
    blank_len = Bit(8)
    blank_value = Bit((blank_len - 2) << 3)
```

```
class tcp_data(layout):
    active_lwnd = Bit(32, init=0)
    passive_lwnd = Bit(32, init=0)
    active_wscale = Bit(32, init=0)
    passive_wscale = Bit(32, init=0)
    active_wsize = Bit(32, init=(1 << 32) - 1)
    passive_wsize = Bit(32, init=(1 << 32) - 1)
    fin_seq1 = Bit(32, init=0)
    fin_seq2 = Bit(32, init=0)


class tcp_temp(layout):
    wnd = Bit(32)
    wnd_size = Bit(32)
    data_len = Bit(32)


def tcp_parser(ip):
    tcp = ConnectionOriented()

    tcp.header = tcp_hdr
    tcp.header += If(tcp.cursor < tcp.header.hdr_len << 2) >> AnyUntil(
        [
            tcp_eol,
            tcp_nop,
            tcp_mss,
            tcp_ws,
            tcp_SACK_permitted,
            tcp_SACK,
            tcp_TS,
            tcp_cc_new,
            tcp_blank,
        ],
        (tcp.cursor < tcp.header.hdr_len << 2) & (tcp.payload_len != 0),
    )

    tcp.selector = (
        [ip.header.saddr, tcp.header.sport],
        [ip.header.daddr, tcp.header.dport],
    )

    tcp.perm = tcp_data
    tcp.temp = tcp_temp

    CLOSED = PSMState(start=True)
    SYN_SENT, SYN_RCV, EST, FIN_WAIT_1, CLOSE_WAIT, LAST_ACK = make_psm_state(6)
    TERMINATE = PSMState(accept=True)

    tcp.prep = Assign(tcp.temp.data_len, tcp.payload_len)
    tcp.prep = (
        If(tcp.header.syn == 1) >> Assign(tcp.temp.data_len, 1) >> Else() >> tcp.prep
    )
    tcp.prep = (
        If(tcp.header.fin == 1)
        >> Assign(tcp.temp.data_len, tcp.payload_len + 1)
        + (
            If(tcp.current_state == EST)
            >> Assign(tcp.perm.fin_seq1, tcp.header.seq_num + tcp.payload_len)
            >> Else()
            >> Assign(tcp.perm.fin_seq2, tcp.header.seq_num)
        )
        >> Else()
        >> tcp.prep
    )
    def update_wnd(oppo_lwnd, oppo_wscale, oppo_wsize, cur_lwnd, cur_wscale, cur_wsize):
        x = If(tcp.header_contain(tcp_ws)) >> Assign(oppo_wscale, tcp.header.ws_value)
        x += Assign(oppo_wsize, tcp.header.window_size)
        x += Assign(oppo_lwnd, tcp.header.ack_num)
        x += Assign(tcp.temp.wnd, cur_lwnd)
        x += Assign(tcp.temp.wnd_size, cur_wsize << cur_wscale)
        return x

    tcp.prep += If(tcp.to_active == 1) >> update_wnd(
        tcp.perm.passive_lwnd,
        tcp.perm.passive_wscale,
        tcp.perm.passive_wsize,
        tcp.perm.active_lwnd,
        tcp.perm.active_wscale,
        tcp.perm.active_wsize,
    )
    tcp.prep += If(tcp.to_passive == 1) >> update_wnd(
        tcp.perm.active_lwnd,
        tcp.perm.active_wscale,
        tcp.perm.active_wsize,
        tcp.perm.passive_lwnd,
        tcp.perm.passive_wscale,
        tcp.perm.passive_wsize,
    )

    tcp.seq = Sequence(
        meta=tcp.header.seq_num,
        zero_based=False,
        data=tcp.payload[: tcp.temp.data_len],
        data_len=tcp.temp.data_len,
        window=(tcp.temp.wnd, tcp.temp.wnd + tcp.temp.wnd_size),
    )

    tcp.psm = PSM(CLOSED, SYN_SENT, SYN_RCV, EST, FIN_WAIT_1, CLOSE_WAIT, LAST_ACK, TERMINATE)

    tcp.psm.orphan = (CLOSED >> TERMINATE) + Pred(tcp.header.syn == 0)
    tcp.psm.hs1 = (CLOSED >> SYN_SENT) + Pred(
        (tcp.header.syn == 1) & (tcp.header.ack == 0)
    )
    tcp.psm.hs2 = (SYN_SENT >> SYN_RCV) + Pred(
        (tcp.to_active == 1) & (tcp.header.syn == 1) & (tcp.header.ack == 1)
    )
    tcp.psm.hs3 = (SYN_RCV >> EST) + Pred(tcp.v.header.ack == 1)

    tcp.psm.buffering = (EST >> EST) + Pred(
        (tcp.header.fin == 0) & (tcp.header.rst == 0)
    )

    tcp.psm.wv1 = (EST >> FIN_WAIT_1) + Pred(tcp.v.header.fin == 1)
    tcp.psm.wv2 = (FIN_WAIT_1 >> CLOSE_WAIT) + Pred(
        (tcp.v.header.ack == 1) & (tcp.v.header.fin == 0)
        & (tcp.perm.fin_seq1 + 1 == tcp.v.header.ack_num)
    )
    tcp.psm.wv2_fast = (FIN_WAIT_1 >> LAST_ACK) + Pred(
        (tcp.v.header.ack == 1) & (tcp.v.header.fin == 1)
        & (tcp.perm.fin_seq1 + 1 == tcp.v.header.ack_num)
    )
    tcp.psm.wv3 = (CLOSE_WAIT >> LAST_ACK) + Pred(tcp.v.header.fin == 1)
    tcp.psm.wv4 = (LAST_ACK >> TERMINATE) + Pred(
        (tcp.v.header.ack == 1) & (tcp.perm.fin_seq2 + 1 == tcp.v.header.ack_num)
    )

    for i, state in enumerate(tcp.psm.states()):
        setattr(tcp.psm, f"rst{i}", (state >> TERMINATE) + Pred(tcp.header.rst == 1))

    tcp.event.asm = If(tcp.psm.buffering) >> Assemble()
    return tcp
```

C.3 GTP Stack

The following code builds a GTP stack (Eth→IP→UDP→GTP→IP→TCP) by composing the reusable parsers.

```
stack = Stack()
stack.eth = eth_parser()
stack.ip1 = ip_parser()
stack.udp = udp_parser()
stack.gtp = gtp_parser()
stack.ip2 = ip_parser()
stack.tcp = tcp_parser(stack.ip2)

stack += (stack.eth >> stack.ip1) + Pred(1)
stack += (stack.ip1 >> stack.udp) + Pred(
    (stack.ip1.psm.dump | stack.ip1.psm.last) & (stack.ip1.header.protocol == 17)
)
stack += (stack.udp >> stack.gtp) + Pred(1)
stack += (stack.gtp >> stack.ip2) + Pred(stack.gtp.header.MT == 255)
stack += (stack.ip2 >> stack.tcp) + Pred(
    (stack.ip2.psm.dump | stack.ip2.psm.last) & (stack.ip2.header.protocol == 6)
)
```

Appendix D Peephole Optimizations

D.1 Direct Fast Forward

Pattern: `CreateInst`→`InsertSeq`→`Assemble`

Analysis: `CreateInst` creates a new instance. As such, the sequence must be empty and `InsertSeq` will insert the first block, which will be directly ejected by `Assemble`. To this end, the insertion is redundant and the assembled data is identical to the payload of this packet. To this end, the insertion and assemble instructions can be removed, and all the reference to p.`sdu` in this code block can be replaced with p.`payload`.

Output: `CreateInst`, and references to p.`sdu` in this code block can be replaced with p.`payload`, *e.g.*, `NextLayer(SDU)` should be modified to `NextLayer(Payload)`.

D.2 Fast Forward

Pattern: `InsertSeq`→`Assemble`

Analysis: Fast Forward optimizes the assemble operations for existing instances. To be specific, we could make a fast peek to the sequence before we insert the block: if the sequence is empty and the block's meta is aligned with the sequence's window, this block can be fast forwarded, *i.e.*, passing the payload instead of SDU; otherwise the code maintains the same.

Output: `If(IsEmpty&IsAlign)`→(replace SDU with payload) →`Else`→`InsertSeq`→`Assemble`

D.3 Fast Assemble

Pattern: `InsertSeq`→`Assemble` and without any `NextLayer` and `Callback`

Analysis: The `false` branch of the Fast Forward optimization means the current sequence is not empty or the current block is not aligned with the window. We can further optimize this branch, if it has no external function call, *i.e.*, `NextLayer` and `Callback`. Specifically, if the packet sequence is implemented using linked list like `libnids`, `Assemble` that collects the continuous blocks will invoke several times of data copy. However, if there is no external function that needs the assembled data, such copy is useless and can be eliminated. On the other hand, if the packet sequence is implemented using ring buffer like mOS, the data copy is still necessary when there are holes in the sequence and memory compaction is performed. In such cases, `Assemble` instruction can be eliminated. Note that we cannot eliminate `InsertSeq`, because this block may be useful for next packets' assemble in other branches.

Output: Remove `Assemble`.

D.4 Fast Destroy

Pattern: `CreateInst`→`DestroyInst`

Analysis: If an instance is created and destroyed by the same packet, it means that such instance will not impact any permanent data, and all sequence and PSM operations are meaningless. As a result, we can eliminate such creation and deletion as well as most of the instructions between them, except `Callback` and `NextLayer`.

Output: A mostly empty instruction block except `Callback` and `NextLayer`.