Programming Network Stack for Physical Middleboxes and Virtualized Network Functions

Hao Li, Yihan Dang, Guangda Sun, Changhao Wu, Peng Zhang, Danfeng Shan, *Member, IEEE*, Tian Pan, and Chengchen Hu, *Member, IEEE*

***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.**

***Index Terms*— Middleboxes, network function virtualization (NFV), software defined networking.**

I. 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 to accommodate different networks (*e.g.*, IEEE 802.11 [2]), support new protocols (*e.g.*, QUIC [3]), realize customized functions (*e.g.*, P4 INT [4]), and capture new events (*e.g.*, IP fragmentation [5]), *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 [6]. Then, Alice needs to re-organize the parsers. (3) *Adding new functions.* 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. (4) *Building service function chain.* Finally, the high-level logic implemented by the user should be efficiently chained for serving as a service function chain, regardless of the underlying stack. To achieve this, Alice needs to abstract stack interfaces agnostic of the L2-L4 protocols and manages the high-level logics without halting the underlying stack.

The difficulty of writing new parsers. Currently, protocol parsers are written in low-level native code to ensure 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 [7]. Someone may argue that 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 [3]), 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` [5] 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 that needs 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 [8], 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.

The difficulty of chaining network functions. Over the past decade, Network Function Virtualization (NFV) has promised the operator that chaining the software net-

Manuscript received 3 April 2022; revised 30 December 2022, 18 April 2023, and 1 August 2023; accepted 2 August 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor V. Subramanian. Date of publication 6 October 2023; date of current version 18 April 2024. This work was supported by the National Natural Science Foundation of China under Grant 62172323 and Grant 62272382. The preliminary version of this paper is published in [1]. *(Corresponding author: Hao Li.)*

Hao Li, Yihan Dang, Peng Zhang, and Danfeng Shan are with the MOE Key Laboratory for Intelligent Networks and Network Security, School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: hao.li@xjtu.edu.cn).

Guangda Sun is with the Department of Computer Science, School of Computing, National University of Singapore, Singapore 119077 (e-mail: sung@comp.nus.edu.sg).

Changhao Wu is with Intel Corporation Santa Clara, CA 95054 USA (e-mail: changhao.wu@intel.com).

Tian Pan is with the State Key Laboratory of Networking and Switching Technology, BUPT, Beijing 100876, China (e-mail: pan@bupt.edu.cn).

Chengchen Hu is with NIO, Shanghai 201804, China (e-mail: huc@ieee.org).

Digital Object Identifier 10.1109/TNET.2023.3307641

TABLE I
EXISTING APPROACHES TO PROGRAM MIDDLEBOX STACK AND THEIR SUPPORT OF THE FOUR PROGRAMMING TASKS

Approaches	Proto. Parser	Stack Hierarchy	Stack Func.	Service Chain
Packet Parser (*e.g.*, P4 [9], VPP [10])	◐	◐	○	○
TCP-Specific Stacks (*e.g.*, mOS [11])	○	○	◐	○
NFV Frameworks (*e.g.*, NetBricks [12])	◐	●	◐	◐

● : *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*

work functions (NFs) could be both time- and cost-efficient. However, these heterogeneous NFs could contain redundant logic, largely because many of them have to incorporate the (same) stacks. To prevent the packets from being processed by the same stacks, Alice needs to abstract a stack interface to NFs, such that the latter can just focus on the high-level logic. The large variation in stack details makes it barely possible to define such a common interface.

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 I: the packet parsers like P4 [9] cannot efficiently buffer the packets; the NFV frameworks like NetBricks [12] and ClickNF [13] often rely on TCP-specific modules that are pre-implemented with many native LOC. We discuss these related works in detail in §II-B.

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 §II-C).

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 (§III and §IV).
- 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 (§V).
- In addition to monolithic middleboxes, we also build an NFV framework on top of Rubik, which provides event-driven stack interfaces as a service and meets several critical requirements of NFV (§VI).
- We prototype Rubik, and build various real cases on it, including 12 reusable protocol parsers, 5 network stacks, and 2 open-source middleboxes (§VII). Experiments show that Rubik is at least 30% faster than the state of the art (§VIII).

II. MOTIVATION AND CHALLENGES

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

A. Programming Middlebox Stack Matters

As presented in §I, 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 [14], [15], [16], [17], [18], [19], [20]. 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 [21], [22].

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 [23], `libnids` will view this as a broken flow and directly drop it for higher performance [5], while mOS will keep the flow and offer an interface to access the fragmented sequence for maximumly collecting the data [11].

Third, middleboxes are constantly evolving for providing value-added functions, *e.g.*, adding a new layer [24], [25], measuring performance [26], inspecting encrypted data [27], [28], [29] and migrating/accelerating NFV [30], [31], [32], [33], [34]. 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 [35]. This heavily impacts the user experience [36], [37], and will finally result in the ossification of underlying networks [38].

B. Related Work

Many attempts have been made to facilitate the middlebox development, as shown in Table I. In the following, we show why they are insufficient to program the middlebox stack.

Programmable packet parsers like P4 [9] and VPP [10] can dissect arbitrarily-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 [39], Ultrapac [40], FlowSifter [41] and COPY [42], 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 [43], Modnet [44], Seastar [45], F-Stack [46], only manage the unidirectional protocol state for a certain end host, while middlebox stacks must monitor the bidirectional behaviors of both sides. Although it is possible to adapt host-end stacks for middlebox development, it is not practical, because they do not provide programmatic control over implementation details needed by middleboxes, *e.g.*, PSM transitions. As a result, middlebox developers may face significant challenges in building 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 [47] and Zeek [8], therefore cannot be reused when developing new applications. `libnids` [5] decouples the TCP middlebox stack from the high-level functions, making the stack reusable. Recent works like mOS [11] and Microboxes [48] 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 [49] and ClickNF [13] can manipulate the stack hierarchy using Click model [50], but they rely on pre-implemented elements, *e.g.*, ClickNF implements the TCP-related elements with 156 source files (12K LOC in C++) [51]. NetBricks [12] 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 [52] and Metron [53] 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.

Attempts for decoupling middlebox stack. NFV consolidates the NFs on a single physical machine, which, among other benefits like low deployment cost, promises two highly desirable features for operators: (F1) flexibility in NF management, *e.g.*, attaching/detaching NFs from SFC during runtime, and (F2) comparable performance to physical middleboxes. Recalling the motivation discussed in §I, it would be more beneficial if we can further (F3) decouple the stack from NFs and make the stack programmable

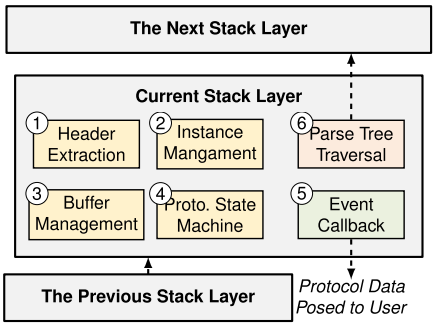


Fig. 1. The three key modules in a middlebox stack layer: protocol parser (yellow boxes), event callback (green box), and parse tree traversal (red box).

Many native NFV systems have aimed to address F1 and/or F2. OpenNetVM [54] allows NFs to run in processes and flexibly load/unload them by a centralized controller (F1). On top of that, NFVNice [55] leverages back-pressure on the chain to further reduce the redundant operations caused by downstream packet drops (F2). Both OpenBox [52] and Metron [53] enable static NF merging to eliminate redundant operations (F2). However, few NFV frameworks support decoupling stack from NF, making it very tedious when porting NFs to diverse workloads. mOS [11] and Microboxes [48] decouple stack and NF by providing NFs event-based interfaces, but they still cannot address F3 since their stack is not programmable.

C. Challenges and Our Approach

A DSL that fully captures the L2-L4 abstractions can be a cure to the above problems. Next, we will 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 as the end-host stacks, as shown in Fig. 1. Specifically, each stack layer starts by *parsing the protocol* (①–④), which extracts the header, manages the instance,[1] 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 though 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 disrupt 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"*

[1]We use instance here to avoid confusion with flow, which is most commonly used to signify TCP flows. Here, instance is unidirectional for connectionless protocols and bidirectional for connection-oriented protocols.

```
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)
```

Fig. 2. IP layer and fragmentation event written in Rubik.

that hides the retransmission exception and "virtual ordered packet" that hides the out-of-order exception, which can maximumly correspond to the intuitive pipeline (§IV).

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 (§V).*

III. 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.

Fig. 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.

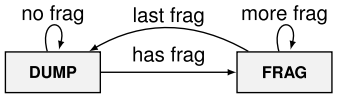


Fig. 3. Simplified PSM for IP fragmentation.

Parsing the header fields (Line 5–18). One has to first define the header format before referencing 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`. It is later used to compose the header parser with only one line in Line 18. Once the header parser is set up, Rubik allows referencing the fields by their names, *e.g.*, `ihl` can be referenced by `ip.header.ihl`.

Managing the instance table (Line 20). After the headers being declared, the stack layer finds the instance that the packet relates 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 containing 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.

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 Fig. 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 [5], [11], while Rubik can program two types of BIEs for arbitrary stacks.

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 [56].

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.

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, 7 LOC to hook the expected event, and 4 LOC to build the parse tree. As a comparison, `libnids` consumes 604 C LOC to implement a similar stack [5].

IV. Rubik Programming Abstractions

§III shows the potential of reducing coding effort with Rubik. However, as discussed in §II-C, 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.

A. 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.

B. 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 indentify the full 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 the 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`.

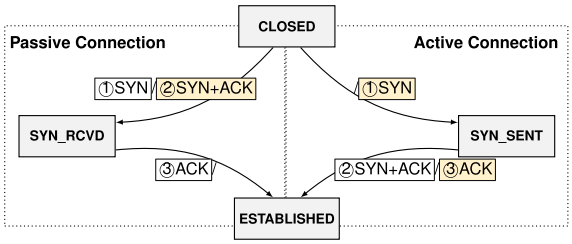
Transport protocols use a window to control the transmitting rate. Operators can pass a `window=(wnd,wnd_size)` parameter to `Sequence()`, which specifies the valid range of meta. The out-of-window packets will not be inserted into the sequence, but raise an `out_of_wnd` event.

C. Virtual Ordered Packets

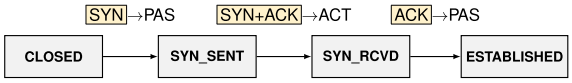
The sequence abstraction will sort the out-of-order packets, which, however, can still cause problems to stack processing. Consider the IP PSM shown in Fig. 3. In the possible out-of-order cases, the "last frag" packet can arrive earlier than a "more frag" packet. With the transitions defined in Fig. 2, such a packet will trigger `ip.psm.last` and form an incomplete SDU for the next layer. To avoid such a 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 this 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)
```



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



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

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

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 §V-D).

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()`.

D. 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 Fig. 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. Fig. 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 Fig. 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 Fig. 4a, SYN is the same packet with 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.

E. Event Ordering

By default, all the events will be raised after proceeding PSM and before parsing the next layer (see §V-B). 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
```

V. 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 (§V-A). To this end, we translate the Rubik program into an intermediate representation (IR) to reveal the factual control flow of the stack (§V-B). Then the middle-end of the compiler performs domain-specific optimizations on the IR to avoid the performance traps (§V-C), and finally the back-end translates the IR into performant C code (§V-D).

A. Avoiding Performance Issues Is Hard

As mentioned in §II-C, 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 if 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 a 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.

B. Intermediate Representation in Rubik

Rubik addresses the 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 §V-C). 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. 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 Fig. 1, *i.e.*, header parsing, instance table, packet sequence and PSM transition, and elaborates it with more information relevant to the optimization, *e.g.*, the operations on the instance table and the sequence. The conditional statements, *e.g.*, PSM transitions and event callbacks, will be translated into 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 relationships 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.

Fig. 5 shows a partial CFG for the IP program presented in §III. 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 instance into the instance table, and `InsertSeq()` to insert the payload into its sequence. Blocks 1–9 are for the IP protocol parser (only one PSM transition, *i.e.*, `ip.psm.dump`, is shown); blocks 10 and 11 are for the sequence assemble event; blocks 12 and 13 are for the next layer parsing, and blocks 14 and 15 handle the accept PSM state.

Revealing the dependency. Given the real control flow with CFG, the compiler can then reveal the dependency

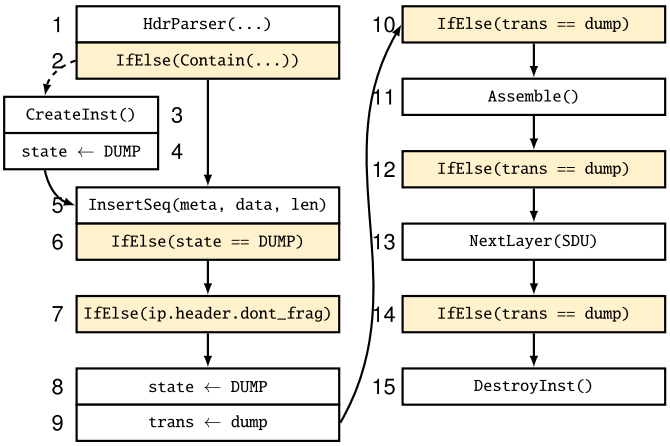


Fig. 5. Partial CFG of the IP program. The yellow boxes are the branching blocks. Only one PSM transition (`ip.psm.dump`) is shown. Most of the `false` branches (dashed edge) are also omitted.

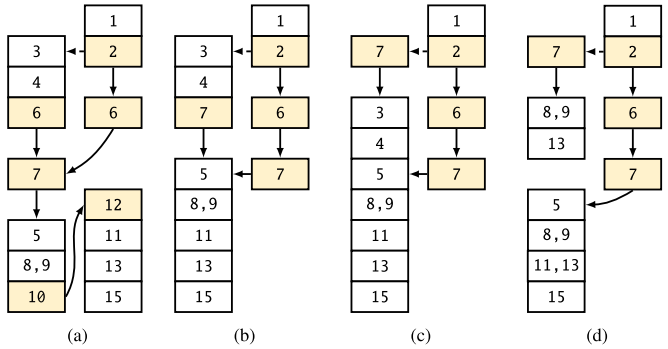


Fig. 6. (a) First-round branch lifting, where 6 is bounded by 2 and 4. (b) Constant analysis, where 6, 10 and 12 are eliminated because they are always `true` guarded by 4 and 9. (c) Second-round branch lifting, where 7 is further lifted above 3, due to the absence of 6. (d) Peephole optimization, where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

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 predefine the implicit dependencies for all instructions.

C. 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. Fig. 6a shows the CFG with branching blocks lifted. We take 6 as an example: it is lifted to top of the `true` branch of 2, because in this branch, it only depends on the conditions in 2; in contrast, in the `false` branch it

can only be lifted below block 4, because it reads the variable `state`, which is written by block 4. Through this process, 6 is duplicated, as both branches should traverse it. Note that some `false` branches are omitted in Fig. 6a, *e.g.*, the `false` branches of 6 and 7 that contain the duplicated 5.

The branch lifting process merges the basic blocks, which helps 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 6 in the `false` branch in Fig. 6a. We can easily assert that its condition (`state==DUMP`) is always `true`, because 4 has just assigned `state` with `DUMP`. Similar analysis takes place in block 9, 10, 12, where `trans==dump` in the latter two must be `true`. Those always-`true` branch blocks can be removed, as shown in Fig. 6b. Note that this elimination may create new opportunities to iteratively lift the branch, *i.e.*, Step 1. For example, 7 can be further lifted above 3, due to the absence of 6, as shown in Fig. 6c.

Step 3: Peephole optimizations. After the first two steps, the complete processing logic on each packet set is revealed. For example, 3–15 in Fig. 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 to identify and eliminate performance traps.

Considering 3–15, we have the following easy optimizations. (1) For 3, 5, 11, 13, it is obvious that the first and only inserted block is directly assembled. Hence, 5 and 11 can be eliminated, and 13 can be rewritten into `NextLayer(Payload)`. (2) 3 and 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. Fig. 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 is 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. We put the peephole optimizations that are currently employed in [57].

D. 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 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 [53]. Specifically, the back-end leverages the symmetric receive-side scaling (S-RSS) technique [58], 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 §VIII-A).

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

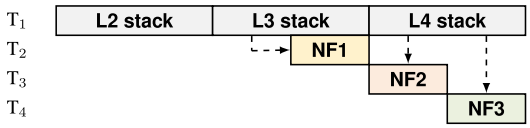
VI. Stack as a Service for Network Functions

In the era of network function virtualization (NFV), middleboxes are usually implemented as network functions (NFs) and are composed into a service function chain (SFC), in order to process the packets in a flexible and efficient fashion.

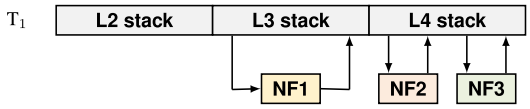
In this section, we advocate a novel NFV framework based on Rubik, where NFs can be implemented upon the stack events and managed in runtime without significant overhead, namely *Middlebox Stack as a Service (MSaaS)*. Specifically, the operator checks the SFC and collects all protocol stacks and BIEs that the NFs in the SFC take care of. Then, she creates a configuration with Rubik DSL, composing protocol layers into stack, and specifying a UDE for each BIE. After that, Rubik can generate a protocol parser according to the configuration. Finally, she links the generated code with NFV's controller and BIE-walking logic. The NFV framework is built, and she can run the SFC on it.

Recall F1–F3 raised in Section II-B. Since stack and NF are decoupled, a Rubik-assisted NFV framework can achieve F1 by dynamically loading NFs onto events. As for F2, Rubik's middle-end optimization lifts BIE events as much as possible, such that the backend C compiler can produce dead code elimination wisely. Additionally, we adopts run-to-completion execution model (as in Fig. 7b), ensures the cache consistency and avoids the potential synchronization between cores, resulting in a much higher per-core performance compared to pipeline model adopted by Microboxes (as in Fig. 7a). Finally, the usage of Rubik naturally and uniquely enables F3.

The following sections present our implementation of MSaaS upon Rubik (§VI-A and §VI-B).



(a) Microboxes run the stack and NFs in separate threads, resulting in the overhead of cache misses.



(b) A run-to-completion model uses a single thread to sequentially execute each component of the stack as well as the NFs built upon it.

Fig. 7. Microboxes exploits multiple threads for the stack and NFs (a), while RTC consolidates all execution in a single thread (b).

A. Dynamically Chaining NFs With Minor Cost

There are two types of overhead when implementing MSaaS: (1) overhead of composing the SFC structure, and (2) overhead of packets traversing different NFs. As discussed earlier, RTC reduces the second overhead by minimizing inter-core communication. However, for the first overhead, existing approaches often have to halt the entire SFC, recompile and reconfigure the NFs, and then restart the SFC. This results in a significant loss of packets and states in NFs.

Our insight comes from the nature of stack abstraction: given a list of BIEs, NFs only need to write handlers and hook the BIEs they are interested in. Therefore, changes in SFC structure are essentially adding/removing functions to/from the corresponding BIEs, which can be easily modified at runtime. And there have been lots of efforts to the end of runtime code modification, among which the most lightweight and straightforward solution would be the runtime library loading from `glibc`, *i.e.*, `dlopen` in `ld.so`. To be specific, each NF is compiled as a shared library (`.so` file), which hooks a certain set of BIEs posed by Rubik. When operators want to chain such NF into SFC, the controller would `dlopen` the corresponding NF, and the stack would invoke the callback function (located by `dlsym`) upon the proper BIE. Similarly, `dlclose` would be called for removing the NFs from SFC.

We use a simple example shown in Fig. 8 to detail this procedure. From the view of an operator, she only needs to write two functions: an initialization function with a fixed name that specifies the NF's name and the BIE it attaches (Line 2–6), and the entrance function that processes the data posed by the BIE. Here we simply attach an NF to the example BIE presented in §III, *i.e.*, `ip.event.ip_frag`, which counts the number of IP fragments (Line 8–16).

The MSaaS controller and the generated BIE handlers are linked together into an executable. In runtime they takes up separate cores: The MSaaS controller would periodically check the management messages from the operators (Line 22), which contain the names of shared libraries to be attached. Having the file name, the controller would `dlopen` the library (Line 23), locate and execute the initialization function (Line 25–26), and finally attach the entrance function to the corresponding BIE (Line 27–29). Note that we only include how the controller loads the NF here for brevity. The full-fledged controller is also responsible for generating new chain topology, unloading an NF, updating NF configuration, etc. The BIE handler generated by Rubik runs on another core and would walk through and execute all entrances attached to it (Line 36–37). Since the controller runs concurrently with the BIE handler, it will not bring noticeable processing overhead.

```
1  /*--- A simple NF that counts the IP fragments ---*/
2  static void init_nf(struct nf_config *cfg)
3  {
4      cfg->entrance = "count_ip_frags";
5      cfg->bie = IP_FRAG;
6  }
7
8  static int ip_frag_count = 0;
9
10 static void count_ip_frags(void *args, ...)
11 {
12     ip_frag_count++;
13     if (ip_frag_count > T) {
14         alert("Too many IP frags.\n");
15     }
16 }
17
18 /*--- The MSaaS controller that loads the NF into SFC ---*/
19 static void controller() {
20     struct nf_config cfg;
21     /* Check the message of NF management */
22     while ((so_name = check_nf_msg()) != NULL) {
23         nf_so = dlopen(so_name, RTD_LAZY);
24         /* Execute the initialization function */
25         init_nf_func = dlsym(nf_so, "init_nf");
26         init_nf_func(&cfg);
27         /* Locate and attach the entrance function */
28         nf_entrance = dlsym(nf_so, cfg.entrance);
29         attach_nf(nf_entrance, cfg.bie);
30     }
31 }
32
33 /*--- The BIE generated by Rubik that executes the NF ---*/
34 static void ip_frag(struct ipc * c) {
35     /* Walk through the NFs attached to this BIE */
36     while ((nf_entrance = walk_bie(IP_FRAG)) != NULL) {
37         nf_entrance(c);
38     }
39 }
```

Fig. 8. Pseudo code of the NF (Line 1–16), the MSaaS controller (Line 18–31) and the generated BIE (Line 33–39).

In sum, the runtime overhead of attaching a new NF is basically the time cost of `dlopen` and `dlsym`, which is quite lightweight (usually less than 100ms). More importantly, this procedure is performed by the controller in an asynchronous way, so the factual runtime cost could only include a spin lock of the BIE entrance array, which can be neglected, considering the SFC modification rarely happens.

B. Scheduling NFs in MSaaS

Despite the ability of chaining NFs with minor cost, we still need to address other issues to make MSaaS practical, *e.g.*, how to schedule NFs, how to profile and scale NFs. Here we briefly discuss one of them, *i.e.*, scheduling NFs.

Consider an SFC of NF1→NF2. If NF1 attaches to BIE1, NF2 attaches to BIE2, and BIE2 happens before BIE1, we need to correctly schedule NF1 and NF2, *i.e.*, invoke them in the order of SFC instead of the order of BIEs, otherwise the side effect of NFs (*e.g.*, header rewrite, packet drop) would impact the correct processing. In the following, we present two methods to schedule NFs in the correct order inside an SFC.

The first method simply collects all BIEs through the stack processing, and triggers the corresponding NFs in the order of SFC. However, this could result in unnecessary processing in the stack. For example, if the first NF drops the packet, the downstream stack processing can be skipped, while this method would finish the whole stack processing before invoking the first NF.

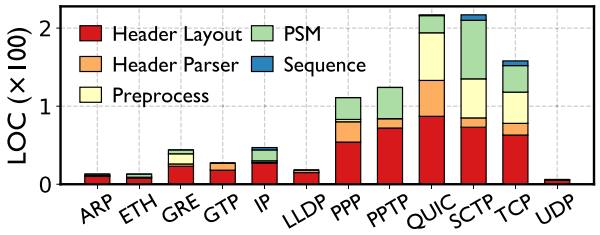


Fig. 9. LOC breakdown of protocol parsers.

The second method respects the factual order of BIEs. To be specific, each time a BIE occurs, Rubik compares its NF entrances with the order of SFC: ones that match with the SFC order would be directly invoked, and others would be queued, which will be again compared to the SFC order (and dequeued) when the next BIE happens. This method invokes the corresponding NFs as soon as possible, and thus would avoid the unnecessary stack processing. However, this method might harm the processing performance. The NF to be invoked is not fixed for each BIE, and such unpredictable control flow would break the CPU execution pipeline, and is not friendly to the instruction cache.

In practice, the first method is preferred if NFs do not drop packets.

VII. 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 (§VII-A), and developing typical high-level middlebox functions (§VII-B).

A. 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 Fig. 9, we have the following observations.

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) [59] 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 [60], making SCTP much less deployed [61]. Using Rubik, it only takes 210 LOC for implementing the SCTP layer. Another example is QUIC [62]: 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 [63]. Using Rubik, merely 216 LOC is enough for prototyping a QUIC parser (without the decryption feature, see §IX).

TABLE II
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*

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

B. Developing Applications With Rubik

In this section, we will show in detail how users can develop real-world applications leveraging Rubik's stack.

As discussed in §I, the code of middleboxes can be divided into two complementary parts: the stack and the high-level functions. In traditional middleboxes, the two parts are most commonly implemented with the same language, and stack invokes the high-level functions via their exported interfaces. However, when using Rubik, the two parts are decoupled at language level: the stack needs to be written in Rubik's DSL, while the high-level functions can be implemented in C/C++. The stack utilizes a unified interface `Callback()` to invoke the high-level functions every time the BIEs are triggered.

The most typical example can be a monitor that tracks the payload length of every active TCP flow. In this case, the developer can pose an event happening with the assemble event (`tcp.event.asm`).

```
class cnt_data(layout):
    length = Bit(32)
    sip = Bit(32)
    dip = Bit(32)
    sport = Bit(16)
    dport = Bit(16)
tcp.event.on_assemble =
    Assign(cnt_data.length, tcp.sdu.length) + \
    Assign(cnt_data.sip, tcp.sdu.sip) + \
    Assign(cnt_data.dip, tcp.sdu.dip) + \
    Assign(cnt_data.sport, tcp.sdu.sport) + \
    Assign(cnt_data.dport, tcp.sdu.dport) + \
    Callback(cnt_data)
tcp.event_relation +=
    tcp.event.asm >> tcp.event.on_assemble
```

Since the developer needs to count payload length of every flow, she needs to assign the four-tuple, as well as the payload length of every packet in the BIE handler. Provided with such information, the Rubik compiler will generate the handler as in Fig. 10:

The Rubik compiler adopts a two-phase compiling procedure. First, the stack written in DSL is compiled into a highly optimized stack (as in §V) and empty handlers hooking the specified BIEs (as in highlighted lines of Fig. 10). As instructed in the DSL configuration, the auto-generated handler can access the corresponding fields of the stack via

```
1  // Auto-generated data structure for cnt_data
2  typedef struct {
3    WV_U32 _204;  // cnt_data.length
4    WV_U32 _205;  // cnt_data.saddr
5    WV_U32 _206;  // cnt_data.daddr
6    WV_U16 _207;  // cnt_data.sport
7    WV_U16 _208;  // cnt_data.dport
8  }__attribute__((packed)) H12;
9  // A map storing flow status
10 std::map<five_tuple, std::size_t> flow_stat;
11 WV_U8 cnt_data(H12 *args, WV_Any *user_data) {
12   struct five_tuple tmp;
13   tmp.sip = args->_205;
14   tmp.dip = args->_206;
15   tmp.sport = args->_207;
16   tmp.dport = args->_208;
17   std::map<five_tuple, std::size_t>::iterator it;
18   if ((it = flow_stat.find(tmp)) == flow_stat.end()) {
19     flow_stat.insert(std::make_pair(tmp, args->_204));
20   } else {
21     it->second += args->_204;
22   }
23   return 0;
24 }
```

Fig. 10. The high-level function of payload counting application. Highlighted lines are generated by Rubik compiler.

a `struct` (Line 2–Line 8). Then, the developer fills in the empty handlers with custom actions. Under this application, the user may need a hash table to store the status of each flow (Line 10), and update the table when the stack triggers `cnt_data` (Line 12–Line 22). After filling the BIE handlers, the developer compiles and links the stack and the handler together using a C/C++ compiler (*e.g.*, `g++`), which forms the second phase of compiling. Since Rubik DSL only provides abstractions for L2-L4, the developers can still implement or reuse high-level functions in plain C/C++, without any restrictions.

We present how we port Snort [47] 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 [64] 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. §IX discusses the UDE programming in detail.

VIII. 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 better performance than the hand-written stacks? (§VIII-A)
2) Do Rubik-ported applications work correctly and efficiently on various stacks? (§VIII-B)
3) Do the middle-end optimizations help improve the performance of Rubik? (§VIII-C)
4) Does the RTC model used in Rubik improve the performance when chaining multiple network functions? (§VIII-D)

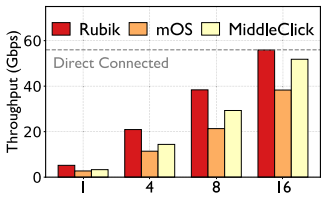


Fig. 11. The multi-core scalability (DA, 1KB file).

A. Microbenchmarks

To measure Rubik's performance under specific traffic loads, 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×Intel Xeon 2.2Ghz, 192GB memory) with three dual-port 40G NICs (Intel XL710). We use another six machines (8×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 (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 I, 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 [7] and MiddleClick [65] 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 [66] and S-RSS. The clients and servers are implemented using mTCP [43].

Fig. 11 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%.

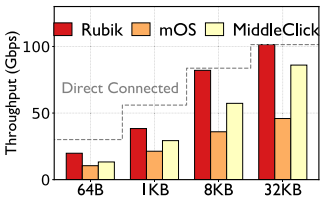


Fig. 12. The file size scalability (DA, 8 cores).

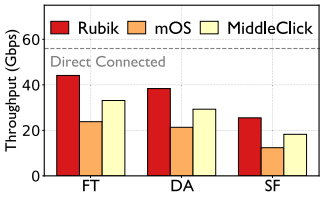


Fig. 13. Throughput vs. functions (1KB file, 8 cores).

Fig. 12 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.

Fig. 13 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 handle 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 performance 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 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.

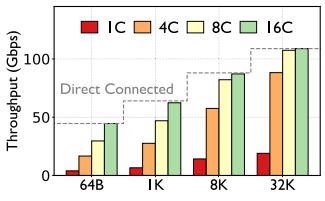


Fig. 14. Performance of the GTP stack (DA).

TABLE III
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

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.

Fig. 14 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 files, 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.

B. 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 III. 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 up a PPTP server with MPPE and PPP compression disabled, and capture the trace by accessing random websites. For QUIC stack, we set up a pair of client and server using `ngtcp2` [67], 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 §IX). For SCTP stack, we set the server and client using `usrsctp` [68], 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.

Applications. We port two well-known middlebox applications, Snort [47] and nDPIReader [69], to Rubik. For Snort, we port it as presented in §VII-B, 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

TABLE IV
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

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 the 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 IV 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. +16.8% for Snort, −2.6% vs.−3.7% for nDPI) [11]. 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.

C. Middle-End Optimizations

We inject the same traces used in §VIII-B 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 V 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 sizes, 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 a 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 V shows that such overhead is minor in practice, *i.e.*, all stacks are compiled within 15 seconds.

TABLE V
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

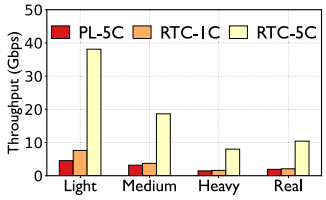


Fig. 15. Throughput comparison between PL and RTC.

D. Chaining Network Functions

We implement a multi-thread pipeline (PL) model to simulate Microboxes [48] and an RTC model used in Rubik that can dynamically chain the NFs in a single thread.

Implementation. For PL model, the stack and NFs are executed by individual threads, *i.e.*, the stack thread and the worker threads. The stack thread receives the packets, proceeds the stack, and moves the packet descriptors to the downstream workers. To synchronize the NFs in the order of SFC, the stack thread would notify the corresponding workers for each triggered BIE. If the worker has the packet descriptor *and* the notification, it would invoke its NF; otherwise it just silently forwards the packet to the next worker. We finally involve a dedicated TX thread to send the packets out of the NIC.

For RTC model, we implement the first method discussed in VI-B, where only one thread is involved in the processing of SFC. When that thread receives a packet, it processes the whole stack, and records all triggered BIEs. The corresponding NFs are then invoked in the order of SFC by the same thread, which would finally send the packet out.

Setup. We use an SFC of 3 synthesized NFs, which hook IP reassembly, TCP connection establishment and termination, respectively. All the NFs perform string matching against their payload, and the workload varies by the number of patterns being used. The throughput and end-to-end latency of PL and RTC under light, medium and heavy loads are measured.

We also build another SFC to evaluate the real-world workload, which copies the packet on IP reassembly, looks up a hash table of source IP and port on TCP establishment, and scans the payload of TCP packets on receiving TCP data. All four SFCs are tested under the same TCP trace used in Table III.

Results. Fig. 15 shows that the single-thread RTC implementation offers a throughput boost of 68%, 17% and 10% under three kinds of loads, 8% under real SFC, compared to its PL counterpart, which takes up to five threads, including one stack thread, one TX thread and three worker threads. The throughput gain of RTC is more obvious when the load is more lightweight, because the overhead of inter-core communication and ring-based message queues is fixed, which is less significant under heavy load. The performance of RTC using five worker cores is also presented, for S-RSS can

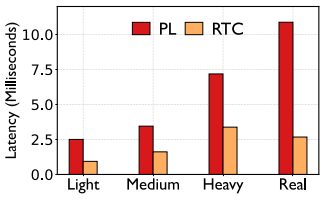


Fig. 16. Latency comparison between PL and RTC.

distribute the traffic evenly among all workers, approximately achieving a 5× linear boost.

RTC also achieves 61% less end-to-end latency on average as shown by Fig. 16. This indicates that the RTC model used in Rubik has a large performance advantage over the PL model, as well as maintaining flexibility.

IX. 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. Rubik can well support programming the former type, for all protocols to the best of our knowledge. 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. Currently, we present the capability and limitation of Rubik DSL in our Github repository [70].

Difference between Rubik and P4. The semantics of Rubik DSL is a strict superset of P4, which means that it is possible to translate an arbitrary P4 program into Rubik. Nevertheless, due to distinct design goals, such translation will not always be ideal. First, Rubik is designed for reusable protocol layers, while P4 lacks this feature and treats the layers like header declarations. Second, in P4, all parsing must be done before computing any variable. On the other hand, Rubik allows conditional parsing of upper layer based on instance or packet states that may just be updated by lower layers. Finally, the selector semantics of Rubik is also far more sophisticated than that of P4. For connection-oriented protocols, Rubik maintains one single instance for bidirectional traffic, finding instances and determining the direction during lookups. It is possible to simulate this logic in P4, but it would require a significant engineering effort.

Runtime reconfiguration. Rubik does not support *dynamically* reconfiguring the stack. As discussed in §V, the stack is compiled from one auto-generated and optimized C file, mainly to benefit from optimizations within the same translation unit. This means each time the stack changes, the operator needs to recompile the Rubik code, relink the compiled stack with the high-level functions, and reboot the middlebox.

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 focused on how to facilitate the middlebox development by offering friendly UDE interfaces. mOS [11] unifies the TCP stack and provides a BSD-style socket interface, so the developers can dynamically register/deregister their UDEs. Microboxes [48] 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.

Reusing NFs to build SFC. An SFC can be more generally applied to versatile network traffic only if it is hosted by an NFV framework with a more configurable network stack. For example, consider a simple VNF that counts the number of fragmented IP packets (similar to the example in Figure 8). In a previous NFV framework like NetBricks [12] or Microboxes [48], such an NF can only apply to raw IP traffic. If later the operator wants to count for tunneled traffic as well, she must parse the tunneled packets by herself. Even if the NFV framework provides the parsing building block she needs, the exposed interface by those tunneling stacks may not be the same as the interface of raw IP stack. For example, the events emitted by different μstacks of Microboxes may have distinct names and semantics. As a result, the operator cannot reuse the existing NF easily. She has to adapt it or write a new NF for another kind of traffic for every time. On the contrary, Rubik provides full support on configuring network stack and its interface to UDEs. This allows operators to switch to a new kind of traffic with few lines of change in Rubik DSL and no change in her own NF implementation.

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++ [71] and eRSS [72] 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 [73] or a programmable switch associated with the middlebox [53] that can dispatch the packets by arbitrary headers could be a cure for this problem.

Middlebox deployments. While Rubik facilitates the development of a single middlebox, there are literatures that deploy middleboxes in a distributed way [74], [75] or as a cloud service [76]. These works can be complementary to Rubik.

Isolation in RTC model. RTC execution model trades isolation among NF instances for better performance. Since all NF instances typically run in the same process, each of them can access the packet buffer as well as the states of others. There have been several works tackling this challenge: NetBricks [12] isolates each instance with a safe language. Quadrant [77] runs each instance in a container, and copies packet once to isolate the packet buffer. Vigor [78] verifies if an NF's behavior strictly according to its specification.

We plan to discover how to enforce isolation among instances under MSaaS in the future.

X. 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.

References

[1] H. Li et al., "Programming network stack for middleboxes with Rubik," in *Proc. USENIX NSDI*, 2021, pp. 551–570.

[2] (2018). *IEEE 802.11 Wireless Local Area Networks*. [Online]. Available: http://grouper.ieee.org/groups/802/11/

[3] A. Langley et al., "The QUIC transport protocol: Design and internet-scale deployment," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 183–196.

[4] (2016). *In-Band Network Telemetry (INT)*. [Online]. Available: https://p4.org/assets/INT-current-spec.pdf

[5] (2018). *Libnids*. [Online]. Available: http://libnids.sourceforge.net/

[6] (1995). *IP in IP Tunneling*. [Online]. Available: https://tools.ietf.org/html/rfc1853

[7] (2018). *mOS-Networking-Stack*. [Online]. Available: https://github.com/ndsl-kaist/mos-networking-stack

[8] (2018). *The Zeek Network Security monitor*. [Online]. Available: https://www.zeek.org/

[9] (2018). *P4 Language*. [Online]. Available: https://p4.org/

[10] (2018). *Vector Packet Processing*. [Online]. Available: https://fd.io/

[11] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. S. Park, "mOS: A reusable networking stack for flow monitoring middleboxes," in *Proc. USENIX NSDI*, 2017, pp. 113–129.

[12] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *Proc. USENIX OSDI*, 2016, pp. 203–216.

[13] M. Gallo and R. Laufer, "ClickNF: A modular stack for custom network functions," in *Proc. USENIX ATC*, 2018, pp. 745–757.

[14] C. Raiciu et al., "How hard can it be? Designing and implementing a deployable multipath TCP," in *Proc. USENIX NSDI*, 2012, pp. 399–412.

[15] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An untold story of middleboxes in cellular networks," in *Proc. ACM SIGCOMM Conf.*, Aug. 2011, pp. 374–385.

[16] (2018). *MiddleBoxes: Taxonomy and Issues*. [Online]. Available: https://tools.ietf.org/html/rfc3234

[17] B. Hesmans, F. Duchene, C. Paasch, G. Detal, and O. Bonaventure, "Are TCP extensions middlebox-proof?" in *Proc. Workshop Hot Topics Middleboxes Netw. Function Virtualization*, Dec. 2013, pp. 37–42.

[18] R. Craven, R. Beverly, and M. Allman, "A middlebox-cooperative TCP for a non end-to-end internet," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 151–162, Feb. 2015.

[19] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, "Is it still possible to extend TCP?" in *Proc. ACM IMC*, 2011, pp. 181–194.

[20] A. Bhartia et al., "Measurement-based, practical techniques to improve 802.11ac performance," in *Proc. Internet Meas. Conf.*, Nov. 2017,pp. 205–219.

[21] (2018). *The New Waist of the Hourglass*. [Online]. Available: http://tools.ietf.org/html/draft-tschofenig-hourglass-00

[22] A. Medina, M. Allman, and S. Floyd, "Measuring the evolution of transport protocols in the internet," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 2, pp. 37–52, Apr. 2005.

[23] (2018). *Snort: Re: Is There a Snort/Libnids Alternative*. [Online]. Available: http://seclists.org/snort/2012/q4/396

[24] (2018). *Middlebox Cooperation Protocol Specification and Analysis*. [Online]. Available: https://mami-project.eu/wp-content/uploads/2015/10/d32.pdf

[25] K. Edeline and B. Donnet, "Towards a middlebox policy taxonomy: Path impairments," in *Proc. IEEE Conf. Comput. Commun. Workshops*, Apr. 2015, pp. 402–407.

[26] (2018). *Report From the IAB Workshop on Stack Evolution in a Middlebox Internet (SEMI)*. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7663.txt

[27] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "BlindBox: Deep packet inspection over encrypted traffic," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 213–226.

[28] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu, "Embark: Securely outsourcing middleboxes to the cloud," in *Proc. USENIX NSDI*, 2016, pp. 255–273.

[29] J. Han, S. Kim, J. Ha, and D. Han, "SGX-Box: Enabling visibility on encrypted traffic using a secure middlebox module," in *Proc. 1st Asia–Pacific Workshop Netw.*, Aug. 2017, pp. 99–105.

[30] A. Gember-Jacobson et al., "OpenNF: Enabling innovation in network function control," in *Proc. ACM Conf. SIGCOMM*, Aug. 2014, pp. 163–174.

[31] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the way for NFV: Simplifying middlebox modifications using StateAlyzr," in *Proc. USENIX NSDI*, 2016, pp. 239–253.

[32] A. Tootoonchian et al., "ResQ: Enabling SLOs in network function virtualization," in *Proc. USENIX NSDI*, 2018, pp. 283–297.

[33] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *Proc. USENIX NSDI*, 2018, pp. 299–312.

[34] S. Palkar et al., "E2: A framework for NFV applications," in *Proc. 25th Symp. Operating Syst. Princ.*, Oct. 2015, pp. 121–136.

[35] (2018). *The Impact on Network Security Through Encrypted Protocols—Quic*. [Online]. Available: https://bit.ly/2xw7z8y

[36] (2015). *Do You Guys Block UDP Port 443 for Your Proxy/Web Filtering?* [Online]. Available: https://bit.ly/2OBM511

[37] (2018). *Quic Protocol | Cisco Community*. [Online]. Available: https://community.cisco.com/t5/switching/quic-protocol/td-p/3402269

[38] G. Papastergiou et al., "De-ossifying the internet transport layer: A survey and future perspectives," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 1, pp. 619–639, 1st Quart., 2017.

[39] R. Pang, V. Paxson, R. Sommer, and L. Peterson, "binpac: A yacc for writing application protocol parsers," in *Proc. 6th ACM SIGCOMM Conf. Internet Meas.*, Oct. 2006, pp. 289–300.

[40] Z. Li et al., "NetShield: Massive semantics-based vulnerability signature matching for high-speed networks," in *Proc. ACM SIGCOMM Conf.*, Aug. 2010, pp. 279–290.

[41] 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 *Proc. IEEE INFOCOM*, Mar. 2012, pp. 1746–1754.

[42] H. Li, C. Hu, J. Hong, X. Chen, and Y. Jiang, "Parsing application layer protocol with commodity hardware for SDN," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, May 2015, pp. 51–61.

[43] E. Jeong et al., "mTCP: A highly scalable user-level TCP stack for multicore systems," in *Proc. USENIX NSDI*, 2014, pp. 489–502.

[44] S. Pathak and V. S. Pai, "ModNet: A modular approach to network stack extension," in *Proc. USENIX NSDI*, 2015, pp. 425–438.

[45] (2019). *Seastar*. [Online]. Available: http://seastar.io/

[46] (2019). *F-Stack*. [Online]. Available: http://www.f-stack.org/

[47] (2018). *Snort—Network Intrusion Detection and Prevention System*. [Online]. Available: https://www.snort.org/

[48] G. Liu, Y. Ren, M. Yurchenko, K. K. Ramakrishnan, and T. Wood, "MicroBoxes: High performance NFV with customizable, asynchronous TCP stacks and dynamic subscriptions," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 1–14.

[49] T. Barbette, C. Soldani, and L. Mathy, "Combined stateful classification and session splicing for high-speed NFV service chaining," *IEEE/ACM Trans. Netw.*, vol. 29, no. 6, pp. 2560–2573, Dec. 2021.

[50] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[51] (2018). *ClickNF*. [Online]. Available: https://github.com/nokia/ClickNF

[52] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A software-defined framework for developing, deploying, and managing network functions," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 511–524.

[53] G. P. Katsikas, T. Barbette, D. Kostic, R. Steinert, and G. Q. Maguire, "Metron: NFV service chains at the true speed of the underlying hardware," in *Proc. USENIX NSDI*, 2018, pp. 171–186.

[54] W. Zhang et al., "OpenNetVM: A platform for high performance network service chains," in *Proc. Workshop Hot Topics Middleboxes Netw. Function Virtualization*, Aug. 2016, pp. 26–31.
[55] S. G. Kulkarni et al., "NFVnice: Dynamic backpressure and scheduling for NFV service chains," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 71–84.
[56] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, "Design principles for packet parsers," in *Proc. Archit. Netw. Commun. Syst.*, Oct. 2013, pp. 13–24.
[57] (2022). *Peephole Optimizations in Rubik*. [Online]. Available: https://github.com/ants-xjtu/rubik/blob/master/doc/04-peephole-optimizations.markdown
[58] S. Woo, E. Jeong, S. Park, J. Lee, S. Ihm, and K. Park, "Comparison of caching strategies in modern cellular backhaul networks," in *Proc. 11th Annu. Int. Conf. Mobile Syst., Appl., Services*, Jun. 2013, pp. 319–332.
[59] (2007). *Stream Control Transmission Protocol*. [Online]. Available: https://tools.ietf.org/html/rfc4960
[60] (2013). *Packet-Sctp*. [Online]. Available: https://github.com/boundary/wireshark/blob/master/epan/dissectors/packet-sctp.c
[61] D. A. Hayes, J. But, and G. Armitage, "Issues with network address translation for SCTP," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 23–33, Dec. 2008.
[62] (2017). *Quic: A UDP-Based Secure and Reliable Transport*. [Online]. Available: https://tools.ietf.org/html/draft-ietf-quic-transport-11
[63] (2019). *Packet-Quic.c*. [Online]. Available: https://bit.ly/32ED3YK
[64] X. Wang et al., "Hyperscan: A fast multi-pattern regex matcher for modern cpus," in *Proc. USENIX NSDI*, 2019, pp. 631–648.
[65] (2020). *Middleclick*. [Online]. Available: https://github.com/tbarbette/fastclick/tree/middleclick
[66] (2018). *DPDK*. [Online]. Available: http://www.dpdk.org/
[67] (2018). *Ngtcp2*. [Online]. Available: https://github.com/ngtcp2/ngtcp2
[68] (2020). *USRSCTP: A Portable SCTP Userland Stack*. [Online]. Available: https://github.com/sctplab/usrsctp
[69] (2018). *NDPI*. [Online]. Available: https://www.ntop.org/products/deep-packet-inspection/ndpi/
[70] (2022). *Survey of Rubik's Semantics Boundary*. [Online]. Available: https://github.com/ants-xjtu/rubik/blob/master/doc/03-semantic-completeness.markdown
[71] T. Barbette, G. P. Katsikas, G. Q. Maguire, and D. Kostić, "RSS++: Load and state-aware receive side scaling," in *Proc. 15th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2019, pp. 318–333.
[72] A. Rucker, M. Shahbaz, T. Swamy, and K. Olukotun, "Elastic RSS: Co-scheduling packets and cores using programmable NICs," in *Proc. 3rd Asia–Pacific Workshop Netw.*, Aug. 2019, pp. 71–77.
[73] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading distributed applications onto smartNICs using iPipe," in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2019, pp. 318–333.
[74] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *Proc. ACM SIGCOMM Conf. SIGCOMM*, Aug. 2013, pp. 27–38.
[75] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, "The middlebox manifesto: Enabling innovation in middlebox deployment," in *Proc. 10th ACM Workshop Hot Topics Netw.*, Nov. 2011, pp. 1–6.
[76] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, Aug. 2012, pp. 13–24.
[77] J. Wang, T. Lévai, Z. Li, M. A. M. Vieira, R. Govindan, and B. Raghavan, "Quadrant: A cloud-deployable NF virtualization platform," in *Proc. 13th Symp. Cloud Comput.*, Nov. 2022, pp. 493–509.
[78] A. Zaostrovnykh et al., "Verifying software network functions with no verification expertise," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, Oct. 2019, pp. 275–290.

Hao Li received the Ph.D. degree in computer science from Xi'an Jiaotong University in 2016. He is currently an Associate Professor with the School of Computer Science and Technology, Xi'an Jiaotong University. He is also with the MOE Key Laboratory for Intelligent Networks and Network Security. His main research interests include programmable networks and network functions.

Yihan Dang received the B.Eng. degree in computer science from Xi'an Jiaotong University in 2021, where he is currently pursuing the Ph.D. degree. His research interests mainly focus on optimizing virtualized network function systems.

Guangda Sun received the bachelor's degree in computer science from Xi'an Jiaotong University in 2020. He is currently pursuing the Ph.D. degree with the National University of Singapore. His research interests include designing fault-tolerance systems, protocols for both high-speed data center networks, and large-scale decentralized networks.

Changhao Wu, photograph and biography not available at the time of publication.

Peng Zhang received the Ph.D. degree in computer science from Tsinghua University in 2013. He was a Visiting Researcher with The Chinese University of Hong Kong and Yale University. He is currently a Professor with the School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China. He is also with the MOE Key Laboratory for Intelligent Networks and Network Security. His research interests include verification, measurement, and security in computer networks.

Danfeng Shan (Member, IEEE) received the B.E. degree in computer science and technology from Xi'an Jiaotong University, China, in 2013, and the Ph.D. degree in computer science and technology from Tsinghua University, China, in 2018. He is currently an Associate Professor with the School of Computer Science and Technology, Xi'an Jiaotong University. His research interests include datacenter networks and traffic management.

Tian Pan received the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, in 2015. He is currently an Associate Professor with the Beijing University of Posts and Telecommunications. His primary research interests include cloud data center networks, programmable data plane, and satellite networks.

Chengchen Hu (Member, IEEE) is currently the Chief Expert and an AVP with NIO Inc. Prior to joining NIO, he was a Principal Engineer and the Founding Director of the Xilinx Laboratories Asia–Pacific, Singapore. Before his experience with Xilinx, he was a Professor and the Department Head with the Department of Computer Science and Technology, Xi'an Jiaotong University, China. He was a recipient of the New Century Excellent Talents in University Award from the Ministry of Education, China; and a fellowship from the European Research Consortium for Informatics and Mathematics (ERCIM) and the Microsoft "Star-Track" Young Faculty. His research theme is to monitor, diagnose, and manage networking and distributed computing through hardware optimized and software-defined systematical approaches.