Parsing Application Layer Protocol with Commodity Hardware for SDN

Hao Li[†], Chengchen Hu[†‡], Junkai Hong[†], Xiyu Chen[†], Yuming Jiang[‡]
[†]Xi'an Jiaotong University [‡]Norwegian University of Science and Technology

ABSTRACT

The de facto implementation of Software Defined Networking (SDN), *i.e.*, OpenFlow, only parses L2-L4 headers, which limits the use of SDN to employ control intelligence in application layer. In this paper, we advocate content parsing to empower SDN with finer grained control ability over traffic. Specifically, we propose a scalable content parser, called COPY, to identify and parse application layer protocols. COPY creates a distinguishable counting context free grammar (DCCFG) to specify the protocol's semantics in application layer, and translates multiple DCCFGs into one distinguishable counting automaton (DCA). DCA is generated without semantic loss from the single DCCFG, and thus provides accurate and scalable parsing ability. Our experiments show that COPY precisely identifies every packet in a labeled trace. When comparing with other six approaches on the real traces, COPY performs 4.2Gb/s and 24.7Gb/s with single- and eight-thread models, respectively, which improves 20%–860% than others, and consumes acceptable offline overhead in time and space.

1. INTRODUCTION

Software Defined Networking (SDN) is recognized as a promising direction of the future Internet, which separates the control plane and the data plane physically, abstracts network functions and controls the network with a global view. OpenFlow is the de facto SDN southbound interface between the control plane (controller) and the data plane (Openflow Switch) [23].

Although the number of fields which OpenFlow checks increases from 12 with OpenFlow 1.0 up to 41 with OpenFlow 1.4, it is so far still limited to L2 – L4 and would become very complicated to extended into more use cases [23]. To better support functions of higher layer appliances, it is advocated to extend SDN with the capability of identifying arbitrarily user-defined protocols, *i.e.*, application layer protocols [10, 11, 30]. This empowers SDN more flexible programmability and finer-grained control with detailed knowledge obtained from the traffic. A simple example is fine-grained traffic engineering, *e.g.*, distribute images and videos in high definition to lightly loaded path, limit download speed from certain applications, *etc.*

With the above motivation, we investigate, in the present paper, application layer protocol parsing for SDN, which is able to extract the field values of protocols up to L7. In the literature, application layer parsing techniques have been studied in the context of middleboxes located in the edge of the network, but they have difficulties to satisfy the speed and flexibility requirements with large fine-grained policy set for SDN. Binpac [24] and Ultrapac [18] requires considerable efforts on specifying new protocols, thus they are preferred to serve as the intrusion detection system (IDS), where fewer protocols are involved. FlowSifter [20] eases the specification and provides higher throughput, but like Binpac and Ultrapac, it focuses on the single protocol parsing, and its scalability becomes the major concern with much more protocols. Simple extensions of these methods by introducing a prior protocol identifier or sequential processing are not valid, as demonstrated later in this paper.

Recently, several papers have proposed to use protocol-independent parsers for SDN, whose implementations mainly rely on specialized hardware to provide high throughput, such as ASIC chip [30, 31] and vender-specialized network processor [10, 11, 14]. Such none-commodity constraint on hardware is rigid and expensive, which consequently causes it hard to support high layer protocols that are often defined with flexible structure. In addition, these methods are basically optimized for "binary-based" protocol and may face difficulties to efficiently parse the "character-based" protocols, *e.g.*, HTTP.

In this paper, we present a novel COntent Parser methodologY (COPY) for application layer protocol parsing in SDN, which is accurate, fast, flexible and scalable. Specifically, COPY enables higher layer visibility (L7 header and payload) and provides wire-speed processing ability without compromise at the semantic level or of accuracy. In the control plane, COPY generates a parsing automaton according to the application layer protocol specifications. The parsing automaton is then issued into the data plane, where the parsing component, based on the parsing automaton, extracts the network traffic knowledge and pipelines it to the matching component or the network applications through the northbound. We have implemented COPY in a commodity platform without any specialized hardware.

Overall, we have three key contributions in the design of COPY (§3):

Expressive and distinguishable specification (§4). We propose a distinguishable counting context free grammar (DCCFG) to specify an application by its L7 header or payload. This expressive and user-friendly grammar can distinguish multiple extracting behaviors across protocols. Our evaluations show that DCCFG can express complex protocols in the application layer within only tens of lines of code.

High speed parsing structure for multiple protocols (§5, §6). We employ a distinguishable counting automaton (DCA) to provide linear-complex parsing on the input length for multiple protocols. DCA identifies the protocol and extracts the field values simultaneously, where the parsing automaton is similar to the traditional deterministic finite automaton (DFA) with only minor overhead.

Commodity implementation and evaluations using real traces (§7). We have built a real prototype implementation of COPY on a commodity platform, and evaluated it using real traces. The experiments show that, with 38 complex L7 protocols, COPY achieves 4.2Gb/s with single thread, and 24.7Gb/s with eight threads. A comprehensive comparison of COPY with six related works further demonstrates that COPY strikes the best performance, improving the factual throughput, or goodput (*i.e.*, throughput excluding the fault identification), by 23% – 860% only at a memory cost of about 450MB.

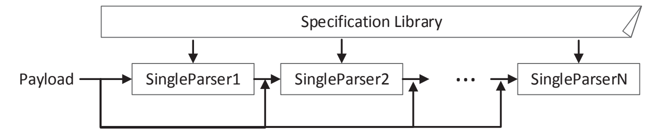
2. GOALS AND CHALLENGES

In this section, we detail the goals as well as the corresponding challenges mentioned in §1.

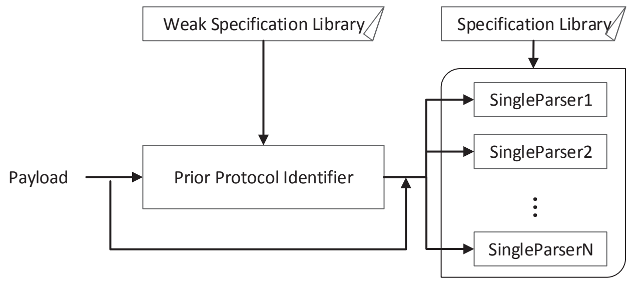
2.1 High layer visibility and programmability

Goal. A recent survey indicated that the number of L7 appliances (*e.g.*, middleboxes) in SDN's infrastructures is comparable to the number of routers [29], making high layer processing in SDN highly challenging. Particularly, to empower SDN with such functions, high layer visibility and protocol parsing in the device are necessary. For example, one operator aims to redirect the traffic of high-definition videos on Youtube to lightly loaded paths so as to provide better QoS based on the SLAs. To realize this, the operator could specify a rule {`App:"Youtube",Resolution:"HD"`} with an action, *e.g.*, forwarding to port 2 where sufficient resource is reserved. This function is like an intelligent load balancer, which inspects not only L7 header (*e.g.*, `Host` field in HTTP request), but also L7 payload (*e.g.*, a certain value in HTML in the response) in the proprietary Youtube protocol. In addition, besides the visibility, the parser is also required to offer the high layer programmability to avoid the frequent update of supporting new protocols [10, 30].

Challenge. Unlike L2-L4 headers located in fixed positions, it is not easy to specify proprietary application protocols since they may be organized in a highly different way. The most common such cases include using a recursive grammar for markup languages (`S`→`[S]` in HTML), and a type-length-value (TLV) field for counting-sensitive need (`Content-Length` field in HTTP). In addition, the regular grammar (RG) is widely used to describe a L7 protocol, which is of low expressiveness. For example, RG uses `GET /index.html HTTP/1.1\r\n` to represent an HTTP request header with method "GET" and URI "index.html", but it also could appear in other L7 payload (intentionally or unintentionally). RG cannot handle the above recursive cases and counting-sensitive cases, which brings risk of inaccuracy and is not feasible for parsing the aforementioned SDN content. The context-free grammars (CFG) could handle the recursive cases, but its recursive descent parsing needs to maintain a symbol stack, which makes it much slower than parsing the finite automaton generated by RG. Moreover, due to the uncertain length of the stack, CFG cannot handle the counting sensitive cases either. The counting sensitive feature normally requires the context sensitive grammar (CSG), but it is very expensive to implement a CSG parser.



(a) Sequentially parsing tries each protocol one by one. The payload will be parsed N times in the worst case (N is the number of the protocols). Speed is the major concern.



(b) The prior identifier dispatches the payload by a much weaker specification library such as RegEx, which may mislead the single protocol parser. Besides, it still needs to parse the whole payload in the single parser, bringing parsing redundancy.

Figure 1: Two simple extensions to support parallel parsing. Both suffer from one of two drawbacks: accuracy and speed.

2.2 Fast, accurate, and scalable processing

Goal. The parser for SDN works in the core of the network, which should process the packets in real time. Traditional middleboxes such as IDS also inspect L7 content in the packets, while they are mainly deployed in the edge of the network with low line rate. In addition, IDS typically cares about only a few protocols, while our goal is to have a parser that is scalable when new protocols are added, which happens ordinarily in the era of SDN. Furthermore, many other middleboxes work on the mirror traffic that will not impact the real network, or provide lower visibility level that barely makes mistake. To accelerate the flexible parser, many works proposed hardware-specialized approaches, which either are based on the elaborately designed ASIC chip [14] or rely on the vender-specialized hardware [30]. These approaches set an uncertain timetable for their practical use. In general, for use in SDN, we need a parser that has accurate processing ability at wire-speed and is scalable with the number of protocols.

Challenge. The fundamental challenge to achieve the above goal is to provide high parsing speed without compromising the accuracy when handling multiple protocols, denoted as "parallel parsing". If it can be realized, the approach then meets both the accuracy and the speed goals. Actually, despite the L2-L4 parsers and the inaccurate RG-based parsers, most current parsers focus on single protocols, and they can only support parallel parsing in two degraded ways. They either parse each application specification sequentially until it hits the input, or need a prior identifier to dispatch the packets to a specific application parser [18, 26]. The former sequential parsing (SP), depicted in Figure 1(a), obviously cannot scale with the number of protocols. The latter prior identification (PI), shown in Figure 1(b), is based on regular expressions (RegExs) or well-known ports, which lowers the accuracy and may mislead the post-parsers. To be specific, SP will try N times on a same payload in the

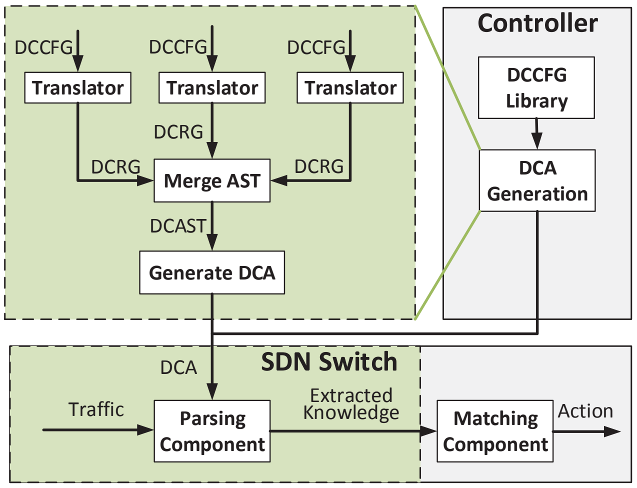


Figure 2: COPY architecture.

worst case (N is the number of parsers), if none of the candidate parsers hits the payload. PI distributes the packets to their single parser by a weak specification library (well-known ports or RegExs mostly), which risks the accuracy of the whole approach. Since the parsing result will impact the SDN immediately with its corresponding actions, PI is not best suitable for parallel parsing in SDN in the first place. Besides, the single parser needs to re-parse the payload that the prior identifier has checked, which brings redundant parsing. In general, the trade-off between accuracy and speed in the traditional L7 awareness schemes does not meet the above goal, *i.e.*, the challenge remains open.

3. OVERVIEW OF COPY

Our goal in this paper is to address the challenges described in §2 and realize a real prototype for the existing SDN switches on the commodity platforms. Our solution, called COPY, is a parsing component for SDN with higher layer awareness ability, which provides user-friendly specifications for complex proprietary protocols, and implements a fast and scalable parsing scheme for multiple protocols without any semantic loss.

Figure 2 gives an overview of the COPY architecture showing the inputs and outputs, as well as the basic workflow. In the control plane, the operators specify application specifications in DCCFG, and COPY translates them into DCA, which is issued to the data plane. In the data plane, the switches utilize DCA as the parsing structure, and pipeline the obtained knowledge to the matching component for further actions. Basically, we need to add message between the controller and the switches for issuing DCA. And if using current OpenFlow version, experimental message in OpenFlow can be used for this purpose.

Specifically, COPY consists of three stages: the first stage takes DCCFG as the input, and translates it into the distinguishable counting regular grammar (DCRG); the second stage merges multiple DCRGs and constructs a distinguishable counting abstract syntax tree (DCAST); the third stage generates the DCA, which is utilized as the parser of SDN. The basic idea in this process is to transform and maintain the unique identifiers specified in the DCCFGs. We use these identifiers to record the possibilities across multiple protocols and to determine the extraction behavior when parsing with DCA. Therefore, COPY will not bring on-line overhead for distinguishing protocols, and it is able to achieve high throughput and good scalability on the growing protocols, and keep their original semantics at the same time. Note that the off-line construction of DCA in the controller will bring some acceptable overhead in time and space, which will be discussed in our evaluations.

In the next sections, we will first present DCCFG and its translation process to DCRG (§4), then merge them into DCA (§5), and show the parsing process with it (§6).

1	S	$\to$ T L V
2	T	$\to$ <key, 0, “[A-Z]”>
3	L	$\to$ “[0-9]” [len=getnum()]
4	[len>0]V	$\to$ <value,0,“[A-Za-z]*”> [len=reduce()]
5	[len=0]V	$\to \epsilon$

Figure 3: A TLV specification Γ in DCCFG.

4. DCCFG: MAKE IT EXPRESSIVE AND DISTINGUISHABLE

4.1 DCCFG Formulation

We propose a new specification form DCCFG, which can be formulated as a six-tuple, $\Gamma = (\mathbb{N}, \Sigma, \mathbb{C}, \mathbb{R}, S, \mathbb{E})$, where $\mathbb{N}$, Σ, $\mathbb{C}$, $\mathbb{R}$, S, $\mathbb{E}$ are the finite set of non-terminals, terminals, counters, production rules, start non-terminal, and extraction tokens, respectively. The non-terminals are the symbols where the terminals can be derived. The terminals can be a single character or a RegEx. The production rules can be described as <`guard`>:<`non-terminal`>→<`body`><`action`>. The `guard` and `action` denote the constrains and operations on the counters. The production rules can be derived only if the guard is true, and the `action` will be executed simultaneously [20]. The `body` consists of the terminals, non-terminals, and the extraction tokens. The extraction tokens are a set of three-tuples, denoted as <`type, id, value`>. The `type` indicates the extracted characters should be treated as a key or a value. There are three extracting types: *key* type extracts the value as a field key, *value* type treats the value as a field value, and *key_named* type indicates a new key with a self-defined name. The key and value of the same field are with the same `id`, according to which they can be correctly paired. The `value` indicating the extracting target can be an existing terminal/non-terminal, an action-defined variable or a self-defined name, which can be paired in `type` way.

Figure 3 illustrates a simple TLV specification example, which uses a counter to control the length of V and two extraction tokens to extract the type-value pair. The grammar could produce the string “A5hello”, and (“A”, “hello”) will be extracted. To be specific, rule 2 can produce “A” which will be extracted as a key with the id 0. Then rule 3 produces the number “5”, and stores it in the counter *len*. After that, rule 4 and rule 5 will produce the rest string and update the counter until it meets enough characters, which be extracted as a value with the same id 0.

Note that DCCFG can resolve both character- and binary-based protocols. RegEx is natural for the character-based protocols, and DCCFG can use `action` to parse the binary-based protocols by shifting the current character. For example, we can specify an action like `[bit = cur_ch >> 7]`

Algorithm 1 Translate(G)

```
1: $G' \leftarrow$ Regularize($G$)
2: Push the start terminal $S$ of $G'$ into $Q$
3: while $Q$ contains non-terminals do
4:    $T \leftarrow$ Pop($Q$)
5:    if $T$ is terminal then
6:       Push $T$ into $Q$
7:    else
8:       for Each rule $R$ whose head is $T$ do
9:          for Each terminal/non-terminal $N$ in $R$ do
10:            Push $N$ into $Q$
11:            if $T$ is extractive and $T$ is $R$'s head then
12:               Set a start extraction token for $N$
13:            if $T$ is extractive and $T$ is $R$'s tail then
14:               Set an end extraction token for $N$
15:            if $T$ contains actions for counters then
16:               Set the actions for $N$
17:            if $T$ contains conditions for counters then
18:               Set the conditions for $N$
19: return $Q$
```

`<key,start,0,0>`[A-Z]`<key,end,0,0>`[0-9]
`<len:=getnum()><value,start,0,0><len:>0>`
[A-Za-z]*`<len:=reduce()><value,end,0,0>`

Figure 4: The DCRG translated from the DCCFG in Figure 3.

to obtain the first bit of the character, and an extraction token like `<value, 0, bit>` to extract that bit. In this way, DCCFG can support all protocols, as long as the protocol is octet-aligned. And to our best knowledge, all protocols are octet-aligned for easier transmission.

4.2 DCRG Translation

It is well known that parsing CFG is very expensive, since CFG maintains an unknown-length stack to track the derivation. CCFG extended from CFG can be translated into counting regular grammar (CRG), which can be parsed without a stack, bringing much faster speed [20]. We translate DCCFG into DCRG in a similar way. First, we regularize the DCCFG to remove the recursions. Second, we construct the DCRG from the non-recursive grammar by inheriting the counters and extraction tokens. The extraction tokens in DCRG are transformed into a new four-tuple, <`type, pos, id, app`>. The `type` and `id` can be inherited from the extraction tokens in DCCFG. The `pos` indicates this character is a *start* or an *end*. The `app` is an auto-generated unique id to distinguish different applications. Notice the `type` in DCRG does not include *key_named*, since no character carries such information. The *key_named* keys are stored independently and will not be used until pairing the keys and values.

Algorithm 1 demonstrates the pseudo-code of constructing DCRG from DCCFG, which regularizes the DCCFG firstly to remove the recursive cases, and inherits the counter and extraction tokens into the target DCRG. Figure 4 illustrates the DCRG translated from the DCCFG in Figure 3. The non-terminals are derived sequentially, and the extraction tokens are correctly set in their positions, as well as the counter conditions and actions.

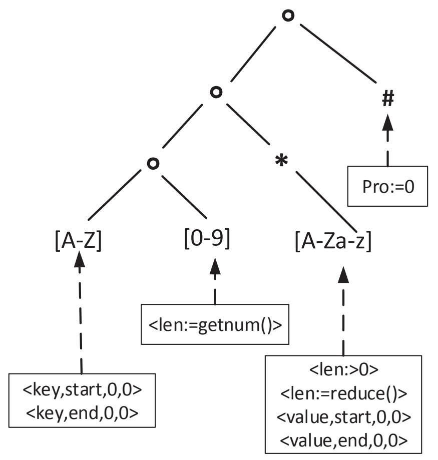


Figure 5: The DCAST translated from the DCRG in Figure 4.

1	S	$\rightarrow$ T L V
2	T	$\rightarrow$ <key,0,"[A-Z]">
3	L	$\rightarrow$ "[0-9]*" [len2=getnum()]
4	[len2>0]V	$\rightarrow$ <value,0,"[A-Za-z]*"> [len2=reduce()]
5	[len2=0]V	$\rightarrow \epsilon$

Figure 6: An alternative TLV specification Γ' in DCCFG.

5. DCA: MERGE TO ACCELERATE

5.1 Generate and Merge AST

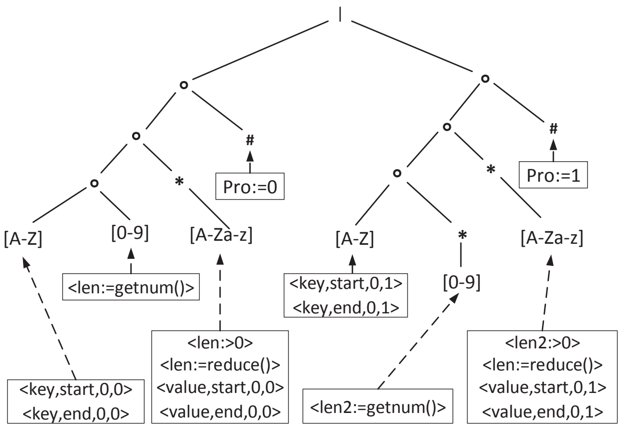
We have demonstrated that the parallel parsing needs a merged automaton from multiple protocol specifications. In this section, we firstly translate DCRG into its abstract syntax tree, called distinguishable counting abstract syntax tree (DCAST), and further merge multiple DCASTs, according to which the DCA can be generated.

The translation from DCRG to DCAST is similar to that from RG to AST [8], since the extraction tokens and the counters are attached with the characters, which are the leaf nodes in the target AST. Thus, we first ignore these extended information in DCRG, and construct the original AST. After that we attach the extraction tokens and counters in the related nodes. The DCAST translated from the DCRG in Figure 4 is illustrated in Fig 5. Notice that there is an end mark "#" indicating the protocol's unique id (0 in this case).

We use `OR` operation to merge multiple DCRGs. A sequence of DCRGs Θ_i can be rewritten into Θ' as $\Theta_1|\Theta_2|...|\Theta_n$, which has the equivalent semantics with the original sequence due to the uniqueness of the extraction tokens and counters. Consequently, we can also use `OR` node to merge multiple DCASTs. Consider an alternative TLV grammar Γ' shown in Figure 6, it requires none or multiple characters for L instead of one character in Γ. We build its DCAST and merge it with the DCAST in Figure 5 as depicted in Figure 7, where `app` in Γ' is 1 to distinguish it from Γ.

5.2 Generate DCA

Previous works have demonstrated that RegEx and DFA are equivalent [8]. Based on a given AST, we can obtain its DFA by computing the first and follow positions of each leaf



|
∘
#
Pro:=0
Pro:=1
*
[A-Z]
[0-9]
[A-Za-z]
<len:=getnum()>
<key,start,0,0> <key,end,0,0>
<len:>0> <len:=reduce()> <value,start,0,0> <value,end,0,0>
<key,start,0,1> <key,end,0,1>
<len2:=getnum()>
<len2:>0> <len:=reduce()> <value,start,0,1> <value,end,0,1>

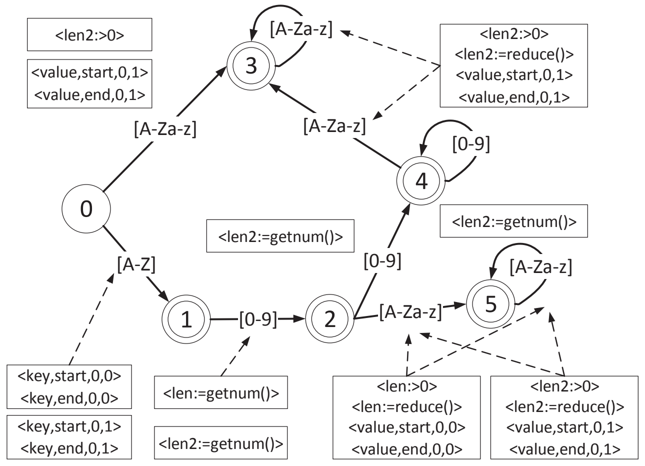
Figure 7: The merged DCAST from Γ and Γ'.

Algorithm 2 ConstructDCA(*root*)

```
1: $root \leftarrow$ The root node of the DCAST
2: $states \leftarrow$ firstpos($root$)
3: while $states$ contains non-tagged state $n$ do
4:     Tag $n$
5:     for Each input $m$ in $n$ do
6:         $i \leftarrow$ The node for $m$ in the DCAST
7:         $s \leftarrow \cup$followpos($i$)
8:         $r \leftarrow \cup$(Every app in $s$)
9:         $e \leftarrow \cup$(Every extraction token in $s$)
10:        $c \leftarrow \cup$(Every condition in $s$)
11:        $a \leftarrow \cup$(Every actions in $s$)
12:        if $s$ is not in $states$ then
13:            Untag $s$ and push $s$ into $states$
14:            $endsets[s] \leftarrow r$
15:        $trans[n, m] \leftarrow s$
16:        $extracts[n, m] \leftarrow e$
17:        $conds[n, m] \leftarrow c$
18:        $actions[n, m] \leftarrow a$
19: return $root$
```

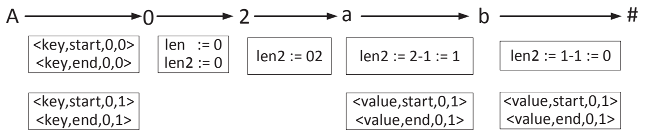
node [12]. The extraction tokens and counters in DCAST are independent of those positions, thus we can construct DCA from DCAST in a similar way but adding extended information. First, the extraction tokens and counters are only attached with the transitions in the automaton. The counters' conditions are checked before activating a transition, and the extraction is triggered if the conditions are true. After that, the counters' actions are executed accordingly. Thus, we need to map the extraction tokens and counters to the corresponding transitions. Besides, the original position automaton is not distinguishable: it does not keep the protocols' id and thus loses the end nodes' information [38]. The automation can only accept or reject the input, but cannot give the hit protocol id in some cases, which is important to determine the extraction. As a result, we need to separate the end nodes when building the position automaton.

The pseudo-code of constructing the DCA from DCAST is demonstrated in Algorithm 2. The `firstpos()` and `followpos()` functions calculate the related node positions, which are defined in position automaton algorithms [12]. After this process, all the transitions are stored in *trans* and the related conditions, actions and extraction tokens are stored in *conds*, *actions* and *extracts*, respectively, which can be paired with their indexes. And we save the end nodes' information in *endsets* to distinguish different protocols, where one node may contain multiple extraction/counter information across different protocols.



0
1
2
3
4
5
[A-Z]
[0-9]
[A-Za-z]
<len2:>0>
<value,start,0,1> <value,end,0,1>
<len2:>0> <len2:=reduce()> <value,start,0,1> <value,end,0,1>
<len2:=getnum()>
<key,start,0,0> <key,end,0,0>
<key,start,0,1> <key,end,0,1>
<len:=getnum()>
<len:>0> <len:=reduce()> <value,start,0,0> <value,end,0,0>

Figure 8: The DCA generated from the merged DCAST in Figure 7.



A
0
2
a
b
#
<key,start,0,0> <key,end,0,0>
<key,start,0,1> <key,end,0,1>
len := 0 len2 := 0
len2 := 02
len2 := 2-1 := 1
len2 := 1-1 := 0
<value,start,0,1> <value,end,0,1>

Figure 9: Parsing process with the input string `A02ab` in Figure 8's DCA

Take Figure 7 as an example, we use Algorithm 2 to calculate the transitions and other information. The generated DCA is illustrated in Figure 8. All the extraction and counter information are attached with the transitions, and we can easily obtain the protocols' end nodes as well: node 1 and node 4 are the end nodes for Γ'; node 2, node 3 and node 5 are the end nodes for both of Γ and Γ', which means if the DCA stops at these nodes, both protocols accept the input payload.

Complexity analysis. It has been proven the `first` and `follow` positions can be computed in $O(N^2)$ time, where N is the unique leaf nodes' number in AST [12]. Algorithm 2 further checks the extraction tokens and counters in each node, making the complexity $O((|extracts| + |conds| + |actions|)N + N^2)$, where the denotation "$|\cdot|$" means the size of the operand. Since $|extracts|$, $|conds|$ and $|actions|$ are limited to much smaller constants compared with N, it still needs quadratic time to construct the DCA from DCAST. However, the growth of DCCFGs does not linearly increase the unique leaf nodes because of the very similarities between multiple DCCFGs. As a result, the real cost is far lower than $O(M^2)$ (M is the number of DCCFGs), which will be shown in our experiments in §7.

6. PARSING WITH DCA

There are two stages when parsing with DCA: (1) parse the input payload to get the accepted protocol(s), and (2) filter the extraction tokens obtained through the parsing and give the final extraction result. In the first stage, DCA jumps to the proper states by checking the current input

Algorithm 3 Parse()

```
1: $s \leftarrow$ The start state of DCA
2: $c \leftarrow$ The first character of the input payload
3: $pos \leftarrow 0$
4: while $c$ is not $null$ do
5:     $pos \leftarrow pos + 1$
6:     if $trans[s, c]$ is $null$ or $conds[s, c]$ is $false$ then
7:         return Not acceptable
8:     if $extracts[s, c]$ is not $null$ then
9:         for Each extraction token $ef$ in $extracts[s, c]$ do
10:            push $ef$ into $e$
11:            push $pos$ into $p$
12:    if $actions[s, c]$ is not $null$ then
13:        Execute $actions[s, c]$
14:    $s \leftarrow trans[s, c]$
15:    $c \leftarrow$ next character in the input.
16: if $endsets[s]$ is $null$ then
17:     return Not acceptable
18: $M \leftarrow endsets[s]$
```

Algorithm 4 Extract()

```
1: $i \leftarrow 0$
2: for Each $ef$ in $e$ do
3:     $i \leftarrow i + 1$
4:     $r \leftarrow$ The app of $ef$
5:     if $r$ is not in $M$ then
6:         Continue
7:     $n \leftarrow$ The id of $ef$
8:     $t \leftarrow$ The type of $ef$
9:     if $t$ is $start$ then
10:        $E[r][n][0] \leftarrow p[i]$
11:    if $t$ is $end$ then
12:        $E[r][n][1] \leftarrow p[i]$
```

and the counters' conditions, and stops when the input is accepted or rejected. Meanwhile, it records all the extraction tokens and their positions in the input, which will be used in the second stage. We use M to denote the accepted protocols, e and p to denote the extraction tokens and their positions. Algorithm 3 demonstrates the first parsing stage to get these values. Taking Figure 8 as an example, assume an input string `A02ab`, the DCA activates node 0, node 1, node 2, node 4, node 3, sequentially, and finally stops at node 3. Thus, $M = endsets(3) = 1$, which means this input is accepted by Γ' only. We can further compute e and p from the node path. Figure 9 shows the parsing process with this input. We can see that *len* controls the length of the value part. The DCA saves 8 extraction tokens in total, including ones from Γ and Γ'. It also saves their positions in the input string.

After the first stage, we get all the extraction tokens and their positions. We can further filter these tokens according to M. The idea is to drop all the tokens whose `app` is not in M, and pair the rest to construct a key-value map. Algorithm 4 illustrates the second stage to obtain the set of the final extraction tokens, denoted as E. Taking Figure 9 as an example, we have obtained $M = 1$ and eight extraction tokens with their positions. For Γ' is the accepted protocol, all the extraction tokens whose `app` is 0 should be ignored. And the rest extraction tokens would be paired by their *id*. Notice the extraction tokens with the same `type` and `id` should be merged as one extraction. Therefore, the final extraction tokens are paired as follows: {<key,start,0,1>,<key,end,0,1>} in position {1,1}, {<value,start,0,1>,<value,end,0,1>} in position {4,5}. The DCA identifies the input string as Γ', and extracts {A, ab} according to its DCCFG correctly.

Complexity analysis. The time complexity of Algorithm 3 is $O(L + |conds| + |actions|)$, where L is the input length, $|conds|$ and $|actions|$ are the size of *conds* and *actions*, respectively. The time complexity of Algorithm 4 is $O(|E|)$. Since $|conds|$, $|actions|$ and $|E|$ can be bounded to much smaller constants compared with L in practice, the parsing complexity remains $O(L)$.

7. PERFORMANCE EVALUATION

7.1 Experimental Settings

7.1.1 Methodologies

In this section, we compare the performance of COPY with related works in three aspects: accuracy, parsing throughput, and memory consumption. We use a piece of trace with known ground truth, *i.e.*, the protocol proportion, which is used to compare with the output of the prototypes to evaluate the accuracy. In throughput experiments, we define a new metric named "goodput" to evaluate the real performance of the involved approaches, which denotes the throughput with correct parsing result. In addition, we preload the testing traces without changing their segments' order into memory to remove the effect of packet capturing. In memory cost experiments, we define a static memory cost to denote the size of parsing structures such as DCA(s) and prior DFA, and a real-time memory cost to illustrate the dynamic memory used for the parsing.

We involve two parsing models for multiple protocols in the comparisons: SP in Figure 1(a), and PI in Figure 1(b). To the best of our knowledge, FlowSifter performs the highest throughput with expressive protocol specification [20], so we extend FlowSifter to SP and PI processing models. To fairly compare the parallel model embedded in COPY, we also build DCA-based parser with single DCCFG in the above models. In PI model, we combine all the RegExs in its weak library to accelerate the identification. We use `lexertl` [3] as the RegEx engine.

We implement the prototype of COPY with about 3,000 lines C++ code, and directly use the executive binary of FlowSifter provided by its authors. We perform all the experiments on a platform with Intel i7 920 (8-core 2.66GHz), 12GB memory and Linux 3.3 kernel. Note that while most of the experiments are performed with single thread, we have also implemented a multiple thread prototype of COPY using the model proposed in [22] to tap the potential of this approach. We use `libnids` [4] to reassemble packets in layer-4, which dispatches segments to multiple cores in multi-thread implementation.

7.1.2 Protocol specifications

We have investigated 38 protocols specifying various applications in L7 header or payload. We give their DCCFG forms as our protocol specifications. The average #lines of code of these specifications is 39.4. This is because many protocols share the common semantic of HTTP, which can be imported as a protocol library. This simple statistic

Table 1: The Selected Specifications and Extraction Tokens

L7 Protocol	Applications	Extractions
HTTP	Video: Youku, Youtube, *etc*	Name, Duration, Resolution
	SNS: Weibo, Facebook, *etc*	Account name, Repost count
	Online Shopping: Taobao, eBay, *etc*	Item name, Price
	App Market: AppStore, GooglePlay, *etc*	App name, Download count
	...	...
QQ	QQ	Sender/Receiver id
DNS	DNS	Domain name, Name server
Total	**38**	

Table 2: The Features of the Selected Real Traces

	Univ. trace	ISP trace
Time	15/12/2012	11/17/2013
Duration	58 min.	70 min.
Size	22GB	7.9GB
Avg. length	818B	566B

Table 3: The Proportion of the Tiny Trace

HTTP					QQ	DNS
Weibo	Youku	Tudou	Amazon	Taobao		
23.6%	11.4%	3.1%	1.4%	25.9%	30.2%	4.4%
Total: 100%						

echoes the demonstration in §1 that COPY can express complex protocols within tens of lines of code.

The extraction tokens in DCCFG depend on the purpose of the applications. For example, Youku-HTTP extracts the video name, duration and resolution, while Weibo-HTTP extracts the account name, repost and reply count. Table 1 summarizes the specification library and its major extraction information. Notice some protocols are defined in L7 header, some others are defined in L7 payload. DCCFG provides the flexibility to specify the applications in both layers. All the DCCFGs are translated and merged to be a combined DCA for the parsing. The translation and combination are pre-computed, thus they will not impact the parsing throughput. More impact from the pre-computation will also be checked in this section. We also generate the corresponding CCFG specifications for FlowSifter-based approaches by removing the extraction tokens in DCCFGs. In the context of single protocol parsing, DCCFG and CCFG are of the same expressiveness.

In PI model, the accuracy of the whole system relies on the accuracy of the prior identification specifications, *i.e.*, the RegExs. However, more accurate RegExs may lead to a much larger DFA, consuming larger memory. Besides, such RegExs tend to check more payload, which aggravates the parsing redundancy as we have discussed in §2. To demonstrate the effect of the RegExs, we give two RegEx libraries according to our DCCFG library. The first one is coarse-grained from l7-filter [2], in which the RegExs only specify the very first bytes of the protocol, such as the request/response line in HTTP protocol. The second is much finer than the first one, that we give the RegExs by only removing the extraction tokens and counters in the corresponding DCRGs.

7.1.3 *Traces*

We use two traces to evaluate the prototypes, each of which contains all or partial above applications. One trace is captured from an edge switch of a university named as "Univ. trace", and the other comes from a mobile service provider named as "ISP trace". Table 2 lists the features of these two traces. We further give a tiny piece of trace with known application proportion named "Tiny trace", which is used in accuracy evaluation, as depicted in Table 3.

Table 4: The Involved Approaches in the Experiments

	SP	PI-Coarse	PI-Fine
FlowSifter	SP-FS	PI-FS-Coarse	PI-FS-Fine
DCA-based	SP-DCA	PI-DCA-Coarse	PI-DCA-Fine

Table 5: The Fault Ratios of the Four Approaches on Univ. trace

	SP-based	PI-Coarse	PI-Fine	COPY
HTTP	0.0%	7.1%	1.1%	0.0%
QQ	0.0%	22.2%	4.9%	0.0%
DNS	0.0%	8.4%	5.4%	0.0%
Overall	**0.0%**	**11.7%**	**2.4%**	**0.0%**

7.2 Experimental Results

As we mentioned above, we involve six approaches in the comparisons besides COPY. The denoting names and settings are depicted in Table 4.

7.2.1 *Accuracy*

We measure the accuracy by the fault ratio, which is defined as the percentage of the "mis-identified" packets. The packet is "mis-identified" if it is parsed by a misleading protocol parser or is ruled out by the prior identifier mistakenly. Notice that if the packets not related to any applications are sent to a misleading protocol parser, it will not produce any false positives, since the single protocol parser filters them and gives negative results. Such mis-identifications will only affect the throughput due to the aggravated parsing redundancy. In contrast, if the packets carrying useful information are dispatched to a wrong parser or are mistakenly ruled out, it will lose the traffic semantic and will impact the network. In our experiments for accuracy, we only count the second case to depict the real impact of the network.

The DCA-based parsers are of same expressiveness with FlowSifter parsers with the single protocol, thus we compare COPY with SP-based, PI-Coarse, and PI-Fine approaches. We use the Tiny trace shown in Table 3 as the input. Table 5 demonstrates the fault ratios compared with the ground truth, where we cluster the applications by their L7 protocols to summarize the results.

According to the experimental results, COPY and SP precisely classifies every packet into the correct application, while PI-based approaches mis-identify 11.7% and 2.4% with

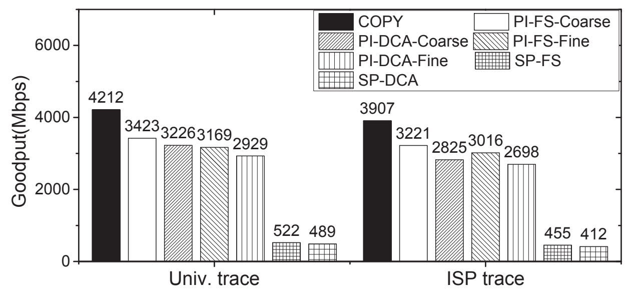


Figure 10: The goodput of seven prototypes on the two real traces.

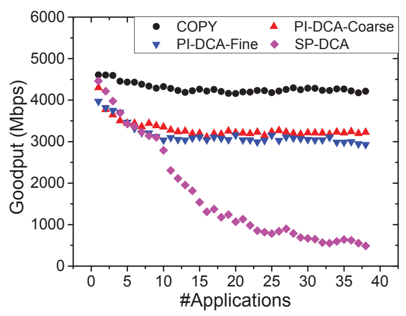


Figure 11: The scalability of COPY and other three prototypes on Univ. trace. To fairly compare our parallel parsing model with SP and PI, we only involve DCA-based prototypes.

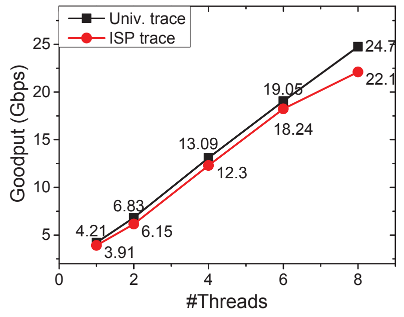


Figure 12: The goodput of COPY with multiple threads on the two real traces.

coarse-grained and fine-grained libraries, respectively. It is shown that SP is of equivalence semantic of the original protocol specifications, which is also assured in the DCA of COPY. On the contrary, PI-based approaches use incomplete regular expressions as the specification library, which inevitably brings identification error through parsing. In PI-Coarse-based approaches, most faults result from the QQ protocol, which specifies a TLV field in the very beginning of the packets. In addition, RegEx fails to handle the protocols that need binary parsing, *e.g.*, DNS. And this limitation is also the major cause to the mis-identifications in PI-Fine-based approaches. The fault ratios of the prior identifier would be even higher if single protocol parsers do not filter the possible false positives. In SDN, the parsing component guides the matching process and actions of the real packets in the core of the network, so the mis-identifications will significantly impact the real network there and then.

7.2.2 *Parsing goodput*

As we have mentioned in §2, SP- and PI-based approaches actually trade off the speed and accuracy. To compare the real parsing speed of the involved approaches, we define a new metric named "goodput" instead of the throughput. The goodput is defined as the total parsed bits of the payload with correct parsing result divided by the used time. This metric removes the effect of the trade-off between the speed and accuracy, showing the factual parsing speed of the system. In practice, we can get the goodput for approach i in the following way:

$$goodput_i = \frac{TotalBits - DiffBitsSP(i)}{time_i} \tag{1}$$

$DiffBitsSP(i)$ calculates the #bits with different parsing results between SP-based approaches and approach i. For the accuracy experiments suggest the SP-based approaches can accurately resolve every protocol, we use them as the baseline in goodput calculation.

Figure 10 demonstrates the goodput of the seven approaches on the two real traces. The results show that COPY is the fastest one, which achieves 4.2Gb/s and 3.9Gb/s on Univ. trace and ISP trace, respectively. Not surprisingly, the SP-based approaches are the slowest, since they try all protocol parsers (38 in total) until one/none of them is accepted. The PI-based approaches benefit from the linear-complex prior DFA on the input and achieve much faster goodput than SP-based ones. However, it is also affected by the RegEx libraries. To be specific, all PI-based approaches are slower than COPY because of their parsing redundancy, as depicted in Figure 1(b). Besides, RegEx-based specification may mislead the protocol identifier to parse the meaningless packets, which increases $DiffBitsSP$ in Eq (1), and lowers the goodput. Due to the similar reason, PI-Coarse-based approaches are only slightly faster than PI-Fine-based approaches, since they produce much more false results. On the other aspect, FlowSifter parsers are slightly faster than DCA-based parsers with the same configurations, because FlowSifter parsers do not filter extraction tokens across multiple protocols. Generally, COPY using the combined parsing structure avoids the parsing redundancy and performs better than others. The gap of parsing speed would be enlarged when the specifications get more complex.

To test the scalability of the goodput, we perform these approaches with the shifting numbers of applications (1 to 38). We only involve DCA-based approaches to exclude other factors impacting the scalability. The experimental results of the two real traces show similar trends, and we only use the result of Univ. trace due to the page limitation, as depicted in Figure 11. In the figure, COPY scales with the number of applications very well, keeping its goodput while more applications are involved. It mostly benefits from the combined DCA with the linear-complexity on the input length. PI-based approaches also show their scalability because the combined RegEx is also liner-complex on the input. In contrast, the goodput of SP-DCA drops significantly with more applications.

To tap the potential of COPY, we further implement an eight-thread prototype and evaluate it with the above traces. The results show that COPY achieves 24.7Gb/s and 22.1Gb/s on the two real traces, respectively, which approves the feasibility of deploying such approach in the real network. Figure 12 shows the goodput trend with the growth of the threads used in the prototype.

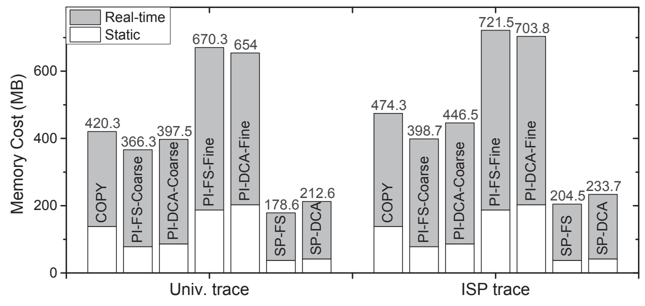


Figure 13: The memory costs on the seven approaches in stacked columns.

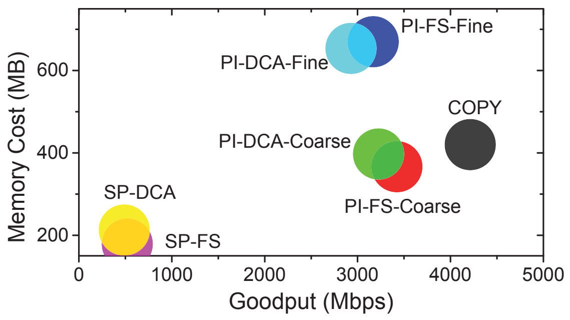


Figure 14: The bubble chart of the seven approaches. COPY's bubble lies in the bottom right corner, striking the best performance-to-cost ratio among all the approaches.

7.2.3 *Memory consumption*

We classify the memory cost into two catalogs. One is to keep the parsing structures, such as the DCA(s), CAs (for FS-based approaches) and the prior identifier, *i.e.*, the DFA constructed by the weak specifications in PI-based approaches. This memory cost is called "static memory consumption". The other is used for maintaining the intermediate results or conditions through the parsing, such as DCA's or CA's current state, and its counters and extraction tokens, called "real-time memory consumption". We evaluate both memory costs of the seven prototypes on the aforementioned two real traces. Figure 13 shows that COPY occupies slightly more memory with PI-based approaches using coarse-grained RegEx library, but much less than the ones using fine-grained RegEx library. It is because COPY combines the DCCFGs into one big DCA, which could use more static memory than building them independently. On the other hand, PI-based approaches build single protocol parsers separately, but employ a combined DFA. The fine-grained RegExs will bring much larger DFA than the coarse-grained ones, due to their similarities to the DCRGs used in COPY. When parsing in real time, COPY does not need to maintain the conditions of the prior DFA, and uses less real-time memory. No extra space is needed in SP-based approaches, since they do not construct any new structure or combine the existing ones. That is why SP-based approaches consume the least memory in both static and real-time cases.

7.2.4 *Comprehensive comparison*

To comprehensively compare the involved approaches, we draw a bubble chart to demonstrate all the metrics in Figure 14. The horizontal axis represents the goodput, the vertical axis denotes the memory cost. Note that the goodput metric takes the factor of the accuracy into account, thus these two metrics can show a big picture in every aspect. COPY occupies the bottom right corner, which means it performs the highest goodput with relative small memory, striking the best performance-to-cost ratio.

In fact, COPY's benefits are achieved by the merged DCA, which embeds the distinguishable information of the protocols and can be pre-constructed off-line, bringing extra time and space of construction. On the contrary, other approaches leveraging PI and SP model avoid such off-line overhead and distinguish the protocols on-line. In other words, COPY compromises between the off-line time and space for higher real-time processing speed. To the time cost, as discussed in §5.2, the construction costs quadratic time on the node numbers, but the application numbers and the node numbers are not linear-related. To the space cost,

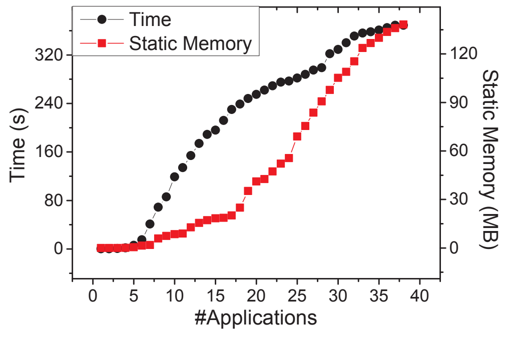


Figure 15: The time and memory cost trends of building DCA in COPY.

the COPY's static memory consumption is determined by the similarity of the protocols: more similar protocols lead to a smaller DCA. In practice, most applications are with the same L7 protocols (HTTP), and this feature reduces the risk of high memory usage. The trends of the above two costs with the application growth is depicted in Figure 15, which shows that COPY would not consume unacceptable time and memory with more applications in practice.

8. RELATED WORK AND DISCUSSIONS

SDN implementation. The latest OpenFlow specification defines 41 match fields that determine the corresponding actions, all of which are in L2-L4 [23]. To parse these fields in the fixed positions, hardware approaches such as NetFGPA [6, 21] and ONetSwitch [16] and software approaches such as CPqD [1], LINC [5] and trema [7] identify the protocols and extract the values by counting the offsets in the packets, which are not sufficient in flexibly-defined L7 applications.

Recent works propose the programmable parser, *a.k.a.*, protocol-independent parser, such as P4 [10] and PoF [30]. However, it is hard and expensive to achieve the high-layer programmable parser, since their implementations rely on specialized hardware [11, 14, 31].

Many languages for the network are proposed to ease composing SDN [13, 15, 32, 35, 36], whose flexibility is limited by the current parser. They need to update their languages to specify more actions in higher layers.

L7 Appliances. Automated parsing technique has been proved more accurate than hand-coded parsing [17, 28]. GAPA defines a protocol language which is type-safe and recursionfree [9]. Binpac uses recursive descent method to parse a pre-constructed AST [24]. Ultrapac improves Binpac's performance by selectively parsing the leaf node in the AST [18]. FlowSifter is the closest work to COPY, which proposes CCFG and CA to translate the counting sensitive grammar into a corresponding automaton, improving the parsing speed on a single specification [20]. CCFG is much like DCCFG but lacks the distinguishable extraction information, which is the key to support parallel parsing ability. Additionally, simple extensions such as SP and PI cannot satisfy the requirements in SDN as shown in §7.

On the other hand, many works propose high performance architectures for traffic classification and intrusion detection, which has achieved over 10Gb/s throughput with GPU acceleration or on a commodity platform [19, 27, 33, 34]. The contribution of these approaches is the elaborately designed architectures/pipelines, instead of the core processing component, *i.e.*, classification and detection component, which is insufficient for fine-grained control in SDN. COPY can gain more improvements if implemented with such architectures as a hybrid approach.

Discussions. There is a debate that high level parsing should be handled in the edge by middleboxes, leaving the core network to push packets only. Though several techniques have been proposed to ease the deployment of middleboxes in the edge, directly embedding parser into the data plane of SDN is obviously more flexible and fine-grained. Besides, the throughput of a board router can also reach tens of Gb/s, and to the best of our knowledge, few middlebox can satisfy such high speed in the context of application layer parsing. COPY scales with the #specifications and the #cores (Figure 11, 12, 15), and costs much lower price and power comparing with hardware-based approach, standing the best chance to realize the content parsing in SDN. Generally, no matter from where, efficiently grasping the semantics from the traffic helps the control plane to better manage SDN, and our motivation in §1 still holds.

Writing a protocol specification takes lots of manual efforts. There have been some algorithms to extract the protocol format from the training traffic [25, 37], yet none of them can generate the fine-grained specifications like DCCFG. The number and accuracy of the protocol specification determine the capacity of the parser and the whole SDN. We will look into this question in our future work.

Encrypted traffic increases continuously in today's Internet for better security performance, which obstructs the fine-grained identification on the packets. However, we argue that the content parsing is still necessary and feasible for SDN. Generally, content providers encrypt their traffic for the security sensitive connections only to save the computing resources, *e.g.*, connections of register, login, and purchase. Other connections, especially the heavily loaded ones with image and video content remain unencrypted. Even all these traffic is encrypted, in since the traffic has been decrypted in the gateway, so SDN can still parses the content of the packets inside content provides' own data center networks.

9. CONCLUSION

In this paper, we have proposed COPY as the content parser for SDN in the application layer instead of L2-L4 headers. COPY employs DCCFG to specify the applications and generates DCA to distinguish protocols without on-line overhead. In this way, COPY achieves fast and scalable parsing with accurate semantics. We have implemented the prototype of COPY and evaluated it on real traces. The experimental results demonstrate better performance of COPY than related works with an accepted and scalable off-line cost.

10. ACKNOWLEDGMENTS

This paper is supported by the 863 plan (2013AA013501), the NSFC (No. 61272459), Program for New Century Excellent Talents in University, Jiangsu Future Internet Innovation Project (BY2013095-1-12) and CETC 54 project (ITD-U14001/KX142600008).

11. REFERENCES

[1] CPqD OpenFlow 1.3 switch. `https://github.com/CPqD/ofsoftswitch13`.

[2] l7-filter. `http://l7-filter.clearfoundation.com/`.

[3] lexertl. `http://www.benhanson.net/lexertl.html`.

[4] Libnids. `http://libnids.sourceforge.net/`.

[5] LINC. `https://github.com/FlowForwarding/LINC-Switch`.

[6] NetFPGA. `https://www.netfpga.org`.

[7] trema. `https://github.com/trema/trema`.

[8] A. V. Aho. *Compilers: Principles, Techniques and Tools (for Anna University), 2/e*. Pearson Education India, 2003.

[9] N. Borisov, D. J. Brumley, and H. J. Wang. A generic application-level protocol analyzer and its language. In *In 14th Annual Network & Distributed System Security Symposium*, 2007.

[10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[11] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013*, pages 99–110, New York, NY, USA, 2013. ACM.

[12] A. Brggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120:87–98, 1996.

[13] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, pages 279–291, New York, NY, USA, 2011.

[14] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '13, pages 13–24, Piscataway, NJ, USA, 2013. IEEE Press.

[15] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical declarative network management. In *Proceedings of the 1st ACM*

Workshop on Research on Enterprise Networking, pages 1–10, New York, NY, USA, 2009. ACM.

[16] C. Hu, J. Yang, H. Zhao, and J. Lu. Design of all programable innovation platform for software defined networking. In *Presented as part of the Open Networking Summit 2014*, Santa Clara, CA, 2014.

[17] A. Kumar, V. Paxson, and N. Weaver. Exploiting underlying structure for detailed reconstruction of an internet-scale event. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 33–33. USENIX Association, 2005.

[18] Z. Li, G. Xia, H. Gao, Y. Tang, Y. Chen, B. Liu, J. Jiang, and Y. Lv. Netshield: massive semantics-based vulnerability signature matching for high-speed networks. In *ACM SIGCOMM*, pages 279–290, New York, NY, USA, 2010. ACM.

[19] Y.-s. Lim, H.-c. Kim, J. Jeong, C.-k. Kim, T. T. Kwon, and Y. Choi. Internet traffic classification demystified: On the sources of the discriminative power. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 9:1–9:12, New York, NY, USA, 2010. ACM.

[20] C. Meiners, E. Norige, A. Liu, and E. Torng. Flowsifter: A counting automata approach to layer 7 field extraction for deep flow inspection. In *INFOCOM, 2012 Proceedings IEEE*, pages 1746 –1754, march 2012.

[21] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an openflow switch on the netfpga platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 1–9, New York, NY, USA, 2008. ACM.

[22] T. Nelms and M. Ahamad. Packet scheduling for deep packet inspection on multi-core architectures. In *ANCS 2010*, pages 1 –11, oct. 2010.

[23] ONF. Openflow switch specification version 1.4.0. In *Open Networking Founddation*, 2012.

[24] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: a yacc for writing application protocol parsers. In *IMC 2006*, IMC '06, pages 289–300, New York, NY, USA, 2006. ACM.

[25] B.-C. Park, Y. Won, M.-S. Kim, and J. Hong. Towards automated application signature generation for traffic identification. In *NOMS 2008*, pages 160–167, 2008.

[26] V. Paxson. Bro: a system for detecting network intruders in real-time. In *USENIX Security Symposium*, pages 3–3, Berkeley, CA, USA, 1998.

[27] P. M. Santiago del Rio, D. Rossi, F. Gringoli, L. Nava, L. Salgarelli, and J. Aracil. Wire-speed statistical classification of network traffic on commodity hardware. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 65–72, New York, NY, USA, 2012. ACM.

[28] N. Schear, D. R. Albrecht, and N. Borisov. High-speed matching of vulnerability signatures. In *in Proc. RAID, 2008*, pages 155–174.

[29] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The middlebox manifesto: Enabling innovation in middlebox deployment. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 21:1–21:6, New York, NY, USA, 2011. ACM.

[30] H. Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 127–132, New York, NY, USA, 2013. ACM.

[31] H. Song, J. Gong, and H. Chen. Coherent sdn forwarding plane programming. In *Presented as part of the Open Networking Summit 2014 (ONS 2014)*, Santa Clara, CA, 2014. USENIX.

[32] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 213–226, New York, NY, USA, 2014. ACM.

[33] G. Szabó, I. Gódor, A. Veres, S. Malomsoky, and S. Molnár. Traffic classification over gbit speed with commodity hardware. *IEEE J. Communications Software and Systems*, 5, 2010.

[34] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. Midea: A multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 297–308, New York, NY, USA, 2011. ACM.

[35] A. Voellmy, H. Kim, and N. Feamster. Procera: A language for high-level reactive network control. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 43–48, New York, NY, USA, 2012. ACM.

[36] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying sdn programming using algorithmic policies. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 87–98, New York, NY, USA, 2013. ACM.

[37] Y. Wang, X. Yun, M. Z. Shafiq, L. Wang, A. X. Liu, Z. Zhang, D. Yao, Y. Zhang, and L. Guo. A semantics aware approach to automated reverse engineering unknown protocols. In *ICNP 2012*, pages 1–10, Washington, DC, USA, 2012.

[38] G. Xia, X. Wang, and B. Liu. Srd-dfa: Achieving sub-rule distinguishing with extended dfa structure. In *Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing*, pages 723 –728,