High-Speed Application Protocol Parsing and Extraction for Deep Flow Inspection

Alex X. Liu, Chad R. Meiners, Eric Norige, and Eric Torng

***Abstract*—In this paper, we propose FlowSifter, a framework for automated online application protocol field extraction. FlowSifter is based on a new grammar model called *Counting Regular Grammars* (CRG) and a corresponding automata model called *Counting Automata* (CA). The CRG and CA models add counters with update functions and transition guards to regular grammars and finite state automata. These additions give CRGs and CAs the ability to parse and extract fields from context sensitive application protocols. These additions also facilitate fast and stackless approximate parsing of recursive structures. These new grammar models enable FlowSifter to generate optimized Layer 7 field extractors from simple extraction specifications. We compare FlowSifter against both BinPAC and UltraPAC, which represent the state-of-the-art field extractors. Our experiments show that when compared to BinPAC parsers, FlowSifter runs more than 21 times faster and uses 49 times less memory. When compared to UltraPAC parsers, FlowSifter extractors run 12 times faster and use 24 times less memory.**

***Index Terms*—Deep flow inspection, L7 field extraction, content-aware policy control.**

I. Introduction

A. Motivation

CONTENT inspection is the core of a variety of network security and management devices and services such as Network Intrusion Detection/Prevention Systems (NIDS/NIPS), load balancing, and traffic shaping. Currently, content inspection is typically achieved by Deep Packet Inspection (DPI), where packet payload is simply treated as a string of bytes and is matched against content policies specified as a set of regular expressions. However, such regular expression based content policies, which do not take content semantics into consideration, can no longer meet the increasing complexity of network security and management.

Semantic-aware content policies, which are specified over application protocols fields, i.e., Layer 7 (L7) fields, have been increasingly used in network security and management devices and services. For example, a load balancing device would like to extract method names and parameter fields from flows (carrying SOAP [1] and XML-RPC [2] traffic for example) to determine the best way to route traffic. A network web traffic analysis tool would extract message length and content type fields from HTTP header fields to gain information about current web traffic patterns. Another example application that demonstrates the need for and the power of semantic-aware content policies is vulnerability-based signature checking for detecting polymorphic worms in NIDS/NIPS. Traditionally, NIDS/NIPS use exploit-based signatures, which are typically specified as regular expressions. The major weakness of an exploit-based signature is that it only recognizes a specific implementation of an exploit. For example, the following exploit-based signature for Code Red, *urlcontent:"ida?NNNNNNNNNNN…"*, where the long string of Ns is used to trigger a buffer overflow vulnerability, fails to detect Code Red variant II where each N is replaced by an X. Given the limitations of exploit-based signatures and the rapidly increasing number of polymorphic worms, vulnerability-based signatures have emerged as effective tools for detecting zero-day polymorphic worms [3]–[6]. As a vulnerability-based signature is independent of any specific implementation, it is hard to evade even for polymorphic worms. Here is an example vulnerability-based signature for Code Red worms as specified in Shield [3]: $c = MATCH\_STR\_LEN(\gg P\_Get\_Request.URI, \text{“}id[aq]\backslash?(.^*)\$\text{"}, limit);\ IF\ (c > limit)\ \#\ Exploit!$. The key feature of this signature is its extraction of the string beginning with “ida?” or “idq?”. By extracting this string and then measuring its length, this signature is able to detect any variant of the Code Red polymorphic worm. A further use of field extraction is to detect malformed SIP messages [7], [8].

We call the process of inspecting flow content based on semantic-aware content policies Deep Flow Inspection (DFI). DFI is the foundation and enabler of a wide variety of current and future network security and management services such as vulnerability-based malware filtering, application-aware load balancing, network measurement, and content-aware caching and routing. The core technology of DFI is L7 field extraction, the process of extracting the values of desired application (Layer 7) protocol fields.

Manuscript received December 31, 2013; revised May 12, 2014; accepted June 3, 2014. Date of publication September 18, 2014; date of current version November 26, 2014. This work was supported in part by the National Science Foundation under Grants CNS-1017588, CNS-1017598, CCF-1347953, and CNS-1318563, the National Natural Science Foundation of China under Grants 61472184 and 61321491, and in part by a research gift from Cisco Systems, Inc. The preliminary version of this paper titled “FlowSifter: A Counting Automata Approach to Layer 7 Field Extraction for Deep Flow Inspection” was published in the proceedings of the 31th INFOCOM Conference, Orlando, Florida, March 2012, pages 1746–1754.

A. X. Liu, E. Norige, and E. Torng are with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA (e-mail: alexliu@cse.msu.edu; norigeer@cse.msu.edu; torng@cse.msu.edu).

C. R. Meiners was with Michigan State University, East Lansing, MI 48824 USA. He is now with MIT Lincoln Laboratory, Lexington, MA 02421 USA (e-mail: chad.meiners@ll.mit.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2014.2358817

B. Problem Statement

In this paper, we aim to develop a high-speed online L7 field extraction framework, which will serve as the core of next generation semantic aware network security and management devices. Such a framework needs to satisfy the following six requirements. First, it needs to parse at high-speed. Second, it needs to use a small amount of memory per flow so that the framework can run in SRAM; otherwise, it will be running in DRAM, which is hundreds of times slower than SRAM.

Third, such a framework needs to be *automated*; that is, it should have a tool that takes as input an extraction specification and automatically generates an extractor. Hand-coded extractors are not acceptable because each time when application protocols or fields change, they need to be manually written or modified, which is slow and error prone.

Fourth, because of time and space constraints, any such framework must perform *selective parsing*, i.e., parsing only relevant protocol fields that are needed for extracting the specified fields, instead of *full parsing*, i.e., parsing the values of every field. Full protocol parsing is too slow and is unnecessary for many applications such as vulnerability-based signature checking because many protocol fields may not be referenced in given vulnerability signatures [5]. Selective parsing that skips irrelevant data leads to faster parsing with less memory. To avoid full protocol parsing and improve parsing efficiency, we want to dramatically simplify parsing grammars based on extraction specification. *We are not aware of existing compiler theory that addresses this issue*.

Fifth, again because of time and space constraints, any such framework must support *approximate protocol parsing* where the actual parser does not parse the input exactly as specified by the grammar. While *precise parsing*, where a parser parses input exactly as specified by the grammar, is desirable and possibly feasible for end hosts, it is not practical for high-speed network based security and management devices. First, precise parsing for recursive grammars is time consuming and memory intensive; therefore, it is not suitable for NIDS/NIPS due to performance demand and resource constraints. Second, precise parsers are vulnerable to Denial of Service (DoS) attacks as attackers can easily craft an arbitrarily deep recursive structure in their messages and exhaust the memory used by the parser. However, it is technically challenging to perform approximate protocol parsing. Again, since most existing compiler applications run on end hosts, *we are not aware of existing compiler theory that addresses this issue*.

At last, any such framework must be able to parse application protocols with field length descriptors, which are fields that specify the length of another field. An example field length descriptor is the HTTP Content-Length header field, which gives the length of the HTTP body. Field length descriptors cannot be described with CFG [9], [10], which means that CFGs are not expressive enough for such framework.

C. Limitations of Prior Art

To the best of our knowledge, there is no existing online L7 field extraction framework that is both automated, selective and approximate. The only automated and selective solution is Zebu [11], but it is not approximate. Furthermore, *prior work on approximate protocol parsing is too inaccurate*. To reduce memory usage, Moscola *et al.* proposed ignoring the recursive structures in a grammar [12]–[14]. This crude approximate parsing is not sufficient because recursive structures often must be partially parsed to extract the desired information; for example, the second field in a function call.

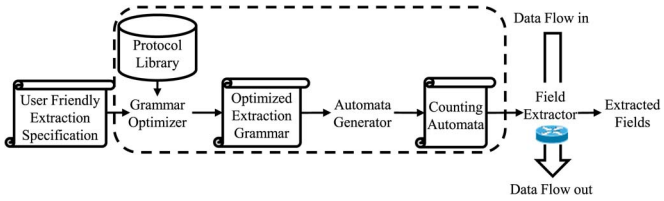


Fig. 1. FlowSifter architecture.

D. Proposed Approach

To address the above limitations of prior work on application protocol parsing and extraction, in this paper, we propose FlowSifter, an L7 field extraction framework that is *automated*, *selective*, and *approximate*. The architecture of FlowSifter is illustrated in Fig. 1. The input to FlowSifter is an extraction specification that specifies the relevant protocol fields that we want to extract values. FlowSifter adopts a new grammar model called *Counting Regular Grammars (CRGs)* for describing both the grammatical structures of messages carried by application protocols and the message fields that we want FlowSifter to extract. To our best knowledge, CRG is the first protocol grammar model that facilitates the automatic optimization of extraction grammars based on their corresponding protocol grammars. The extraction specification can be a partial specification that uses a corresponding complete protocol grammar from FlowSifter's built-in library of protocol grammars to complete its specification. Changing the field extractor only requires changing the extraction specification, which makes FlowSifter highly flexible and configurable. FlowSifter has three modules: *a grammar optimizer*, *an automata generator*, and *a field extractor*. The grammar optimizer module takes the extraction specification and the corresponding protocol grammar as its input and outputs an optimized extraction grammar, by which FlowSifter selectively parses only relevant fields bypassing irrelevant fields. The automata generator module takes the optimized extraction grammar as its input and outputs the corresponding *counting automaton*. The field extractor module applies the counting automaton to extract relevant fields from flows.

FlowSifter achieves low memory consumption by stackless approximate parsing. Processing millions of concurrent flows makes stacks an unaffordable luxury and a vulnerability for DoS attacks. For some application protocols such as XML-RPC [2], an attacker may craft its flow so that the stack used by the parser go infinitely deep until memory is exhausted. To achieve a controlled tradeoff between memory usage and parsing accuracy, we use a formal automata theory model, counting automata (CA), to support approximate protocol parsing.

CAs facilitate stackless parsing by using counters to maintain parsing state information. Controlling memory size allocated to counters gives us a simple way of balancing between memory usage and parsing accuracy. With this solid theoretical underpinning, FlowSifter achieves approximate protocol parsing with well-defined error bounds.

FlowSifter can be implemented in both software and hardware. For hardware implementation, FlowSifter can be implemented in both ASIC and Ternary Content Addressable Memory (TCAM). For ASIC implementation, FlowSifter uses a small, fast parsing engine to traverse the memory image of the CA along with the flow bytes, allowing real-time processing of a vast number of flows with easy modification of the protocol to be parsed. For TCAM implementation, FlowSifter encodes the CA transition tables into a TCAM table, thus allowing FlowSifter to extract relevant information from a flow in linear time over the number of bytes in a flow. FlowSifter uses optimization techniques to minimize the number of TCAM bits needed to encode the transition tables. Note that TCAM has already been installed on many networking devices such as routers and firewalls.

E. Limitations

FlowSifter, as presented here, is optimized for L7 protocol parsing, so it is not the best choice for layer 2–5 packet header parsing, which can be done efficiently with much simpler tools. FlowSifter is designed to process the packet payload in largely a byte-by-byte manner, so protocols that are not aligned to octets will also cause problems. To the authors' knowledge, all L7 protocols are aligned to octets, so this is unlikely to be a problem.

F. Key Contributions

In this paper, we make the following key contributions:

1) We propose the first L7 field extraction framework that is automated, selective, and approximate.
2) We propose for the first time to use Counting Context-Free Grammar and Counting Automata to support approximate protocol parsing. By controlling memory size allocated to counters, we can easily tradeoff between memory usage and parsing accuracy.
3) We propose efficient algorithms for optimizing extraction grammars.
4) We propose an algorithm for the automatic generation of stackless parsers from non-regular grammars.
5) Open source code available at https://github.com/MSU-SSL/FlowSifter.

II. Related Work

Prior work on application protocol parsing falls into four categories: (1) hand-coded, full, and precise parsing, (2) hand-coded, selective, and precise parsing, (3) automated, full, and precise parsing, and (4) automated, selective and precise parsing.

Hand-Coded, Full, and Precise Parsing: Although application protocol parsers are still predominantly hand coded [9], hand-coded protocol parsing has two major weaknesses in comparison with automated protocol parsing. First, hand-coded protocol parsers are hard to reuse as they are tightly coupled with specific systems and deeply embedded into their working environment [9]. For example, Wireshark has a large collection of protocol parsers, but none can be easily reused outside of Wireshark. Second, such parsers tend to be error-prone and lack robustness [9]. For example, severe vulnerabilities have been discovered in several hand-coded protocol parsers [15]–[20]. Writing an efficient and robust parser is a surprisingly difficult and error-prone process because of the many protocol specific issues (such as handling concurrent flows) and the increasing complexity of modern protocols [9]. For example, the NetWare Core Protocol used for remote file access has about 400 request types, each with its own syntax [9].

Hand-Coded, Selective, and Precise Parsing: Full protocol parsing is not necessary for many applications. For example, Schear *et al.* observed that full protocol parsing is not necessary for detecting vulnerability-based signatures because many protocol fields are not referenced in vulnerability signatures [5]. Based on such observations, Schear *et al.* proposed selective protocol parsing [5], which is three times faster than binpac. However, the protocol parsers in [5] are hand-coded and henceforth suffer from the weaknesses of hand-coded protocol parsing. Note that to perform field extraction, we first need to identify the exact application protocol of the flow under inspection. There are a variety of hand-coded, selective parsers that detect protocol type by recognizing patterns in the start of the payload. For example, NetFilter [21], Dreger *et al.* work [22], and TStat [23] each use hand-coded detection routines to identify a variety of different protocols. While their parsing strategies are appropriate for protocol identification, they do not extend to field extraction of complex protocols as they are hand-coded and precise.

Automated, Full, and Precise Parsing: Recognizing the increasing demand for application protocol parsers, the difficulty in developing efficient and robust protocol parsers, and the many defects of home-brew parsers, three application protocol parser generators have been proposed: binpac [9], GAPA [10], and UltraPAC [24]. Most network protocols are designed to be easily parsed by hand, but this often means their formal definitions turn out complex in terms of standard parsing representations. Pang *et al.*, motivated by the fact that the programming language community has benefited from higher levels of abstraction for many years using parser generation tools such as yacc [25] and ANTLR [26], developed the protocol parser generator BinPAC. GAPA, developed by Borisov *et al.*, focuses on providing a protocol specification that guarantees the generated parser to be type-safe and free of infinite loops. The similarity between BinPAC and GAPA is that they both use recursive grammars and embedded code to generate context sensitive protocol parsers. The difference between BinPAC and GAPA is that BinPAC favors parsing efficiency and GAPA favors parsing safety. BinPAC uses C++ for users to specify code blocks and compile the entire parser into C++ whereas GAPA uses a restricted and memory-safe interpreted

language that can be proven free of infinite loops. UltraPAC improves on BinPAC by replacing BinPAC's tree parser with a stream parser implemented using a state machine to avoid constructing the tree representation of a flow. UltraPAC inherits from BinPAC a low-level protocol field extraction language that allows additional grammar expressiveness using embedded C++ code. This makes optimizing an extraction specification extremely difficult, if not impossible. In contrast, FlowSifter uses high-level CA grammars without any inline code, which facilitates the automated optimization of protocol field extraction specifications. When parsing HTTP, for example, BinPAC and UltraPAC need inline C++ code to detect and extract the Content-Length field's value whereas FlowSifter's grammar can represent this operation directly. In addition, FlowSifter can automatically regularize non-regular grammars to produce a stackless approximate parser whereas an UltraPAC parser for the same extraction specification must be converted manually to a stackless form using C++ to embed the approximation. The Ragel framework [27] is another parsing engine that mixes code with regular expressions to produce a parser in a variety of output languages. It has developed an impressive list of features, but has no automated support for either selective parsing or approximate parsing. Further, non-regular grammatical features are only supported by manual development of embedded code to redirect the parsing state around where it would normally go. FlowSifter is able to automate these steps while natively supporting non-regular language features in its protocol grammar.

Automated, Selective, and Precise Parsing: Zebu [11] is a parser generator that also takes a protocol grammar and specification of what fields to extract and automatically builds a selective parser for just those fields. Zebu can do strict validation of message formats and compiles its parser to native C code. FlowSifter is different from Zebu in that it can handle recursive grammars through its use of Counting Automata. While Zebu is unable to handle any recursive structures in its protocols, FlowSifter can handle recursive structures. FlowSifter automatically builds an approximate parser for these recursive structures so that per-flow resource usage is constant.

III. Protocol and Extraction Specifications

FlowSifter produces an L7 field extractor from two inputs: a protocol specification, and an extraction specification. The protocol specification gives a grammar for parsing the full protocol of the flow being inspected. The extraction specification indicates what fields need to be extracted. Having a separate extraction specification allows the protocol specification to be reused in different applications that need different fields to be extracted. Both specifications are specified using a new grammar model called Counting Context Free Grammar (CCFG), which augments rules for context-free grammars with counters, guards, and actions. These augmentations increase grammar expressiveness, but still allows the grammars to be automatically simplified and optimized. In this section, we first formally define CCFG, and then explain how to write protocol specification and extraction specification using CCFG.

A. Counting Context Free Grammar

Formally, a counting context-free grammar is a five-tuple $\Gamma = (\mathbb{N}, \Sigma, \mathbb{C}, \mathbb{R}, S)$ where $\mathbb{N}$, Σ, $\mathbb{C}$, and $\mathbb{R}$ are finite sets of *nonterminals*, *terminals*, *counters*, and *production rules*, respectively, and S is the *start nonterminal*. The terminal symbols are those that can be seen in strings to be parsed. For L7 field extraction, this is usually a single octet. A *counter* is a variable with an integer value, which is initialized to zero. The counters can be used to store parsing information. For example, in parsing an HTTP flow, a counter can be used to store the value of the "Content-Length" field. Counters also provide a mechanism for eliminating parsing stacks.

A production rule is written as $\langle \text{guard} \rangle : \langle \text{nonterminal} \rangle \rightarrow \langle \text{body} \rangle$. The *guard* is a conjunction of unary predicates over the counters in $\mathbb{C}$, i.e., expressions of a single counter that return `true` or `false`. An example guard is $(c_1 > 2; c_2 > 2)$, which checks counters c_1 and c_2, and evaluates to `true` if both are greater than 2. If a counter is not included in a guard, then its value does not affect the evaluation of the guard. Guards are used to guide the parsing of future bytes based on the parsing history of past bytes. For example, in parsing an HTTP flow, the value of the "Content-Length" field determines the number of bytes that will be included in the message body. This value can be stored in a counter. As bytes are processed, the associated counter is decremented. A guard that checks if the counter is 0 would detect the end of the message body and allow a different action to be taken.

The *nonterminal* following the guard is called the *head* of the rule. Following it, the *body* is an ordered sequence of terminals and nonterminals, any of which can have associated actions. An empty body is written ϵ. An *action* is a set of unary update expressions, each updating the value of one counter, and is associated with a specific terminal or nonterminal in a rule. The action is executed after parsing the associated terminal or nonterminal. An example action in CCFG is $(c_1 := c_1 * 2; c_2 := c_2 + 1)$. If a counter is not included in an action, then the value of that counter is unchanged. An action may be empty, i.e., updates no counter. Actions in CCFG are used to write "history" information into counters, such as the message body size.

To make the parsing process deterministic, CCFGs require leftmost derivations; that is, for any body that has multiple nonterminals, only the leftmost nonterminal that can be expanded is expanded. Thus, at any time, production rules can be applied in only one position. We use leftmost derivations rather than rightmost derivations so that updates are applied to counters in the order that the corresponding data appears in the data flow.

B. Protocol Specification in CCFG

An application protocol specification precisely specifies the protocol being parsed. We use CCFG to specify application protocols. For example, consider the Varstring language consisting of strings with two fields separated by a space: a length field, B, and a data field, V, where the binary encoded value of B specifies the length of V. This language cannot be specified as a Context Free Grammar (CFG), it can be easily specified

1	$\mathrm{S} \rightarrow \mathrm{B\,V}$	
2	$\mathrm{B} \rightarrow$ '0' $(c := c * 2)\,\mathrm{B}$	
3	$\mathrm{B} \rightarrow$ '1' $(c := c * 2 + 1)\,\mathrm{B}$	
4	$\mathrm{B} \rightarrow$ '␣'	
5	$(c > 0)\ \mathrm{V} \rightarrow \Sigma\,(c := c - 1)\,\mathrm{V}$	
6	$(c = 0)\ \mathrm{V} \rightarrow \epsilon$	
	examples: "1 a", "10 ba", "101 xyzab"	
	(a)	

1	$\mathrm{S} \rightarrow \epsilon$
2	$\mathrm{S} \rightarrow \mathrm{I\,S}$
3	$\mathrm{I} \rightarrow$ '[' S ']'
	"[[]]", "[][[]]"
	(b)

Fig. 2. Two protocol specifications in CCFG. (a) Varstring Γ_v. (b) Dyck Γ_d.

Derivation	c	Rule #	Note
S	0	(1)	Decompose into len and body
B V	0	(3)	Produce '1', $c = 0 * 2 + 1$
1 B V	1	(2)	Produce '0', $c = 1 * 2$
10 B V	2	(4)	Eliminate B, c is the length of body
10␣V	2	(5)	Produce 'b', decrement c
10␣b V	1	(5)	Produce 'a', decrement c
10␣ba V	0	(6)	$c = 0$, so eliminate V
10␣ba	0		No nonterminals left, done

Fig. 3. Derivation of "10 ba" from the Varstring grammar in Fig. 2(a).

in CCFG as shown in Fig. 2(a). Note that a CFG is a CCFG without counters. The CCFG specification of another language called Dyck consisting of strings with balanced parentheses '[' and ']', is shown in Fig. 2(b). We adopt the convention that the head of the first rule, such as S in the Varstring and Dyck grammars, is the start nonterminal.

We now explain the six production rules in Varstring. The first rule $\mathrm{S} \rightarrow \mathrm{BV}$ means that a message has two fields, the length field represented by nonterminal B and a variable-length data field represented by nonterminal V. The second rule $\mathrm{B} \rightarrow$ '0'$(c := c * 2)\,\mathrm{B}$ means that if we encounter character '0' when parsing the length field, we double the value of counter c. Similarly, the third rule $\mathrm{B} \rightarrow$ '1'$(c := c * 2 + 1)\,\mathrm{B}$ means that if we encounter '1' when parsing the length field, we double the value of counter c first and then increase its value by 1. The fourth rule $\mathrm{B} \rightarrow$ '␣' means that the parsing of the length field terminates when we see a space. These three rules fully specify how to parse the length field and store the length in c. For example, after parsing a length field with "10", the value of the counter c will be $2(= ((0 * 2) + 1) * 2)$. Note that here "10" is the binary representation of value 2. The fifth rule $(c > 0) : \mathrm{V} \rightarrow \Sigma(c := c - 1)\mathrm{V}$ means that when parsing the data field, we decrement the counter c by one each time a character is parsed. The guard allows use of this rule as long as $c > 0$. The sixth rule $(c = 0) : \mathrm{V} \rightarrow \epsilon$ means that when $c = 0$, the parsing of the variable-length field is terminated.

We now demonstrate how the Varstring grammar can produce the string "10 ba". Each row of the table in Fig. 3 is a step in the derivation. The c column shows the value of the counter c at each step. The number in parentheses is the rule from Fig. 2(a) that is applied to get to the next derivation. Starting with the Varstring's start symbol, S, we derive the target string by replacing the leftmost nonterminal with the body of one of its production rules. When applying rule 5, the symbol Σ is shorthand for any character, so it can produce 'a' or 'b' or any other character.

C. Extraction Specification in CCFG

An extraction specification is a CCFG $\Gamma_x = (\mathbb{N}_x, \Sigma, \mathbb{C}_x, \mathbb{R}_x, S_x)$, which may not be complete. It can refer to nonterminals specified in the corresponding protocol specification denoted $\Gamma_p = (\mathbb{N}_p, \Sigma, \mathbb{C}_p, \mathbb{R}_p, S_p)$. However, Γ_x cannot modify Γ_p and cannot add new derivations for nonterminals defined in Γ_p. This ensures that we can approximate Γ_p without changing the semantics of Γ_x.

The purpose of FlowSifter is to call application processing functions on extracted field values. Based on the extracted field values, these application processing functions will take application specific actions such as stopping the flow for security purposes or routing the flow to a particular server for load balancing purposes. Calls to these functions appear in the actions of a rule. Application processing functions can also return a value back into the extractor to affect the parsing of the rest of the flow. Since the application processing functions are part of the layer above FlowSifter, their specification is beyond the scope of this paper. Furthermore, we define a shorthand for calling an application processing function f on a piece of the grammar $f\{\langle body \rangle\}$, where $\langle body \rangle$ is a rule body that makes up the field to be extracted. If a field is less than one byte, the extraction function called on that field can handle the process of extracting the target bits from the octet given to it.

Some applications may require the extraction of a composite field and some sub-fields within it. In such cases, we can first use FlowSifter to extract the composite field and then feed the extracted composite field to an application processing function that further extracts sub-fields from the composite field. If the grammar of the sub-fields is simple, then this application processing function could be a simple program customized for extracting these sub-fields. For example, if we want to extract both the URL of an HTTP header and the file name within the URL, we can first extract the URL and then feed it to a function that can quickly extract the file name from an URL. In rare cases that the grammar within the composite field is too complex, we can feed the extracted composite field to another instance of FlowSifter to further extract relevant sub-fields.

We next show two extraction specifications that are partial CCFGs, using the function-calling shorthand. The first, Γ_{xv} in Fig. 4(a), specifies the extraction of the variable-length field V for the Varstring CCFG in Fig. 2(a). This field is passed to an application processing function `vstr`. For example, given input stream "`101 Hello`", the field "`Hello`" will be extracted. This example illustrates several features. First, it shows how FlowSifter can handle variable-length field extractions. Second, it shows how the user can leverage the protocol library to simplify writing the extraction specification. The second extraction specification, Γ_{xd} in Fig. 4(b), is associated with the Dyck CCFG in Fig. 2(b) and specifies the extraction of the contents of the first pair of square parentheses; this field is passed to an application processing function named `param`. For example, given the input stream [[[]]][[]][], the [[]] will be extracted. This example illustrates how FlowSifter can extract specific fields within a recursive protocol by referring to the protocol grammar.

$1 \mid \mathrm{X} \to \mathrm{B}\ \texttt{vstr}\{\mathrm{V}\}$ (a)

$1 \mid \mathrm{X} \to \text{`['}\ \texttt{param}\{\mathrm{S}\}\ \text{`]'}\ \mathrm{S}$ (b)

Fig. 4. Two extraction CCFGs Γ_{xv} and Γ_{xd}. (a) Varstring Γ_{xv}. (b) Dyck Γ_{xd}.

IV. Grammar Optimization

In this section, we introduce techniques to optimize the input protocol specification and extraction specification so that the resulting CA uses less memory and runs faster.

A. Counting Regular Grammar

Just as parsing with non-regular CFGs is expensive, parsing with non-regular CCFGs is also expensive as the whole derivation must be tracked with a stack. To resolve this, FlowSifter converts input CCFGs to Counting Regular Grammars (CRGs), which can be parsed without a stack, just like parsing regular grammars. A CRG is a CCFG where each production rule is regular. A production rule is *regular* if and only if it is in one of the following two forms:

$$\langle \text{guard} \rangle : X \to \alpha\, [\langle \text{action} \rangle]\, Y \tag{1}$$

$$\langle \text{guard} \rangle : X \to \alpha\, [\langle \text{action} \rangle] \tag{2}$$

where X and Y are nonterminals and α is a terminal. CRG rules that fit (1) are the *nonterminating* rules whereas those that fit (2) are the *terminating* rules as derivations end when they are applied.

For a CCFG $\Gamma = (\mathbb{N}, \Sigma, \mathbb{C}, \mathbb{R}, S)$ and a nonterminal $X \in \mathbb{N}$, we use $\Gamma(X)$ to denote the CCFG subgrammar $\Gamma = (\mathbb{N}, \Sigma, \mathbb{C}, \mathbb{R}, X)$ with the nonterminals that are unreachable from X being removed. For a CCFG $\Gamma = (\mathbb{N}, \Sigma, \mathbb{C}, \mathbb{R}, S)$ and a nonterminal $X \in \mathbb{N}$, X is *normal* if and only if $\Gamma(X)$ is equivalent to some CRG.

Given the extraction CCFG Γ_x and L7 grammar Γ_p as inputs, FlowSifter first generates a complete extraction CRG $\Gamma_f = (\mathbb{N}_x \cup \mathbb{N}_p, \Sigma, \mathbb{C}_x \cup \mathbb{C}_p, \mathbb{R}_x \cup \mathbb{R}_p, S_x)$. Second, it prunes any unreachable nonterminals from S_x and their corresponding production rules. Third, it partitions the nonterminals in $\mathbb{N}_p$ into those we can guarantee to be normal and those we cannot. Fourth, for each nonterminal X that are guaranteed to be normal, FlowSifter regularizes X; for the remaining nonterminals $X \in \mathbb{N}_p$, FlowSifter uses counting approximation to produce a CRG that approximates $\Gamma_f(X)$. If FlowSifter is unable to regularize any nonterminal, it reports that the extraction specification Γ_x needs to be modified and provides appropriate debugging information. Last, FlowSifter eliminates idle rules to optimize the CRG. Next, we explain in detail how to identify nonterminals that are guaranteed to be normal, how to regularize a normal terminal, how to perform counting approximation, and how to eliminate idle rules.

B. Normal Nonterminal Identification

Determining if a CFG describes a regular language is undecidable. Thus, we cannot precisely identify normal nonterminals. FlowSifter identifies nonterminals in $\mathbb{N}_p$ that are guaranteed to be normal using the following sufficient but not necessary condition. A nonterminal $X \in \mathbb{N}_p$ is guaranteed to be normal if it satisfies one of the following two conditions:

1) $\Gamma_f(X)$ has only regular rules.
2) For the body of any rule with head X, X only appears last in the body and for every nonterminal Y that is reachable from X, Y is normal and X is not reachable from Y.

Although we may misidentify a normal nonterminal as not normal, fortunately, as we will see in Section IV-D, the cost of such a mistake is relatively low; it is only one counter in memory and some unnecessary predicate checks.

$$\begin{array}{l|l} 1 & \mathrm{S} \to \mathrm{B}' \\ 2 & \mathrm{B}' \to 0\,(c := c * 2)\,\mathrm{B}' \\ 3 & \mathrm{B}' \to 1\,(c := 1 + c * 2)\,\mathrm{B}' \\ 4 & \mathrm{B}' \to \sqcup\mathrm{V} \\ 5 & (c = 0)\ \ \mathrm{V} \to \epsilon \\ 6 & (c > 0)\ \ \mathrm{V} \to \Sigma\,(c := c - 1)\,\mathrm{V} \end{array}$$

Fig. 5. Varstring after decomposition of rule $\mathrm{S} \to \mathrm{BV}$.

C. Normal Nonterminal Regularization

In this step, FlowSifter replaces each identified normal nonterminal's production rule with a collection of equivalent regular rules. Consider an arbitrary non-regular rule

$$\langle guard \rangle : X \to \langle body \rangle.$$

We first express the body as $Y_1 \cdots Y_n$ where $Y_i, 1 \le i \le n$, is either a terminal or a nonterminal (possibly with an action). Because this is a non-regular rule, either Y_1 is a nonterminal or $n > 2$ (or both). We handle the cases as follows.

- If Y_1 is a non-normal nonterminal, Γ_x was incorrectly written and needs to be reformulated.
- If Y_1 is a normal nonterminal, we define CRG $\Gamma' = (\mathbb{N}', \Sigma, \mathbb{C}', \mathbb{R}', \mathbb{Y}'_{\not\vdash})$ to be $\Gamma_f(Y_1)$ where the nonterminals have been given unique names. We use Γ' to update the rule set as follows. First, we replace the rule $\langle guard \rangle : X \to \langle body \rangle$ with $\langle guard \rangle : X \to \mathbb{Y}'_{\not\vdash}$. Next, for each terminating rule $r \in \mathbb{R}'$, we create a new rule r' where we append $Y_2 \cdots Y_n$ to the body of r and add r' to the rule set; for each nonterminating rule $r \in \mathbb{R}'$, we add r to the rule set.
- If Y_1 is a terminal and $n > 2$, the rule is decomposed into two rules: $\langle guard \rangle : X \to Y_1 X'$ and $X' \to Y'_2 \cdots Y_n$ where X' is a new nonterminal.

The above regularization process is repeatedly applied until there are no non-regular rules.

For example, consider the Varstring CCFG Γ with non-regular rule $\mathrm{S} \to \mathrm{BV}$. As both $\Gamma(B)$ and $\Gamma(V)$ are CRGs, so S is a normal non-terminal. Decomposition regularizes $\Gamma(S)$ by replacing $\mathrm{S} \to \mathrm{BV}$ by $\mathrm{S} \to \mathrm{B}'$ and $\mathrm{B} \to \sqcup$ by $\mathrm{B}' \to \sqcup\mathrm{V}$. We also add copies of all other rules where we use B' in place of B. Fig. 5 illustrates the resulting rule set excluding unreachable rules. For example, the nonterminal B is no longer referenced by any rule in the new grammar. For efficiency, we remove unreferenced nonterminals and their rules after each application of regularization.

Theorem 4.1: Given a normal nonterminal X in grammar Γ, applying regularization to any rule $\langle guard\rangle : X \to Y_1 \cdots Y_n$ in Γ produces an equivalent grammar $\bar{\Gamma}$.

Proof: We define rule $r = \langle guard\rangle : X \to Y_1 \cdots Y_n$ as the rule in Γ that is replaced by other rules in $\bar{\Gamma}$. We consider two cases: Y_1 is a terminal, and Y_1 is a normal nonterminal.

For the case that Y_1 is a terminal, the only difference between Γ and $\bar{\Gamma}$ is that rule $r = \langle guard\rangle : X \to Y_1 \cdots Y_n$ is replaced by rules $r_1 = \langle guard\rangle : X \to Y_1 X'$ and $r_2 = X' \to Y_2 \cdots Y_n$ to get $Y_1 \cdots Y_n$. Consider any leftmost derivation with Γ that applies the rule r. We get an equivalent leftmost derivation with $\bar{\Gamma}$ that replaces the application of r with the application of rule r_1 immediately followed by the application of rule r_2 to produce the exact same result. Likewise, any leftmost derivation in $\bar{\Gamma}$ that applies rule r_1 must then immediately apply rule r_2 since X' is the leftmost nonterminal in the resulting string. We get an equivalent leftmost derivation with Γ by replacing the application of r_1 and r_2 with r. Finally, r_2 can never be applied except immediately following the application of r_1 because rule r_1 is the only derivation that can produce nonterminal X'. Therefore, Γ and $\bar{\Gamma}$ are equivalent.

For the case that Y_1 is nonterminal, rule $r = \langle guard\rangle : X \to Y_1 \cdots Y_n$ in Γ is replaced by rule $r_1 = \langle guard\rangle : X \to \mathbb{Y}'_{\not\Vdash}$ in $\bar{\Gamma}$. Furthermore, we add copies of all rules with head Y_1 that now have head Y_1' where all nonterminals are replaced with new equivalent nonterminals. This also applied to other nonterminals that are in the body of rules in Γ with head Y_1. Finally, for any terminating rule r_t in $\Gamma(Y_1)$, we add a rule r'_t where $Y_2 \cdots Y_n$ is appended to the body of r'_t and add r'_t to $\bar{\Gamma}$.

Consider any leftmost derivation with Γ that applies the rule r. We get an equivalent leftmost derivation with $\bar{\Gamma}$ as follows. First, we replace the application of r with the application of r_1. Next, until we reach a terminating rule, we replace each application of a rule with head Y_1 or other nonterminal in $\Gamma(Y_1)$ with the equivalent new rule using the new nonterminal names. Finally, we replace the application of terminating rule r_t with terminating rule r'_t. Now consider any leftmost derivation in $\bar{\Gamma}$ that applies rule r_1. This leftmost derivation must eventually apply some terminating rule r'_t. We get an equivalent leftmost derivation in $\bar{\Gamma}$ by replacing r_1 with r, r'_t with r_t, and intermediate rule applications with their original rule copies. We also note that no derivations in $\bar{\Gamma}$ can use any of the new rules without first invoking r_1 since that is the only path to reaching the new nonterminals. Therefore, Γ and $\bar{\Gamma}$ are equivalent. ■

D. Counting Approximation

For nonterminals in $\mathbb{N}_p$ that are not normal, we use counting approximation to produce a collection of regular rules, which are used to replace these non-normal nonterminals. For a non-normal nonterminal X, our basic idea is to parse only the start and end terminals for $\Gamma(X)$ ignoring any other parsing information contained within subgrammar $\Gamma(X)$. This approximation is sufficient because our extraction specification does not need to precisely parse any subgrammar starting at a nonterminal in $\mathbb{N}_p$. That is, we only need to identify the start and end of any subgrammar rooted at a nonterminal $X \in \mathbb{N}_p$. By using the counters to track nesting depth, we can approximate the parsing stack for such nonterminals. For nonterminals in the input extraction grammar, we require them to be normal. In practice, this restriction turns out to be minor, and when a violation is detected, our tool will give feedback to aid the user in revising the grammar.

$$\begin{array}{l|l}
1 & (cnt = 0)\ \mathrm{X} \to \epsilon \\
2 & (cnt \geq 0)\ \mathrm{X} \to start\,(cnt := cnt + 1)\,\mathrm{X} \\
3 & (cnt > 0)\ \mathrm{X} \to stop\,(cnt := cnt - 1)\,\mathrm{X} \\
4 & (cnt > 0)\ \mathrm{X} \to other\,\mathrm{X}
\end{array}$$

Fig. 6. General approximation structure.

$$\begin{array}{l|l}
1 & (cnt = 0)\ \mathrm{S} \to \epsilon \\
2 & (cnt \geq 0)\ \mathrm{S} \to \text{'['}\,(cnt := cnt + 1)\,\mathrm{S} \\
3 & (cnt > 0)\ \mathrm{S} \to \text{']'}\,(cnt := cnt - 1)\,\mathrm{S}
\end{array}$$

Fig. 7. Approximation of Dyck S.

Given a CCFG Γ_f with a nonterminal $X \in \mathbb{N}_p$ that does not identify as normal, FlowSifter computes a counting approximation of $\Gamma_f(X)$ as follows. First, FlowSifter computes the sets of start and end terminals for $\Gamma_f(X)$, which are denoted as *start* and *stop*. These are the terminals that mark the start and end of a string that can be produced by $\Gamma(X)$. The remaining terminals are denoted as *other*. For example, in the Dyck extraction grammar Γ_{xd} in Fig. 4(b), the set of start and end terminals of $\Gamma_{xd}(S)$ are {'['} and {']'}, respectively, and *other* has no elements. FlowSifter replaces all rules with head X with the four rules in Fig. 6 that use a new counter cnt. The first rule allows exiting X when the recursion level is zero. The second and third increase and decrease the recursion level when matching start and stop terminals. The final production rule consumes the other terminals, approximating the grammar while $cnt > 0$.

For example, if we apply counting approximation to the nonterminal S from the Dyck extraction grammar Γ_{xd} in Fig. 4(b), we get the new production rules in Fig. 7.

We can apply counting approximation to any subgrammar $\Gamma_f(X)$ with unambiguous starting and stopping terminals. Ignoring all parsing information other than nesting depth of start and end terminals in the flow leads to potentially faster flow processing and fixed memory cost. In particular, the errors introduced by counting approximation do not interfere with extracting fields from correct locations within protocol compliant inputs. However, counting approximations do not guarantee that all extracted fields are the result of only protocol compliant inputs. Therefore, application processing functions should validate any input that its behavior depends upon for proper operation.

E. Idle Rule Elimination

The CRG generated as above may have production rules without terminals. When implemented as a parser, no input is consumed when executing such rules. We call such rules *idle rules*, and they have the form: $X \to Y$ without any terminal α. FlowSifter eliminates idle rules by hoisting the contents of

Y into X, composing the actions and predicates as well. For a CRG with n variables, to compose a rule

$$(q_1 \wedge \cdots \wedge q_n) : Y \rightarrow \alpha(act)Z(g_1, \cdots, g_n)$$

into the idle rule

$$(p_1 \wedge \cdots \wedge p_n) : X \rightarrow Y(f_1, \cdots, f_n),$$

we create a new rule

$$(p'_1 \wedge \cdots \wedge p'_n) : X \rightarrow \alpha(act)Z\,(f'_1, \cdots, f_n)$$

where $p'_i = p_i \wedge q_i$ and $f'_i = f_i \circ g_i$ for $1 \leq i \leq n$. That is, we compose the actions associated with Y in X into Z's actions and merge the predicates.

V. Automated Counting Automaton Generation

The *automata generator* module in FlowSifter takes an optimized extraction grammar as its input and generates an equivalent counting automaton (CA) at output. The field extractor module will use this CA as its data structure for performing field extraction.

One of the challenges of field extraction with a CA is resolving conflicting instructions from different CRG rules. For example, consider a CA state A with two rules: $A \rightarrow /ab/B$ and $A \rightarrow /a/[x := x + 1]C$. After processing input character `a`, the state of the automaton is indeterminate. Should it increment counter x and transition to state C, or should it wait for input character `b` so it can transition to state B?

We solve this problem by using a DFA as subroutines in counting automtata. The DFA will inspect flow bytes and return a decision indicating which pattern matched. The DFA will use a priority system to resolve ambiguities and return the highest priorty decision. For the above example, the DFA will have to lookahead in the stream to determine whether /ab/ or /a/ is the correct match; in practice, the lookahead required is small and inexpensive. We describe this novel integrated CA and DFA model in this section.

A. Counting Automata

A Counting Automata (CA) is a 6-tuple $(Q, C, q_0, c_0, \mathbb{D}, \delta)$ where Q is a set of CA states, C is a set of possible counter configurations, $q_0 \in Q$ is the initial state and $c_0 \in C$ is the initial counter configuration. Normally, the transition function δ of the CA is a function that maps the current configuration (state $q \in Q$ and counter configuration $c \in C$) along with input character $\sigma \in \Sigma$ to a new CA state q' along with some action to update the current counters. We choose a different approach where we use Labeled Priority DFA (LPDFA) as a subroutine to perform this mapping for a given CA. We leave the formal details of LPDFA to Section V-B but describe now how a CA uses LPDFA to define the CA's transition function δ. The set $\mathbb{D}$ defines the set of LPDFA that the CA may use. The transition function δ then maps each configuration ($q \in Q$, $c \in C$) to an appropriate LPDFA. That is, $\delta : Q \times C \rightarrow \mathbb{D}$. The LPDFA will deterministically process the flow and return a decision belonging to set $D = (Q_C \cup \{DONE, FAIL\}) \times (C \rightarrow C)$ where $DONE$ and $FAIL$ are distinct from all CA states; $DONE$ implies the CA has completed processing of the input stream and $FAIL$ implies that the input flow does not meet the protocol expectations.

1		$1 \rightarrow$ [$(p := pos())\,4$
2		$2 \rightarrow$] [$(c := 1)\,5$
3		$2 \rightarrow$]
4	$(c > 0)$	$3 \rightarrow$] $(c := c - 1)\,3$
5		$3 \rightarrow$ [$(c := c + 1)\,3$
6	$(c = 0)$	$3 \rightarrow \epsilon\,(p := token(p))\,2$
7	$(c > 0)$	$3 \rightarrow /$[^ [\ ]]$/\,3$
8		$4 \rightarrow$] $(c := 1)\,3$
9		$4 \rightarrow \epsilon\,(p := token(p))\,2$
10	$(c > 0)$	$5 \rightarrow$] $(c := c - 1)\,5$
11		$5 \rightarrow$ [$(c := c + 1)\,5$
12	$(c = 0)$	$5 \rightarrow \epsilon$
13	$(c > 0)$	$5 \rightarrow /$[^ [\ ]]$/\,5$

Fig. 8. CRG for Dyck example from Figs. 2(b) and 4(b).

FlowSifter generates a CA $(Q, C, q_0, c_0, \mathbb{D}, \delta)$ from a CRG $\Gamma = (N, \Sigma_g, C_g, R, S)$ as follows. We set $Q = N$ and $q_0 = S$. We build C based on C_g but with a bounded maximum size b where typically $b = 2^{sizeof(int)} - 1$; counters are initialized to 0. Formally, $C = \{(c_1, c_2, \ldots, c_{|C_g|}) : 0 \leq c_i < b, 1 \leq i \leq |C_g|\}$ and $c_0 = (0, 0, \ldots, 0)$. If necessary, we can tune the parsing state size by using different numbers of bits for each counter. We set δ as follows. For each configuration (q, c), we identify the set of CRG rules $R(q, c)$ that correspond to (q, c) and build the corresponding LPDFA from those rules; the set $\mathbb{D}$ is simply the set of LPDFA we build. We describe LPDFA construction in detail in Section V-B.

For example, in Fig. 8, for the configuration with CA state 3 and counter $c == 0$, rules 5 and 6 are active, so $\delta(3, c == 0)$ is the LPDFA constructed from the right hand sides of those rules: [$(c := c + 1)3$ and $\epsilon(p := token(p))\,2$. This LPDFA will return decision $(3, (c := c + 1))$ when the flow bytes match [and $(2, p := token(p))$ when the flow bytes match ϵ. On the other hand, $\delta(3, c > 0)$ is constructed from the right hand side of rules 4, 5, and 7.

To apply a CA to a flow, we initialize the CA state as (q_0, c_0). Based on the current state (q_i, c_i), we determine $\delta(q_i, c_i) = dfa_i$, and apply dfa_i to the flow bytes. This LPDFA will always return a decision (q_{i+1}, act_i), and if q_{i+1} is either $DONE$ or $FAIL$, parsing ends. After computing $c_{i+1} = act_i(c_i)$, the CA state becomes (q_{i+1}, c_{i+1}), and we repeat the process until we are out of flow bytes.

For example, consider the optimized CRG and CA for our running Dyck grammar example depicted in Figs. 8 and 9, respectively; this CA uses a single counter cnt. The CA state labels have been replaced with integers, and we show the terminal symbols as strings when possible, or as $/regex/$ when a regular expression is needed, such as for rule 7. The initial CA configuration is $(1, cnt == 0)$. The only CRG rule for state 1 is CRG rule 1 which has no conditions associated with it. Thus, the CA invokes the LPDFA that represents the body of rule 1; this LPDFA matches only the string "[", and the CA transitions to state 4 after updating p to the result of $pos()$. If the flow

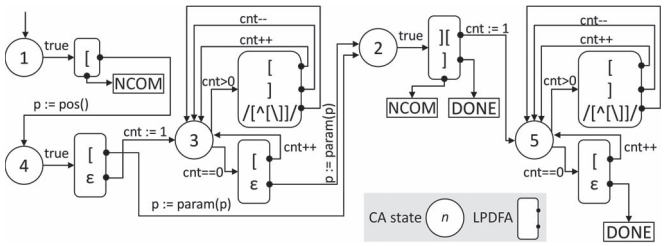


Fig. 9. Exploded CA for Dyck in Figs. 2(b) and 4(b); each cluster has start CA state on left, and destination CA state on right.

bytes do not start with [, then the LPDFA returns $FAIL$ and the CA stops processing input because the input does not conform to the CRG. Suppose later the configuration is $(3, cnt > 0)$. In this case, the CA will invoke an LPDFA that matches the input against to increment cnt, to decrement cnt, or any other input character making no change to cnt; this is based on the counting approximation to parse nested brackets. In all cases, the LPDFA sets the next state to be 3. If the configuration is $(3, cnt == 0)$, then the CA will invoke an LPDFA that matches the input against [to increment cnt and return to state 3; otherwise no input is consumed and $p := param(p)$ will be evaluated and the CA will leave state 3 for state 2.

The functions pos and $param$ allow the CA to interact with its runtime state and the outside world. The function $pos()$ is built into the FlowSifter environment and returns the current parsing offset in the flow. The CA stores this information in a counter so that it can report the start and end positions of a token when it calls $param(p)$ to report this token to the layer above. The CA waits for a return value from the called application processing function so that it can update the counters before it continues processing the input flow. In many cases, the application processing function never needs to return an actual value to the CA; in such cases, it immediately returns a null value, and the CA immediately resumes processing the input flow.

B. LPDFA

In this section, we formally define LPDFA focusing on non-standard LPDFA construction and operation details. A Labeled Priority DFA (LPDFA) is a 7-tuple $(Q, \Sigma, \delta, q_0, D, DF, \pi)$ where Q is a set of states, Σ is an alphabet, q_0 is the initial state, $\delta : Q \times \Sigma \to Q$ is the transition function, as normal. The new properties are D, the set of possible decisions, $DF : Q \to D$, a partial function assigning a subset of the states (the accepting states) a decision, and $\pi : Q \to \mathbb{N}$ a total function assigning every state a priority.

An LPDFA works as a subroutine for a CA, examining the input for various patterns, consuming the highest priority observed pattern, and returning this pattern's associated decision. This allows deterministic operation as there is only one result from matching the given regular expressions against the packet payload. We construct and run an LPDFA in a manner similar to a DFA with some modifications to return values instead of just accept/reject and to only consume the correct amount of the input. We focus on the construction of $DF : Q \to D$, a partial function assigning a subset of the states (the accepting states) a decision, and $\pi : Q \to \mathbb{N}$ a total function assigning every state a priority. As a reminder, $D = (Q_C \cup \{DONE, FAIL\}) \times (C \to C)$, where $DONE$ and $FAIL$ are distinct from all CA states.

For each configuration ($q \in Q$ and $c \in C$) in the δ function of the CA, we construct an LPDFA from the bodies of the rules matching the given configuration. Because we start from a CRG, we can assume the rule bodies are written as (rx_i, act_i, q_i), a terminal regular expression, action and non-terminal. If no rule has a regular expression that matches the empty string ϵ, then we add an ϵ-rule with body $(\epsilon, (), FAIL)$ to guarantee the LPDFA will always return a value. We use the standard process for constructing an automaton from a collection of regular expressions.

We now describe how we set DF. For any accepting state q, we identify all CRG rules' whose regular expressions were matched. If more than one match, we choose the rule r with body (rx, act, q) that has the highest priority (the ϵ-rule has lowest priority). If r is the ϵ-rule, the CA state value is set to $FAIL$. If q is empty, the CA state value is set to $DONE$. Otherwise, the CA state value is set to q. The appropriate action is set to act.

We now describe how an LPDFA operates. We give all states in the LPDFA a priority which is the highest priority decision that is reachable from that state. As the LPDFA processes the input flow, it remembers the highest priority decision state encountered. To prevent the LPDFA from continuing to consume input due to a potential low priority match such as a low priority $/.*/$ rule, the LPDFA stops processing the input once it reaches a state with an equal or lower priority. The LPDFA then returns the appropriate decision and consumes only the appropriate input even if more has been seen.

We illustrate the importance of prioritizing CRG rules using the following two rules from the protocol specification for HTTP headers.

```
HEADER 50 -> /(?i:Content-Length):\s*/
    [bodylength := getnum()];
HEADER 20 -> TOKEN/:/VALUE;
```

The second rule is given a lower priority (20) than the first rule's priority (50) to ensure that the first rule is used when the flow prefix is "Content-Length:". In such a case, the rule stores the size of the HTTP body in a counter bodylength for later use. If the priorities were inverted, the first rule would never be used. We must ensure the relative priorities of rules are maintained through all optimizations that we apply. Maintaining these priorities is straightforward but tedious, so we omit these details.

C. CA Specific Optimizations

We have implemented two key optimizations to speed up our CA implementation. We first avoid processing many bytes of the flow by having actions modify the flow offset in the parsing state. Specifically, if our CA is processing an HTTP flow and does not need to parse within the body, an action can call

$skip(n)$ to skip over n bytes of payload. This allows FlowSifter to avoid stepping through the payload byte-by-byte to get to the headers of the next request in the same flow.

We also eliminate some LPDFA to CA transitions. Suppose the optimized CRG has a nonterminal X with a single rule with no actions such as $X \rightarrow /rx/Y$. We can eliminate the switch from LPDFA to CA at the end of $/rx/$ and the switch back to LPDFA at the beginning of Y by inlining Y into X. This is similar to idle rule elimination, but because our terminals are regular expressions, we can concatenate two regular expressions into a single terminal, keeping the grammar in normal form. We also perform this optimization when Y has a single rule and all of X's rules that end in Y have no actions. This increases the number of states in the LPDFA for each nonterminal but improves parsing speed by decreasing the number of context switches between LPDFA and CA. This optimization has already been performed on the CA in Fig. 9, specifically in state 2. The pattern "][" is not part of any of the input regular expressions, but is composed of the closing ']' of the Dyck extraction grammar and the opening '[' of the S nonterminal following it.

VI. Counting Automaton Implementation

In this section, we first describe incremental packet processing, which is needed because flow data arrives in packets and should be processed as it is received. The alternative solution of buffering large portions of flow is problematic for two reasons. First, it may require large amounts of dynamically allocated memory. Second, it will increase latency in scenarios where undesirable traffic must be blocked.

We then describe two implementations of CA: simulated CA and compiled CA. In a simulated CA, an automaton-independent process parses a flow by referencing a memory image of the simulated CA. In a compiled CA, the structure of the CA is encoded in the process that parses the flow. Typically, compiled CA are more efficient but simulated CA are easier to deploy and modify.

A. Incremental Packet Processing

Packet processing is difficult because the flow data arrives packet by packet rather than all at once. There are two main ways to process packets: *incrementally* which means processing each packet as it arrives or *buffering* which means buffering a number of packets until a certain amount of flow data is gathered. A DFA supports incremental processing by processing each byte of the current packet and then saving the active state of the DFA in a parsing state to be used when the next packet of the flow arrives. BinPAC and UltraPAC do not support incremental packet processing. Instead, they buffer packets until they can guarantee sufficient flow data is available to match an entire token. This has the two drawbacks of (i) requiring large amounts of dynamically allocated memory and (ii) increasing latency in scenarios where undesirable traffic must be blocked.

As FlowSifter is built on automata-theoretic constructs, we present an incremental packet processing approach similar to DFA, though we do require some buffering of flow data since an LPDFA may look at more flow data than it consumes. Unlike other buffering solutions, FlowSifter only buffers input data that the LPDFA needs to determine the highest priority match; this is typically a small amount of data, much smaller than the amount of data needed to guarantee a token can be matched. There are three parts of the per-flow state that we must record: the state of the input flow, the state of the CA and the state of the LPDFA.

Note that FlowSifter is a stream parser (like UltraPAC), which means that FlowSifter does not store extracted fields. Once a field is extracted, FlowSifter passes it to the upper layer for application specific processing. The only memory used by FlowSifter is to store the CA and context information needed for traversing the CA. Such memory is allocated on the start of a new flow and deallocated at the end of the flow.

We record two byte offsets for each flow: an offset within the current packet of the next byte to be processed, and a flow offset of the current packet, indicating the position in the flow of the first byte of that packet. We keep both of these in our parsing state to be able to check for errors in flow reassembly and to aid in supporting the $skip()$ builtin. After processing a packet, we subtract its size from the packet offset, which normally resets the packet offset. The $skip()$ action can be implemented by increasing the packet offset by the number of bytes to be skipped. If the resulting offset is within the current packet, processing will resume there. If the offset is within the next packet, the subtraction rule will correctly compute the offset within that next packet to resume processing. If the offset is beyond the next packet's data, that packet can be skipped entirely and the subtraction rule will account for that packet's bytes being skipped.

As both the CA and LPDFA are deterministic, they each have only one active state. Furthermore, when an LPDFA has an active state, the CA state is not needed, and when a CA state is active, no LPDFA state is active; thus we only need to store one or the other. In addition to the active state, the CA also must keep the values of each counter. In addition to the active state, the LPDFA must track the state id and flow offset of the current highest priority match, if any. A small buffer may be needed when the LPDFA lookahead occurs at a packet boundary, to buffer the flow data since the last match was found. This allows the LPDFA to consume only the bytes needed to reach its decision while allowing the next consumer of input bytes to resume from the correct location, even if that is in the previous packet.

Finally, we must address incremental parsing for actions that alter the parsing state. For example, $skip()$ can change the packet offset. We might create other actions such as $getByte()$ that can read a current input character into a counter as an unsigned integer; $getByte()$ is useful in the DNS protocol where the length of a name field is indicated by the byte preceding it. Instead of using $getByte()$, we could use an LPDFA that transitions to a different state for each possible value of that byte and have an action for each value that would set the counter correctly. Using a function like $getByte()$ makes this much easier to implement and faster to execute. However, this does introduce a corner case where the byte we need may be in

the next packet. In general, the CA must be able to resume processing from the middle of an action. Our simulated CA and compiled CA take different approaches to solving this problem, as described in their sections below.

B. Simulated CA

In an ASIC implementation of CA, we create a fixed *simulator* that cannot be changed after construction. The simulator uses a memory image of the CA to be simulated when processing an input flow. To assess the challenges of a simulated CA implementation and develop a proof of concept, we implemented a software CA simulator. The most challenging issues we faced were (i) efficiently implementing the function δ to find the appropriate LPFDA based on the current CA counter values and (ii) incremental packet processing.

The biggest challenge to implementing δ efficiently is the potentially huge size of the counter state space. With two 16-bit counters and just 10 CA states, a direct lookup table would have over 40 billion entries, ruling out this solution for the entirety of δ. Instead, we break the lookup into two steps. In the first step, we use the CA state number as a direct lookup.

For a CA with C counters, a given CA state may correspond to a CRG nonterminal with n rules. We must find the correct LPFDA that implements the combination of these rules that can be applied, based upon the current counter values. We present three different solutions to solving this LPDFA selection problem. We evaluate each solution based upon two criteria: (i) space and (ii) time. The space complexity of a solution corresponds to the number of LPDFA that need to be precomputed and stored. The time complexity of a solution corresponds to the number of predicate evaluations that must be performed. Each solution differs in how to index the LPDFA and how to perform the predicate evaluations.

Our first solution indexes LPDFA by rule. That is, we construct 2^n LPDFA, one for each combination of rules and store pointers to these LPDFA in an array of size 2^n. Note that not all combinations of rules are possible. For example, in the Dyck CA, state 3 has 4 rules, so the LPDFA table has 16 entries. However, only two LPDFAs are possible for state 3: one encoding rules 5 and 6 when $cnt == 0$, and one encoding rules 4, 5, and 7 when $cnt > 0$. We perform an analysis of rule predicates and save space by leaving as many of the LPDFA array entries empty as possible. To find the current predicate, we evaluate the predicates for each rule and pack these true/false results as 1/0 bits in an unsigned integer used to index the array of LPDFA. In the worst case, this requires nC counter predicate evaluations. Of course, some counters may not need to be evaluated for some rules, and the results from some counter evaluations may eliminate the need to perform other counter evaluations.

Our second solution indexes LPDFA by predicate, taking advantage of redundancy in predicates among different rules. For example, using state 3 from the Dyck CA, we previously observed there are only two relevant LPDFA: one for $cnt == 0$ and one for $cnt > 0$. We can store this list of relevant predicates and index the LPDFA by their truth values. For this example, we would have just these two predicates and two LPFDA. For this example, we would evaluate both predicates and use these two bits to choose the LPDFA to execute.

A third solution is to build an optimal decision tree. Each node of the decision tree would correspond to a counter and the children of that node would either be the LPDFA to execute or further nodes corresponding to other counters. Each edge would be labeled with the values of the counter it can be traversed on. In this example, the decision tree would have one node, with an edge to the rule {5, 6} LPDFA labeled 0 and an edge to the rule {4, 5, 7} LPDFA on all other values. This solution can reduce the number of counter evaluations required by optimally exploiting redundancies and optimally using the results of previous predicate evaluations. We plan on investigating this potential solution further in future work. In our software simulator, we observed that indexing by rule appeared to be more efficient than indexing by predicate in our experiments.

Our simulator uses closures to handle saving and resuming state. Whenever it runs out of input, it returns a new function that takes the next packet's payload as input and continues processing where it left off. This is possible because OCaml allows us to encapsulate a portion of our environment of variables inside a newly created function, allowing this new function to make use of any local state available at the point of its creation. The details of the function differ depending on where we run out of input. In general, this resume function does some bookkeeping on the input string to put it into the main shared state record, and then it calls a function to resume processing at either a CA state (to processes any delayed actions) or at an LPDFA state, to continue consuming input.

C. Compiled CA

In this section, we give a compilation from CA to C++. This non-ASIC compiled CA implementation can achieve very high performance because modern CPUs are highly optimized simulators for their machine language. The major difficulty is minimizing the overhead in the compilation process; the Turing completeness of CPUs guarantees that this is always possible for any reasonable automata construction, but the efficiency of the result is strongly dependent on the semantics of the automaton and the specifics of the translation process.

The basic structure of the compiled CA is similar to that of the simulated CA. We briefly describe all of the details but focus on the key differences. We encapsulate the parsing state of a flow using a C++ struct that contains unsigned integers for each counter, a few fields to hold a pointer to the flow data and the offsets relative to the pointer and to the start of the flow. We also include fields for the LPDFA to store the offset, state, and priority of the highest priority matching state seen so far. Finally, we store the current LPDFA state, if any, to support incremental packet processing.

We represent each CA state, each LPDFA, and each action update function with a procedure. Each procedure exhibits tail call behavior where it ends by calling some other procedure. For example, after the CA state procedure determines which LPDFA to run, it ends its operation by calling the appropriate LPDFA procedure. Likewise, an LPDFA procedure will

typically end by calling a series of update actions and then the next CA state procedure. To ensure the stack does not grow to unbounded size, our CA must implement Tail Call Optimization (TCO). Ideally, the C/C++ compiler will implement TCO. If not, we manually implement TCO by having a main dispatch loop. Within this loop, we maintain a function pointer that represents the next procedure to be called. Each procedure then ends by returning a pointer to the next function to be called rather than calling the next function.

The procedure for each CA state has access to all the flow state, and its job is simply to determine which LPDFA to run. As with simulated CA, the compiled CA can index the LPFDA in many different ways such as by rule, by predicate, or by decision tree. Our compiler produces a CA that indexes by rule where the resulting Boolean values are concatenated into a single integer. The CA state procedure uses a switch statement on this integer to determine which LPDFA is initialized and started. If all the rules for a CA state have no predicates, then there is only one LPDFA that is always called. We can eliminate this CA state procedure and simply call the corresponding LPDFA procedure instead.

We run each LPDFA using a its own LPDFA simulator procedure. The transitions and priorities are stored in integer arrays. To make a transition, the current state and input character are used to index the transition array which stores the next state that this automaton will be in. The simulator only proceeds to the next state q if q's priority exceeds the priority of the highest priority match found so far. When we have determined the highest priority match, the simulator runs a switch statement to implement the actions and the transition to the next CA state dictated by this match. We choose to implement a separate simulator for each LPDFA to support finely tuned branch prediction for the CPU and to facilitate pausing and resumption caused by incremental packet processing. The cost of replicating the very small common simulation code is negligible.

It is possible that multiple CA states may call identical LPDFA. With each new LPDFA that we create, we compare it to previously created ones and only keep the new one if it is indeed different than all previous ones.

Finally, we discuss our procedures which implement the actions of a CA. We create a procedure for each action so that we can pause and resume them at packet boundaries. To support incremental packet processing, we ensure that each input-consuming function in an action is in its own action function. We improve performance by leaving inline actions that do not consume any input as the CA will never need to resume from such actions.

VII. Experimental Results

We evaluate field extractor performance in both speed and memory. Speed is important to keep up with incoming packets. Because memory bandwidth is limited and saving and loading extractor state to DRAM is necessary when parsing a large number of simultaneous flows, memory use is also a critical aspect of field extraction. Note that FlowSifter is 100% accurate at extracting all fields in the extraction specification from traffic that matches the protocol specification, just like BinPAC and UltraPAC. Our FlowSifter implementation is available as open source (GPL) at https://github.com/MSU-SSL/FlowSifter.

A. Methodology

Tests are performed using two types of traces, HTTP and SOAP. We use HTTP traffic in our comparative tests because the majority of non-P2P traffic is HTTP and because HTTP field extraction is critical for L7 load balancing. We use a SOAP-like protocol to demonstrate FlowSifter's ability to perform field extraction on flows with recursive structure. SOAP is a very common protocol for RPC in business applications, and SOAP is the successor of XML-RPC. Parsing SOAP at the firewall is important for detecting parameter overflows.

Our trace data format has interleaved packets from multiple flows. In contrast, BinPAC/UltraPAC use evaluation traces where each trace is a sequence of pre-assembled flows. We use the interleaved packet format because it is impractical for a network device to pre-assemble each flow before passing it to the parser. Specifically, the memory costs of this pre-assembly would be very large and the resulting delays in flow transmission would be unacceptably long. For our experiments, we assemble flows at runtime, switching the parser to a flow as the next payload data for that flow is available. We determine the CPU cost of this assembly by timing a null parser that performs assembly but otherwise ignores the input. We subtract this assembly time when reporting FlowSifter's parsing time.

Our HTTP packet data comes from two main sources: the MIT Lincoln Lab's (LL) DARPA intrusion detection data sets [28] and captures done in our research lab. This LL data set has 12 total weeks of data from 1998 and 1999. We obtained the HTTP packet data by pre-filtering for traffic on port 80 with elimination of TCP retransmissions and delaying out-of-order packets. Each day's traffic became one test case. We eliminated the unusually small traces ($<$ 25 MB) from our test data sets to improve timing accuracy. The final collection is 45 LL test traces, with between 0.16 and 2.5 Gbits of data and between 27 K and 566 K packets per trace, totaling 17 GB of trace data.

The real life packet traces come from two sources: 300 MB of HTTP spidering traces (HM) and 50 GB (roughly 2/3 of which is HTTP) of Internet use (IU) over a period of one month. We capture the HTTP spidering traces using the HarvestMan [29] web spider on web directories like dmoz.org and dir.yahoo.com. This method produces mostly smaller requests for `.html` files instead of downloading large graphic, sound or video files. HarvestMan also uses HTTP 1.1's keep-alive pervasively, resulting in many HTTP requests/responses being included in a single TCP flow. Thus, it is imperative to support the Content-Length header to identify the end of one request's data. The IU trace was recorded in 100 MB sections. We use each section as a separate datapoint for testing.

We construct SOAP-like traces by constructing 10,000 SOAP-like flows and then merging them together into a trace. We create one SOAP-like flow by encapsulating a constructed SOAP body in a fixed HTTP and SOAP header and footer. We use a parameter n ranging from 0 to 16 to vary the depth of our flows as follows. The SOAP body is composed of nested tag; on level l, a child node is inserted with probability $\left(0.8^{\max(0,l-n)}\right)$.

After inserting a child node, the generator inserts a sibling node with probability $(.6 * .8^{\max(0,l-n)})$. The average depth of a flow with parameter n is about $n + 5$.

For each value of n, we generate 10 traces for a total of 170 traces. Each trace consists of 10,000 flows generated using the same parameter n. These 10,000 flows are multiplexed into a stream of packets as follows. Initially we set no flow as active. During each unit of virtual time, one new flow is added to the set of active flows. Each active flow generates data that will be sent in the current unit of virtual time as follows: with equal probability, it sends `0`, `rand(50)`, `rand(200)`, `rand(1000)` and `1000 + rand(500)` bytes. If the transmission amount for a flow exceeds its remaining content, that flow sends all remaining data and is then removed from the set of active flows. All the data sent in one unit of virtual time is merged and accumulated into a virtual packet flow. The typical trace length is roughly 28 K packets for $n = 0$ and 106 K packets for $n = 16$. The total length of all 170 traces is 668 MB.

1) Field Extractors: We have two implementations of FlowSifter: a compiled FlowSifter (csift) and a simulated FlowSifter (sift). We have written one FlowSifter package that generates both implementations. This package is written in 1900 lines of Objective Caml (excluding LPDFA generation, which uses normal DFA construction code) and runs on a desktop PC running Linux 2.6.35 on an AMD Phenom X4 945 with 4 GB RAM. The package includes additional optimizations not documented here for space reasons. We emphasize that in our experiments, the compiled FlowSifter outperforms the simulated FlowSifter because we are only doing software simulation. If we had an ASIC simulator, the simulator would run much more quickly and likely would outperform our compiled implementation.

We constructed HTTP field extractors using FlowSifter, BinPAC from version 1.5.1 of Bro, and UltraPAC from NetShield's SVN r1928. The basic method for field extractor construction with all three systems is identical. First, a base parser is constructed from an HTTP protocol grammar. Next, a field extractor is constructed by compiling an extraction specification with the base parser. Each system provides its own method for melding a base parser with an extraction specification to construct a field extractor. We used UltraPAC's default HTTP field extractor which extracts the following HTTP fields: `method`, `URI`, `headername`, and `headervalue`. We modified BinPAC's default HTTP field extractor to extract these same fields by adding extraction actions. FlowSifter's base HTTP parser was written from the HTTP protocol spec. We then wrote an extraction specification to extract these same HTTP fields.

For SOAP traffic, we can only test the two implementations of FlowSifter. We again wrote a base SOAP parser using a simplified SOAP protocol spec. We then made an extraction specification to extract some specific SOAP fields and formed the SOAP field extractor by compiling the extraction specification with the base SOAP parser. We attempted to develop field extractors for BinPAC and UltraPAC, but they seem incapable of easily parsing XML-style recursive structures. BinPAC assumes it can buffer enough flow data to be able to generate a parse node at once. UltraPAC's Parsing State Machine can't represent the recursive structure of the stack, so it would require generating the counting approximation by hand.

2) Metrics: For any trace, there are two key metrics for measuring a field extractor's performance: *parsing speed* and *memory used*. We use the term *speedup* to indicate the ratio of FlowSifter's parsing speed on a trace divided by another field extractor's parsing speed on the same trace. We use the term *memory compression* to indicate the ratio of another parser's memory used on a trace divided by FlowSifter's memory used on the same trace. The *average speedup* or *average memory compression* of FlowSifter for a set of traces is the average of the speedups or memory compressions for each trace. Parser Complexity is measured by comparing the definitions of the base HTTP protocol parsers. We only compare the HTTP protocol parsers since we failed to construct SOAP field extractors for either BinPAC or UltraPAC.

We measure parsing speed as the number of bits parsed divided by the time spent parsing. We use Linux process counters to measure the user plus system time needed to parse a trace. We record the memory used by a field extractor on a trace by taking the difference between the memory used by the extractor at the end of processing the trace and the memory used by the extractor just before processing the trace. This is not the peak memory across the trace. Instead, it is a somewhat random sample of memory usages for each trace. In particular, traces end at arbitrary points typically with many active flows. BinPAC, UltraPAC and compiled FlowSifter use manual memory management, so we get our memory used values via tcmalloc's [30] `generic.current_allocated_bytes` parameter. This allows us to precisely identify the exact amount of memory allocated to the extractor and not yet freed. Although this does not measure stack usage, none of the implementations makes extensive use of the stack. Simulated FlowSifter runs in a garbage collected environment that provides an equivalent measure of live heap data.

B. Experimental Results

We show empirical CDFs for all three field extractors' memory usage and extraction speed on each dataset. These show FlowSifter's memory use dominates both BinPAC and UltraPAC. There is slight overlap in parsing speed, but FlowSifter clearly has better best case, worst case and average speed than both BinPAC and UltraPAC. The efficiency curve nearly matches the speed curve, with FlowSifter having infrequent worst and best efficiency, and still showing much improvement over BinPAC and UltraPAC.

1) Parsing Speed: As shown in Fig. 10, compiled FlowSifter (siftc) is significantly faster than simulated FlowSifter (sift) which is faster than both BinPAC (bpac) and UltraPAC (upac). Each cluster of points has its convex hull shaded to make it easier to separate the clusters while still showing the individual data points. Across the range of traces, compiled FlowSifter's average speedup over BinPAC ranged from 11 (LL) to 21 (IU). UltraPAC was able to parse faster than BinPAC, but compiled FlowSifter still kept an average speedup of between 4 (LL) to 12 (IU). This means that on a real-life dataset, FlowSifter was able to achieve an average performance of over 12 times

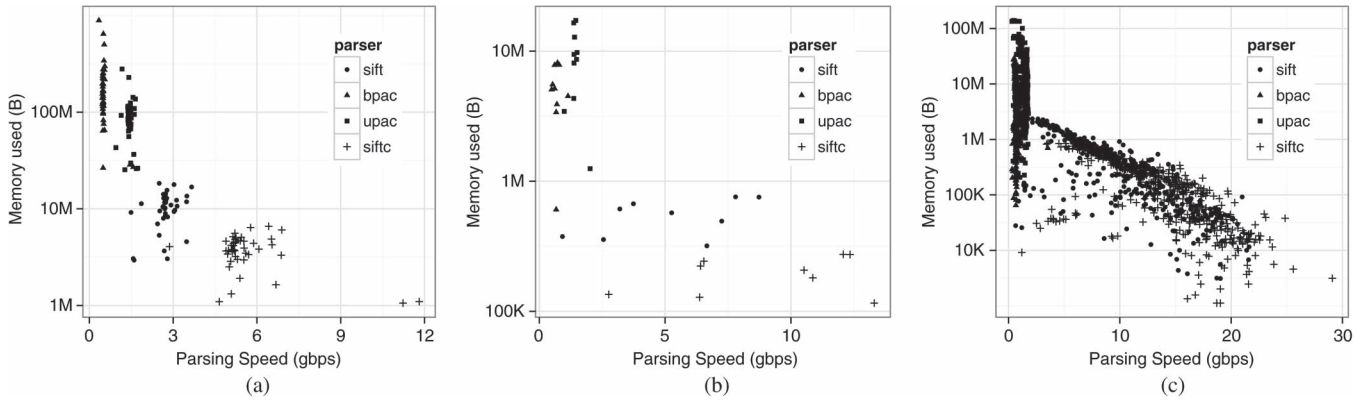


Fig. 10. Comparison of parsers on different traces. (a) LL traces. (b) HM traces. (c) IU traces.

that of the previous best field extraction tool. Further speed improvement is possible with ASIC implementation, which could make one LPDFA transition per cycle at 1 GHz, resulting in up to 80 Gbps performance on the IU traces with a single engine.

Simulated FlowSifter's LPDFA speed is 1.8 Gbps; compiled FlowSifter's LPDFA speed is 2.6 Gbps. These speeds were measured by running a simple LPDFA on a simple input flow. As shown in Fig. 10, the FlowSifter implementations can run both faster and slower than their LPDFA speed. FlowSifter can traverse flows faster by using the CA to perform selective parsing. For example, for an HTTP flow, the CA can process the `ContentLength` field into a number and skip the entire body by ignoring that number of bytes from the input. BinPAC and UltraPAC improve their performance similarly through their `&restofflow` flag.

BinPAC and UltraPAC have much slower parsing speed. This is because their per-flow state is much larger than that of FlowSifter. The per-flow states of BinPAC and UltraPAC are much larger than FLowSifter because their parsing strategy requires much more of the flow to be buffered to ensure that an entire "token" is consumed at once. FlowSifter is able to use much more fine-grained tokens and can save/resume state even in the middle of a token, allowing us to buffer much less. When switching between many flows, the CPU cache will often have a cache miss on the flow state, causing a much higher context switch cost.

However, the CA introduces two factors that can lead to slower parsing: evaluating expressions and context switching from an LPDFA to a CA and then back to an LPDFA. Evaluating predicates and performing actions is more costly in the simulated FlowSifter implementation than in the compiled implementation. Context switches are also more costly for the simulated implementation than the compiled implementation. Further, the compiled implementation can eliminate some context switches by compiler optimizations. That is, it can inline the code for a CA state into the decision for the LPDFA so that one LPDFA can directly start the next one.

To test FlowSifter's approximation performance, we made a SOAP field extractor that extracts a single SOAP data field two levels deep and then ran it on our 10 traces for each value of n ranging from 0 to 16. The resulting average parsing speeds with 95% confidence intervals for each value of n are shown in Fig. 11. For both compiled and simulated implementations, the parsing speed on this grammar is much less than on plain HTTP. The major reason for this is that with HTTP, there are no fields to extract from the body, so the body can be skipped. With SOAP traces, we cannot exploit any skip.

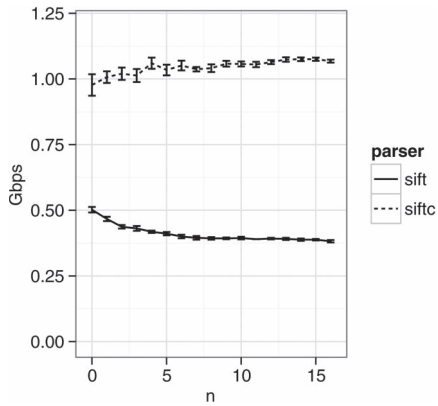


Fig. 11. Average parsing speed for SOAP-like flows versus recursion depth n.

As the recursion level increases, the number of CA transitions per DFA transition increases. This causes FlowSifter to check and modify counters more often, slowing execution.

2) Memory Use: While the graphs in Fig. 10 show FlowSifter using less memory than BinPAC and UltraPAC, the different number of active flows in each capture make direct comparison harder. Each point in Fig. 12 shows the total memory used divided by the number of flows in progress when the capture was made. This shows FlowSifter uses much less memory per flow (and thus per trace) than either BinPAC or UltraPAC. On average over our 45 LL traces, FlowSifter uses 16 times less memory per flow (or trace) than BinPAC and 8 times less memory per flow (or trace) than UltraPAC.

FlowSifter's memory usage is consistently 344 bytes per flow for simulated automaton and 112 bytes for compiled automaton. This is due to its use of a fixed-size array of counters to store almost all of the parsing state. BinPAC and UltraPAC use much more memory averaging 5.5 KB and 2.7 KB per flow, respectively. This is mainly due to their buffering requirements; they must parse an entire record at once. For HTTP traffic, this means an entire line must be buffered before they parse it. When matching a regular expression against flow content, if there is

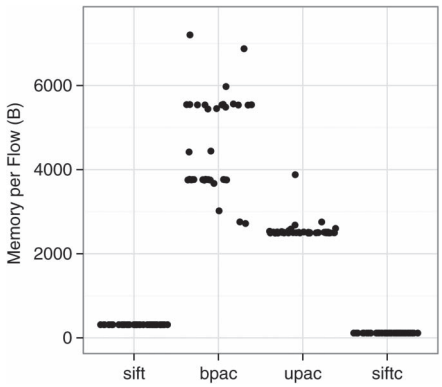


Fig. 12. Memory per flow on LL traces.

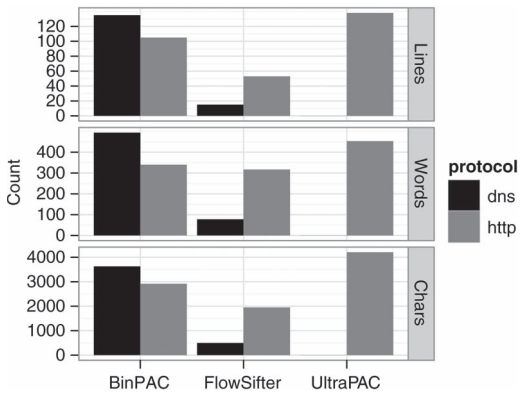


Fig. 13. Complexity of base protocol parsers.

not enough flow to finish, they buffer additional content before trying to match again.

3) Parser Definition Complexity: The final point of comparison is possibly less scientific than the others, but is relevant for practical use of parser generators. The complexity of writing a base protocol parser for each of these systems can be approximated by the size of the parser file. We exclude comments and blank lines for this comparison, but even doing this, the results should be taken as a very rough estimate of complexity. Fig. 13 shows the DNS and HTTP parser size for BinPAC and FlowSifter and HTTP parser size for UltraPAC. UltraPAC has not released a DNS parser. The FlowSifter parsers are the smallest of all three, with FlowSifter's DNS parser being especially small. This indicates that CCFG grammars are a good match for application protocol parsing.

VIII. Conclusion

In this work, we performed a rigorous study of the online L7 field extraction problem. We propose FlowSifter, the first systematic framework that generates optimized L7 field extractors. Besides the importance of the subject itself and its potential transformative impact on networking and security services, the significance of this work lies in the theoretical foundation that we lay for future work on this subject, which is based on well-established automata theory.

With this solid theoretical underpinning, FlowSifter generates high-speed and stackless L7 field extractors. These field extractors run faster than comparable state of the art parsers, use much less memory, and allow more complex protocols to be represented. The parsing specifications are even by some measures simpler than previous works. There are further improvements to be made to make our field extractor even more selective and efficient by further relaxing the original grammar.

Appendix
HTTP Protocol Grammar

```
HTTP -> HTTP_START [bodychunked := 0;
    bodylength := 0]
       HEADERS CRLF BODY HTTP;
HTTP -> ;
CRLF -> /\r?\n/;
SP -> /\x20/;
LWS -> /(\r?\n)?[ \t]+/;
CHAR -> /[\x00-\x7f]/; #ascii 0-127
NONWS -> /[^\x00-\x1f\x7f ]+/;
    # excludes cr, lf, tab, space
TEXT -> /[^\x00-\x1f\x7f]+/; # excludes
    cr, lf, tab
QDTEXT -> /[^\x00-\x1f\x7f"]+/;
    # excludes cr, lf, tab, quote
TOKEN ->/[^\x00-\x1f()<>@,;:\\"\/[]?={}]
   +/; # excludes CTLs and separators
URL -> NONWS;
#Response
HTTP_START -> VERSION SP STATUS TAILOP
    CRLF;
#Request
HTTP_START 10 -> TOKEN SP URL SP VERSION
    CRLF [httprequest := 1];
#Response part
TAILOP -> SP TEXT;
TAILOP -> ;
# capture the HTTP version string
VERSION-> /HTTP\/1\.0/ [httpversion := 0];
VERSION-> /HTTP\/1\.1/ [httpversion := 1];
VERSION 10 -> /HTTP\/[0-9]+\.[0-9]+/;
# Status code
STATUS -> /\d\d\d/;
#Header Field overall structure
    (including final CRLF)
HEADERS -> HEADER CRLF HEADERS;
HEADERS -> ;
#Each individual header - special
    attention to content length &
    transfer encoding as they determine
    body format
HEADER -> /(?i:Content-Length):\s*/
    [bodylength := getnum()];
HEADER -> /(?i:Transfer-Encoding:
    \s*chunked)/ [bodychunked := 1];
HEADER 10 -> TOKEN /:/ VALUE;
```

```
VALUE -> TEXT VALUE;
VALUE -> LWS VALUE;
VALUE -> ;
#body types
## Content length known
BODY [bodylength > 0] -> // [bodylength
    := skip(bodylength)];
BODY [bodylength == 0] -> BODY_NO_LEN;
## Chunked body
BODY_NO_LEN [bodychunked == 1] ->
    CHUNK_BODY;
BODY_NO_LEN [bodychunked == 0] ->
    BODY_VERSION;
## HTTP/1.0: skip rest of flow
BODY_VERSION [httpversion == 0] -> //
    [bodylength := drop_tail()];
## HTTP/1.1, assume bodylength = 0, eat
    any nulls used as keepalive
BODY_VERSION [httpversion == 1] ->
    /\x00*/ ;
#BODY_VERSION [httpversion == 1;
    httprequest == 0] -> // [bodylength
    := drop_tail()] ;
BODY_XML -> CRLF [bodyend := bodyend +
    pos()] XML;
CHUNK_BODY -> // [chunksize :=
    gethex()] CHUNK_EXTENSION CRLF
    [chunksize := skip(chunksize)] CRLF
    CHUNK_BODY;
CHUNK_BODY 99 -> /0/ CRLF HEADERS CRLF;
CHUNK_BODY 99 -> /0;/ TEXT CRLF HEADERS
    CRLF;
CHUNK_EXTENSION -> /;/ TEXT;
CHUNK_EXTENSION -> ;
XML -> XVER XENV [bodyend :=
    skip_to(bodyend)];
XVER -> /-><?xml version="1.0"?>\s*/ ;
XENV -> /<soap:Envelope\
    xmlns:soap="http:\/\/www\.w3\.org\
   /2003\/05\/soap-envelope">/ XHDR XBDY
    /<\/soap:Envelope>/ ;
XHDR -> /\s*<soap:Header>\s*<\/soap:
   Header>\s*/ ;
XBDY -> /\s*<soap:Body>\s*/ ARRAY
    /\s*<\/soap:Body>\s*/;
ARRAY -> /<array>/ ARRAY /<\/array>/
    ARRAY ;
ARRAY -> ;
```

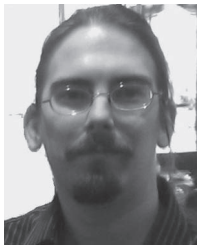
ACKNOWLEDGMENT

We would like to thank Sailesh Kumar for describing the importance of the Layer 7 field extraction problem, and for reviewing an earlier version of this paper. We thank Zhichun Li and Hongyu Gao for their help with the NetShield codebase. We also would like to thank the anonymous reviewers for their encouraging comments and excellent suggestions for improving the quality of this work.

REFERENCES

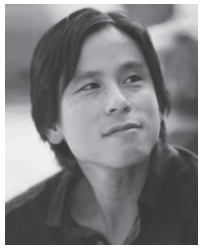
[1] Simple Object Access Protocol (Soap). [Online]. Available: www.w3.org/tr/soap/
[2] XML-RPC. [Online]. Available: http://www.xmlrpc.com/spec
[3] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier, "Shield: Vulnerability-driven network filters for preventing known vulnerability exploits," in *Proc. SIGCOMM*, 2004, pp. 193–204.
[4] Z. Li, L. Wang, Y. Chen, and Z. Fu, "Network-based and attack-resilient length signature generation for zero-day polymorphic worms," in *Proc. IEEE ICNP*, 2007, pp. 164–173.
[5] N. Schear, D. R. Albrecht, and N. Borisov, "High-speed matching of vulnerability signatures," in *Proc. Int. Symp. RAID*, 2008, pp. 155–174.
[6] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *Proc. IEEE Symp. Security Privacy*, 2007, pp. 1–15.
[7] H. Li, H. Lin, X. Yang, and F. Liu, "A rules-based intrusion detection and prevention framework against sip malformed messages attacks," in *Proc. 3rd IEEE IC-BNMT*, Oct. 2010, pp. 700–705.
[8] R. Ferdous, R. Cigno, and A. Zorat, "On the use of svms to detect anomalies in a stream of sip messages," in *Proc. 11th ICMLA*, Dec. 2012, vol. 1, pp. 592–597.
[9] R. Pang, V. Paxson, R. Sommer, and L. Peterson, "binpac: A yacc for writing application protocol parsers," in *Proc. ACM IMC*, 2006, pp. 289–300.
[10] N. Borisov, D. J. Brumley, and H. J. Wang, "A generic application-level protocol analyzer and its language," in *Proc. NDSS*, 2007, pp. 216–231.
[11] L. Burgy, L. Reveillere, J. Lawall, and G. Muller, "Zebu: A language-based approach for network protocol message processing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 4, pp. 575–591, Jul./Aug. 2011.
[12] J. Moscola, Y. H. Cho, and J. W. Lockwood, "Reconfigurable context-free grammar based data processing hardware with error recovery," in *Proc. 20th IPDPS*, 2006, pp. 1–4.
[13] J. Moscola, J. W. Lockwood, and Y. H. Cho, "Reconfigurable content-based router using hardware-accelerated language parser," *ACM Trans. Design Autom. Electron. Syst.*, vol. 13, no. 2, pp. 1–25, Apr. 2008.
[14] Y. H. Cho, J. Moscola, and J. W. Lockwood, "Context-free-grammar based token tagger in reconfigurable devices," in *Proc. ACM/SIGDA 14th Int. Symp. FPGA*, 2006, pp. 237–237.
[15] Ethereal OSPF Protocol Dissector Buffer Overflow Vulnerability, [Online]. Available: http://www.idefense.com/intelligence/vulnerabilities/display.php?id=349
[16] Snort TCP Stream Reassembly Integer Overflow Exploit. [Online]. Available: http://www.securiteam.com/exploits/5bp0o209ps.html
[17] tcpdump ISAKMP Packet Delete Payload Buffer Overflow. [Online]. Available: http://xforce.iss.net/xforce/xfdb/15680
[18] Symantec Multiple Firewall NBNS Response Processing Stack Overflow. [Online]. Available: http://research.eeye.com/html/advisories/published/ad20040512a.html
[19] C. Shannon and D. Moore, The Spread of the Witty Worm. [Online]. Available: http://www.caida.org/research/security/witty/
[20] A. Kumar, V. Paxson, and N. Weaver, "Exploiting underlying structure for detailed reconstruction of an internet-scale event," in *Proc. ACM IMC*, 2005, p. 33.
[21] R. Russel, The Netfilter Packet Filtering Framework, Apr. 2014. [Online]. Available: http://www.netfilter.org/
[22] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer, "Dynamic application-layer protocol analysis for network intrusion detection," in *Proc. 15th Conf.USENIX-SS*, 2006, vol. 15, p. 18, Berkeley, CA, USA: USENIX Association.
[23] M. Mellia, A. Carpani, and R. Lo Cigno, "Measuring IP and TCP behavior on edge nodes," in *Proc. IEEE GLOBECOM*, Nov. 2002, vol. 3, pp. 2533–2537.
[24] Z. Li *et al.*, "NetShield: Massive semantics-based vulnerability signature matching for high-speed networks," in *Proc. SIGCOMM*, 2010, pp. 279–290.
[25] S. C. Johnson, "Yacc—Yet another compiler-compiler," Bell Lab., Murray Hill, NJ, USA, Tech. Rep. 32, 1975.
[26] T. T. J. Parr and R. R. W. Quong, "ANTLR: A predicated-LL(k) parser generator," *Softw.—Pract. Exp.*, vol. 25, no. 7, pp. 789–810, Jul. 1995.
[27] A. D. Thurston, "Parsing computer languages with an automaton compiled from a single regular expression," in *Proc. 11th Int. CIAA*, 2006, pp. 285–286.
[28] Darpa Intrusion Detection Evaluation Data Set, 1998. [Online]. Available: http://www.ll.mit.edu/mission/communications/cyber/CSTcorpora/ideval/data
[29] Harvestman, 2013. [Online]. Available: http://code.google.com/p/harvestman-crawler/
[30] Tcmalloc, 2011. [Online]. Available: http://goog-perftools.sourceforge.net/doc/tcmalloc.html

Alex X. Liu received the Ph.D. degree in computer science from The University of Texas at Austin, Austin, TX, USA, in 2006. His research interests focus on networking and security. He is an Editor of the IEEE/ACM TRANSACTIONS ON NETWORKING and the Journal of Computer Communications. He is the TPC Co-Chair of ICNP 2014. Dr. Liu received the IEEE & IFIP William C. Carter Award in 2004, an NSF CAREER Award in 2009, and the Michigan State University Withrow Distinguished Scholar Award in 2011. He received Best Paper Awards from ICNP 2012, SRDS 2012, LISA 2010, and TSP 2009.



Chad R. Meiners received the Ph.D. in computer science at Michigan State University, East Lansing, MI, USA, in 2009. He is currently a Member of Technical Staff with MIT Lincoln Laboratory, Lexington, MA. His research interests include networking, algorithms, and security.

Eric Norige is pursuing the Ph.D. degree in computer science from Michigan State University, East Lansing, MI, USA. He is currently working for NetSpeed Systems, a Silicon Valley startup in the area of NoC IP. His research interests are algorithms, optimization, and security.



Eric Torng received the Ph.D. degree in computer science from Stanford University, Standford, CA, USA, in 1994. He is currently an Associate Professor and Graduate Director in the Department of Computer Science and Engineering at Michigan State University, East Lansing, MI, USA. He received an NSF CAREER award in 1997. His research interests include algorithms, scheduling, and networking.