FlowSifter: A Counting Automata Approach to Layer 7 Field Extraction for Deep Flow Inspection

Chad Meiners Eric Norige Alex X. Liu Eric Torng
Dept. of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824-1266, U.S.A.
{meinersc, norigeer, alexliu, torng}@cse.msu.edu

***Abstract*—In this paper, we introduce FlowSifter, a systematic framework for online application protocol field extraction. FlowSifter introduces a new grammar model *Counting Regular Grammars* (CRG) and a corresponding automata model *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. In our experiments, we compare FlowSifter against both BinPAC and UltraPAC, which are the freely available state of the art field extractors. Our experiments show that when compared to UltraPAC parsers, FlowSifter extractors run 84% faster and use 12% of the memory.**

I. Introduction

In the past, most network devices were content-unaware; such devices extracted only transportation information contained in Layer 3 (L3) and Layer 4 (L4) headers such as source IP address and destination port number instead of Layer 7 (L7) packet payload content to manage network traffic and implement network security. The main reason for using content-unaware networking devices is that it is much cheaper and easier to extract L3 and L4 packet header information than it is to extract L7 packet payload content.

However, modern network management now requires networking devices that can extract specific content from within packet payloads. A typical application will require these content-aware devices to extract particular L7 fields. For example, data loss prevention tools (DLP) [1], [2] often extract HTTP fields to detect covert data channels. Intrustion detection systems [3]–[6] rely on L7 field extraction as a primitive operation. Load balancing devices may extract method names and parameters from flows carrying SOAP [7] and XML-RPC [8] traffic and then route the request to the appropriate server that is best able to respond to the request. Finally, existing network monitoring tools such as SNORT [9] and BRO [10] extract L7 fields for behavioral analysis.

A. Problem Statement

We address the problem of online L7 field extraction that occurs within content-aware networking devices. To do this well, we need to support automatic translation from grammar representations to automata implementations and automated optimization of the resulting automata implementations. Unfortunately, such automated translation and optimization is difficult because network protocols include features that are not easily represented using standard parsing models such as context-free grammars (CFGs) or regular expressions (REs). For example, the HTTP header field, "Content-Length", specifies the length of the HTTP body. Unaugmented, a CFG would require a new rule for each legitimate field length, which makes them impractical for L7 parsing [11], [12].

Online L7 field extraction in a content-aware networking device is fundamentally different than end host protocol parsing because the content-aware network devices must handle millions of concurrent multiplexed network flows. This difference has several technical implications. First, buffering a flow before parsing should be avoided; thus parsing and field extraction should occur incrementally. Second, online L7 field extraction must support efficient context-switching; this requires minimizing the parsing state size of flows. Third, the online L7 field extraction must occur at line-speed.

B. Limitations of Prior Art

Prior online L7 field extraction solutions suffer from one of two drawbacks. They are either hand optimized for better performance [5], or they are derived from an unoptimizable parsing model: recursive descent parsing with code execution [11], [12]. Hand optimized solutions suffer from a high production cost and are prone to errors [11], [12]. The recursive descent solutions offer an excessively rich parsing model that can not be automatically optimized.

To the best of our knowledge, there is no existing solution for online L7 field extraction that supports automated translation from a grammar-based extraction specification to an automata implementation with automated optimization. To illustrate one dimension where previous solutions struggle, we highlight the conflict between automated translation and optimization with line-speed extraction. One technique that can be exploited to achieve line-speed extraction is to ignore (not parse) unnecessary data [5]; we refer to this as *selective parsing*. Previous selective parsing work achieves higher throughput through hand pruning rather than automated translation and optimization [5], [13].

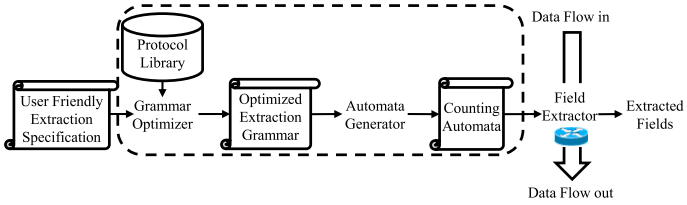

Fig. 1. FlowSifter architecture

C. Proposed Approach: FlowSifter

We propose FlowSifter, a systematic, online L7 field extraction solution, which uses Counting Context-Free Grammars (CCFG), Counting Regular Grammars (CRG) and Counting Automata (CA) to facilitate the automated translation and optimization of L7 field extractors. The architecture of FlowSifter is illustrated in Figure 1. The input to FlowSifter is an extraction specification that specifies the relevant protocol fields. 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. If the corresponding L7 grammar is not in the library, the user can provide a complete extraction specification. 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. The automata generator module takes the optimized extraction grammar as its input and outputs a CA, which is a special type of finite automaton that is augmented with counters. The field extractor module uses the counting automaton to extract relevant fields from data flows. In essence, the counting automaton serves as an L7 protocol configuration for the field extractor module.

Our work recognizes that the automated translation and optimization of an online L7 field extractor requires grammar and automata models that are weaker than standard automata augmented with inline code but richer than finite state automata. CRGs and CA satisfy this requirement and address the other technical challenges. Because CA are state machines, they may be efficiently implemented in either software or hardware. CA have a fixed number of counters rather than a stack, so the parsing state size of any extractor is small and bounded. CRG based extractors are automatically derived from grammar specifications. By using CCFGs to define protocols and extractor specifications, FlowSifter automatically transforms grammar specifications into CRGs, which are executed as CA. For protocols that contain recursively nested fields, FlowSifter uses an approximation method to generate a CRG that navigates the recursive structures of the protocol to locate and extract the desired L7 fields.

D. Key Contributions

In this paper, we make five key contributions in addition to addressing the technical challenges: (1) We propose CRG and CA, a grammar and automata model for selective parsing of L7 protocols with field length descriptors. (2) We propose efficient algorithms for optimizing extraction grammars. (3) We propose automatic generation of stackless parsers from non-regular grammars. (4) We implemented our algorithms and performed experiments on a suite of extraction specifications using real network traces. (5) We compare our algorithms against the state of the art algorithms [10], [13]. In these comparisons, we show that FlowSifter extractors run 85% faster than UltraPAC extractors and four times as fast as BinPAC extractors. We also show that FlowSifter extractors have very small parsing state memory requirements. In our comparisons, the size of a FlowSifter extractor's parsing state is on average eight times smaller than that of the corresponding UltraPAC extractor's parsing state and sixteen times smaller than that of the corresponding BinPAC extractor's parsing state.

II. Related Work

Hand-coded parsing: Although L7 parsers are still predominantly hand coded [11], 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 [11]. 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 [11]. For example, severe vulnerabilities have been discovered in several hand-coded protocol parsers [14]–[19]. Writing an efficient and robust parser is a surprisingly difficult and error-prone process because of the many protocol specific issues and the increasing complexity of modern protocols [11]. For example, the NetWare Core Protocol used for remote file access has about 400 request types, each with its own syntax [11].

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]. Spending scarce CPU and memory resources for parsing useless fields should be avoided for Intrusion Prevention Systems, which are resource constrained. Based on such observations, Schear *et al.* showed that hand-coded selective protocol parsers [5] can run 3 times faster than binpac.

Parsing Generators: Recognizing the increasing demand for L7 parsers and the difficulty in developing robust protocol parsers, three L7 parser generators have been proposed: binpac [11], GAPA [12], and UltraPAC [13]. 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 [20] and ANTLR [21], developed the network protocol parser generator binpac. GAPA, developed by Borisov *et al.*, focuses on providing a protocol specification that guarantees that the generated parser is 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.

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. Binpac uses C++ to specify code blocks and compiles the entire parser into C++ whereas GAPA uses an interpreted language. 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. 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.

III. CCFG Grammars

Given the difficultly of optimizing in-lined code segments, we define a parsing model that augments rules for context-free grammars and regular grammars with counters, guards, and actions. These augmentations increase grammar expressiveness, but the grammars are still amenable to automatic simplification and optimization.

FlowSifter produces an L7 field extractor from two inputs: a protocol specification, and an extraction specification. Protocol specifications are CCFGs that precisely specify how to parse the network protocol. Protocol specifications are generic for any desired extraction and are kept in the protocol library. Extraction specifications specify in detail the exact L7 fields to be extracted. They are written as annotated partial CCFGs, and reference a protocol specification for parts of the grammar that need no special handling.

We next formally define the grammar model for precisely expressing L7 grammars. We then describe FlowSifter's requirements for a user-friendly extraction specification.

A. Formal Definition

Formally, a counting context-free grammar is a five-tuple $\Gamma = (\mathbb{N}, \Sigma, \mathbb{C}, \mathbb{R}, S)$ where $\mathbb{N}, \Sigma, \mathbb{C}$, 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 symbols that will 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, initialized to zero. The counters store parsing information such as the value of length fields. In parsing an HTTP flow, a counter stores 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 predicate evaluates to `true`, and its value does not affect the evaluation of the guard. Guards are used to guide the parsing based on computed values.

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 run 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.

Producing a language from a CCFG works in the same way as a leftmost derivation for a CFG. The derivation starts with the start symbol and with all counters initialized to zero and produces a string of nonterminals by application of the production rules. Each of the following simplifications applies to the leftmost item in the derivation that is not a terminal symbol. If that item is an action, it is removed from the derivation by applying it to the counters. If that item is a nonterminal, it is expanded by replacing it by the body of any of its production rules for which the guard of that rule evaluates to true. Whenever the result of repeating this procedure results in a string of terminals, that string is in the language of the CCFG. This leftmost derivation procedure matches the parsing semantics we will use.

1) Protocol Specification in CCFG: The Varstring CCFG in Figure 2 illustrates how FlowSifter can easily specify an application protocol feature that is difficult for CFGs. The Varstring language consists 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*. We also present a Dyck language CCFG; the Dyck language is the set of strings of balanced parentheses '[' and ']'. We adopt the convention that the head of the first rule is the start nonterminal.

1	$\text{S} \rightarrow \text{B V}$
2	$\text{B} \rightarrow$ '0' $(c := c * 2)$ B
3	$\text{B} \rightarrow$ '1' $(c := c * 2 + 1)$ B
4	$\text{B} \rightarrow$ '␣'
5	$(c > 0)$ $\text{V} \rightarrow \Sigma\,(c := c - 1)$ V
6	$(c = 0)$ $\text{V} \rightarrow \epsilon$

examples: "1 a", "10 ba", "101 xyzab"

(a) Varstring Γ_v

1	$\text{S} \rightarrow \epsilon$
2	$\text{S} \rightarrow \text{I S}$
3	$\text{I} \rightarrow$ '[' S ']'

"[[]]", "[][[][]]"

(b) Dyck Γ_d

Fig. 2. Two protocol specifications in CCFG

We now demonstrate how the Varstring grammar can produce the string "10 ba". Each row of the table in Figure 3 is a step in the derivation. The c column shows the value of the variable c at each step. The number in parentheses is the rule from Figure 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.

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" in Varstring

2) Counting Regular Grammars: Just as parsing with CFGs is expensive, plain CCFGs shouldn't be used for this kind of parsing, as the whole derivation must be tracked. To resolve this, we will convert them to Counting Regular Grammars (CRGs), analogous to Regular Grammars, those grammars parsable without a stack. For CRGs, all rules in the grammar must use one of the following two forms: (1) $\langle \text{guard} \rangle : X \to \alpha \langle \text{action} \rangle Y$ or (2) $\langle \text{guard} \rangle : X \to \alpha \langle \text{action} \rangle$ where X and Y are nonterminals and $\alpha \in \Sigma$. CRG rules that fit equation (1) are the *nonterminating* rules whereas those that fit equation (2) are the *terminating* production rules as derivations end when they are applied. CCFG rules that fit either equation are *regular* rules; other rules are *non-regular* rules. The details of this conversion are given in section IV.

B. Extraction Specification Requirements

FlowSifter's design imposes some requirements on the extraction specification, but also gives it additional freedoms. The extraction specification is a CCFG $\Gamma_x = (\mathbb{N}_x, \Sigma, \mathbb{C}_x, \mathbb{R}_x, S_x)$, but is not required to be complete. It can refer to nonterminals specified in the protocol grammar for its L7 protocol, denoted $\Gamma_p = (\mathbb{N}_p, \Sigma, \mathbb{C}_p, \mathbb{R}_p, S_p)$. However, Γ_x is not allowed to modify Γ_p; its rules must not 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 largest restriction on the extraction grammar is that it must be *normal*; that is, it must be convertible into an equivalent CRG. 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.

1) Extraction Annotations: The purpose of FlowSifter is to call application processing functions on user-specified fields. Based on the extracted field values that they receive, 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. FlowSifter allows calls to these functions in the actions of a rule. Application processing functions can also return a value back into the extractor to affect the remaining parsing. Since the application processing functions are part of the layer above FlowSifter, their specification is beyond the scope of this paper. Further, we include 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.

1 | X → B `vstr`{V}

(a) Varstring Γ_{xv}

1 | X → '[' `parameter`{S} ']' S

(b) Dyck Γ_{xd}

Fig. 4. Two extraction CCFGs Γ_{xv} and Γ_{xd}

We next show two user-friendly extraction specifications that are annotated partial CCFGs. The first, Γ_{xv} in Figure 4(a), specifies the extraction of the variable-length field V for the Varstring CCFG in Figure 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. While the Varstring protocol CCFG is not large, it is much easier to write a one production, incomplete CCFG Γ_{xv} rather than a complete extraction grammar. The second extraction specification, Γ_{xd} in Figure 4(b), is associated with the Dyck CCFG in Figure 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 `parameter`. 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.

IV. Grammar Operations

Given protocol and extraction specifications, FlowSifter automatically produces an optimized CRG extraction grammar. This process is accomplished using three techniques: normal identification, regularization, and counting approximation. Normal identification identifies the nonterminals we will approximate. Regularization converts normal rules into equivalent regular rules. Counting approximation converts the remaining non-regular rules into regular rules that approximately parse recursive L7 field structures. After applying all these processes and inlining rules that do not consume any input, the input CCFGs are converted into a CCFG with regular rules (a CRG), which is suitable for conversion into a CA.

FlowSifter takes the potentially incomplete extraction CCFG Γ_x and the L7 grammar Γ_p and turns them into a complete extraction CRG $\Gamma_f = (\mathbb{N}_f, \Sigma, \mathbb{C}_f, \mathbb{R}_f, S_f)$. Recall that $\mathbb{N}_x$ and $\mathbb{N}_p$ are disjoint and that $\mathbb{R}_x$ may include nonterminals from $\mathbb{N}_p$ only in the result of a production rule. Furthermore, the nonterminals in $\mathbb{N}_x$ do not appear in any production rules in $\mathbb{R}_p$. We refer to $\mathbb{N}_x$ as the *extraction nonterminals* and $\mathbb{N}_p$ as the *protocol nonterminals*.

For a given CCFG $\Gamma = (\mathbb{N}, \Sigma, \mathbb{C}, \mathbb{R}, S)$, we let $\Gamma(X)$ for $X \in \mathbb{N}$ denote the grammar $(\mathbb{N}, \Sigma, \mathbb{C}, \mathbb{R}, X)$; that is, X is the start nonterminal, and we say that X is *normal* if $\Gamma(X)$

is normal. If we treat all protocol nonterminals $Y \in \mathbb{N}_p$ as terminals in Γ_x, then we assume that Γ_x is normal. It follows that for each $X \in \mathbb{N}_x$, X is normal if we treat all protocol nonterminals as terminals. However, it is possible that some protocol nonterminal $Y \in \mathbb{N}_p$ that is reachable from S_x is not normal. For example, $\Gamma_p(Y)$ may define a feature such as nesting of balanced opening and closing parentheses that require unlimited memory to precisely parse.

FlowSifter combines Γ_x with Γ_p to produce a CRG as follows. It creates a new CCFG $\Gamma_f = (\mathbb{N}_x \cup \mathbb{N}_p, \Sigma, \mathbb{C}_x \cup \mathbb{C}_p, \mathbb{R}_x \cup \mathbb{R}_p, S_x)$ and prunes any unreachable nonterminals from this composite. We then partition the nonterminals in $\mathbb{N}_p$ into those we can guarantee to be normal and those we cannot. FlowSifter regularizes normal nonterminals and does counting approximation on those not identified as normal. After FlowSifter replaces each parsing nonterminal using either regularization or counting approximation, FlowSifter then regularizes the extraction nonterminals. If FlowSifter is unable to regularize any extraction nonterminal, it reports that the extraction specification Γ_x needs to be changed and provides appropriate debugging information.

A. Normal Identification

Determining if a context-free grammar 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. Each nonterminal $X \in \mathbb{N}_p$ is normal if

1) $\Gamma_f(X)$ is regular OR
2) For all rules with head X,
   a) X only appears last in the body AND
   b) for every Y that is reachable from X
      - Y is normal AND X is not reachable from Y.

That is, FlowSifter first checks to see if $\Gamma_f(X)$ is regular. If so, it stops and returns that X is normal. Otherwise, it checks each production rule with head X to confirm that if X appears, X is the last symbol in the body. If X appears in a non-final position, FlowSifter decides that it is not normal, even though it may be. Otherwise, FlowSifter finally recursively performs the normal check on any other nonterminals that are reached. When performing these recursive checks, if FlowSifter reaches X again, FlowSifter decides that X is not normal, even though it might be. If any nonterminal that is checked in the process is determined to be not normal, FlowSifter decides that X is not normal, even though it might be. Otherwise, FlowSifter decides that X is normal.

Once FlowSifter has identified each nonterminal as normal or not, we regularize the normal nonterminals as described in Section IV-B and approximate the rest as described in Section IV-C. Since our process for identifying nonterminals is not accurate, we may misidentify a normal nonterminal as not normal. Fortunately, as we will see in Section IV-C, the cost of such a mistake is relatively low; it is only one counter in memory and some unnecessary predicate checks.

B. Regularization

Regularization replaces a normal nonterminal's rules with a collection of equivalent regular rules. The basic idea behind regularization is to use standard decomposition techniques to turn nonregular rules into a collection of equivalent regular rules. Consider an arbitrary nonregular rule $\langle guard \rangle : X \rightarrow \langle body \rangle$. We first express the body as $Y_1 \ \cdots \ Y_n$ where $Y_i, 1 \leq i \leq n$ is either a terminal (possibly with an action) or a nonterminal. Because this is a nonregular 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 use the assumed CRG $\Gamma' = (\mathbb{N}', \Sigma, \mathbb{C}', \mathbb{R}', \mathbb{S}')$ that is equivalent to $\Gamma_f(Y_1)$ to replace the rule as follows. First, we add rule $\langle guard \rangle : X \rightarrow \mathbb{S}'$. Next, for each terminating rule $r \in R'$, we append $Y_2 \ \cdots \ Y_n$ to the body of r and the resulting rule to the regularized rule set. Finally, we add all the nonterminating rules $r \in R'$ to the regularized rule set.
- If Y_1 is a terminal and $n > 2$, the rule is decomposed into two rules: $\langle guard \rangle : X \rightarrow Y_1 \ X'$ and $X' \rightarrow Y_2 \ \cdots \ Y_n$ where X' is a new nonterminal.

1		$\mathrm{S} \rightarrow \mathrm{B}'$
2		$\mathrm{B}' \rightarrow 0\,(c := c * 2)\,\mathrm{B}'$
3		$\mathrm{B}' \rightarrow 1\,(c := 1 + c * 2)\,\mathrm{B}'$
4		$\mathrm{B}' \rightarrow \textvisiblespace\,\mathrm{V}$
5	$(c = 0)$	$\mathrm{V} \rightarrow \epsilon$
6	$(c > 0)$	$\mathrm{V} \rightarrow \Sigma\,(c := c - 1)\,\mathrm{V}$

Fig. 5. Varstring after decomposition of rule S → B V.

For example, consider the Varstring CCFG Γ_v with nonregular rule S→B V. Both $\Gamma_v(B)$ and $\Gamma_v(V)$ are CRGs. Decomposition regularizes $\Gamma_v(S)$ by replacing S→B V by S→ B′ and B →␣ by B′ →␣V. We also add copies of all other rules where we use B' in place of B. Figure 5 illustrates this complete result. Note that 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 decomposition.

C. Counting Approximation

We use counting approximation to produce regular rules for L7 protocol structures that are not normal. We can do this for nonregular subgrammars that have such balanced nesting structures with computable start and end terminals. The basic idea is to parse only the start and end terminals for $\Gamma(X)$ ignoring any other parsing information contained within this subgrammar. By using the counters to track nesting depth, we can approximate the parsing stack for nonterminals in our protocol grammar. We only apply this to nonterminals from $\mathbb{N}_p$, so we don't affect extraction on grammatical streams.

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 we denote as *other*. For example, in the Dyck extraction grammar Γ_{xd} in Figure 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 following rules that use a new counter cnt:

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

Fig. 6. General Approximation Structure

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 Figure 4(b), we get the new production rules in Figure 7.

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

Fig. 7. Approximation of Dyck S

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. Most importantly, the errors introduced do not interfere with field extraction because we do not approximate extraction specification nonterminals.

V. Automata Generator

The *automata generator* module in FlowSifter takes an optimized extraction grammar as its input and generates an equivalent counting automaton, which will serve as the data structure (or say the "configuration") of the field extractor module. Counting Automata (CA) allow efficient use of CRGs in online field extraction by leveraging Deterministic Finite State Automata (DFA) for matching flow data. Much work has been done on efficient implementation of DFAs on network and security devices [22]–[29]. We build on this work by using Regular Expressions as the terminal symbols in our CCFGs and CRGs. This implies that each transition in the resulting CA uses its own DFA to process the flow payload, determine the next CA state, and update the CA counters.

A. Counting Automata

We first define a DFA with labeled decisions. A Labeled DFA is a 5-tuple $DFA(\Sigma, D) = (Q, \Sigma, \delta, q_0, DF)$ where Q is a set of states, Σ is an alphabet, q_0 is the initial state, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function and $DF : Q \rightarrow D$ is a partial function assigning a subset of the states a decision from the decision set D. The notation $DFA(\Sigma, D)$ denotes the set of DFA over an alphabet Σ and a decision set D. A Counting Automata (CA) is a 5-tuple $(Q, \Sigma, C, \delta, q_0, c_0)$ where Q_c is a set of states, Σ is an alphabet, C is a set of possible counter configurations, q_0 is the initial state, and c_0 is the initial counter configuration. The transition function is $\delta : Q \times C \rightarrow \mathrm{DFA}(\Sigma, (Q \times (C \rightarrow C))$; that is, given the current state q_i and counter configuration c_i, the transition function δ specifies a DFA over alphabet Σ whose decision is a state $q_j \in Q$ and an action function act_i that updates the counter configuration.

FlowSifter generates a CA $(Q, \Sigma, C, \delta, q_0)$ from a CRG $\Gamma = (N, \Sigma_g, C_g, R, S)$ as follows. Some components of the grammar are directly inherited by the CA. The states of the CA are exactly the set of nonterminals of the CRG, and the initial state is also the start nonterminal, so $Q = N$ and $q_0 = S$. The CA works over the same alphabet as the Grammar, so $\Sigma = \Sigma_g$. For the set of possible counter configurations C, we assume each counter from C_g has some maximum size, typically $2^{sizeof(int)} - 1$. We could reduce the size of each counter to reduce the final parsing state size of the CA. Formally, $C = \{(c_1, c_2, \ldots, c_{|C_g|}) : |c_i| < 2^{sizeof(int)}, 1 \leq i \leq |C_g|\}$. The most complex assignment is the transition function δ. For each state q, consider the corresponding nonterminal X. For each possible counter configuration $c \in C$, we identify the set of production rules $r(c) \in R$ with head X whose guards are satisfied by c. The body of each rule $r_i \in r(c)$ consists of a regular expression rx_i (the terminals in the CRG are regular expressions), an action act_i which updates some of the counters, and possibly a next nonterminal nt_i. For this q and c, we construct a DFA built from the rx_i of the rules $r_i \in r(c)$. The decision for each rx_i is (act_i, nt_i).

To apply a CA to a flow, we first identify $\delta(q_0, c_0) = dfa_0$ and run this DFA on the flow until it returns a decision (q_1, act_0). If the DFA does not return any decision, the flow does not match the grammar, and we can stop processing. We apply the action function act_0 to get the new counter configuration $c_1 = act_0(c_0)$. We then identify the appropriate DFA $\delta(q_1, c_1) = dfa_1$ which resumes processing the flow. The CA continues in this fashion alternating between CA states where counters are updated and predicates computed and DFA states where flow input is consumed until the entire flow is processed. The parsing state of the CA consists of a DFA state, a counter configuration, plus some flow state variables such as the flow offset that the next DFA should start at.

A CA reports extraction events by having its actions call application processing functions which are defined in the extraction specification. The CA waits for a return value from the called application processing function so it can complete updating 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, so it can immediately return a null value so that the CA can immediately resume processing the input flow.

The text above has assumed that only one regular expression from the rules in $r(c)$ will match the flow data at a time. However, multiple regular expressions may match the same flow data and have different actions. We address this by assigning priorities to the different rules in $r(c)$ and take

these priorities into account when constructing the DFA that corresponds to $\delta(X, c)$. For example, we use

```
HEADER -> /(?i:Content-Length):\s*/
     [bodylength := getnum()];
HEADER 99 -> TOKEN /:/ VALUE;
```

as part of our protocol specification for processing HTTP headers. We give the first rule higher priority which allows us to easily differentiate the special case of `TOKEN` where the header name is "Content-Length" so we can act differently.

VI. Experimental Evaluation

We evaluate field extractor performance in three areas: speed, memory and extractor definition complexity. 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. Lastly, the complexity of writing field extractors determines the rate at which new protocol field extractors can be deployed.

A. Methods

1) Traces: 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 is interleaved packets from multiple flows. In contrast, previous work has used traces that consist of pre-assembled complete 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.

Our HTTP packet data comes from the MIT Lincoln Lab's (LL) DARPA intrusion detection data sets [30]. 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 ($<$ 25MB) from our test data sets to improve timing accuracy. This left 45 test traces, with between 0.16 and 2.5 Gbits of data and between 27K and 566K packets per trace.

2) Field Extractors: Our FlowSifter implementation is written in 1900 lines of Objective Caml (excluding DFA generation) and runs on a desktop PC running Linux 2.6.35 on an AMD Phenom X4 945 with 4GB RAM. It generates the CA from protocol and extraction grammars and simulates it on trace payloads. The implementation includes a few optimizations not documented here for space reasons.

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`, `header name`, and `header value`. 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 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.

3) Metrics: For any trace, there are two basic metrics for measuring a field extractor's performance: *parsing speed* and *memory used*. We define a third metric, *efficiency*, which we define as the parsing speed divided by the $\log_{10}$ of the memory needed. Higher efficiency indicates higher speed with less memory needed.

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 could 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 measure the memory taken by a field extractor on a trace by measuring the memory use of the extractor before and right at the end of processing the given trace and taking the difference. BinPAC and UltraPAC use manual memory management, so we measure memory use by using tcmalloc's [31] `generic.current_allocated_bytes` parameter.

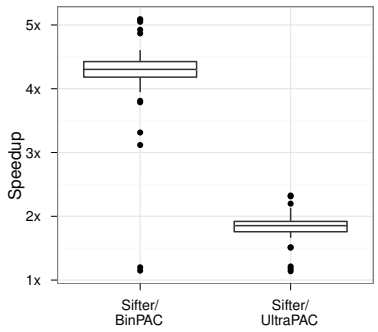

Fig. 12. Boxplots of speedup on 45 LL traces

This allows us to precisely identify the exact amount of memory allocated to the extractor and not yet freed. Since FlowSifter runs in a garbage collected environment, its environment provides an equivalent measure of live heap data.

B. Experimental Results

We show empirical CDFs for all three field extractors' memory usage, parsing speed and efficiency on the 45 Lincoln Labs traces in Figure 8. By all three metrics of memory use, parsing speed, and efficiency, FlowSifter clearly outperforms both BinPAC and UltraPAC.

1) Parsing Speed: As shown in Figure 12, FlowSifter parses the input faster than either BinPAC or UltraPAC. On average, FlowSifter runs 4.1 times faster than BinPAC and 1.84 times faster than UltraPAC.

FlowSifter's optimal DFA parsing speed is 1.8Gbps. We determined this by running a simple DFA on a simple input flow. As shown in Figure 8, FlowSifter can run both faster and slower than 1.8Gbps. 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.

However, the CA introduces two factors that can lead to slower parsing: evaluating expressions and context switching. In our current implementation, both predicates and actions are interpreted. A more efficient implementation could compile these so they run at the speed of the processor. Each CA transition also leads to potentially a new DFA that will process the next piece of the flow. FlowSifter suffers a context switching cost with each such DFA change.

To test FlowSifter's approximation performance, we made a SOAP field extractor that extracts a single 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 Figure 9. As we expected, FlowSifter's SOAP field extractor had a slower parsing speed than FlowSifter's HTTP field extractor. There are two main reasons for the slowdown. First, there are fewer opportunities for selective parsing. For example, FlowSifter cannot skip any fields such as the HTTP body. Second, 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: Each point in Figure 10 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. This is due to FlowSifter's use of a fixed-size array of counters to store almost all of the parsing state. BinPAC and UltraPAC use much more memory respectively averaging 5.5KB and 2.7KB per flow. This is mainly due to their buffering requirements, as 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 not enough flow to finish, they buffer additional content before trying to match again.

3) Parser Definition Complexity: The final point of comparison is 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. Figure 11 shows a 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.

VII. Conclusions

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.

References

[1] K. Borders and A. Prakash, "Web tap: Detecting covert web traffic," in *Proc. CCS*, 2004.

[2] ——, "Towards quantification of netword-based information leaks via HTTP," in *Proc. USENIX Hotsec*, 2008.

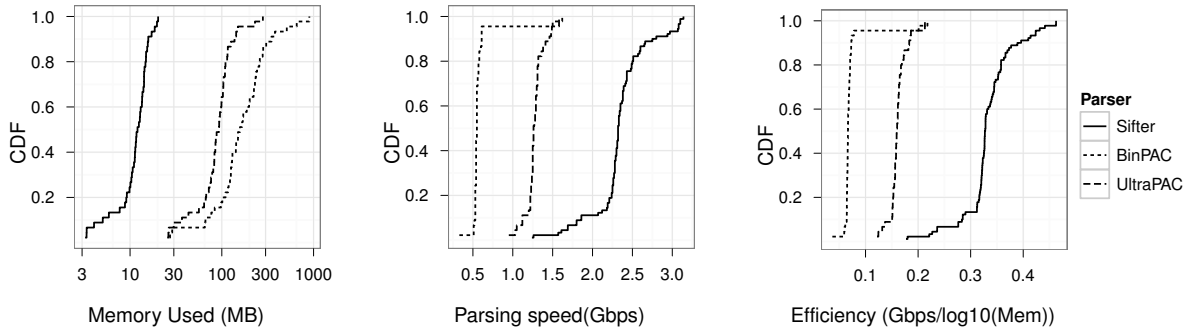

Fig. 8. CDF comparison of HTTP Field Extractors on 45 LL traces

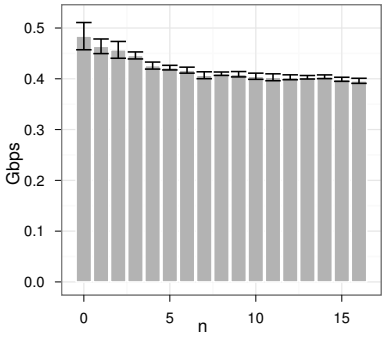

Fig. 9. Average parsing speed for SOAP-like flows versus recursion depth n

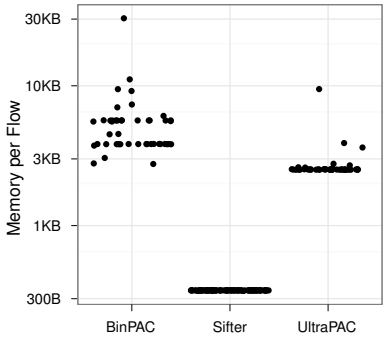

Fig. 10. Memory per flow on LL traces

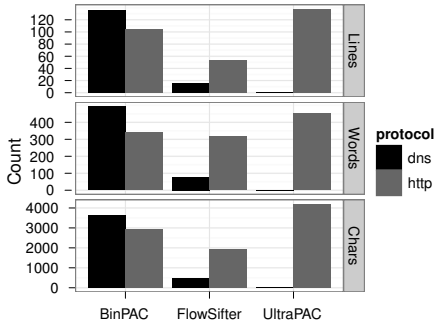

Fig. 11. Complexity of base protocol parsers

[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.

[4] Z. Li, L. Wang, Y. Chen, and Z. Fu, "Network-based and attack-resilient length signature generation for zero-day polymorphic worms," *IEEE Int. Conf. Network Prorocols (ICNP)*, pp. 164–173, 2007.

[5] N. Schear, D. R. Albrecht, and N. Borisov, "High-speed matching of vulnerability signatures," in *Proc. Int. Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[6] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," *IEEE Symposium Security and Privacy*, 2007.

[7] "Simple object access protocol (soap), www.w3.org/tr/soap/."

[8] "XML-RPC, http://www.xmlrpc.com/spec."

[9] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *Proc. 13th Systems Administration Conference (LISA), USENIX Association*, November 1999, pp. 229–238.

[10] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999. [Online]. Available: citeseer.ist.psu.edu/paxson98bro.html

[11] R. Pang, V. Paxson, R. Sommer, and L. Peterson, "binpac: a yacc for writing application protocol parsers," in *Proc. ACM Internet Measurement Conference (IMC)*, 2006.

[12] N. Borisov, D. J. Brumley, and H. J. Wang, "A generic application-level protocol analyzer and its language," in *Proc. Network and Distributed System Security Symposium (NDSS)*, 2007.

[13] Z. Li, G. Xia, H. Gao, Y. Tang, Y. Chen, B. Liu, J. Jiang, and Y. Lv, "NetShield: Massive semantics-based vulnerability signature matching for high-speed networks," in *Proc. SigCOMM*, 2010.

[14] "Ethereal OSPF protocol dissector buffer overflow vulnerability. http://www.idefense.com/intelligence/vulnerabilities/display.php?id=349."

[15] "Snort TCP stream reassembly integer overflow exploit, http://www.securiteam.com/exploits/5bp0o209ps.html."

[16] "tcpdump ISAKMP packet delete payload buffer overflow. http://xforce.iss.net/xforce/xfdb/15680."

[17] "Symantec multiple firewall NBNS response processing stack overflow. http://research.eeye.com/html/advisories/published/AD20040512A.html."

[18] C. Shannon and D. Moore, "The spread of the witty worm. http://www.caida.org/research/security/witty/."

[19] A. Kumar, V. Paxson, and N. Weaver, "Exploiting underlying structure for detailed reconstruction of an internet-scale event," in *Proc. ACM Internet Measurement Conference (IMC)*, 2005.

[20] S. C. Johnson, "Yacc - yet another compiler-compiler," Bell Laboratories, Technical Report 32, 1975.

[21] T. T. J. Parr and R. R. W. Quong, "Antlr: A predicated-ll(k) parser generator," *Software, Practice and Experience*, vol. 25, 1995.

[22] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," in *Proc. INFOCOM*. IEEE, 2007.

[23] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proc. ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*, 2007.

[24] C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high speed networks," in *Proc. 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Washington, DC, 2004.

[25] S. Dharmapurikar and J. W. Lockwood, "Fast and scalable pattern matching for network intrusion detection systems," *IEEE Journal on Selected Areas in Communications*, vol. 24, pp. 1781 – 1792, 2006.

[26] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proc. ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*, 2007, pp. 155–164.

[27] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proc. SIGCOMM*, 2006, pp. 339–350.

[28] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," in *Proc. SIGCOMM*, 2008, pp. 207–218.

[29] S. Kong, R. Smith, and C. Estan, "Efficient signature matching with multiple alphabet compression tables," in *Proc. SecureComm*, 2008.

[30] "Darpa intrusion detection evaluation data set," www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/1998data.html, 1998.

[31] "Tcmalloc," http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2011.