Protocol Normalization Using Attribute Grammars

Drew Davidson[1], Randy Smith[1], Nic Doyle[2], and Somesh Jha[1]

[1] Computer Sciences Department, University of Wisconsin, Madison, WI 53706
[2] ERBU XE Security group, CISCO systems

Abstract. Protocol parsing is an essential step in several networking-related tasks. For instance, parsing network traffic is an essential step for Intrusion Prevention Systems (IPSs). The task of developing parsers for protocols is challenging because network protocols often have features that cannot be expressed in a context-free grammar. We address the problem of parsing protocols by using attribute grammars (AGs), which allow us to factor features that are not context-free and treat them as attributes. We investigate this approach in the context of protocol normalization, which is an essential task in IPSs. Normalizers generated using systematic techniques, such as ours, are more robust and resilient to attacks. Our experience is that such normalizers incur an acceptable level of overhead (approximately 15% in the worst case) and are straightforward to implement.

1 Introduction

Parsing application-layer protocols is a fundamental step in several networking-related tasks. Programs that operate over application-level traffic semantics, such as systems that investigate Email traffic and Internet attacks, use a protocol parser as an integral component. Parsing network traffic is also an essential step for Intrusion Prevention Systems (IPSs) because protocols allow many representations of the same message. *Protocol normalization* is meant to reverse the transformations and obfuscations that an attacker performs on a message to a canonical form [7]. An IPS that does not perform normalization is vulnerable to evasion attacks [7,15,17]. In order to perform normalization, IPSs must know certain fields in a protocol, e.g., to normalize URLs an IPS system has to extract the URL field from HTTP traffic. In this paper we focus on protocol parsing in the context of intrusion prevention, but the results are applicable to related areas such as firewalls, URL filtering, and HTTP server load balancing.

At first glance implementing application protocol parsers seems like a straightforward task. One strategy would be to use standard parser generators such as *yacc* [9] or *ANTLR* [14] to implement an application protocol parser. This strategy often does not work, however, because many protocols have constructs that are not context-free. For example, data fields that are preceded by their actual length (which is common in several network protocols) cannot be expressed in a context-free grammar [13]. In this work we consider a systematic approach to the

M. Backes and P. Ning (Eds.): ESORICS 2009, LNCS 5789, pp. 216–231, 2009.

problem of parsing application protocols with features that are not context-free by using attribute grammars. We posit that a systematic approach to generating parsers leads to more robust applications.

Formally, an *attribute grammar (AG)* [8,12] is a way to define *attributes* for the productions of a *grammar*, associating these attributes to values. The evaluation occurs in the nodes of the *abstract syntax tree (AST)*, when the language is processed by a parser. The attributes are divided into two groups: *synthesized* attributes and *inherited* attributes. The synthesized attributes are the result of the attribute evaluation rules, and may also use the values of the inherited attributes. The inherited attributes are passed down from parent nodes. Attributes have been used in the past for network protocol parsing [1]. They are a natural and systematic way to represent context-sensitive features of network protocols, such as fixed-length bodies of HTTP messages. Once we have an attribute grammar for a protocol, we can use it to generate a normalizer for the protocol which can be deployed in a IPS. This paper makes the following contributions:

- We propose using Attribute Grammars to generate parsers for common network protocols. We compare parsers generated using our approach to existing parsers for these protocols. We find that expressing the syntax of network protocols as attribute grammars helps to clarify other tasks related to parsing, such as protocol normalization.
- We demonstrate the practicality of the Attribute Grammar approach by implementing an AG-based normalizer directly into the popular IPS `Snort`. We show that our normalizer is more principled than the unmodified version of `Snort`, and that our normalizers only incur a modest performance penalty of 15.5% in the worst case. We have made our normalizer publicly available at `http://www.cs.wisc.edu/~davidson/ag_normalizer`.
- We show that our approach can be adopted easily by using existing tools such as *bison* and *flex*. These tools have the advantage of being well tested, widely deployed, and accessible via a familiar syntax.

2 Related Work

IPS evasion was first explored by Ptacek and Newsham [15]. They pioneered a number of techniques to transform a malicious payload to escape signature detection. The most relevant technique to our work is the evasion attack, where an attacker crafts packets that are accepted by an end-system but rejected by a signature matcher. Handley, Paxson, and Kreibich [7] introduced the *normalizer* as a software module to eliminate potential ambiguities in a packet stream to detect evasion attacks. While this work operated at a lower level in the protocol stack than our work, it forms the foundation of protocol normalization for signature matching.

The difficulty of parsing network protocols due to context-sensitive features was first observed by Pang *et al.* [13]. They implemented a tool called *binpac* to address this problem. Although *binpac* was a good first step towards addressing

this problem, it is not as disciplined as standard parser generators (such as *yacc* and *ANTLR*). Moreover, the syntax of the specification language of *binpac* is new, so it would require users well versed in existing parser generator tools to learn an entirely new set of constructs. A tool in a similar vein is GAPA [2], which uses a custom protocol description language to build protocol parsers. Unlike our tool, GAPA is meant for protocol analysis, rather than normalization. Our approach of using attribute grammars enables us to use familiar parser generators, which use syntax that users already know.

Chapman [5] suggested using AGs to specify network protocols. Chapman focused on formalising protocols using attribute grammars in order to characterize protocol properties, such as deadlock proneness. Chapman did not consider the application to protocol normalization. His attribute grammars only accept valid input and reject invalid, they do not perform any transformation on the input stream. Our technique is meant to be integrated into an online system that performs a larger task of which parsing is a critical step, such as in an IPS. We take inspiration from this work to use AGs as a basis for specifying network protocols.

Anderson and Landweber [1] explored extensions of AGs to specify various network protocols. They introduced a formalism called Real-time Asynchronous Grammars (RTAG) for specifying protocols. In RTAG, terminal symbols of a parse tree correspond to messages sent and received by the protocol. Each production in RTAG could have a Boolean expression over attribute values called a *start condition*. In order for the production to be evaluated, the start condition must first evaluate to *true*. Anderson and Landweber did not consider an application of their technique to normalization. Our technique is a more fine-grained approach than that taken by Anderson and Landweber, as we use tokens from an input stream as our terminal symbols rather than messages or events.

3 Overview

We motivate our technique by explaining some of the difficulties in parsing network protocols using a running example of a fragment of the HTTP protocol. We selected this protocol as our central example because it is a popular protocol that has historically been a vector for numerous attacks. HTTP also contains features that exemplify the obstacles to using principled parsing techniques. In addition to HTTP, we have also applied our techniques to the FTP and SMTP protocols.

3.1 HTTP Protocol Characteristics

Attackers can gain unauthorized, privileged access to an HTTP server by remotely supplying malicious payloads designed to trigger a vulnerability in the remote host. For example, the DNS-tools attack (CVE-2002-0613 [11]) allows a malicious client to gain administrator privileges on a DNS server using the `DNStool version 2.0 beta 4` auto configuration system, by placing the string shown in Figure 1 in the URL of an HTTP get request.

Table 1. URL String Encodings in HTTP

Encoding	Description	Example
UL	Convert lowercase letters to uppercase	ATTACK
percent	Replace characters with a corresponding hexadecimal value	%61%74%74%61%63%6b
percent + UL	Apply percent encoding followed by UL encoding	%61%74%74%61%63%6B
UL + percent	Apply UL encoding followed by percent encoding	%41%54%54%41%43%4b
double percent	Apply percent decoding twice	%25%36%31%25%37%34 %25%37%34%25%36%31 %25%36%33%25%36%62

```
dnstools.php?section=hosts&user_logged_in=true
```

Fig. 1. Attack URL string for the DNS-tools attack

Intrusion prevention systems (IPS) have arisen as a necessary layer of defense to identify and filter out such attacks. For the exploit above, a simple signature can identify and remove packets that carry this malicious payload and inform an administrator of attempted attacks.

Unfortunately, writing a database of attack signatures is not straightforward. As per the HTTP standard [6], URL strings may be alternatively encoded in a variety of ways without changing the semantics. Table 1 shows some of these encodings with examples over the URL string `attack`. In the *UL* encoding, lowercase letters are transformed to uppercase. Beyond case-sensitivity, syntactic isomorphisms such as the *percent* encoding allow for further encoding, transforming a character to a percent followed by two hexadecimal characters representing the ASCII equivalent of that value. Multiple encodings may be applied to the same string and may overlap. Note that the ordering of these encodings alters the appearance of the final URL string.

Finally, individual servers may vary from the formal HTTP specification and allow additional semantics-preserving transformations. A bug in older versions of Microsoft IIS causes the server to perform percent decoding routines twice [4]. For example, `%25%35%30` decodes to `%50` after one pass, and `%50` decodes to `P` on a second pass. This transformation is shown as *double percent* in Table 1. By applying all of these encodings simultaneously to the malicious URL in the DNS-tools attack, one can transform the URL string in Figure 1 to the one in Figure 2. Each of these alternate encodings have the same semantics and are characteristic of the thousands of distinct, specific exploits that target this general vulnerability.

As this example illustrates, writing distinct signatures for each exploit quickly becomes untenable and stresses a signature matching engine. Higher-level vulnerability signatures [3] can reduce the number of signatures needed in some

cases, but the problem here stems from variations in the *encoding* itself rather than in distinct vulnerabilities. Thus, an IPS typically includes a normalizer module to decode the alternate encodings that are part of the HTTP standard and also those encodings that are the result of bugs in popular software. The IPS then only matches decoded strings against canonical signatures.

3.2 Normalization for Context-Free Grammars

Rather than attempting to hand-code a normalizer, we propose formalizing a protocol using a grammar and adding normalizing transformations to the productions of that grammar. Our motivation for this approach is that writing code for a correct normalizer is a difficult and error-prone task. There are many complications that give rise to this difficulty: The normalizer must be able to discern which fields of the protocol are appropriate to normalize, it must be aware of each possible encoding, and it must be able to account for multiple encodings being applied to the same token (for example, UL and percent encoding). Also, the normalizer must be extensible, since it may be necessary to normalize new encodings as the standard for new protocols evolve and new bugs are found that create unintended encodings. Dealing with normalization as a grammar parsing problem allows one to create a declarative specification of the protocol and think compositionally. This in turn ensures that the appropriate normalizations are applied to the correct fields, and makes it easier to deal with multiple encodings on the same token. As a toy example of this technique, we show a restricted *context-free grammar* (CFG) for HTTP URLs, and demonstrate how the grammar can be extended to achieve normalization. In the next section, we will show that a context-free grammar is not powerful enough to recognize the syntax of full network protocols like HTTP.

Definition 1. *A CFG is a four-tuple* (T, N, P, Z) *where*

- T *is a set of terminal symbols*
- N *is a set of non terminal symbols*
- P *is a set of productions, of the form* $\alpha \rightarrow \gamma_1, \ldots, \gamma_n$ *where* $\alpha \in N$ *and* $\gamma_i \in (T \cup N), 1 \leq i \leq n$
- $Z \in N$ *is the start symbol.*

Figure 3(a) shows a CFG for URL strings that can have the UL or percent encodings described Table 1. Many of the rules in this grammar have an intuitive correlation with encodings. However, a context-free grammar such as in Figure 3(a) is unsuitable for normalization, because it has no concept of output; it may only accept valid strings and reject invalid ones. One possible approach would be to add output rules directly to the Context-Free Grammar, but that

```
dNsToOls.%25%35%30h%25%35%30?section=hosts&user_logged_in=true
```

Fig. 2. Obfuscated URL string for the DNS-tools attack

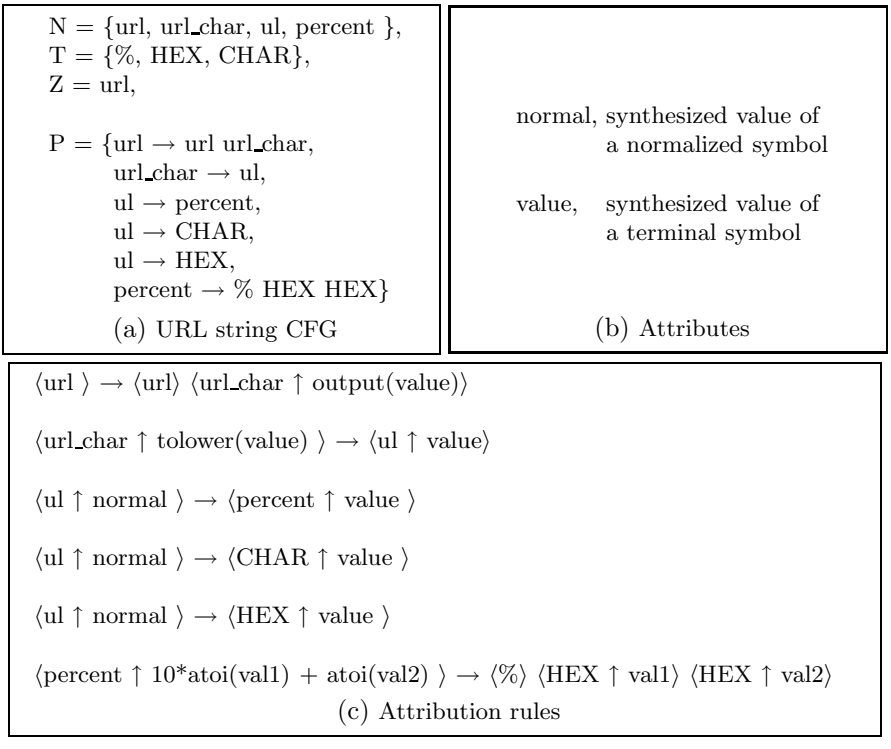

Fig. 3. URL string Normalizer

approach has the disadvantage that every character needs to be represented as a distinct symbol. The resultant explosion in symbols and productions is cumbersome. Instead, we extend our CFG to an *attribute grammar* (AG).

Definition 2. *An attribute grammar is a 3-tuple* (G, A, R)*, where*

- $G = (T, N, P, Z)$ *is a context-free grammar.*
- A *is a finite set of attributes. The finite set of attributes* $A(X)$ *is associated with each symbol* $X \in T \cup N$*. A is partitioned into disjoint subsets* $I(X)$*, the inherited attributes, and* $S(X)$*, the synthesized attributes. A is defined as*

$$A(X)|X \in T \cup N$$

- R *is a finite set of attribution rules. A production*

$$p : X_0 \rightarrow X_1 ... X_n | (n \geq 0, p \in P)$$

has an attribute occurrence $X_i.a$ *if* $a \in A(X_i)$*,* $0 \leq i \leq n$*. A finite set of attribute evaluation rules* R_p *is associated with the production p with exactly one rule for each synthesized attribute occurrence* $X_0.a$ *and exactly one rule for each inherited attribute occurrence* $X_i.a$*,* $1 \leq i \leq n$*. Thus, an attribute of node t is synthesized if it is computed within the subtree rooted at t, and inherited if it is computed outside of the subtree rooted at t.*

We adopt the notation used by Chapman [5] to specify our attribute grammars. This notation replaces the productions of a CFG with *attributed symbol forms*, each consisting of a (terminal or nonterminal) symbol followed by evaluation rules for that symbol's attributes. Evaluation rules for synthesized attributes are preceded by ↑, inherited ones by ↓. Each *attributed symbol form* is enclosed in angle brackets.

Intuitively, an AG allows the underlying value of a symbol to be carried through the parse tree. For example, a HEX token may represent all strings of hexadecimal digits, and an attribute `val` to capture the numeric value of those digits. Attribute evaluation rules may then apply some function to transfer attributes between rules. This flow of attributes corresponds naturally to normalization. Encoded tokens can easily be represented as attributes of terminal symbols and normalized tokens as attributes of nonterminal symbols. In this way, normalization is completely embedded in the attribution rules of the grammar. We allow the special function `output(`η`)` to occur within R_p to indicate that attribute η should be output as a normalized token. The normalizing extensions to Figure 3(a) are shown in Figure 3(c).

3.3 Normalization for Context Sensitive Grammars

Despite the advantages offered by grammar-based parsers, to our knowledge all modern IPS normalizers are created using ad-hoc techniques; they are either hand-coded, or they use parser generators that are not based on any abstract data structure. This is because the syntax of network protocols is not context-free.

As a specific example of a context sensitive behavior, consider the *HTTP Chunked-Body* type. Chunked bodies allow a message to be sent in pieces called *chunks* over a persistent connection [6]. This is done to improve the efficiency of a transmission, as it allows one party to begin sending data before they know exactly how many bytes are going to be sent, or to avoid the overhead of reestablishing a connection [10]. Figure 4(a) shows the excerpt from the HTTP RFC [6] that pertains to HTTP chunks. Each `chunk` symbol begins with a single line of hexadecimal digits called the `chunk-size`, followed by a stream of data that constitutes the `chunk-data`. The size of the chunk-data must be equal to the value of the chunk-size. The first entry in Figure 4(b) shows a valid HTTP chunk. Note that the *chunk-size* has the value `4`, and the *chunk-data* (`GOOD`) is 4 bytes long. Contrawise, The second entry is invalid, because the *chunk-size* has the value `3` and the length of the *chunk-data* is `5`. An ad-hoc parser might enforce this condition by initializing a counter with the value of the `chunk-size` and then decrementing that counter value for each byte in the `chunk-data`, finishing the chunk-data when the counter has reached zero. Since there is no bound on the value of a chunk-size, there is no practical way to represent this relationship using a Context-Free Grammar [13].

In practice, HTTP contains many features that cannot be parsed with a context-free grammar. Table 2 lists several such context sensitive constructs from HTTP. In each of these examples, the value of some field being parsed affects the

⟨chunked_body⟩ → ⟨chunk⟩ ⟨headers⟩ CRLF
⟨chunk ↑ 0 ⟩ → ⟨HEX ↑ *value*⟩ CRLF
⟨chunk ↑ *length*⟩ → ⟨chunk⟩
⟨chunk_size ↑ *length* ⟩ CRLF
⟨chunk_data ↓ *length*⟩ CRLF
⟨chunk_size ↑ *value*⟩ → ⟨HEX ↑ *value*⟩
⟨chunk_data ↓ 1⟩ → DATA
⟨chunk_data ↓ *length*⟩ → ⟨chunk_data ↓ (*length* − 1)⟩ DATA
⟨headers⟩ → headers header CRLF
⟨header⟩ → `CONTENT_LOCATION` url
⟨url ↑ output(value) ⟩ ⟨url⟩ ⟨url_char ↑ value⟩
⟨url_char ↑ tolower(value) ⟩ → ⟨ul ↑ value⟩
⟨ul ↑ normal ⟩ → ⟨percent↑ value ⟩
⟨ul ↑ normal ⟩ → ⟨CHAR↑ value ⟩
⟨ul ↑ normal ⟩ → ⟨HEX↑ value ⟩
⟨percent ↑ 10*atoi(val1) + atoi(val2) ⟩ → ⟨%⟩ ⟨HEX ↑ val1⟩ ⟨HEX ↑ val2⟩

(a) HTTP Chunk EBNF fragment

Data Stream	chunk-size	chunk-data	valid
`4\r\nGOOD\r\n`	4	`GOOD`	Yes
`3\r\nNOTSO\r\n`	3	`NOT`	No

(b) HTTP Chunk examples

Fig. 4. HTTP Chunk

interpretation of fields to appear later in the token stream, which prevents the use of a CFG. In order to properly represent these constructs, we have selected *Higher-Order Attribute Grammars* (HAGs) [18] as our formalism. Intuitively, a HAG is an Attribute Grammar in which attributes can appear in the left-hand side of a production. This extension allows a grammar to select amongst syntactically equivalent rules based on the value of an attribute from earlier in the parse. This extension is necessary for constructs like the `chunk-data` symbol of Figure 4(a) , where the productions

$$\langle \text{chunk\_data} \downarrow 1 \rangle \rightarrow \text{DATA}$$

$$\langle \text{chunk\_data} \downarrow \textit{length} \rangle \rightarrow \langle \text{chunk\_data} \downarrow (\textit{length} - 1) \rangle \text{DATA}$$

are only distinguishable based on the value of the *length* attribute.

In our experience, this extension is sufficient for parsing the context sensitive features of network protocols. Our strategy for creating network protocol normalizers is as follows:

1. Create a Context-Free Grammar for as much of the protocol as possible. In our experience, most network protocols are largely context-free, with a smattering of context sensitive features.
2. Extend the Context-Free Grammar to an Attribute Grammar. Since the symbols of the underlying Context-Free Grammar are in close correspondance

Table 2. Context Sensitive Constructs of HTTP

Construct	Context Sensitive Aspect
HTTP chunk	length of field specified in a preceding field
fixed length body	length of field specified in a preceding field
HTTP mime type	field delimiter specified in a preceding field

with the normalizations, this step consists of adding attribute evaluation rules that specify how the attributes of encoded symbols are transformed into normalized ones. For tokens that are completely normalized, the special attribute `output` is added to denote that the attribute may be placed in the output stream of the normalizer. As an example of the `output` attribute, consider the `url` token of Figure 4(a). Since all normalizations are applied in sub-rules, if a url token is produced, it is in fully normalized form.

3. Extend the Attribute Grammar to a Higher-Order Attribute Grammar. Pang *et. al.* observed that they syntax of most protocols does not require lookahead in the grammar [13]. This observation, in practice, means that the rules that most context sensitive features of network protocols can be captured by simple attribute evaluation rules in the left-hand side of a production.

4 Technical Details

In this section, we explain how a network protocol normalizer can be specified using a HAG. We demonstrate that a practical implementation of a HAG-based normalizer can be achieved using the parser generator `bison`. We use the running example of an *HTTP Chunked-Body* to demonstrate the need for a HAG, since it is one of the context-sensitive feature of HTTP.

Higher-Order Attribute Grammars

In a conventional attribute grammar, no part of the structure of the parse tree may be defined by means of an attribute value, and vice-versa[18]. For languages with context-free syntax but context sensitive semantics, this boundary does not present a limitation (C is such a language, as variables need to be declared with a type before they are used). Protocols like HTTP have a different form; the syntax of fields is altered by preceding fields. One way to recognize fields of this form is to allow the left hand side of a production to have a attribute in the defining position. This extension allows productions to be applied to an input stream based on the value of attributes. These extensions classify our attribute grammar as a HAG.

⟨chunked_body⟩ → ⟨chunk⟩ ⟨headers⟩ ⟨CRLF ↑ output(*value*) ⟩
⟨chunk ↑ 0 ⟩ → ⟨HEX ↑ output(*value*)⟩ ⟨CRLF ↑ output(*value*) ⟩
⟨chunk ↑ *length*⟩ → ⟨chunk⟩
⟨chunk_size ↑ *length* ⟩ ⟨CRLF ↑ output(*value*) ⟩
⟨chunk_data ↓ *length* ⟩ ⟨CRLF ↑ output(*value*) ⟩
⟨chunk_size ↑ *value*⟩ → ⟨HEX ↑ *value*⟩
⟨chunk_data ↓ 1⟩ → ⟨DATA ↑ output(*value*) ⟩
⟨chunk_data ↓ *length*⟩ → ⟨chunk_data ↓ (*length* − 1)⟩ ⟨DATA ↑ output(*value*) ⟩
⟨headers⟩ → ⟨headers⟩ ⟨header⟩
⟨header⟩ → ⟨`CONTENT_LOCATION` ↑ output(*value*)⟩ url ⟨CRLF ↑ output(*value*) ⟩
⟨url⟩ → ⟨url⟩ ⟨url_char ↑ output(*value*)⟩
⟨url_char ↑ tolower(*value*) ⟩ → ⟨ul ↑ *value* ⟩
⟨ul ↑ *normal* ⟩ → ⟨percent ↑ *value* ⟩
⟨ul ↑ *normal* ⟩ → ⟨CHAR ↑ *value* ⟩
⟨ul ↑ *normal* ⟩ → ⟨HEX ↑ *value* ⟩
⟨percent ↑ 10*atoi(*val1*) + atoi(*val2*) ⟩ → ⟨%⟩ ⟨HEX ↑ *val*1⟩ ⟨HEX ↑ *val*2⟩

(a) Attribution rules

Fig. 5. HTTP Grammar Fragments for the HTTP Chunked-Body

Figure 5 shows an excerpt from RFC 2616 showing the structure of the *HTTP Chunked-Body*, represented by the symbol `chunked_body`. A `chunked_body` is made up of a sequence of 1 or more `chunk` symbols followed by a sequence of 0 or more `header` symbols. Each `chunk` symbol begins with a single line of hexadecimal digits called the `chunk-size`, followed by a stream of data that constitutes the `chunk-data`. The context-sensitive aspect of the `chunk` is that the length of the `chunk-data` must be equal to the value of the `chunk-size`.

A naive normalizer might skip over a sequence of `chunk` symbols entirely, since no terminal in the sequence requires normalization. This approach is insufficient because a normalizer must be able to recognize the `headers` symbol, which does may include url normalizations. Note that this is handled in the normalizer shown here by parsing the chunk structure, and simply applying the output function at every terminal symbol.

4.1 Evaluation Strategy

A traditional HAG is used for parsing features of context sensitive languages, so the primary challenge of this work is to embed the task of normalization in the productions, attributes and symbols of an attribute grammar. We detail our evaluation strategy by stepping through the example of parsing the context-sensitive *HTTP Chunked-Body* construct.

Symbols: There are two types of symbols in a Higher-Order Attribute Grammar:

– Terminal symbols of our HAG correspond to unnormalized symbols from the input stream. We rely on a lexical analysis to determine how to tokenize

characters. The `chunk` can have three different terminals in its subtree: CRLF, denoting the carriage-return and line-feed combination, HEX, denoting a hexadecimal value, and DATA to represent any single byte.
- Nonterminal symbols represent a normalized construct in the protocol. We parse input streams in a bottom-up fashion, so when a symbol is added to the tree, it represents a normalized form of an underlying symbol. Consider the `ul` symbol, which represents a lowercase symbol that may appear in a `uri`.

Attributes: We use the attributes of a HAG for three purposes:

- Attributes reduce the number of symbols in the grammar. For example, rather than using a separate symbol for each ASCII character, an ASCII token is given a *value* attribute denoting that ASCII value. The net result of reducing the number of symbols is to make the grammar more concise and reduce the memory needed by the parser.
- Attributes can represent the normalized value of a token. For example, the `ul` nonterminal has an attribute *normal* to denote that value of a symbol in lowercase form.
- Attributes on the left-hand side of a production can be used to guide the syntactic interpretation of terminal symbols. For example, the *length* attribute of a `chunk_size` symbol controls the number of tokens that can exist in a sequence of `chunk_data` symbols. When the first element in the sequence is parsed, it inherits the length attribute from the preceding `chunk_size` symbol. Every other element in the sequence gets the attribute from the preceding `chunk_data` in the sequence, decremented by 1. If *length* is greater than 1, a `DATA` token is parsed and another `chunk_data` symbol is expected as the next symbol in the parse. If the length attribute is 1, a `DATA` token is parsed, and the next token expected is a CRLF to end the `chunk`.

Productions: Productions in our HAG provide two purposes:

- Productions specify the flow of attributes between symbols. The rule

$$\langle \text{chunk\_size} \uparrow \textit{value} \rangle \rightarrow \langle \text{HEX} \uparrow \textit{value} \rangle$$

  specifies that the synthesized attribute *value* of the nonterminal symbol chunk_size gets the value of the synthesized attribute *value* from the terminal symbol HEX.
- Productions specify normalizations. Consider the following production:

$$\langle \text{url\_char} \uparrow \text{tolower}(\textit{value}) \rangle \rightarrow \langle \text{ul} \uparrow \textit{value} \rangle$$

  This production specifies that a url_char symbol has the lowercase value of the ul symbol. This production corresponds to decoding the UL encoding for URLs.
- Productions specify the structure of a parse tree. This is consistent with the purpose of a parse tree in a Context-Free Grammar.

4.2 Implementation Details

We have implemented proof-of-concept normalizers for HTTP, SMTP, and FTP using unmodified versions of the parser generator `bison` and the lexer generator `flex`. Although `bison` is capable of representing AGs, it does not have a built-in facility for parsing context-sensitive features in the way that they may be presented by a HAG. We simluate the ability to evaluate attribute rules on the left-hand side of a production by manipulating a global variable *context*. The value of context is checked every time the lexer parses a rule, and maps to a given `start condition`, which in turn selects a subset of the lexical rules.

Conceptually, this gives a user the ability to switch tokenizers whenever a token is matched. The intended use of start conditions is to allow the lexer to switch modes when an incoming symbol indicates some type of modifier to the character stream. An example in the C language is that the string `int` should be tokenized as a single token, but if it is within a comment, it should not be tokenized at all. A flex specification to reflect this might contain a start condition for the "normal" int token, and another start condition for rules to discard any sequence of characters within a comment. We allow the start condition to be set from within the parser by providing a shared switch that the parser sets and the lexer checks. Recall the example of an HTTP chunk. Our parser can prompt the lexer to match a string of hexadecimal digits terminated by a carriage return and newline, then initialize a counter with the value of that number. It will then switch the start condition to a accept any byte, and create new nodes in the manner suggested above until a node is created with a length of 0. Then the lexer can be switched to a text-oriented start condition to match the footer of the chunk.

Our experience has shown that this dynamic tree-building ability is sufficient for covering context-sensitive details of protocols that would be impossible or nonintuitive for a context-free grammar.

5 Evaluation

We have evaluated whether HAGs are an appropriate way to express protocol parsing tasks such as protocol normalization. We were particularly interested in answering the following questions about this technique:

1. Is it feasible to represent a protocol using a HAG? Limiting the expressive capabilities of protocol abstraction from the level of source code to the level of a grammar eases the burden of writing a protocol specification. However, it is crucial that the grammar be able to express all of the features of popular protocols. In Section 5.1, we discuss our implementations of parsers generated using HAGs for three widely known protocols: FTP, SMTP, and HTTP. Our implementation proves that HAGs are expressive enough to represent these protocols in sufficient detail to perform common parsing tasks.
2. How efficiently can a HAG-based normalizer execute? In order to determine if our technique is practical, we undertook a case study to compare

the normalizers built into the popular IPS Snort. We tested the running time of Snort's HTTPInspect normalizing preprocessor, SMTP dynamic normalizing preprocessor, and FTPTELNET normalizing dynamic preprocessor against our own normalizers automatically generated from HAG grammars. We foundthat in the worst case, our normalizers incur only 15.5% overhead versus the Snort normalizers, even with the added burden that our normalizers kept track of additional fields that Snort did not. When we restricted our normalizers to only those fields for which Snort checked for a signature, our overhead was reduced to approximately 7%. We explain the details of our performance evaluation in Section 5.2.

3. Is a HAG based normalizer robust against syntax transformations? An IPS is often the last line of defense in a security infrastructure. For this reason, it is critical that the IPS normalizer modules be robust against syntax transformations. We used our normalizers in Snort and tested the number and type of alerts that Snort generated against the alerts that an unmodified version of Snort raised. We found that our version caught all of the malicious requests that were caught by Snort, several of which have eluded previous versions of the Snort normalizers [16,17]. Details of this comparison are in Section 5.2.

5.1 Feasibility Study

To evaluate our approach, we built normalizers for three common protocols - HTTP, SMTP, and FTP - for which the Snort IPS also has normalizers.

The first protocol for which we have implemented an attribute grammar parser is HTTP. Our attribute grammar implements parsing for requests with fixed-length bodies, chunked bodies over a persistent connection, and variable length bodies. We have not integrated normalization for multipart message bodies in our grammar. We handle messages of this type by simply ignoring the message body. This treatment fits with the intention that HTTP treats the body of a multipart messages as a payload, rather than information with a special semantic meaning to the protocol itself. However, we believe that our methodology could be extended for deep inspection of these payloads with subgrammars being used to parse whatever MIME type the message specifies. Although HTTP is not parseable using context-free grammars because of the chunked body content type, and fixed length HTTP bodies, those features can be captured with an attribute grammar so that normalization can be performed. Our Higher-Order Attribute Grammar for HTTP was developed in one week by a single graduate student, concurrently with our overall approach for parsing context-sensitive grammars. We are confident that a developer well-versed in HTTP would be able to re-create a parser similar to ours in a matter of days.

Another protocol that we have modeled is the Simple Mail Transfer Protocol (SMTP). SMTP can include runs of white space that are ignored by SMTP servers when processing commands. Our grammar can recognize these runs in all SMTP commands. We do not recognize limits in the length of the command, header, or response line, but integrating a simple counter into the grammar would

not be a difficult extension. SMTP is an example of a relatively simple protocol that requires little normalization. Creating an attribute grammar required just one day for one graduate student.

The File Transfer Protocol (FTP) is used to transfer data over a network. Our Attribute Grammar FTP parser can recognize all valid FTP commands, and is sensitive to injected telnet escape sequences. As with SMTP, the extra expressive powers of an Attribute Grammars are not strictly necessary: a context-free grammar would be sufficient. However, the convenience of our method shows that an attribute grammar can readily be constructed in a simple, straightforward way. As with the SMTP parser, the FTP grammar was completed in a single day by one graduate student.

5.2 Snort Case Study

Snort is a popular, open-source IPS that performs analysis and normalization for several protocols. We chose to implement our normalizers as modules in Snort because it is widely used, open source, and is designed for pluggable, modular preprocessors.

Performance Evaluation: We tested three different versions of Snort for our experiments. The first is an uninstrumented version of Snort that uses the existing `HTTPInspect` preprocessor for HTTP normalization, the `smtp` dynamic preprocessor for SMTP normalization, and the `ftptelnet` dynamic preprocessor for ftp normalization. We ran Snort in it's default configuration. The second version of Snort, listed as *AG-Maximal*, implements the full set of normalizations described above. The final version of Snort, listed as *AG-Minimal*, uses our attribute grammar method to normalize only those protocol fields that are relevant to a signature in Snort's database. For example, the `HTTPInspect` preprocessor does not do any normalization on HTTP fixed-length bodies, so the *AG-Minimal* grammar includes no normalization rules for those message bodies.

We performed our modifications on Snort version 2.8.0.2. Our tests used a trace of 795,488 packets (approximately 2 gigabytes) that we collected from a campus web server. Our experiment uses average numbers from Snort's performance profiling module, which measures at fine granularity the total time for packet processing.

The results of performance profiling on these versions of Snort are summarized in Table 6(a). Not shown are differences in compilation time, which are negligible for our attribute grammar normalizers versus the uninstrumented version of Snort. In the worst case for *AG-Maximal*, our normalizer incurs 15.5% overhead versus the Snort normalizers. The *AG-Minimal* normalizers, which are more consistent with `Snort's` behavior, reduce the overhead to 7.16%.

Robustness Evaluation: To test the robustness of our normalizers, we crafted obfuscated packets by hand. For the HTTP normalizer, we used the following obfuscations:

– Uppercase to lowercase transformation

Name	Total Packet Processing Time	Overhead (%)
Snort 2.8.0.2	50.53	-
AG-Minimal	54.15	7.16
AG-Maximal	58.36	15.50

(a) Performance summary

Fig. 6. Performance Evaluation

- Percent encoding
- Double percent encoding
- ASCII to Unicode encoding

Our packets included both malicious signatures known to the Snort database, and benign traffic that was obfuscated in a similar way to the malicious traffic. We found that our system correctly normalized all encoded traffic, and did not make changes to any traffic that was already decoded. We observed similar results for SMTP traffic and FTP traffic.

6 Conclusion

We introduced the notion of using a higher order attribute grammar (HAG) to parse many modern protocols for which using context-free grammars are impractical or impossible. We believe that the small decrease in performance that these tools display when compared with ad-hoc approaches is more than outweighed by the gains in ease of use.

We plan to investigate the use of our tool for binary protocols and study the use of systems that directly support attribute grammar parsing, rather than relying on existing tools that are not meant for online parsing speed. We believe that a tool specifically geared towards network protocol parsing would provide even more competitive performance numbers than our existing approach, and may even yield a performance boost.

References

1. Anderson, D.P., Landweber, L.H.: A grammar-based methodology for protocol specification and implementation. In: Proceedings of SIGCOMM (1985)
2. Borisov, N., Brumley, D.J., Wang, H.J.: A generic application-level protocol analyzer and its language. In: 14th Annual Network & Distributed System Security Symposium (2007)
3. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards automatic generation of vulnerability-based signatures. In: SP 2006: Proceedings of the IEEE Symposium on Security and Privacy, pp. 2–16. IEEE Computer Society, Los Alamitos (2006)
4. CERT. Superfluous Decoding Vulnerability in IIS. CA-2001-12 (2001)
5. Chapman, N.P.: Defining, analysing and implementing communication protocols using attribute grammars. In: Formal Aspects of Computing 1990, pp. 359–392 (1990)

6. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext transfer protocol – HTTP/1.1, RFC2616 (1999)
7. Handley, M., Paxson, V., Kreibich, C.: Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In: Proceedings of the 10th conference on USENIX Security Symposium (2001)
8. Knuth, D.E.: The genesis of attribute grammars. In: Proceedings of the International Conference on Attribute grammars and their Applications (1990)
9. Levine, J.R., Mason, T., Brown, D.: lex & yacc, 2nd edn. O'Reilly & Associates, Inc., Sebastopol (1992)
10. Nielsen, H.F., Gettys, J., Baird-Smith, A., Prud'hommeaux, E., Lie, H.W., Lilley, C.: Network performance effects of HTTP/1.1, CSS1, and PNG. SIGCOMM Comput. Commun. Rev. 27(4), 155–166 (1997)
11. NVD. CVE-2002-0613. National Vulnerability Database (June 2002), `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2002-0613`
12. Paakki, J.: Attribute grammar paradigms–a high-level methodology in language implementation. ACM Computing Surveys 27(2) (June 1995)
13. Pang, R., Paxson, V., Sommer, R., Peterson, L.: binpac: A yacc for writing application protocol parsers. In: Proceedings of the Internet Measurement Conference, IMC (2006)
14. Parr, T.: The Complete Antlr Reference Guide. Pragmatic Bookshelf (2007)
15. Ptacek, T.H., Newsham, T.N.: Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc. (January 1998), `http://www.aciri.org/vern/Ptacek-Newsham-Evasion-98.ps`
16. Rubin, S., Jha, S., Miller, B.P.: Automatic generation and analysis of NIDS attacks. In: Annual Computer Security Applications Conference (ACSAC) (December 2004)
17. Vigna, G., Robertson, W., Balzarotti, D.: Testing network-based intrusion detection signatures using mutant exploits. In: Proceedings of the ACM Conference on Computer and Communication Security (ACM CCS) (October 2004)
18. Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher order attribute grammars. In: PLDI 1989: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation, pp. 131–145. ACM, New York (1989)