2008

Deriving Input Syntactic Structure From Execution and Its Applications

Zhiqiang Lin

Xiangyu Zhang
Purdue University, xyzhang@cs.purdue.edu

Report Number:
08-006

http://docs.lib.purdue.edu/cstech/1696
Deriving Input Syntactic Structure From Execution and Its Applications

Zhiqiang Lin
Xiangyu Zhang

CSD TR #08-006
March 2008
Deriving Input Syntactic Structure From Execution

Zhiqiang Lin Xiangyu Zhang
Department of Computer Science
Purdue University, West Lafayette, Indiana 47907

ABSTRACT
Program input syntactic structure is essential for a wide range of applications such as test case generation, software debugging and network security. However, such important information is often not available (e.g., most malware programs make use of secret protocols to communicate) or not directly usable by machines (e.g., many programs specify their inputs in plain text or other random formats). Furthermore, many programs claim they accept inputs with a published format, but their implementations actually support a subset or a variant. Based on the observations that input structure is manifested by the way input symbols are used during execution and most programs take input with top-down or bottom-up grammars, we devise two dynamic analyses, one for each grammar category. Our evaluation on a set of real-world programs shows that our technique is able to precisely reverse engineer input syntactic structure from execution.

Categories and Subject Descriptors
D.3.4 [Programming Languages]: Processors—Parsing; D.2.5 [Software Engineering]: Testing and Debugging—Tracing; D.2.7 [Distribution, Maintenance, and Enhancement]: [Restructuring, reverse engineering, and reengineering]

General Terms
Algorithms, Verification

Keywords
reverse engineering, input structure

1. INTRODUCTION
Most software applications take structural inputs. Document processing software such as XML, PDF and WORD processors require input files in specific formats. Compilers consume inputs written in programming languages. Network applications communicate through sessions in which messages have to follow certain formats. Data processing programs such as audio/video codecs accept structural bit streams. As an integral component, the syntactic structure of program inputs serves in a wide range of applications.

Software Engineering – In software testing, automatically generating tests from input grammar is a technique originated in 1970’s [14, 25], and then continuously studied ever since, e.g. in [8, 20, 27]. Most recently, it has been found that considering input grammars can significantly scale up symbolic execution based test generation techniques [18, 13]. Delta Debugging [32] is a highly effective automatic debugging technique that reduces a large failure inducing input to its minimal subset that still exposes the fault. The reduction is done through a binary-search like procedure in which the program is executed iteratively. Most recently, Hierarchical Delta Debugging (HDD) [21] shows that the search procedure can be greatly accelerated if the input hierarchical structure is provided. Execution Fast Forwarding [35] treats event replay log as the program input that drives program re-execution, and reduces a long execution by reducing the replay log for the purpose of debugging. Considering event hierarchy would avoid producing ill-structured reduced logs.

Computer Security – Input structure, reflected as protocol formats in network security, is critical in a number of scenarios. Protocol structure can be used in penetration testing that evaluate the security of a system by simulating attacks. For instance, packet vaccine [29] is a technique that randomizes the address fields in a network packet in order to simulate control flow hijacking. The information of packet format can actually better guide the vaccine generation as illustrated in ShieldGen [12]. Signature generation techniques construct signatures for exploits, and packet format information such as payload lengths, keywords, field types, state transitions is essential to signature composition [28]. Intrusion detection systems such as snort [3] matches network traffic to pre-defined protocols. Scanning unauthorized services provided at non-standard ports requires understanding the communication protocol.

Despite the importance of input structure, acquiring such information often demands a lot of efforts. First, input structure is often specified in a machine unfriendly way (e.g. textual documents). Hence, in applications such as HDD [21] and the recent work of combining input grammar with symbolic execution based test generation [18, 13], the onus is on users to provide input grammars and parsers, even for inputs such as C programs and XML files. Second, various software applications that claim to accept inputs in a grammar may indeed implement slight variants of the grammar. For example, it is quite common that an implementation of a network protocol does not support part of the specification. Third, input structure is not even specified in many cases. A zombie computer usually communicates with the remote attacker through secret protocols. Analyzing and understanding these protocols has imposed great challenges. Even benign software such as Yahoo Messenger makes use of a closed protocol. It took the open source community
years to understand the protocol and provide a usable open source client [1]. Upon the happening of a failure, modern systems often provide channels to turn in failure reports. As the failure inducing input is the most critical part of a failure report, HDD may be used on the user side to reduce the failure inducing input. However, regular users often have no access to the source code, let alone the input specification. Similarly, penetration testing is often carried out by administrators or regular users after a software is deployed and thus lack of input specification. Therefore, techniques that automatically reverse engineer input structure are highly desirable to circumvent these difficulties.

Recently, research has been conducted on automatic input structure derivation in the context of network security, particularly for protocol reverse engineering [9, 15, 30, 11]. The basic observation is that the protocol implementation that handles incoming protocol messages reveals a wealth of information about protocol format. Therefore, protocol structure can be naturally discovered by analyzing program binary based on dynamic data flow analysis. In particular, Polyglot [9], and [30] exploits the semantics of message payload processing instructions such as loops and comparison to identify keywords, delimiters and thus the fields in messages. Along this way, Tupni [11] makes such analysis applicable to infer record sequences, record types, and input constraints, and can even generalize the format specification over multiple messages, facilitated by instruction semantics. AutoFormat [15] leverages execution contexts (i.e., call stacks and instruction addresses), in which messages are processed, to identify input fields and hierarchical structure.

These systems are able to derive input structure to some extent, but fail to deliver an effective general solution as they catch only part of the problem’s essence. First of all, these techniques assume programs take inputs with top-down grammars, which often implies input structure being reflected in program structure. However, we observe that many programs require inputs with bottom-up grammars, which are parsed by automata. In such a scenario, program structure does not reflect input structure. Our study on SPEC95INT programs shows that 25% of the applications rely on bottom-up grammars. Some network applications such as Webcp require protocol messages with bottom-up grammars as well. Existing reverse engineering techniques fail to derive input structure for these applications. Second, even for inputs with top-down grammars, these techniques do not catch the essence of the problem and provide only partial solutions. For example, Polyglot[9] and [30] rely on delimiter identification to identify network message fields: and delimiter identification is based on a heuristic that a delimiter is a byte that appears in a loop predicate and is compared against multiple bytes in a message. Such heuristic may not work well in many situations such as message fields are not divided by delimiters or delimiters are implicit (e.g., in the case that fields have fixed lengths) so that they do not appear as constants in the code. Similarly, our previous work AutoFormat [15] relies on execution context, and thus if application implementation does not follow the modular programming practice so that multiple message fields are parsed in the same execution context, the identified structure would be too coarse-grained.

In this paper, we propose two dynamic analyses that reverse engineer syntactic structure for inputs with top-down grammars and bottom-up grammars respectively. Given the program binary (without source code) and a program input, our system executes the program with the provided input, and at the end, emits the syntax tree of the input without any user interference. Currently, our technique only derives syntax trees for individual inputs. We leave input grammar derivation to our future work. While handling bottom-up grammars is a feature that has not been supported by existing techniques, our solution to top-down grammars supercedes existing techniques as well because it better reflects the problem’s essence and thus is more systematic. The unique observation we obtain (regarding inputs with top-down grammars) is that dynamic control dependence is the most prominent evidence of input structure, reflecting input syntactic structure at the finest level. Delimiters in [9, 30] and execution contexts in [15] only catch part of the exercised dynamic control dependence, and thus cannot construct the precise syntactic input structure. Programs that accept inputs with bottom-up grammars manifest completely different runtime characteristics, rendering the control dependence based approach not applicable. We observe bottom-up parsing is mostly associated with a parsing stack, and operations on the stack serve as a strong indicator of the input structure. We devise an analysis to extract the stack related sub-execution and build the input syntax tree from the sub-execution. Our evaluation on a set of real-world applications show that the proposed techniques produce input syntax trees with high quality. The contributions of our paper are highlighted as follows.

- We devise a dynamic analysis to reverse engineer the structure of input with top-down grammars. The analysis heavily depends on dynamic program control dependence.
- We have the insight that programs that consume input with bottom-up grammar behave differently at runtime and thus make existing approaches and the proposed dynamic control dependence based approach ineffective.
- We propose a dynamic analysis to handle inputs with bottom-up grammars. It relies on identifying and monitoring the parsing stack.
- We evaluate our technique on a set of benchmarks that employ top-down and bottom-up parsing. Our results show that the proposed analyses are highly effective in producing precise input syntax trees. Particularly, the derived trees for the set of benchmark applications with bottom-up input grammars are identical to the real trees.

2. HANDLING INPUTS WITH TOP-DOWN GRAMMARS

Most programs take inputs with top-down grammars or bottom-up grammars. In this section, we first discuss how to handle inputs with top-down grammars.

A grammar that can be parsed by a top-down parser is called a top-down grammar. A top-down parser parses an input string from the root of the syntax tree (ST) to the leaves. The input of a wide range of applications can be described by top-down grammars. Examples include html/xml pages, http/sip packages, and binary inputs such as audio/video files. Due to its implementation simplicity, many hand written parsers are a top-down parser.

**Example.** Fig. 1 shows a simple language with a top-down grammar, which accepts strings that have structure similar to html pages.

```
Doc → Head Body
Head → H Text /H
Body → B Tag /B
Tag → T Text /T Tag | ε
Text → [a-zA-Z]+
```

**Figure 1:** A Simple Language with Top-Down Grammar
A document consists of two parts, a header and a body. A header is delimited by “H” and “/H”. A body consists of a series of tags that are confined by symbols “T” and “/T”. Fig. 2 presents a string that belongs to the grammar and its corresponding derivation. The derivation is also called the syntax tree (ST). In order to parse the string into its ST, the parser first takes the top rule, i.e., the Doc rule. As the Doc rule is composed of the Head and Body rules, it next takes the Head rule to parse the string. The Head rule accepts the first symbol “H” and continues with the Text rule, so on. The whole procedure is like walking from the top of the derivation tree to the bottom.

A unique characteristic of top-down grammars is that a top-down parser can precisely predict the next rule to parse the remaining input at any particular time based on the current parsing rule and the incoming element. For instance in Fig. 1, if a character “T” is seen and the parser is not in the middle of parsing rule Text. Rule Tag is taken to parse the character. Fig. 3 shows a top-down parser of our sample grammar, which is a highly simplified version of the html parser in tidy. In the implementation, each function corresponds to one nonterminal in the grammar. The parser starts parsing by calling function PDoc on the input, which in turn calls PHead and PBody. PBody verifies if the next character is ‘B’. If not, an error is reported because it violates the parser’s expectation at current state. Otherwise, it calls PTag and then verifies the remaining ending delimiter symbol. PTag parses all Tag expressions through a loop.

Recall the objective of our technique is to derive the structure of an input, given the application binary. The parser is an integral part of the binary, statically indistinguishable from other functional components. The key observation is that dynamic control dependencies disclose the syntactic structure of inputs with a top-down grammar. Intuitively, a top-down parser decides if a grammar rule is taken by comparing the incoming input symbol with the leading symbol of the rule. The parsing of all the constituent symbols of the rule, either terminals or nonterminals, is guarded by the comparison. In other words, the dynamic control dependence caused by the comparison discloses the hierarchical relation between child symbols and their parent. Therefore, our technique traces the dynamic control dependencies that are exercised during execution and constructs the dynamic control dependence graph (DCDG). By observing how input elements are used in the DCDG, the syntactic structure can be derived.

To describe our analyses, we first formally define the problem.

**PROBLEM STATEMENT 1.** Given a pair <P, I>, in which P is a program binary and I is an m-tuple input for P with the format of T^2...T^n, construct the syntactic structure of I with the representation of an ST.

The idea is to derive input structure through dynamic control dependence. Informally, an executed statement x_i, denoting the i-th instance of statement x, is dynamically control dependent on another executed statement y_j, represented by y_j \rightarrow_{dcd} x_i, if and only if y_j is a predicate or a function call site and y_j directly decides the execution of x_i. For example in the execution shown on the left hand side of Fig. 4, produced by applying the input in Fig. 2 to the implementation in Fig. 3, 11 \rightarrow_{dcd} 12, since the branch outcome of 11 directly decides the execution of 12. Similarly, 3_1 \rightarrow_{dcd} 10_1. More formal definition is elided for brevity, interested readers are referred to [31]. If each executed statement is considered as a node and each exercised dynamic control dependence is considered as an edge, a DCDG is constructed. The right hand side of Fig. 4 shows a DCDG subgraph for the trace on the left.

To derive input structure from the execution structure revealed by the DCDG, we label the nodes that use an input value. For example in Fig. 4, node 11 uses the first ‘B’ in the input string (stored in variable c) and thus it is labeled with ‘B'. The labels of other nodes, if any, are also displayed in the figure. Through these labels, the hierarchy of the DCDG is translated to the hierarchy of input elements.

A number of issues need to be addressed to make the runtime analysis work. First, constructing the DCDG for the whole execution entails tremendous space overhead [34], and is not necessary as only the labeled subgraph is needed, which is often a tiny part of the whole DCDG. Second, we need to handle propagation of input values to assign correct labels to DCDG nodes as an input value can be propagated through variable assignments. Third, an online algorithm is highly desirable as post-mortem analyses require collecting and storing traces.

We devise a cost-effective online algorithm to address these issues. It was observed dynamic control dependence has the LIFO characteristic and thus can be maintained by a stack called control dependence stack (CDS) [31, 19]. In particular, an entry is pushed onto CDS when a predicate p_i is executed, and the same entry is popped if the immediate post-dominator of p_i is executed, indicating the end of the execution region that is directly or indirectly dynamic control dependent on p_i. For instance in the execution

1The outline indicates it is a sequence.
At line 10, a leaf node is introduced denoting the node in the ST for each CDS entry if the node has not been created.

The algorithm tests if p refers to the control dependence stack (CDS) state at the moment of its execution. For instance, the execution of 25, after 11, pushes another entry to the stack. Upon the execution of 26, its dynamic control dependence transitive closure is disclosed by the current CDS, i.e., ...

**Definition 1.** Given a statement execution \( s_i \), \( CDS(s_i) = < p_1, p_2, ..., p^n > \) refers to the control dependence stack (CDS) state when \( s_i \) is executed, representing the dynamic control dependence transitive closure of \( s_i \).

Algorithm 1 presents the instrumentation that produces the input ST on the fly. The algorithm first updates the CDS at line 3. If \( s_m \) is a predicate instance, \( updateCDS \) performs a push; if \( s_m \) is a post-dominator instance, it performs \( pop(s) \). More details about \( updateCDS \) such as the proof of the LIFO property and handling irregular control flow can be found in our prior work [31]. At line 4, the algorithm tests if \( s_m \) uses a variable that has been labeled with an input value. If yes, it retrieves the current CDS, and creates a node in the ST for each CDS entry if the node has not been created before. Two nodes that are consecutive in the CDS are connected with an edge. At line 10, a leaf node is introduced denoting the input label. Lines 11-12 handle label propagation, it propagates the label from the source variable \( v \) to the destination \( d \). Note that we only propagate labels for assignment type of instructions. In other words, we do not propagate labels for binary operations on two input related variables. Line 14 turns off the instrumentation after all inputs have entered the execution, implying the parsing phase is over. In our implementation, we initiate input labeling by intercepting input system calls like \( SYS\_READ \), through which we also identify the last input symbol \( i^* \).

**Algorithm 1 Online Analysis**

```plaintext
constructTree updates the ST upon each instruction execution. 
updateCDS maintains the control dependence stack. 
addNode adds a node to the resulting ST. 
addEdge adds an edge to the resulting ST. 
instrumentationOff turns off the instrumentation. 
```

1. \( constructTree(s_m) \) 
2. { 
3. \( updateCDS(s_m) \); 
4. \( if \ (variable \ v \ is \ used \ in \ s_m \ and \ v \ is \ labeled \ with \ input \ i^*) \) 
5. \( addNode(i^*) \); 
6. \( foreach \ p^i \ in \ CDS(s_m) \) in the bottom-up order, 
7. \( addNode(p^i) \); 
8. \( addEdge(p^i-1 \rightarrow p^i) \); 
9. } 
10. \( addEdge(CDS(s_m).top() \rightarrow i^*) \) 
11. \( if \ (s_m \ has \ the \ form \ of \ d = f(v)) \) 
12. \( label \ variable \ d \ with \ i^*; \) 
13. \( if \ (the \ last \ input \ i^n \ has \ been \ used) \ instrumentationOff(); \) 
15. 
16. \( updateCDS(s_m) \) 
17. { 
18. \( while \ (s \ is \ the \ immediate \ post \ dominator) \) 
19. \( of \ CDS(s_m).top() \) 
20. \( CDS(s_m).pop(); \) 
21. \( if \ (s \ is \ a \ predicate \ or \ a \ method \ call) \) 
22. \( CDS(s_m).push(s_m); \) 
23. }
```

**Example.** The left hand side of Fig. 5 shows part of the resulting ST. Statement instance 11, uses a variable labeled with \( B'1 \) (note that although \( c \) is defined with input \( B'1 \), it is not used until 11, and thus \( c \) does not lead to a node creation). \( CDS(11) = < START, 3, 111 > \), and thus the online algorithm generates three corresponding nodes and a leaf node \( B'1 \). Similarly, 14, has the CDS of \( < START, 3, 111, l_4 > \) and it uses a variable labeled with \( T'4 \), resulting in a node whose parent is a sibling of node \( B'1 \).

**2.2 Offline Transformations**

The ST constructed by the online analysis does not precisely mirror the real input structure. The comparison between the left hand side of Fig. 5 and the real derivation in Fig. 2 suggests that further transformations are needed.

**Duplicated Leaf Nodes Elimination.** On the left hand side of Fig. 5, we can see the same leaf node \( T'4 \) appears in three places. Two are the children of nodes 14 and 12, and the third one is a descendant of 25. They correspond to the descendants of 14, 15, and 14, respectively. Such situation arises if the same input value is used in multiple places to control the parser execution. These input values having multiple use points are often delimiters. As a ST has one leaf node for one use of an input symbol, we need to identify the one that reveals the true structure and remove the rest from

\[ \text{Since } s_m \text{ could be the post-dominator for multiple consecutive predicates, we use a loop here.} \]
the resulting tree. Two observations can be exploited to achieve this goal. The first one is that most parsers parse input symbols in order, i.e., one symbol is not parsed until its predecessor is parsed. Second, if a symbol is used in multiple points during execution, like delimiters, the last use point before its successor being parsed is the parsing point of the symbol. The observation behind this is that a delimiter is permanently removed from the input buffer, and thus parsed, right before the next symbol is processed. Note that a symbol may be used beyond its successor’s parsing point, e.g., a printf that prints all input symbols at the end of the program execution. Therefore, we cannot simply consider the last use point as the parsing point.

**Definition 2.** A statement instance $s_m$ is the anchor point of an input value $i^x$, denoted by $AP(i^x) = s_m$, if and only if $s_m$ uses a variable labeled with $i^x$ and there is NO other instance $t_n$ that uses a variable with the same label during the parsing phase.

In other words, if an input element is used at only one place during the parsing phase, the use point is its anchor point. For instance, $AP(‘B’_2) = 16$. Based on anchor points, we define the parsing points of an input symbol.

**Definition 3.** The parsing point of an input element $i^x$ is defined as:

$$PP(i^x) = \begin{cases} 
AP(i^x) & \text{if } AP(i^x)! = \bot; \\
\bot & \text{if } s_m \text{ uses a variable labeled with } i^x \cap \forall y > x. \text{AP}(i^y)! = \bot, \\
s_m \text{ occurs before } AP(i^x) \cap s_m \text{ is the latest point that satisfies the previous two conditions.} \\
\text{otherwise.} 
\end{cases}$$

The symbol $\bot$ stands for undefined. If an input symbol has multiple use points, the above definition identifies the parsing point of the symbol as the one that happens before the next anchor point and has the largest timestamp. Other use points are removed from the tree. In our example, input $‘T’_4$ has three use points, 141, 253, and 351, resulting in the three labels in the left graph of Fig. 5. The next anchor point of $‘T’_4$ is $AP(‘B’_2) = 16$. Since all three uses happen before 161 and 141 has the largest timestamp, 141 is identified as $PP(‘T’_4)$ and the other two points are pruned from the tree.

**Redundant Intermediate Nodes Elimination.** The approximation produced by the online analysis often contains redundant intermediate nodes, which do not provide useful information. An intermediate node is redundant if and only if it has only one child. The redundancy can be removed by replacing the intermediate node with its child. This process continues until no further reduction can be conducted. For instance, nodes 31, 141, 161, and 32 are on the left hand side of Fig. 5 are redundant. Node 12 is also redundant after its leaf labeled with $‘T’_4$ is pruned according to the aforementioned transformation, and hence it is replaced with node 251.

After applying the transformations, the final ST produced for our sample is presented on the right hand side of Fig. 5, which precisely reflects the desired hierarchical structure of the input.

3. **HANDLING INPUTS WITH BOTTOM-UP GRAMMARS**

We observe that many applications consume inputs with bottom-up grammars, e.g., most programming languages have a bottom-up grammar. Due to the different runtime characteristics of top-down and bottom-up parsers, reverse engineering syntactic structure for inputs with bottom-up grammars requires a different solution.

A grammar that can be parsed by a bottom-up parser is called a bottom-up grammar. A bottom-up parser parses a string by constructing the derivation in a bottom-up manner, namely, it starts at the leaf level and works up towards the root by reducing a set of low level nodes to a higher level intermediate node at each step [6]. A large body of applications make use of bottom-up grammars due to their expressiveness. The class of languages that can be expressed by bottom-up grammars is a proper superset of those expressed by top-down grammars. The intuitive explanation is that top-down grammars require parsers to predict a grammar rule by looking at the first (a few) symbol(s), whereas bottom-up parsers delay making this decision till all the symbols of a grammar rule are in sight, which is far less stringent. Although bottom-up grammars feature higher complexity in implementation, there exist tools such as yacc or bison that can automatically generate parsers for bottom-up grammars.

Figure 5: Transformation.

Figure 6: A General Bottom-Up Parsing Algorithm

Deriving input structure for bottom-up parsers is intriguing due to the way they are implemented. Figure 6 presents a general parsing algorithm used by most bottom-up parsers [6]. The algorithm is facilitated by a stack and a DFA, encoded by the action and goto tables. Given the current state of the DFA, which is stored in the stack, and an incoming input symbol, i.e., the leftmost symbol of the input string, there are two possible actions. If the top symbols on the stack do not constitute the righthand side of a grammar rule, indicated by the action table entry indexed by the current state and the incoming input symbol having the value of shift,
as shown at line 5 of the algorithm, the input symbol c is removed from the input string and pushed to the stack, being labeled with the updated state. If the top symbols are indeed the righthand side of a grammar rule $A \rightarrow \beta$, encoded by the action table entry having the value of reduce as shown at line 9, the top $n = |\beta|$ elements are popped from the stack and the lefthand side symbol $A$ is resulted at lines 10–11. At line 12, the current state is updated based on the state on the top of the stack and $A$ according to the goto table. The symbol $A$ labeled with the new state is pushed to the stack at line 13. The process terminates when the start symbol meets with an input symbol, encoded by an accept action. The DFA encoded by tables action and goto can be constructed in various ways, giving rise to different subclasses of bottom-up grammars. Our analysis described later is independent of the way the DFA is constructed and thus is general for bottom-up grammars.

$$\text{(1) } \text{Body} \rightarrow \ B \ \text{Tag} /B \ $$
$$\text{(2) } \text{Tag} \rightarrow \ \text{Tag} \ T \ \text{Text} /T \ $$
$$\text{(3) } \text{Tag} \rightarrow \ \text{T} \ \text{Text} /T \ $$
$$\text{(4) } \text{Text} \rightarrow \ \text{Text} \ a \ $$
$$\text{(5) } \text{Text} \rightarrow \ a \ $$

Figure 7: A Sample Bottom-Up Rule

Table 1: Parsing string “BTa/TTaa/T/B”.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Stack operation trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) B</td>
<td>BTA/TTa/T/B</td>
<td>push(B,1)</td>
</tr>
<tr>
<td>(2) S B</td>
<td>Ta/TTa/T/B</td>
<td>push(T,2)</td>
</tr>
<tr>
<td>(3) S B T</td>
<td>a/TTa/T/B</td>
<td>push(a,3)</td>
</tr>
<tr>
<td>(4) S B T</td>
<td>/T/B</td>
<td>pop(1); push(Text,3)</td>
</tr>
<tr>
<td>(5) S B T</td>
<td>Text</td>
<td>rule: Text → a</td>
</tr>
<tr>
<td>(6) S B T</td>
<td>/T/B</td>
<td>pop(3); push(Text,2)</td>
</tr>
<tr>
<td>(7) S B T</td>
<td>Tag</td>
<td>rule: Tag → T Text /T</td>
</tr>
<tr>
<td>(8) S B T</td>
<td>Tag</td>
<td>push(T,3)</td>
</tr>
<tr>
<td>(9) S B T</td>
<td>Tag</td>
<td>a/TT/B</td>
</tr>
<tr>
<td>(10) S B B</td>
<td>Tag</td>
<td>push(Text,4)</td>
</tr>
<tr>
<td>(11) S B B</td>
<td>Tag</td>
<td>rule: Tag → Text a</td>
</tr>
<tr>
<td>(12) S B T</td>
<td>Tag</td>
<td>push(T,4,5)</td>
</tr>
<tr>
<td>(13) S B B</td>
<td>Tag</td>
<td>/B</td>
</tr>
<tr>
<td>(14) S B B</td>
<td>Tag</td>
<td>push(Text,3)</td>
</tr>
<tr>
<td>(15) S B B</td>
<td>Tag</td>
<td>$</td>
</tr>
<tr>
<td>(16) S B B</td>
<td>Tag</td>
<td>exit the while loop</td>
</tr>
</tbody>
</table>

The superscripts of stack entries are the states associated with symbols. push(s, p) means pushing symbol s to the pth position of the stack, and the state is t. pop(s) means popping the top s stack entries.

Example. Fig. 7 shows a sample bottom-up grammar. It is not a top-down grammar because of the left-recursions in rules (2) and (4), which make a top-down parser fail to predict which rule to follow upon seeing a symbol a or T. Table 1 illustrates how an input in the sample grammar is parsed according to the algorithm in Fig. 6. The grammar is translated to action and goto tables in Table 2. Note tools are available to automate the translation, and interested readers are referred to [6]. At step (1), the next input symbol $B$ is pushed to the stack and the current state is updated to 1, which is decided by action[0, B] = <shift, 1> in Table 2. At step (4), the top element on the stack is popped and reduced to Text, which is decided by action[7, /T] = <reduce, Text → a>, which is pushed to the stack with the new state 6 (goto[3, Text] = 6 in Table 2). The process terminates at step (16) where the stack contains the start symbol Body and the input string becomes empty.

The analysis described in previous section does not work well for bottom-up parsers. The execution structure, illustrated by the exercised control dependences, no longer approximates the input syntactic structure. According to the algorithm in Fig. 6, input symbols are consumed in different iterations of the while loop at runtime. As one iteration is dynamically control dependent on its preceding iteration, a node labeled with an input symbol is dynamically control dependent on the node labeled with its preceding symbol. The resulting AST approximation has a close-to-linear structure.

Fortunately, the execution of a bottom-up parser exposes the input structure nonetheless through a different channel. Consider the stack column in Table 1, the reductions at steps (4), (6), (9), etc., highlighted by boxes, introduce hierarchical relations between symbols. For instance, the reduction at step (6) indicates that the resulting Tag symbol is derived from the T, Text and /T symbols on the stack, which constitute the child nodes of Tag in the ST. The key observation is that reductions reveal the input structure and a reduction can be identified from the behavior of the parsing stack.

**Definition 4.** Given a bottom-up parser $P$ and an input $I$, the stack operation trace of the execution $P(I)$ is defined as the sequence of push and pop operations of the parsing stack.

- A push operation is represented as push(s, p), meaning pushing the symbol s with the new state t to the top position p.
- A pop operation is represented as pop(a), meaning popping the top n entries.

For instance, the stack operation trace column of Table 1 lists the sequence of stack operations. We can observe from Table 1 that reductions are always associated with pop operations. Unfortunately, pop operations are hard to identify from execution trace, assuming no knowledge of the source code, because they are often translated to pointer arithmetic operations on the stack variable, which are indistinguishable from numerous other pointer arithmetic operations during the execution. Furthermore, as pointers are often stored in registers, tracing operations on registers is very expensive. In comparison, push operations are much more visible as they are always associated with memory writes on a specific region (the stack) with certain patterns. Therefore, we decide the

Table 2: Parsing Table For The Grammar in Fig. 7.

<table>
<thead>
<tr>
<th>state</th>
<th>H</th>
<th>/B</th>
<th>T</th>
<th>/T</th>
<th>a</th>
<th>s</th>
<th>Body</th>
<th>Tag</th>
<th>Text</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g10</td>
</tr>
<tr>
<td>1</td>
<td>v4</td>
<td>v3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g2</td>
</tr>
<tr>
<td>2</td>
<td>v7</td>
<td>v6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g6</td>
</tr>
<tr>
<td>3</td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g8</td>
</tr>
<tr>
<td>4</td>
<td>v9</td>
<td>v10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>acc</td>
</tr>
<tr>
<td>5</td>
<td>r3</td>
<td>r2</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r3</td>
<td>r2</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3</td>
<td>r2</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r3</td>
<td>r2</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r3</td>
<td>r2</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>r3</td>
<td>r2</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>r3</td>
<td>r2</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>r3</td>
<td>r2</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>r3</td>
<td>r2</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>r3</td>
<td>r2</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>r3</td>
<td>r2</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>r3</td>
<td>r2</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

s1 denotes shifting in one input and updating the current state to n, r3 denotes reducing according to rule n; $\text{go to}$ denotes updating the current state to n.
to identify reduction steps from push operations. We define a subset of push operations as backward operations as follows.

**Definition 5.** Given a stack operation trace, assume a push operation \( x = \text{push}(s, p) \), and its preceding push operation \( x_{\text{pred}} = \text{push}(s^{\prime}, p') \), \( x \) is a backward push operation if and only if \( p <_{\text{pos}} p' \).

Intuitively, a push operation that pushes to a position that is smaller than its predecessor is a backward push. In Table 1, the push operation at step (4) is a backward operation because it pushes to position 3 and its preceding push operation at step (3) pushes to the same location 3. Similarly, the pushes at steps (6), (9), (11), (13), and (15) are also backward operations. A backward operation implies a step of reduction. This property can be exploited to discover input structure. The algorithm is presented in Algorithm 2. Given a stack operation trace \( T \), the algorithm scans each push operation \( x \) in the time order. Line 5 decides if \( x \) is a backward operation. If so, edges are introduced between the stack entry of push operations as backward operations as follows.

\[
\text{addEdge}(x); \\
\text{addNode}(y); \\
\text{addEdge}(x \rightarrow y);
\]

**Algorithm 2 Construct a ST from a stack operation trace \( T \).**

1: STFromTrace (\( T \))
2: { 
3: foreach operation in \( T \) with the form of \( x = \text{push}(s, p) \) { 
4: \( x \)'s preceding push operation \( x_{\text{pred}} = \text{push}(s^{\prime}, p') \); 
5: if \( p <_{\text{pos}} p' \) { 
6: foreach \( t \in \{ p, p' \} \) { 
7: \( y = \text{push}(\ldots, t) \) precedes \( x \) and is closest to \( x \); 
8: \( \text{addNode}(y); \) 
9: \( \text{addEdge}(x \rightarrow y); \) 
10: } 
11: } 
12: } 
13: }

Figure 8: The Derived ST For the Sample Trace in Table 1.

Consider our example trace in Table 1, the push operation at (6) is a backward operation, which pushes to position 2, and its preceding push operates at position 4. According to lines 7-11 in Algorithm 2, edges are introduced between the push at (6) and those that most recently push to stack positions of 2, 3, and 3, namely, the pushes at steps (2), (4) and (5), which push exactly the righthand side of rule (3) to the stack. The resulting ST is shown in Fig. 8, which faithfully mirrors the tree derivation.

**Extract The Stack Operation Trace.** One issue remains unsolved is to extract the stack operation trace. Recall that we only assume the program binary. It is challenging to identify which part of the binary contributes to operating the parsing stack. Fortunately, this part of execution often demonstrates unique runtime characteristics. To explain the idea, we first define the concept of data lineage.

**Definition 6.** The data lineage of variable \( v \) at an executed statement instance \( s_i \), denoted by \( DL(v@s_i) \), refers to the set of input values that affect the value of \( v \) at \( s_i \), through direct/indirect dynamic data dependence.

A dynamic data dependence exists between two statement instances \( x_i \) and \( y_j \) if and only if a variable is defined at \( x_i \) and then used at \( y_j \). In the below code snippet, the execution instances of statement 3, 4 and 5 are data dependent on that of 1. According to the above definition, \( DL(x@11) = \{ DL(a@3), DL(c@5) \} = \{ \text{INPUT}[1] \} \), \( DL(y@12) = \{ \text{INPUT}[2] \} \), \( DL(a@4) = DL(x@11) \cup DL(y@12) = \{ \text{INPUT}[1], \text{INPUT}[2] \} \). Efficient computation of data lineage can be found in [33].

1: \( x=\text{INPUT}[1]; \)
2: \( y=\text{INPUT}[2]; \)
3: \( a=x; \)
4: \( b=x+y; \)
5: \( c=A[x]; \)
6: ...

Data lineage is crucial to distinguish parsing stack operations. Specifically, multiple instances of instructions for a parsing stack push operation have increasing lineage sets, and the lineage set of each instance contains all the input symbols seen so far. Consider the general algorithm in Fig. 6, this property can be proved by showing \( DL(\text{state}[7]) \supseteq DL(\text{state}[6]) \cup DL(c) \) at lines 7 and 13, where \( \text{state}[6] \) stands for the value of \( \text{state} \) in the previous iteration. It is true for line 7 because for any instance \( i \), \( DL(\text{state}[7]) = DL(\text{state}[6]) \cup DL(c) \) as the value of \( \text{state} \) at 6 is a function of the \( \text{state} \) in the last iteration and \( i \). As for line 13, \( DL(\text{state}[13]) = DL(\text{state}[12]) \cup DL(a) \cup DL(c) \). Given the input string shown in the caption of Table 1, after the first iteration of the while loop, the lineage of the variable has the lineage of \( \{ B' \} \), after the second iteration, it becomes \( \{ B', T' \} \), and so on.

Increasing lineage is not unique to push operations. Other operations that perform accumulative computation on input, such as sum, may manifest the same lineage pattern. Those operations mostly access a single variable while push operations access a set of memory locations. In the bottom-up parsers we have studied, we successfully extract stack push operation trace by searching for write instructions that access a set of memory locations in a fluctuating pattern and have increasing data lineage.

**Deciding The Grammar Category.** As the different natures of top-down and bottom-up grammars lead to two different solutions, it becomes an issue to decide which one to apply given that we have no knowledge about the input grammar category of a program (recall we assume no source code access). In practice, if that happens, we apply both analyses. Our study in the evaluation section shows that by inspecting the two generated trees, one can easily tell which generated tree is the right one because applying the top-down approach to inputs with a bottom-up grammar generates a meaningless tree and vice versa.

**4. EVALUATION.**

Our analyses are implemented using Diablo [26] and Valgrind [23]. Diablo is used to perform post-dominance analysis on binaries, to facilitate Valgrind, which is used to instrument the binary and catch the data as well as control dependency, and build the ST
2003). We used three HTML files with different sizes (range from 63 bytes to 414 bytes). Wireshark is a widely used network trouble shooting tool, which contains many small parsers to handle different protocols. In order to transform one tree into another, a series of local operations are applied, which are: (a) move - change the position of a node; (b) delete - remove a non-root node in the tree by calculating their tree edit distance [7]; (c) insert - insert a node as a child of an existing node. The tree edit distance of two trees is defined as the number of primitive operations required in order to transform one tree to another, assuming each primitive operation has a unit cost.

Top-Down Grammars. We first evaluate our analysis for top-down grammars. The results are shown in Table 3 and Table 4, respectively. Bottom-up parsers are mostly generated by automatic tools. In order to evaluate the robustness of our analysis in the presence of various parser generation tools, for each program in the bottom-up category, we used two most popular open-source parser generators, bison-2.1 and byacc-1.9, to generate two different bottom-up parsers. Each program (version) is tested on a number of inputs. For each input, we compare the derived input tree with the real tree, which is acquired from the input specification.

We compare the derived tree and the real tree by calculating their tree edit distance [7]. Tree edit distance is a technique to compare two trees based on simple local operations of deleting, inserting, and relabeling nodes. A labeled tree is a tree in which each node is assigned a label. Recall that the internal nodes of our syntax trees are not labeled. In order to perform the comparison, we label an internal node \( n \) as the number of input symbols that is the union of all the children’s labels. One can consider that the label of an internal node \( n \) represents the input subsequence whose derivation is the tree rooted at \( n \). Three primitive operations are defined which can be applied to transform a labeled tree. They are: (a) relabel - change the label of a node; (b) delete - remove a non-root node in the tree by connecting its children to its parent; (c) insert - insert a node as a child of an existing node. The tree edit distance of two trees \( t_1 \) and \( t_2 \) is defined as the number of primitive operations required in order to transform \( t_1 \) to \( t_2 \), assuming each primitive operation has a unit cost.

Top-Down Grammars. We first evaluate our analysis for top-down grammars. The results are shown in Table 3. In order to evaluate the derived trees, we used Wireshark [4] to generate the real syntax trees for most programs except tidy. Wireshark is a very popular network trouble shooting tool, which contains manually crafted information about network protocol formats. It was widely used in other projects such as [10, 9, 15] to evaluate the quality of reverse engineered network protocol formats. For tidy, which was not documented by Wireshark, the corresponding real tree is generated under the guidance of HTML grammar [5].

Table 3: Experimental Result for Top-Down Grammars

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description (LOC)</th>
<th>Input size (bytes)</th>
<th>#Derived Node</th>
<th>#Real Node</th>
<th>Edit Dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tidy</td>
<td>An HTML file</td>
<td>126</td>
<td>32</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td>Apache 2.0.59</td>
<td>An HTTP server</td>
<td>1050</td>
<td>255</td>
<td>100</td>
<td>54</td>
</tr>
<tr>
<td>Asterisk 1.4.4</td>
<td>A voice over IP</td>
<td>515</td>
<td>17</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Zebra 0.95a</td>
<td>A GNU routing</td>
<td>190</td>
<td>30</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Samba 3.0.8</td>
<td>A SMB/CIFS pro-</td>
<td>133</td>
<td>35</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Some Leaves in Wireshark ST

Some Leaves in our ST

Figure 9: Tree Comparison Between Wireshark and Ours for apache.

4 According to [7], a tree \( t_1 \) is included in a tree \( t_2 \) if and only if \( t_1 \) can be obtained by deleting nodes from \( t_2 \).
to the implementation complexity. These applications typically contain a standard grammar file which can be used by the parser generation tool. Such grammar files can be used to provide the real trees for our evaluation. In particular, we instrument the grammar files so that if multiple symbols are going to be reduced to a higher level symbol, edges are added between the reduced symbols and the resulting symbol. For instance, we add new actions to the grammar file so that upon a reduction based on the grammar rule `input_item; semicolon_list ENDOF_LINE`, two edges will be added between a node representing `input_item` and the two nodes representing `semicolon_list` and `ENDOF_LINE`. Eventually, a syntax tree is explicitly constructed during parsing. Since this tree is stringently created according to the input grammar, it can be considered as a real tree.

The results for bottom-up grammars are presented in Table 4. Each row of `wc` corresponds to parsing a single file while each row of `wuftp` is for parsing a series of ftp commands in a session. Note that `byacc` failed to generate a parser for the grammar file of `gcc`, and hence we used only `bison`. For these applications, we are able to acquire STs that are identical to the real ones despite different benchmarks considered and different parser generators used.

A possible explanation is that the bottom-up parsers considered are all automatically generated by tools and thus their runtime behavior is well regulated, which makes them highly amenable to our analysis. In contrast, top-down parsers, due to their implementation simplicity, are often hand-coded and thus display significant variety. Potentially, bottom-up parsers in a different paradigm may degrade the effectiveness of our analysis. We plan to study more parsers and parser generators to further validate our technique in the future.

### Table 4: Experimental Result for Bottom-Up Grammars

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description (LOC)</th>
<th>Tool</th>
<th>Input Size (Bytes)</th>
<th>Derived Node</th>
<th># Real Node</th>
<th>Edit Dist.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bin-0.6</td>
<td>Arbitrary precision numeric processing language (14,4K)</td>
<td>Bison</td>
<td>372</td>
<td>253</td>
<td>253</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byacc</td>
<td>891</td>
<td>612</td>
<td>612</td>
<td>0</td>
</tr>
<tr>
<td>Wufpd-2.6.2</td>
<td>An FTP server (27,1K)</td>
<td>Bison</td>
<td>56</td>
<td>24</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byacc</td>
<td>241</td>
<td>97</td>
<td>97</td>
<td>0</td>
</tr>
<tr>
<td>gcc-3.4.6</td>
<td>GNU Compiler Collection (212K)</td>
<td>Bison</td>
<td>60</td>
<td>25</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>241</td>
<td>137</td>
<td>137</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>623</td>
<td>453</td>
<td>453</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9430</td>
<td>5649</td>
<td>5649</td>
<td>0</td>
</tr>
</tbody>
</table>

### 4.2 Deciding the Grammar Category

As we discussed earlier, if an input grammar can not be decided beforehand to be one of the two options, our strategy is to apply both analyses. In this experiment, we applied the top-down analysis to the set of bottom-up applications and applied the bottom-up analysis to the set of top-down programs and observe if we can easily tell which of the two trees is the desired one. Applying the bottom-up analysis to top-down programs failed to produce any trees as the analysis failed to identify the parsing stack. Applying the top-down analysis to bottom-up programs was able to produce trees. However, these trees are mostly meaningless and thus the winner becomes clear when compared to the trees generated by the bottom-up analysis. Due to the space limit, we show the two trees for the benchmark `bc` in Fig. 10 and Fig. 11. The input is a program shown below.

```c
int i=0;
for (i=0;i<3;i++) {
    b=i;
}
```

We can clearly see in Fig. 11, the tree generated by the top-down algorithm does not make sense as the labels on the second layer nodes are meaningless. In comparison, the tree in Fig. 10 clearly depicts the input structure.

![Figure 12: Performance Overhead of Execution Tracing](image)

### 4.3 Performance Overhead

The next experiment is to evaluate performance. Due to the space limit, we use `tidy` and `bc` (the parser is generated by `bison`) to evaluate the performance of our system and its sensitivity to input size. We feed the two programs with inputs of different sizes to observe the overhead imposed by our analyses. The overhead is measured by comparing the execution times against those of the corresponding base line runs on Valgrind without instrumentation.

The execution times of `tidy` for inputs with different sizes (varied from 800 to 40k bytes) are shown in Fig. 12(a). The performance overhead varies from 5X to 45X. This is due to the fact that larger inputs entail more operations. Thus, the control dependence stack becomes deeper and the number of labeled operations becomes larger, and thus the online maintenance induces more overhead. For `bc`, we use inputs with different sizes but with similar structure. This is because inputs with different structure will lead to significantly varied execution times as `bc` is an interpreter, whose execution time heavily depends on the structure of the input program. The execution times of `bc` are shown in Fig. 12(b). The overhead ranges from 6X to 8X for the given experiment inputs.

### 4.4 A Client Study on HDD

Delta debugging [32] is an automatic debugging technique that looks for a valid and minimal subset of a failure inducing input that produces the same failure through an iterative algorithm. Hierarchical Delta Debugging [21] improves the algorithm by considering the hierarchical structure of the input so that invalid input subsets can be avoided. However, HDD requires the programmer to provide the input grammar and the corresponding parser. We have built a completely automated HDD system by integrating our input derivation system with the HDD algorithm. The independence of a priori knowledge of input structure enables new applications such as failure report composition, which often targets on deployed software without source code. More details can be found in our technical report [16].

### 5. RELATED WORK

In the area of network security, research has been conducted to extract protocol formats from a large pool of network traces [10, 2], and from dynamic binary analysis [9, 15, 30, 11]. The network trace based techniques do not look at execution of network
applications. The accuracy of these approaches relies on the size of the trace pool and the heuristics used. In contrast, as demonstrated in [9, 15, 30, 11], by analyzing how the program processes the input data, dynamic binary analysis could be used to reveal the input structure. The difference of our technique compared with these binary analysis approaches is (1) they only handle top-down grammars, whereas we also handle bottom-up grammars, which are the other very important input category; (2) regarding top-down grammars, our system is superior for being more general, robust, and accurate since it captures the essence of the problem - the dynamic control dependence. Indeed, the heuristics used in all the other projects, such as loops and comparisons in [9, 30, 11] and execution contexts in [15], are a subset of control dependence information.

In [17], Lim et al. propose using static analysis to derive output structure. They observe that the structure of a program contains a wealth information of the output format. They use interprocedural control flow graphs, call graphs, and value set analysis to extract output format step by step. Their technique is a static analysis that analyzes output structure instead of input structure. Our technique is dynamic analysis based and is more appropriate for deriving the structure for a single input.

Our technique is also related to grammar inference (GI) in the language research area. Grammar inference concerns the acquisition of the syntax or the grammar of a target language. It is defined as the process of learning a grammar from a set of grammatically correct and, if available, incorrect sentences. More information can be found in [24], a survey by Parekh et al. on computational approaches for learning different classes of formal languages. Compared to our work, which derives input structure from program execution, GI techniques try to tackle the problem from a much harder way, i.e., by only looking at inputs. As reported in [10], GI approaches are too complex to apply and have very limited success.

6. LIMITATIONS AND FUTURE WORK

There are a number of issues that affect the effectiveness of our technique. First, the robustness of our technique needs to be further tested. Although most programs we have seen so far take inputs of top-down or bottom-up grammars, there may exist other grammars. Even with the two types of grammars, individual parser implementations may not fall into the specific paradigms. In a worst scenario, one might write a top-down parser so that the control dependence structure is totally independent of the parsing structure. In the future, we plan to study more parser implementations and evaluate the performance of our technique on obfuscated binaries such as malware and viruses. Second, our technique derives the structure
for individual inputs. It is more desirable to be able to infer the input grammar especially for applications like testing. While combining the syntax trees of multiple inputs into a grammar is our ongoing work, we believe at the end, in order to acquire a complete grammar, we need to address the coverage problem, meaning we need enough inputs to exercise all parts of a grammar. Third, our technique currently only derives the syntactical structure. Many security applications desire semantic information as well, such as the keywords of a protocol, constraints across multiple fields (e.g., the length of field \( B \) is confined by the value of field \( A \)). We plan to extend our technique to solve this problem in our future work.

7. CONCLUSION

Deriving input syntactic structure is very important for a wide variety of applications such as test generation, delta debugging, failure reporting and protocol reverse engineering. We propose two dynamic analyses that construct input structure from program execution. Our technique does not require source code or any symbolic information. Our evaluation shows that the proposed techniques are highly effective and produce input syntax trees with high quality.

8. REFERENCES