# Mining Input Grammars from Dynamic Control Flow

Rahul Gopinath
rahul.gopinath@cispa.saarland
CISPA Helmholtz Center for
Information Security
Saarbrücken, Germany

Björn Mathis
bjoern.mathis@cispa.saarland
CISPA Helmholtz Center for
Information Security
Saarbrücken, Germany

Andreas Zeller
zeller@cispa.saarland
CISPA Helmholtz Center for
Information Security
Saarbrücken, Germany

## ABSTRACT

One of the key properties of a program is its input specification. Having a formal input specification can be critical in fields such as vulnerability analysis, reverse engineering, software testing, clone detection, or refactoring. Unfortunately, accurate input specifications for typical programs are often unavailable or out of date.

In this paper, we present a general algorithm that takes a program and a small set of sample inputs and automatically infers a readable context-free grammar capturing the input language of the program. We infer the syntactic input structure only by observing access of input characters at different locations of the input parser. This works on all stack based recursive descent input parsers, including parser combinators, and works entirely without program specific heuristics. Our *Mimid* prototype produced accurate and readable grammars for a variety of evaluation subjects, including complex languages such as JSON, TinyC, and JavaScript.

## CCS CONCEPTS

• **Software and its engineering** → **Software reverse engineering**; *Dynamic analysis*; • **Theory of computation** → **Grammars and context-free languages**.

## KEYWORDS

context-free grammar, dynamic analysis, fuzzing, dataflow, control-flow

## 1 INTRODUCTION

One of the key properties of a program is its input specification. Having a formal input specification is important in diverse fields such as reverse engineering [18], program refactoring [29], and program comprehension [23, 44]. To generate complex system inputs for testing, a specification for the input language is practically mandatory [12, 27, 32]

However, formal input specifications are seldom available, and when they are, they may be incomplete [14], obsolete, or inaccurate with respect to the program [47]. Unfortunately, determining the input language of a program is a non-trivial problem even when the source code is available [27]. Therefore, *obtaining input models automatically* bears great promise for software engineering.

⟨*START*⟩ ::= ⟨*json_raw*⟩

⟨*json_raw*⟩ ::= '"' ⟨*json_string'*⟩ | '[' ⟨*json_list'*⟩ | '{' ⟨*json_dict'*⟩
 | ⟨*json_number'*⟩ | 'true' | 'false' | 'null'

⟨*json_number'*⟩ ::= ⟨*json_number*⟩+
 | ⟨*json_number*⟩+ 'e' ⟨*json_number*⟩+

⟨*json_number*⟩ ::= '+' | '-' | '.' | '[0-9]' | 'E' | 'e'

⟨*json_string'*⟩ ::= ⟨*json_string*⟩* '"'

⟨*json_list'*⟩ ::= ']'
 | ⟨*json_raw*⟩ (',' ⟨*json_raw*⟩ )* ']'
 | (',' ⟨*json_raw*⟩ )+ (',' ⟨*json_raw*⟩)* ']'

⟨*json_dict'*⟩ ::= '}'
 | ('"' ⟨*json_string'*⟩ ':' ⟨*json_raw*⟩ ',' )*
 '"' ⟨*json_string'*⟩ ':' ⟨*json_raw*⟩ '}'

⟨*json_string*⟩ ::= ' ' | '!' | '#' | '$' | '%' | '&' | '''
 | '*' | '+' | '-' | ',' | '.' | '/' | ':' | ';'
 | '<' | '=' | '>' | '?' | '@' | '[' | ']' | '^' | '_',' ',
 | '{' | '|' | '}' | '~'
 | '[A-Za-z0-9]' | '\' ⟨*decode_escape*⟩

⟨*decode_escape*⟩ ::= '"' | '/' | 'b' | 'f' | 'n' | 'r' | 't'

**Figure 1: JSON grammar extracted from *microjson.py*.**

While researchers have tried to tackle the problem of grammar recovery using black-box approaches [14, 48], the seminal paper by Angluin and Kharitonov [11] shows that a pure black-box approach is doomed to failure as *there cannot be a polynomial time algorithm* in terms of the number of queries needed for recovering a context-free grammar from membership queries alone. Hence, only *white-box* approaches that take program semantics into account can obtain an accurate input specification.

The first white-box approach to extract input structures from programs is the work by Lin et al. [39, 40], which recovers *parse trees* from inputs using a combination of static and dynamic analysis. However, Lin et al. stop at recovering the parse trees with limited labeling, and the recovery of a grammar from the parse trees is non-trivial (as the authors recognize in the paper, and as indicated by the limited number publications in this topic even though recovering such a grammar is important in many areas of software engineering).

The one approach so far that extracts human-readable input grammars from programs is *Autogram* [33] by Höschele et al. Given a program and a set of inputs, *Autogram* extracts an approximate context-free grammar of the program's input language. It does so by tracking the *dynamic data flow* between *variables* at different locations: If a part of the input is assigned to a variable called protocol, this substring forms a ⟨*protocol*⟩ *nonterminal* in the grammar.

While dynamic data flows produce well-structured grammars on a number of subjects, the approach depends on a number of assumptions, the most important being that some data flow to a *unique variable* has to be there in the first place. If a program accepts a structured input where only part of the input is saved and used, there is no data flow to learn from in the unsaved parts.

Second, learning from dynamic data flows requires special heuristics to work around common parsing patterns identified; the data flow induced by a parser lookahead, for instance, has to be ignored as it would otherwise break the model [33]. Finally, common patterns such as passing the complete input as an array with an index indicating current parse status can break the subsumption model. These shortcomings limit learning from data flows to a small class of input processors.

In this paper, we describe a *general algorithm* to recover the *input grammar* from a program without any of these limitations. Rather than being based on *data flows* to unique variables, it recovers the input grammar from *dynamic control flow* and how input characters are *accessed* from different locations in the parser. Our algorithm works regardless of whether and how the parsed data is stored and requires no heuristics to identify parsing patterns. It works on all program stack based recursive descent parsers, including modern techniques such as parser combinators. The recursive descent family of parsers makes up 80% of the top programming language parsers on GitHub [41].

The resulting grammars are well-structured and very readable. As an example, consider the JSON grammar shown in Figure 1, which our *Mimid* prototype extracted from *microjson.py*.[1] Each JSON element has its own production rule; ⟨*json_number*⟩, for instance, lists a number as a string of digits. Rules capture the recursive nature of the input: A ⟨*json_list*⟩ contains ⟨*json_raw*⟩ elements, which in turn are other JSON values. All identifiers of nonterminals are derived from the names of the input functions that consume them. All this makes for very readable grammars that can be easily understood, adapted, and extended.

Why do we emphasize the *readability* of extracted grammars? Recovering readable grammars is important in a number of areas in software engineering [46].

- The recovered grammar represents the input specification of the given program and provides an overview of how the program processes its inputs. This can be used for understanding the program and identifying where possible vulnerabilities lie. The grammar recovered can be useful for identifying the difference between differing implementations, and even for identifying how the input specification changed across revisions, and identifying compatibility issues.
- A large number of recent bugs have been caused by incorrectly implemented parsers for common specifications [43]. Recovering the actual grammar of the inputs accepted by the program can help us identify the problematic parts easily.
- Another important use of grammar is for debugging where techniques such as hierarchical delta debugging [42] can only be applied if one has the program input grammar. An input grammar for a given program can also be of use if one

wants to repair damaged inputs [37]. In all these cases, the inputs need to be decomposed into smaller fragments.
- When using a grammar as an input producer for testing, readable grammars allow testers to refine the grammar with specific inputs such as logins, passwords, or exploits. Given such a grammar, one can contract the grammar such that only specific subparts are generated if one is first able to understand what parts of the grammar correspond to the part that one is interested in. If even a partial human readable grammar is available, it can be expanded with human knowledge on features where the miner may not have sufficient inputs or identify vulnerabilities through human inspection of the grammar (e.g. allowing special characters in usernames). Fuzzers can only allow for such control if the model is human-readable.

In the remainder of this paper, we first illustrate our approach on a simple, self-contained example in Section 2. We then detail our contributions:

(1) We provide a general algorithm for deriving the context-free approximation of an input language from a recursive descent parser. Our approach relies on *tracking character accesses in the input buffer* (Section 3), which is easy to implement for a variety of languages that support string wrappers, or the source can be transformed to support such wrappers, or dynamic taint information is available. From the tracked accesses, we then infer *parse trees* (Section 4), which we generalize by means of *active learning*[2], before passing them to our grammar inference (Section 5). Our approach distills the developer supplied method names and input processing structure to produce human-readable input grammars.

(2) We evaluate our approach using subjects in both *Python* and *C* and recover complex grammars such as *JavaScript* and *TinyC*. For the evaluation, we assess both precision and recall (Section 6) of our grammars. When compared against learning from dynamic data flows (so far the only approach for inference of human-readable grammars), we found our approach to be superior in both precision and recall.

(3) In our evaluation, we also show that our approach is applicable in contexts in which no usable data flow of input fragments to variables exists such as modern parsing techniques like *parser combinators* which make the state of the art for writing secure parsers [17, 19].

After discussing limitations (Section 7) and related work (Section 8), Section 9 closes with conclusion and future work. The complete source code of our approach and evaluation is available.

## 2 OUR APPROACH IN A NUTSHELL

How does our approach work? We use lightweight instrumentation to track *dynamic control flow* and lightweight string wrappers (or dynamic taint information if available) to identify in which control flow nodes specific input characters are accessed. The character accesses as well as the corresponding control flow nodes are then logged. A *parse tree* of the input string is extracted from that trace using the following rules (which mostly follow Lin et al. [39]):

---

[1]We removed rules pertaining to whitespace processing for clarity.

[2] The term *active learning* was first used by Dana Angluin [10] for grammar learning.

```
1    def digit(i):
2        return i in "0123456789"
3
4    def parse_num(s,i):
5        n = ''
6        while i != len(s) and digit(s[i]):
7            n += s[i]
8            i = i +1
9        return i, n
10
11   def parse_paren(s, i):
12       assert s[i] == '('
13       i, v = parse_expr(s, i+1)
14       if i == len(s): raise Ex(s, i)
15       assert s[i] == ')'
16       return i+1, v
17
18   def parse_expr(s, i = 0):
19       expr, is_op = [], True
20       while i < len(s):
21           c = s[i]
22           if digit(c):
23               if not is_op: raise Ex(s,i)
24               i,num = parse_num(s,i)
25               expr.append(num)
26               is_op = False
27           elif c in ['+', '-', '*', '/']:
28               if is_op: raise Ex(s,i)
29               expr.append(c)
30               is_op, i = True, i + 1
31           elif c == '(':
32               if not is_op: raise Ex(s,i)
33               i, cexpr = parse_paren(s, i)
34               expr.append(cexpr)
35               is_op = False
36           elif c == ')': break
37           else: raise Ex(s,i)
38       if is_op: raise Ex(s,i)
39       return i, expr
40
41   def main(arg):
42       return parse_expr(arg)
43
```

**Figure 2: A Python parser for math expressions**

(1) A recursive descent parser tries alternatives rules until the first successful parse, and a character is not accessed after it was successfully parsed. Hence, the method call that accesses a particular input character last *directly consumes* that character. E.g. if a call to digit() is the last to access the digit 3, then 3 is consumed by that call to digit().

(2) A method call *indirectly consumes* a character if one of the nested method calls *consumes* that character.

(3) A control flow node such as a conditional (e.g. *if*) or loop (e.g. *while*) is regarded as a pseudo method. The name for the pseudo method is derived from the parent method name and a unique identifier.

(4) Names of methods that *consume* some part of the input are used as the *nonterminal* symbol for the corresponding node in the parse tree for that part.

As an example, consider Figure 2 showing a complete Python program that accepts mathematical expressions using a recursive descent parser. Running it with an argument 9+3/4 yields the tentative parse tree shown in Figure 3. The method parse_num(), which parses numeric elements, becomes the *nonterminal* ⟨*parse_num*⟩ in the parse tree (Figure 3), representing input numeric elements.
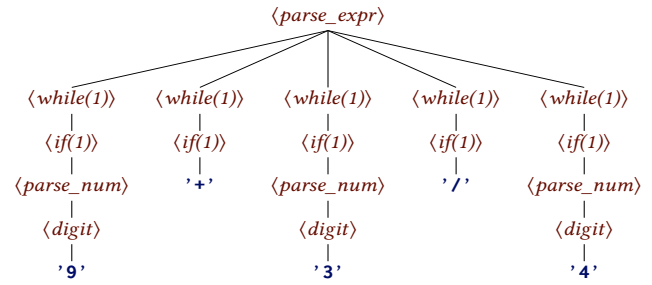


**Figure 3: Derivation tree for 9+3/4**

⟨*START*⟩ ::= (⟨*parse_expr*⟩[*+-/])* ⟨*parse_expr*⟩

⟨*parse_expr*⟩ ::= ⟨*parse_num*⟩ | ⟨*parse_paren*⟩

⟨*parse_paren*⟩ ::= '(' ⟨*parse_expr*⟩ ')'

⟨*parse_num*⟩ ::= ⟨*parse_digit*⟩+

⟨*digit*⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

**Figure 4: A grammar derived from the parser in Figure 2**

The parse tree is then processed further to correctly identify *compatible nonterminal* nodes with the following steps:

(1) Collect and group nodes by the names of their *nonterminal* symbols.

(2) Identify subdivisions within each *nonterminal* group by substituting nodes with each other in the corresponding parse trees and verifying the validity of resulting parse trees.

(3) Generate unique *nonterminal* symbols for identified subgroups and update them in the corresponding nodes in the parse trees. This gives us accurately labelled parse trees.

We then use this accurately labelled parse trees to infer a grammar:

(1) Each node in a parse tree is collected, and the *nonterminal* symbol becomes a *nonterminal* in the generated grammar.

(2) Each sequence of *terminal* and *nonterminal* symbols from child nodes of the *nonterminal* becomes a possible expansion rule for that *nonterminal*.

(3) Apply the prefix-tree acceptor algorithm [25] to identify regular right hand sides [36] due to loops and conditionals.

(4) Apply generalization of tokens such as converting a seen list of integers to an expression that can generate any integer.

(5) Compact and cleanup the grammar.

We extract such parse trees for a number of given inputs. Next, we traverse each tree and identify *loop* nodes that are similar as we detail in Section 4.1. This results in parse trees where similar nodes have similar names. Finally, we construct the grammar by recursively traversing each parse tree and collecting the name and children types and names for each node. The node names become *nonterminal* symbols in the grammar, and each set of children becomes one possible expansion in the grammar being constructed. The child nodes that represent characters become terminal symbols in the constructed grammar.

For our example, the final result is the grammar in Figure 4, which exactly reflects the capabilities of the program in Figure 2. Again, the grammar is readable with near-textbook quality and well reflects the input structure.

# 3 TRACKING CONTROL FLOW AND COMPARISONS

Let us now start with describing the details of how we infer the parse tree from dynamic control flow.

For tracking the control flow, we programmatically modify the parser source. We insert a tracker for both a method entry and an exit as well as trackers for control flow entry and exit for any conditions and loops. For the purposes of our approach, we consider these control structures as *pseudo methods*. Every such method call (both true and pseudo method calls) gets a unique identifier from a counter such that a child of the current method or a later method call gets a larger new method identifier than the current one.

In Python, for tracking the character accesses being made, we simply wrap the input string in a proxy object that log access to characters. We annotate each character accessed with the current method name. For C, the above information is recovered using a lightweight dynamic taint framework.

What if one can not distinguish parsing functions? In such case, one can simply conservatively instrument *all* functions. Given that we only log access to the initial buffer, the non-parsing functions will have no effect (except on performance) on the parse trees generated. Another approach would be to make use of static analysis to identify *parsing* functions that *might* access the initial buffer, and insert trackers only on those functions.

In both languages, we stop tracking as soon as the input is transformed or copied into a different data structure (if there is a lexer, we continue the tracking into the parser stage)[3].

We note that access to the source code is not a requirement. So long as one can track access to the initial input string and identify method calls (by their address if we are analysing a binary) one can recover the required information.

# 4 FROM TRACES TO PARSE TREES

The previous section showed how to annotate the execution tree with indexes of the input string accessed by each method call. At this point, however, there can be multiple accesses by different methods at the same input index. To assign specific indexes to specific method calls, we follow a simple strategy informed from the characteristics of recursive descent parsers: The *last method call* that accessed a character directly *consumes* that character, and its parent method calls indirectly consume that character[4].

We now resolve the *ownership* of input index ranges between nodes. Given any node, we obtain the starting and ending indexes that were *consumed* by the node. If no other node has *consumed* any character within that range, the current node *directly owns* that range, and all its parent nodes *indirectly own* that range. If a range overlap is found between a parent node and a child node, the tie is decided in favor of the child, with the child node *directly owning* the overlap, and the parent node *indirectly owning* the overlap. If an overlap is found between two sibling nodes, the tie is decided in favor of the sibling that accessed the part last (as the last access

to the character defines the consumer method). The sibling that is in the overlap is recursively scanned, and any descendent of that node that are contained in the overlap are removed.

Once the indexes are associated with method call identifiers, we generate a *call tree* with each method identifier arranged such that methods called from a given method are its children. The directly owned input indexes are added as the leaf nodes from the corresponding method call node. As there is no overlap, such a tree can be considered as a *parse tree* for the given input string.

The parse tree at this point is given in Figure 5a, which we call the *non-generalized* parse tree of the input string. In Figure 5a, each pseudo method has a list of values in parenthesis, in the following format. The last value in the parenthesis is the identifier for the control flow node taken. That is, given a node name as ⟨*if(2:0, 3:1)*⟩, the identifier is 3:1. It indicates that the corresponding if statement was the third conditional in the program, and the execution took the first (*if*) branch of the conditional. If the identifier was 3:2, it would indicate that the execution took the *else* branch, and for larger values, it indicates the corresponding branch of a cascading if statement or a case statement. In the case of loops, there is only a single branch that has child nodes, and hence this is indicated by 0. The values before the identifier correspond to the identifiers of the pseudo-method parents of this node until the first method call. That is, the ⟨*if(2:0, 3:1)*⟩ has a parent node that is a loop, and it is the second loop in the program.

While we have produced a parse tree, it is not yet in a format from which we can recover the context-free grammar. To be able to do so, we need accurately labeled parse trees where any given node can be replaced by a node of similar kind without affecting the parse validity of the string. The problem here is that not all iterations of loops are replaceable with each other. That is, loops can be influenced by the previous iterations. For example, consider the derivation tree in Figure 5a. If one considers each iteration of the loop to be one *alternate expansion* to the corresponding *nonterminal*, the rule recovered is:

$$\langle expr \rangle \rightarrow \text{num} \mid + \mid /$$

However, this is incorrect as a single free-standing operator such as + is not a valid value. The problem is that is_op encodes a link between different iterations. Hence, we annotate each individual iteration and leave recovering the actual repeating structure for the next step. A similar problem occurs with method calls too. In the produced parse tree we assume that any given *nonterminal*— say ⟨*parse_num*⟩ — can be replaced by another instance of ⟨*parse_num*⟩ without affecting the validity of the string. However, this assumption may not hold true in every case. The behavior of a method may be influenced by a number of factors including its parameters and the global environment. We fix this in the next step.

## 4.1 Active Learning of Labeling

To determine the precise labeling of each node, we first collect each node from each parse tree and group them by the name of the *nonterminal*. That is, all ⟨*parse_num*⟩ nodes go together, so do all ⟨*if(1:0, 1:1)*⟩.

Next, the challenge is to identify the buckets under the grouped nodes that are compatible with each other. We consider a node as *replaceable* with another if the string produced from a parse tree

---

[3] This is one of the main differences of our technique from both Lin et al. [40] and Höschele et al. [33] who track dynamic taints throughout the program.

[4] We note that this is one of the major difference of our technique from Lin et al. [40] who define *parsing points* as the last point the character was used before the parsing point of its successor. The problem with such an approach is that real-world parsers may access characters out of order. See Section 8.2 for details.

**(a) Non-generalized parse tree. The number before colon indicates the particular pseudo-method, and number after the colon identifies the alternative if any. That is, if(... 5:2) is the fifth if whose else branch was taken. The pseudo method stack is inside '()'.**

**(b) Generalized parse tree. The number in suffix after colon indicates the generalized identifier after validating replacements. As before, the pseudo method stack is contained in the parenthesis, which is also updated when the parent is updated during generalization.**
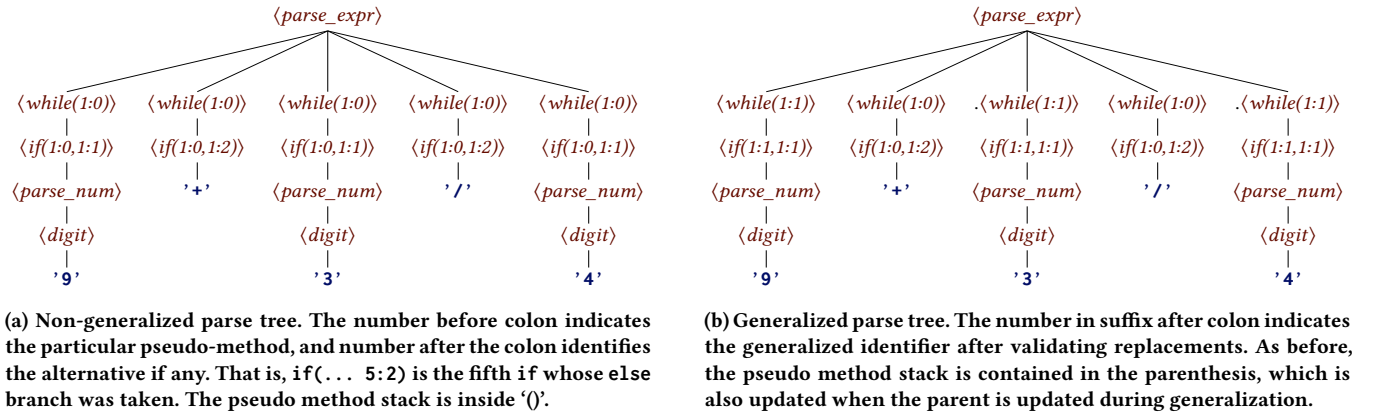
**Figure 5: Parse trees for 9+3/4. The prefix before colon indicates the static identifier of the control structure in the method. That is, the first if gets the prefix 1:. The suffix is explained above.**

where the first node is replaced by the second is parsed correctly – that is, the generated string is parsed without any errors, and the parse tree generated from the new parse has the *same structure* as the tree generated by replacing the node. A node is *compatible* with another if both nodes are *replaceable* with each other.

Unfortunately, compatibility is not *transitive* if one looks at parse validity. For example, say, there are three *words* in a language — a, b, and ac. Each *word* is composed of individual *letters*. In the case of a, and b, the corresponding letter, and for ac, the letters a, and c.

⟨*start*⟩ ::= ⟨*word1*⟩ | ⟨*word2*⟩ | ⟨*word3*⟩

⟨*word1*⟩ ::= ⟨*letter_a*⟩

⟨*word2*⟩ ::= ⟨*letter_b*⟩

⟨*word3*⟩ ::= ⟨*letter_a*⟩⟨*letter_c*⟩

⟨*letter_a*⟩ ::= 'a'

⟨*letter_b*⟩ ::= 'b'

⟨*letter_c*⟩ ::= 'c'

Now, consider the parse trees of a, b, and ac.
(⟨*start*⟩ (⟨*word1*⟩ (⟨*letter_a*⟩ 'a')))
(⟨*start*⟩ (⟨*word2*⟩ (⟨*letter_b*⟩ 'b')))
(⟨*start*⟩ (⟨*word3*⟩ (⟨*letter_a*⟩ 'a') (⟨*letter_c*⟩ 'c')))
Here, the nodes ⟨*letter_a*⟩ across parse trees are compatible because the generated strings are exactly the same. Next, the ⟨*letter_a*⟩ under ⟨*word1*⟩ is compatible with ⟨*letter_b*⟩ under ⟨*word2*⟩. The generated strings are a and b. So, is the node ⟨*letter_b*⟩ under ⟨*word2*⟩ compatible with ⟨*letter_a*⟩ under ⟨*word3*⟩? Unfortunately not, as the string generated from
(⟨*start*⟩ (⟨*word3*⟩ (⟨*letter_b*⟩ 'b') (⟨*letter_c*⟩ 'c')))
is bc which is not in the language.

This means that for the accurate identification of unique node labels, each node has to be compared with all other nodes with the same name, which gives us a complexity of $O(n^2)$ in the worst case in terms of the number of nodes. However, we found that the assumption of *transitivity* rarely breaks, and even then, the inaccuracy induced affects less than 10% of inputs generated from the grammar (See the evaluation of *mathexpr.py*). Since the assumption

of *transitivity* allows us to reduce the computational effort, our evaluation is implemented assuming *transitivity* of compatibility.[5]

Once we have identified the compatibility buckets, we can update the nodes in them with unique suffixes corresponding to each bucket and update the node name of each one with the suffix. In the case of loop nodes, we also update the stack name of this node in all the descendent elements of this node — all descendent nodes up to the next non-pseudo method call. The idea here is that if there are two unique loop iterations that are incompatible with each other, then any other control flow nodes inside that loops such as conditionals should also be considered incompatible even if the same alternative path is taken in the conditional during the execution of both iterations.

Once the process is complete, all the nodes in all the parse trees will be labeled with consistent and correct identifiers. These can then be extracted to produce the correct grammar. The generalized counterpart to Figure 3 is given in Figure 5b.

## 4.2  Active Learning of Nullability

Some of the loops may be skipped completely — e.g., an empty object in JSON. These can be identified using *active learning*. The idea is to replace all *consecutive* loop nodes that are the children of a given node in a given parse tree. Then check the validity of the string produced from that tree. If the parse structure of the new string is correct, and this can be done on all parse trees and at all points where this is possible, the loop is marked as nullable.

For conditional nodes, whether an if node can be skipped can be determined statically without active learning, by simply checking for the presence of an else branch. However, if conditionals may be labeled incorrectly. For example, consider this set of statements:

```
1  if g_validate:
2    validate_header(header)
```

While the if does not have an else branch, we do not know whether the body of the conditional can be skipped or not. In particular, the g_validate may be a global configuration option

---

[5] We note that a user of our technique does not need to rely on this assumption. One can choose to do the complete $O(n^2)$ verification, or can choose anything in between that and the faster but approximate version.

which may mean that it is always enabled or always disabled for specific kinds of parse trees. While we have not found such conditionals in our subjects, if additional accuracy is desired, the optional parts of conditionals may also be verified using active learning.

With this, our trees are accurately labeled and ready for inferring grammars from them.

## 5  GRAMMAR INFERENCE

For constructing a grammar out of parse trees, we traverse each parse tree starting from the top. Each node we see, if it is not a character node, is marked as a *nonterminal* in the grammar. The children are placed as the rule for expansion of the *nonterminal* in the grammar. If the child is a non-character node, the token in the expansion will be a reference to the corresponding *nonterminal* in the grammar. There may be multiple alternate expansions to the same *nonterminal* even from the same tree as the same method call may be made recursively. This is detailed in Algorithm 1.

---

**Algorithm 1** Extracting the basic grammar

---

**function** EXTRACT_GRAMMAR(node, grammar)
    name, uid, children, stack ⇐ node
    a_name ⇐ name + uid + stack
    rule ⇐ []
    **if** a_name ∉ *grammar* **then**
        grammar[a_name] ⇐ {rule}
    **else**
        grammar[a_name].*add*(rule)
    **if** children = ∅ **then**
        **return** *terminal*, a_name
    **else**
        **for** child ← children **do**
            kind, cname ⇐ extract_grammar(child)
            **if** kind = *terminal* **then**
                rule + = to_terminal(cname)
            **else**
                rule + = to_nonterminal(cname)
    **return** *nonterminal*, a_name

---

An additional challenge comes from identifying repeating patterns. From the various solutions of this problem [25], we chose a modification of the *prefix tree acceptor* algorithm[6]. Once we run the modified *PTA* algorithm, the previous grammar is transformed to:

⟨*expr*⟩ → while:1

⟨*expr*⟩ → (while:1 while:2)+ while:1

⟨*while:1*⟩ → if:1

⟨*if:1*⟩ → num

⟨*while:2*⟩ → +

The while:<n> can be replaced by the regular expression summaries of the corresponding rules recursively. Here, this gives us the regular expression:

⟨*expr*⟩ → num | (num [+])+ num

---

[6] Unlike the original *PTA*, which considers only repeating patterns of single characters, we first scan for and identify repeating patterns of any block size. We next scan the inputs for any instances of the identified repeating patterns. These inputs are then chunked and considered as the alphabets as in the original *PTA* algorithm.

While generalizing, we can replace any + with ∗ provided all the items inside the group are nullable. Similarly, when merging regular expressions corresponding to conditionals, one can add (…|) i.e. an $\epsilon$ alternative, provided the corresponding if condition was nullable. These steps generate the right hand side regular expressions in Figure 4 for a simple program given in Figure 2. For details on learning regular expressions from samples, see Higuera [22]. The grammar derived from *microjson.py* after removing differences due to *white spaces* is given in Figure 1.

### 5.1  Generalizing Tokens

The grammar we have generated so far is a faithful reproduction of the parsing process. This grammar is however not complete with respect to the generalization. The reason is that typical languages rely on external *libc* calls such as *strtod*, *strtof*, and *strstr*. The *internal structure* of the portions thus parsed are lost to our simple tracer. Hence, we need to recover the structure of these items separately. A similar problem also occurs in parsers that uses an initial *scanner* that feeds tokens into parser. These result in *nonterminal* symbols which simply contain a long list of tokens such as

⟨*int*⟩ ::= '786022139' | '1101' | '934' | '898880' | '1'

which need to be generalized. The solution here is a simple generalization, with the following steps.

(1) Collect all *terminal* symbols in the grammar.
(2) Split each *terminal* symbol into individual characters.
(3) Widen the character into its parent group. E.g. Given a character '1', widen it first to '⟨*digit*⟩', then to '⟨*alphanum*⟩', and lastly to '⟨*anychar*⟩', checking to make sure that each widening is accepted by the program, and is parsed in the same manner.
(4) Deduplicate the resulting grammar
(5) Apply regular expression learning using the PTA algorithm to obtain the fully generalized grammar.

### 5.2  Producing a Compact Grammar

At this point, the mined grammar is readable but verbose. There are a number of transformations that one can take to reduce its verbosity without changing the language defined by the grammar. These are as follows:

(1) If there is any *nonterminal* that is defined by a single rule with a single token, delete the key from the grammar and replace all references to that key with the token instead.
(2) If there are multiple keys with the same rule set, choose one, delete the rest, and update the references to other keys with the chosen one.
(3) If there are duplicate rules under the same key, remove the redundant rules.
(4) Remove any rule that is the same as the key it defines.
(5) If there is any key that is referred to on a single rule on a single token, and the key is defined by just one rule, delete the key from the grammar and replace the reference to the key with the rule.

We repeat these steps as long as the number of rules in the grammar decreases in each cycle. This produces a smaller grammar that defines the same language.

# 6 EVALUATION

We compare our approach of learning grammars from dynamic control flow with the state of the art, namely learning grammars from dynamic data flow, as embodied in the *Autogram* tool [33]. The original *Autogram* implementation extracts input grammars from Java programs—in contrast to our *Mimid* prototype, which works on Python and C programs. For our comparison, we therefore relied on the code from *Mining Input Grammars* [51] from Zeller et al., which embodies the *Autogram* approach by learning grammars from dynamic data flow of Python programs. As we are interested in comparing algorithms, rather than tools, for all differences in results, we investigate how much they are due to *conceptual* differences.

We also note that the Python implementation of learning from dynamic data flows does not implement generalization of character sets to regular expressions (unlike the original *Autogram* tool [33]). For a fair comparisons of Python subjects, we thus have disabled *Mimid* generalization of character sets to larger regular expressions. For C subjects, generalization of character sets is enabled.

Our research questions are as follows:

**RQ 1.** How accurate are the derived grammars as *producers?*
**RQ 2.** How accurate are the derived grammars as *parsers?*
**RQ 3.** Can one apply *Mimid* to modern parsing techniques such as combinatory parsing?

## 6.1 Subjects

We focused on subjects that demonstrated different styles of writing recursive descent parsers, with different levels of grammars, with and without a lexing stage, and in different implementation languages. Our subjects were the following (the kind of the grammar – regular or context free is indicated in parenthesis).

***calc.py*** (CFG) – a simple recursive descent program written in textbook style from the Codeproject[1], simplified and converted to Python. It also forms the running example in Section 2. We used self generated expressions to mine the grammar and evaluate.

***mathexpr.py*** (CFG) – a more advanced expression evaluator from the Codeproject [5].It includes pre-defined constants, method calls, and the ability to define variables. As in the case with *calc.py*, we used self generated expressions and test cases to mine the grammar and evaluate.

***cgidecode.py*** (RG) – the cgidecode.py Python implementation from the chapter on *Code Coverage* [49] from Zeller et al. This is an example of a parser that is a simple state machine. It is not recursive and hence does not use the stack. For cgidecode.py, we used self generated expressions to mine the grammar and evaluate.

***urlparse.py*** (RG) – the URL parser part of the Python *urllib* library[9]. An example of an ad hoc parsing with little ordering between how the parts are parsed. For initial mining and evaluation, we used the URLs generated from passing tests using the *test_urllib.py* in the Python distribution. We also used a handwritten grammar to generate inputs as we detail later.

***microjson.py*** (CFG) – a minimal JSON parser from Github [6]. We fixed a few bugs in this project during the course of extracting its grammar (merged upstream). For mining, we chose ten simple samples that explored all code paths in the parser. For

**Table 1: Inputs generated by inferred grammars that were accepted by the program (1,000 inputs each)**

| | from data flows (state of the art) | from control flows (our approach) |
|---|---|---|
| *calc.py* | 36.5% | 100.0% |
| *mathexpr.py* | 30.3% | 87.5% |
| *cgidecode.py* | 47.8% | 100.0% |
| *urlparse.py* | 100.0% | 100.0% |
| *microjson.py* | 53.8% | 98.7% |
| *parseclisp.py* | 100.0% | 99.3% |
| *jsonparser.c* | n/a | 100.0% |
| *tiny.c* | n/a | 100.0% |
| *mjs.c* | n/a | 95.4% |

our evaluation, we used 100 samples of JSON generated from the following JSON API end points: *api.duckduckgo.com*, *developer.github.com*, *api.github.com*, *dictionaryapi.com*, *wordsapi.com*, *tech.yandex.com*. We also added sample JSON files from *json.org*, *json-schema.org*, *jsonlint*, *www.w3schools.com*, and *opensource.adobe.com*.

***parseclisp.py*** (CFG) – The conversion of an s-expression Parser using Parsec [4] to PyParsec. We used a *golden grammar* (a given correct grammar) to generate inputs.

***tiny.c*** (CFG) – a minimal C compiler implemented in C [8]. The *tinyc* parser uses a separate lexing stage. We used a golden grammar given in the *tinyc* source to generate the inputs.

***mjs.c*** (CFG) – a minimal Javascript interpreter implemented in C [7]. We used the ANTLR grammar for JavaScript from the ANTLR project to generate inputs.

***jsonparser.c*** (CFG) – a fast JSON parser implemented in C [3].

## 6.2 RQ 1. Grammar Accuracy as Producer

For our experiments, we used both learning from dynamic *data* flows (*Autogram*; [33]) and learning from dynamic *control* flow (*Mimid*; our approach) on the *same* set of samples and generated a grammar from each approach for each program. This grammar was then used to generate inputs to fuzz the program using the *GrammarFuzzer* [50]. The number of inputs that were accepted by the subject program is given in Table 1 .

> *Grammars inferred from dynamic control flow* **produce** *more correct inputs than dynamic data flow.*

To assess and understand the differences in results between learning from data flow and control flow, we manually examined the grammars produced where a large difference was visible. As an implementation for learning from dynamic data flows was only available for Python, we only look at these subjects in detail.

*6.2.1* ***calc.py****.* A snippet of the grammar learned from dynamic data flow for *calc.py* is given below.

⟨*START*⟩ ::= ⟨*init@884:self*⟩

⟨*init@884:self*⟩ ::= ⟨*expr@26:c*⟩'0''0'
  | ⟨*expr@26:c*⟩'3'⟨*expr@26:c*⟩⟨*expr@29:num*⟩
    ⟨*expr@26:c*⟩'*'⟨*expr@29:num*⟩'*''4'

```
|  ⟨expr@26:c⟩'1' ⟨expr@26:c⟩⟨expr@26:c⟩'0' ⟨expr@26:c⟩'2'
|  ⟨expr@26:c⟩'1''0''0') ...
```

The grammar from our approach (dynamic control flow) for the same program using same mining sample is given in Figure 4. An examination shows that the rules derived from dynamic data flow were not as general as ours. That is, the grammar generated from dynamic data flow is enumerative at the expense of generality. Why does this happen? The reason is that parse_expr() and other functions in *calc.py* accept a *buffer* of input characters, with an index specifying the current parse location. learning from dynamic data flow relies on fragmented values being passed into method calls for successful extraction of parts and hence the derivation of tree structure. Here, the first call creates a linear tree with each nested method claiming the entirety of the buffer, and this defeats the learning algorithm.

This is not a matter of better implementation. The original *Autogram* implementation [33] relies on parameters to the method calls to contain only parts of the input. While any algorithm learning from dynamic data flows may choose to ignore the method parameters, any use of a similar buffer inside the method will cause the algorithm to fail unless it is able to identify such spurious variables.

*6.2.2* **mathexpr.py**. For *mathexpr.py*, the situation was again similar. Learning from dynamic data flow was unable to abstract any rules. The *mathexpr.py* program uses a variation of the strategy used by *calc.py* for parsing. It stores the input in an internal buffer in the class and stores the index to the buffer as the location being currently parsed. For similar reasons as before, learning from dynamic data flows fails to extract the parts of the buffer. Our approach of learning from dynamic control flow, on the other hand, produced an almost correct grammar, correctly identifying constants and external variables. The single mistake found (which was the cause of multiple invalid outputs) was instructive. The *mathexpr.py* program pre-defines letters from a to z as constants. Further, it also defines functions such as exp(). The function names are checked in the same place as the constants are parsed. *Mimid* found that the function names are composed of *letters*, and some of the letters in the function names are compatible with the single letter variables—they can be exchanged and still produce correct values. Since we assumed *transitivity*, *Mimid* assumed that all letters in function names are compatible with single letter constants. This assumption produced function names such as *eep()*, which failed the input validation.

*6.2.3* **microjson.py**. The *microjson.py* grammar inferred from dynamic data flow produces more than 50% valid inputs when compared to 98.2% from *Mimid*. Further, we note that the 50% valid inputs paints a more robust picture than the actual situation. That is, the grammar recovered from dynamic data flows for *microjson.py* is mostly an *enumeration* of the values seen during mining. The reason is that *microjson.py* uses a data structure JStream which internally contains a StringIO buffer of data. This data structure is passed as method parameters for all method calls, e.g. _from_json_string(stm). Hence, every call to the data structure gets the complete buffer with no chance of breaking it apart. We note that it is possible to work around this problem by essentially ignoring method parameters and focusing more on return values.

The problem with the data structure can also be worked around by modifying the data structure to hold only the remaining data to be parsed. This however, requires some specific knowledge of the program being analyzed. Learning from dynamic control flow with *Mimid*, on the other hand, is not affected by the problem of buffers at all and recovers a complete grammar.

*6.2.4* **urlparse.py**. For *urlparse.py*, neither grammars learned from data flow nor those learned from control flows performed well (the inferred grammar could recognize less than 10% of the samples) due to the inability to generalize strings. Since we were interested in comparing the capabilities of both algorithms in detecting a structure, we restricted our mining sample to only contain a specific set of strings. The *urlparse.py* program splits the given input to ⟨*scheme*⟩, ⟨*netloc*⟩, ⟨*query*⟩, and ⟨*fragment*⟩ based on delimiters. In particular, the internal structure of ⟨*netloc*⟩ and ⟨*query*⟩ were ignored by the *urlparse.py*.

Hence, we wrote a grammar for *urlparse.py* which contained a list of specific strings for each part. Next, we generated 100 inputs each using the Grammar Fuzzer and validated each by checking the string with the program. We then used these strings as a mining set. With this mining set, the grammars learned from both data flow and control flow could produce 100% correct inputs.

*6.2.5* **parseclisp.py**. Similar to other subjects, the grammar generated by *Autogram* was enumerative which resulted in 100% generated inputs accepted by the program (only previously seen inputs could be generated) while resulting in much lower inputs produced by the golden grammar being accepted by the recovered grammar. The grammar recovered by *Mimid* on the other hand could parse 80.6% of the inputs produced by the golden grammar.

*6.2.6* **jsonparser.c**. The program *jsonparser.c* is in *C*, and hence there is no evaluation using *Autogram*. However, we note that grammar recovered using *Mimid* had 100% accuracy in producing valid inputs when used for fuzzing.

*6.2.7* **tiny.c**. The program *tiny.c* is also in *C*, and hence there is no evaluation using *Autogram*. However, like for *jsonparser.c*, the grammar recovered using *Mimid* could produce 100% valid inputs when fuzzing.

*6.2.8* **mjs.c**. The program *tiny.c* is also a *C* subject with no evaluation possible using *Autogram*. We found that the inputs produced by the grammar recovered using *Mimid* were almost always valid (95.4%).

## 6.3  RQ 2. Grammar Accuracy as Parser

For our second question, we want to assess whether correct inputs would also be accepted by our inferred grammars. In order to obtain correct inputs, we used various approaches as available for different grammars. For *calc.py* and *mathexpr.py*, we wrote a grammar by hand. Next, we used this grammar to generate a set of inputs that were then run through the subject programs to check whether the inputs were valid. We collected 1,000 such inputs for both programs. Next, these inputs were fed into parsers using grammars mined from dynamic control flow (*Mimid*) and dynamic data flow (*Autogram*). We used the *Iterative Earley Parser* from [52] for verifying that the inputs were parsed by the given grammar.

**Table 2: Inputs generated by a golden grammar that were accepted by the inferred grammar parser (1,000 inputs each except *microjson.py* which used 100 external inputs)**

| | from data flows | from control flows |
|:---:|:---:|:---:|
| | (state of the art) | (our approach) |
| *calc.py* | 0.0% | 100.0% |
| *mathexpr.py* | 0.0% | 92.7% |
| *cgidecode.py* | 35.1% | 100.0% |
| *urlparse.py* | 100.0% | 96.4% |
| *microjson.py* | 0.0% | 93.0% |
| *parseclisp.py* | 37.6% | 80.6% |
| *jsonparser.c* | n/a | 83.8% |
| *tiny.c* | n/a | 92.8% |
| *mjs.c* | n/a | 95.9% |

For *urlparse.py*, we used the same grammar for parsing that we already had used to generate mining inputs. We again collected a set of valid inputs and verified that the inferred grammar is able to parse these inputs. For *microjson.py*, we used the collected JSON documents as described above. The largest document was 2,840 lines long. We then verified whether the grammar inferred by each algorithm could parse these inputs. Our results are given in Table 2. As one would expect, grammars from data flow cannot parse the expressions from *calc.py* and *mathexpr.py* grammars. For *cgidecode.py*, grammars from data flow performed poorly, while grammars from dynamic control flow (*Mimid*) achieved 100% accuracy. As we expected, grammars from dynamic control flow performed better for *microjson.py* too, with more than 90% of the input samples recognized by the inferred grammar.

> *Grammars inferred from dynamic control flow **accept** more correct inputs than those from dynamic data flow.*

The outlier is *urlparse.py*, for which grammars from dynamic data flow achieved 100% while grammars from dynamic control flow (*Mimid*) performed slightly worse (but still more than 90% input strings recognized by the inferred grammar). An inspection of the source code of the subject program reveals that it violated one of the assumptions of *Mimid*. Namely, *urlparse.py* searches for character strings in the entirety of its input rather than restricting searches to unparsed parts of the program. For example, it searches for URL fragments (delimited by #) starting from the first location in the input. When this happens, *Mimid* has no way to tell these spurious accesses apart from the true parsing.

We have no comparisons from *Autogram* for the *C* subjects. However, the grammar recovered by *Mimid* for each of the *C* subjects could parse almost all inputs produced by the golden grammar — *jsonparser.c* (83.8%), *tiny.c* (92.8%), and *mjs.c* (95.9%).

## 6.4 RQ 3. Modern Parsing Techniques

While the large majority of parsers is written by hand in the traditional recursive descent approach [41], another parsing technique has become popular recently. *Parser combinators* [19] are recommended over parser generators due to the various inflexibilities such as handling ambiguities, context-sensitive features [38], and bad

error messages [35] [7] when using parser generators. Indeed, parser combinators are often recommended [17] as recognizers where no usable dataflow to unique variables exist. Combinatory parsers are parsers that are built from small primitive parsers that recognize a single literal at a time. These are combined using two operations — sequencing (AndThen), and alternation (OrElse). Given that these parsers follow a fixed recipe for combinations, recovering the grammar from a representative combinatory parser that uses primitive parsers as well as the two operations is sufficient to show that *Mimid* can recover grammars from any such combinatory parsers however complex. Hence, to verify that our technique can indeed recover the grammar from parsers written using the *combinatory parsing* approach, we include a representative combinatory parsing recognizer (*parseclisp.py*) in our subjects.

As our results in Table 1 and Table 2 show, our technique is not challenged by this kind of parsers.

> *Mimid is not challenged by combinatory parsers.*

## 7 LIMITATIONS

Our work is subject to the following important limitations.

**Approximation.** An interpreter for a programming language or a data format for specific applications often have additional restrictions beyond the initial parsing stage. We only try to recover the grammar from the initial parser, and hence, the grammar recovered is an approximation and captures only the syntactic part not the semantic rules. We assume that input is *parsed* as long as it is subject to "syntactic" character and string comparisons and storage only; and *processed* as other "semantic" operations such as arithmetics are being applied.

**Table-driven parsers.** In *table-driven* parsers, control flow and stack are not explicitly encoded into the program, but an implicit part of the parser state. We do not attempt grammar recovery from table driven parsers with *Mimid*.

**Sample inputs.** The features of grammars produced by *Mimid* reflect the features of the inputs it is provided with: If a feature is not present in the input set, it will not be present in the resulting grammar either. New test generators specifically targeting input processors [41] could be able to create such input sets automatically.

**Reparsing.** Since *Mimid* tracks only the *last* access of a character, it can get confused if an ad hoc parser reparses a previously parsed input. This problem can be addressed by exploring multiple candidates for consumption and comparing the resulting grammar structure.

## 8 RELATED WORK

Learning the input language of a given program is an established line of research. There is a large body of work [22, 30, 31] and a community [2] devoted to learning grammars through black-box

---

[7] In the words of a commenter (https://news.ycombinator.com/item?id=18400717) "getting a reasonable error message out of YACC style parser generators is as fun as poking yourself in the eye with a sharp stick". GCC (http://gcc.gnu.org/wiki/New_C_Parser), and CLANG (http://clang.llvm.org/features.html#unifiedparser) use handwritten parsers for the same reason.

approaches. However, as noted in Section 1, there are fundamental limits [11] to this technique, which also apply to statistical approaches such as Learn&Fuzz [28], PULSAR [26], *Neural byte sieve* [45], and *NEUZZ* [24]. We thus focus on gray and white-box techniques.

## 8.1 Learning Context-Free Grammars

*Autogram* [33] is the approach closest to ours. *Autogram* uses the program code in a *dynamic*, *white-box* fashion. Given a program and a set of inputs, *Autogram* uses *dynamic taints* to identify the *data flow* from the input to string fragments found during execution. These entities are associated with corresponding method calls in the call tree, and each entity is assigned an *input interval* that specifies start and end indices of the string found in that entity during execution. Using a subsumption relation, these intervals are collated into a parse tree; the grammar for that parse tree can be recovered by recursively descending into the tree. While learning from dynamic data flows can produce very readable and usable grammars, its success depends on having a data flow to track. If parts of the input are *not* stored in some variable, there is no data flow to learn from. If the parser skips parts of the input (say, to scan over a comment), this will not result in a data flow. Conversely, data can flow into *multiple variables,* causing another set of problems. If a parser uses multiple functions, whose parameters are a buffer pointer and an index into the buffer, then each of these functions gets the entire buffer as a data flow. Such programming idioms may be less frequent in Java (the subjects *Autogram* aims at), but in general would require expensive and difficult disambiguation.

In contrast, our approach tracks *all* accesses of individual characters, no matter whether they would be stored. Our assumption that the last function accessing a character is the *consumer* of this character (and hence parsing a *nonterminal*) still produces very readable and accurate grammars.

## 8.2 Recovering Parse Trees

Lin et al. [39, 40] show how to recover parse trees from inputs using a combination of static and dynamic analysis. They observe that the structure of the input is induced by the way its parts are used during execution, and provide two approaches for recovering bottom-up and top-down parse trees. Similar to our approach, they construct a call tree which contains the method calls, and crucially, the conditionals and individual loop iterations. Next, they identify which nodes in the call tree consume which characters in the input string. Their key idea is a *parsing point* where they consider a particular character to have been consumed. The parsing point of a character is the last point that the character was *used* before the parsing point of its successor. A character is used when a value *derived from it* is accessed — that is, the input labels are propagated through variable assignments much like taints.

A problem with this approach is that it *only* considers well written parsers in the text book style that consumes characters one by one before the next character is parsed. Unfortunately, in real world handwritten parsers, one cannot always have this guarantee. For example, the Python URL parser first checks if a given URL contains any fragment (indicated by the delimiter #), and if there is, the fragment is split from the URL. Next, in the remaining *prefix*,

the query string is checked, which is indicated by the delimiter ?, which is then separated out from the path. Finally, the parameters that are encoded in the path using ; are parsed from the path left over from the above steps. This kind of processing is by no means rare. A common pattern is to split the input string into fields using delimiters such as commas and then parse the individual fields. All this means that the parsing points by Lin et al. will occur much before the actual parse. Lin et al. note that one cannot simply use the *last use* of a label as its parsing point because the values derived from it may be accessed after the parsing phase.

*Mimid* uses the same *last use* strategy, but gets around this problem by only tracking access to the original input buffer. *Mimid* stops tracking as soon as the input is transformed, which makes the *Mimid* instrumentation lightweight, and grammars accurate.

Finally, Lin et al. stop at parse trees. While they show how the function names can form the *nonterminal* symbols, their approach stops at identifying control flow nodes and makes no attempt to either identify compatible nodes or the iteration order, or to recover a grammar which needs something similar to the *prefix tree acceptor* algorithm to generalize over multiple trees, each of which is needed to *accurately label* the parse tree.

## 8.3 Testing with Grammar-Like Structures

*GRIMOIRE* by Blazytko et al. [15] is an end-to-end grey-box fuzzer that uses the new coverage obtained by inputs to synthesize a *grammar like structure* while fuzzing. There are two major shortcomings with the grammar like structures generated by *GRIMOIRE*. First, according to the authors [15, Section 3, last paragraph], the grammar like structure contains a flat hierarchy and contains a single *nonterminal* denoted by □. This *nonterminal* can be expanded to any of the "production rules" which are input fragments with the same *nonterminal* □ inserted in them, producing gaps that can be filled in. Real world applications, however, often have multiple nestings, where only a particular kind of items can be inserted—e.g numbers, strings, etc. These kinds of structures cannot be represented by the grammar like structure without loss of accuracy. Second, as the grammar structure derived by *GRIMOIRE* is essentially a long list of templates, the grammar is likely to be uninterpretable by humans.

Other tools that infer grammar like structures during test generation include *GLADE* [14] and *REINAM* [48] (which is based on *GLADE*), both of which are black-box approaches, and thus subject to the constraints of black-box approaches. Bastani et al. provide a proof that a parenthesis (Dyck) language can be inferred by the *GLADE* algorithm in $O(n^4)$ time in terms of the seed length. This, however, can not be generalized to the general context-free class [11].

Neither *GRIMOIRE*, *GLADE*, or *REINAM* allow to export the inferred input structures, as they focus on test generation rather than grammar extraction.

## 8.4 Learning Finite State Models

The idea of using dynamic traces for inferring models of the underlying software goes back to Hungar et al.[34], learning a finite state model representation of a program; Walkingshaw et al. [47] later refined this approach using queries. Such models represent legal sequences of (input) *events* and thus correspond to the input

language of the program. While our approach could also be applied to event sequences rather than character sequences, it focuses on recovering syntactic (context-free) input structures. Another related work by Bafante et al. [16] uses a method similar to ours to group similar nodes during automata approximation.

## 8.5 Domain-Specific Approaches

Polyglot [18] and Prospex [20] reverse engineer network protocols. They track how the program under test accesses its input data, recovering fixed-length fields, direction fields, and separators. Tupni [21] from Cui et al. uses similar techniques to reverse engineer binary file formats; for instance, element sequences are identified from loops that process an unbounded sequence of elements. AuthScan [13] from Bai et al. uses source code analysis to extract web authentication protocols from implementations. None of these generalizes to recursive input structures.

## 9 CONCLUSION AND FUTURE WORK

Many activities of software engineering benefit from formal input models for a program. However, formal input models are seldom available, and even then, the model can be incomplete, obsolete, or inaccurate with respect to the actual implementation. Inferring input grammars from dynamic control flow, as introduced in this paper, produces readable grammars that accurately describe input syntax. Improving over the state of the art, which uses data flow to identify grammars, our approach can infer grammars even in the absence of data flow and does not require heuristics for common parsing patterns. As we show in our evaluation, our approach is superior to the state of the art both in precision and recall and is applicable to a wide range of parsers, up to input languages such as JSON, JavaScript, or TinyC whose complexity far exceeds the previous state of the art.

The complete code of our approach, including subjects and experimental data, is available as a self-contained Jupyter notebook:

https://github.com/vrthra/mimid

## REFERENCES

[1] [n.d.]. The Codeproject. https://www.codeproject.com/Articles/88435/Simple-Guide-to-Mathematical-Expression-Parsing.
[2] [n.d.]. Home of the International Community interested in Grammatical Inference. https://grammarlearning.org.
[3] [n.d.]. JSON Parser. https://github.com/HarryDC/JsonParser.
[4] [n.d.]. Lisp Parser. https://hackage.haskell.org/package/lispparser.
[5] [n.d.]. Mathematical Expressions Parser. https://github.com/louisfisch/mathematical-expressions-parser.
[6] [n.d.]. Microjson – a minimal JSON parser. https://github.com/phensley/microjson.
[7] [n.d.]. MJS. https://github.com/cesanta/mjs.
[8] [n.d.]. TinyC. https://github.com/TinyCC/TinyCC.
[9] [n.d.]. URL Lib Parser. https://github.com/python/cpython/blob/3.6/Lib/urllib/parse.py.
[10] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. Inf. Comput. 75, 2 (Nov. 1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6
[11] Dana Angluin and Michael Kharitonov. 1995. When Won't Membership Queries Help? J. Comput. System Sci. 50, 2 (1995), 336–355.
[12] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In The Network and Distributed System Security Symposium.
[13] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. 2013. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In The Network and Distributed System Security Symposium. The Internet Society.

[14] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, New York, NY, USA, 95–110.
[15] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In 28th USENIX Security Symposium (USENIX Security 19). 1985–2002. https://www.syssec.ruhr-uni-bochum.de/media/emma/veroeffentlichungen/2019/06/03/grimoire.pdf
[16] Guillaume Bonfante, Jean-Yves Marion, and Thanh Dinh Ta. 2014. Malware Message Classification by Dynamic Analysis. In Foundations and Practice of Security - 7th International Symposium, FPS 2014, Montreal, QC, Canada, November 3-5, 2014. Revised Selected Papers (Lecture Notes in Computer Science), Frédéric Cuppens, Joaquín García-Alfaro, A. Nur Zincir-Heywood, and Philip W. L. Fong (Eds.), Vol. 8930. Springer, 112–128. https://doi.org/10.1007/978-3-319-17040-4_8
[17] Sergey Bratus, Lars Hermerschmidt, Sven M. Hallberg, Michael E. Locasto, Falcon Momot, Meredith L. Patterson, and Anna Shubina. 2017. Curing the Vulnerable Parser: Design Patterns for Secure Input Handling. ;login: 42 (2017).
[18] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In ACM Conference on Computer and Communications Security. ACM, New York, NY, USA, 317–329. https://doi.org/10.1145/1315245.1315286
[19] Pierre Chifflier and Geoffroy Couprie. 2017. Writing parsers like it is 2017. In 2017 IEEE Security and Privacy Workshops (SPW). IEEE, 80–92.
[20] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. 2009. Prospex: Protocol Specification Extraction. In IEEE Symposium on Security and Privacy. IEEE Computer Society, Washington, DC, USA, 110–125. https://doi.org/10.1109/SP.2009.14
[21] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. 2008. Tupni: Automatic Reverse Engineering of Input Formats. In ACM Conference on Computer and Communications Security. ACM, New York, NY, USA, 391–402.
[22] Colin De la Higuera. 2010. Grammatical inference: learning automata and grammars. Cambridge University Press.
[23] Jean-Christophe Deprez and Arun Lakhotia. 2000. A Formalism to Automate Mapping from Program Features to Code.. In IWPC. 69–78.
[24] Kexin Pei Dongdong Shi. 2019. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. IEEE S&P (Jan. 2019). http://par.nsf.gov/biblio/10097303
[25] Henning Fernau. 2009. Algorithms for learning regular expressions from positive data. Information and Computation 207, 4 (2009), 521–541.
[26] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In International Conference on Security and Privacy in Communication Systems. Springer, 330–347.
[27] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, New York, NY, USA, 206–215.
[28] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017). IEEE Press, Piscataway, NJ, USA, 50–59. http://dl.acm.org/citation.cfm?id=3155562.3155573
[29] Benedikt Hauptmann, Elmar Juergens, and Volkmar Woinke. 2015. Generating refactoring proposals to remove clones from automated system tests. In Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension. IEEE Press, 115–124.
[30] Jeffrey Heinz, Colin De la Higuera, and Menno Van Zaanen. 2015. Grammatical inference for computational linguistics. Synthesis Lectures on Human Language Technologies 8, 4 (2015), 1–139.
[31] Jeffrey Heinz and José M Sempere. 2016. Topics in grammatical inference. Vol. 465. Springer.
[32] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In Proceedings of the 21st USENIX Conference on Security Symposium (Security'12). USENIX Association, Berkeley, CA, USA, 38–38. https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final73.pdf
[33] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In IEEE/ACM International Conference on Automated Software Engineering. ACM, New York, NY, USA, 720–725.
[34] H. Hungar, T. Margaria, and B. Steffen. 2003. Test-based model generation for legacy systems. In International Test Conference, 2003. Proceedings. ITC 2003., Vol. 2. 150–159 Vol.2. https://doi.org/10.1109/TEST.2003.1271205
[35] Clinton L Jeffery. 2003. Generating LR syntax error messages from examples. ACM Transactions on Programming Languages and Systems (TOPLAS) 25, 5 (2003), 631–640.
[36] Trevor Jim and Yitzhak Mandelbaum. 2010. Efficient Earley parsing with regular right-hand sides. Electronic Notes in Theoretical Computer Science 253, 7 (2010), 135–148.
[37] Lukas Kirschner, Ezekiel O. Soremekun, and Andreas Zeller. 2020. Debugging Inputs. In International Conference on Software Engineering. ACM.
[38] Nicolas Laurent and Kim Mens. 2016. Taming context-sensitive languages with principled stateful parsing. In Proceedings of the 2016 ACM SIGPLAN International

Conference on Software Language Engineering. ACM, 15–27.

[39] Zhiqiang Lin and Xiangyu Zhang. 2008. Deriving Input Syntactic Structure from Execution. In *ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, New York, NY, USA, 83–93.

[40] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Reverse Engineering Input Syntactic Structure from Program Execution and Its Applications. *IEEE Transactions on Software Engineering* 36, 5 (Sept. 2010), 688–703. https://doi.org/10.1109/TSE.2009.54

[41] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Mathias Höschele, and Andreas Zeller. 2019. Parser Directed Fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA.

[42] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical Delta Debugging. *Proceedings - International Conference on Software Engineering* 2006, 142–151. https://doi.org/10.1145/1134307

[43] F. Momot, S. Bratus, S. M. Hallberg, and M. L. Patterson. 2016. The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them. In *2016 IEEE Cybersecurity Development (SecDev)*. 45–52. https://doi.org/10.1109/SecDev.2016.019

[44] Václav Rajlich and Norman Wilde. 2002. The role of concepts in program comprehension. In *Proceedings 10th International Workshop on Program Comprehension*. IEEE, 271–278.

[45] Mohit Rajpal, William Blum, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. *CoRR* abs/1711.04596 (2017). arXiv:1711.04596

http://arxiv.org/abs/1711.04596

[46] Andrew Stevenson and James R. Cordy. 2014. A Survey of Grammatical Inference in Software Engineering. *Science of Computer Programming* 96, P4 (Dec. 2014), 444–459. https://doi.org/10.1016/j.scico.2014.05.008

[47] Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. 2007. Reverse Engineering State Machines by Interactive Grammar Inference. In *Working Conference on Reverse Engineering*. IEEE Computer Society, Washington, DC, USA, 209–218. https://doi.org/10.1109/WCRE.2007.45

[48] Zhengkai Wu, Evan Johnson, Wei Yang, Osbert Bastani, Dawn Song, Jian Peng, and Tao Xie. 2019. REINAM: reinforcement learning for input-grammar inference. In *ACM SIGSOFT Symposium on The Foundations of Software Engineering*. 488–498. https://doi.org/10.1145/3338906.3338958

[49] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. Code Coverage. In *Generating Software Tests*. Saarland University. https://www.fuzzingbook.org/html/Coverage.html

[50] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. Efficient Grammar Fuzzing. In *Generating Software Tests*. Saarland University. https://www.fuzzingbook.org/html/GrammarFuzzer.html

[51] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. Mining Input Grammars. In *Generating Software Tests*. Saarland University. https://www.fuzzingbook.org/html/GrammarFuzzer.html

[52] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. Parsing Inputs. In *Generating Software Tests*. Saarland University. https://www.fuzzingbook.org/html/GrammarFuzzer.html