“Synthesizing Input Grammars”: A Replication Study

Bachir Bendrissou
CISPA Helmholtz Center For Information Security
Germany
bachir.bendrissou@cispa.de

Rahul Gopinath
CISPA Helmholtz Center For Information Security
Germany
rahul.gopinath@cispa.de

Andreas Zeller
CISPA Helmholtz Center For Information Security
Germany
zeller@cispa.de

Abstract

When producing test inputs for a program, test generators (“fuzzers”) can greatly profit from grammars that formally describe the language of expected inputs. In recent years, researchers thus have studied means to *recover* input grammars from programs and their executions. The GLADE algorithm by Bastani et al., published at PLDI 2017, was the first black-box approach to claim context-free approximation of input specification for non-trivial languages such as XML, Lisp, URLs, and more.

Prompted by recent observations that the GLADE algorithm may show lower performance than reported in the original paper, we have reimplemented the GLADE algorithm from scratch. Our evaluation confirms that the effectiveness score (F1) reported in the GLADE paper is overly optimistic, and in some cases, based on the wrong language. Furthermore, GLADE fares poorly in several real-world languages evaluated, producing grammars that spend megabytes to enumerate inputs.

CCS Concepts: • **Security and privacy** → Software reverse engineering; • **Software and its engineering** → *Software maintenance tools*; *Parsers.*

Keywords: context-free grammar, inference, GLADE

ACM Reference Format:
Bachir Bendrissou, Rahul Gopinath, and Andreas Zeller. 2022. “Synthesizing Input Grammars”: A Replication Study. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI ’22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3519939.3523716

PLDI ’22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523716

1 Introduction

Generating test inputs for a program (“fuzzing”) is much more effective if the fuzzer knows the *input language* of the program under test—that is, the set of valid inputs that actually leads to deeper functionality in the program. Input languages are typically characterized by *context-free grammars,* and the recent interest in fuzzing thus has fueled research in recovering *input grammars* from existing programs.

The GLADE algorithm by Bastani et al., published in “Synthesizing Input Grammars” at PLDI 2017 [6], automatically approximates an input grammar from a given program. In contrast to other approaches, GLADE does not make use of program code to infer input properties. Instead, it relies on feedback from the program whether a given input is valid or not, and synthesizes a multitude of trial inputs to infer the input grammar. GLADE claims substantial improvement over existing algorithms both in terms of accuracy as well as in terms of speed of inference. In particular, GLADE claims better performance over even the current best regular language inference techniques such as L-Star [4] and RPNI [20]. Further, GLADE claims to be able to recover the input grammar for complex languages such as Ruby, Python, and JavaScript in a couple of hours[6, Figure 6.].

In recent work [18], however, Kulkarni, Lemieux, and Sen found that the F1 scores—a measure for the accuracy of the inferred grammar—produced by the GLADE tool were much lower than the scores reported in the GLADE paper, for instance *XML* (0.42 compared to 0.98) and *Lisp* (0.38 compared to 0.97) [18, Table I].

This observation prompted us to investigate the GLADE algorithm in detail. Given that the algorithm reported in the paper is the central contribution, we reimplemented the GLADE algorithm completely from scratch using the algorithm description given in the paper [6].

We call the implementation by Bastani et al. GLADE-I[1], and we call our implementation GLADE-II to differentiate both where there is ambiguity. We used our implementation to evaluate synthesized grammars for programs given in the original paper [6]. These include URL, Grep, Lisp, and XML. We further evaluated GLADE-II on several other small grammars such as different parenthesis grammars, Ints, Decimals, and a few real-world complex grammars such as Lua,

[1] Available in GitHub [2].

MySQL, Pascal, XPath, C, TinyC, Tiny, and Basic. Our evaluation uncovers a number of problems and limitations, which we summarize as follows.

1. The F1 score that we obtained from GLADE-II is much lower than the F1 scores reported by Bastani et al. This confirms the observation by Kulkarni et al. [18].
2. Bastani et al. use handwritten grammars for computing precision and recall. We found that the handwritten *Grep* grammar was far more permissive than the program, resulting in spurious results.
3. The precision scores of simple real-world grammars such as Decimals (0.84), and JSON (0.53) is lower than expected considering the high values reported by GLADE for other programs, and considering their simplicity.
4. The recall of JSON (0.79) is lower than expected considering the simplicity of its specification.
5. Similar to *Grep*, the *XML* grammar used by GLADE-I was more permissive than the actual XML specification.
6. GLADE is unable to learn and synthesize valid XML even when a correct XML grammar is used to learn from.
7. GLADE cannot learn trivial context-free languages such as $a^n b^n$ or the language of palindromes.
8. The synthesized grammars are extremely large, often megabytes in size that enumerate inputs.

Our implementation GLADE-II and all experiments are available online for inspection and replication.

2 Background

Formal specifications for input formats have a long tradition in computer science. Beyond specifying input languages and parsing inputs, grammars have been used for program input generation [15], reverse engineering [7], program refactoring [14], program comprehension [9, 21], and many more. The potential of grammars for producing syntactically valid inputs during test generation and fuzzing has raised interest in methods that *recover input grammars* from programs and/or given inputs.

An early result in grammar inference was the discovery by Gold [10] that learning an accurate input specification from exclusively positive examples was impossible, even when the specification complexity was limited to regular languages; even when negative examples are given, it is NP-hard [11].

Consequently, current practical approaches to inferring context-free grammars inference all make use of *program executions.* "Whitebox" approaches analyze code and dynamic control and data flow to extract compact input grammars that follow the structure of input processing [12, 16, 19]. "Blackbox" approaches, in contrast, extract grammars from *membership queries,* executing the program only to determine if an input is valid or not. Clark's algorithm [8] uses a *minimally adequate teacher* (which can be simulated by membership queries [4]) to learn a subclass of context-free grammars. More recent "blackbox" approaches include GLADE by Bastani et al. [6] and Arvada by Kulkarni et al. [18], both set to learn *general* context-free grammars.

However, all "blackbox" approaches for learning general context-free grammars are fundamentally limited: In 1995, Angluin and Kharitonov showed that unless RSA encryption is broken, there is no polynomial time prediction algorithm with query memberships for context-free grammars [5]. To be efficient, "blackbox" approaches thus need algorithms that are tailored towards the features of commonly used input languages.

3 The GLADE Algorithm

GLADE [6] is a grammar inference algorithm. It infers the context-free grammar of a black-box oracle capable of saying *yes* or *no* to membership queries. It also bootstraps itself with a set of positive examples. The GLADE paper implies that it can produce the context-free grammar even if the positive examples given do not cover all "interesting behaviors".

The GLADE algorithm starts with a seed input α_{in}. Such a single seed input (or a set of seed inputs) is a finite choice grammar [17] with high precision (because it will never generate an invalid input) but very low recall. From this, GLADE performs a series of precision-preserving generalization steps that attempts to increase the *recall.* Each step produces more and more general *regular expressions.*

While the algorithm attempts to preserve precision, doing so during transformations is hard. This is because we only have access to a membership oracle, and it is impossible to guarantee that precision is preserved without an infinite number of queries in general. Hence, GLADE uses a series of heuristic checks to ensure that the candidate is *potentially* precision preserving.

The GLADE algorithm has two main phases.

3.1 Phase I: Regular Expression Synthesis

In the first phase, the idea is to synthesize a representative *regular expression.* The algorithm first attempts to generalize substrings of the seed as repetition (*rep*) or alternation (*alt*).

The seed input α_{in} is first annotated as $[\alpha_{in}]_{rep}$. Then the following rules are followed to successively generalize the internal substrings.

- Given any partly annotated string $P[\alpha]_{rep}Q$ such that P is the non annotated prefix, Q is the non annotated suffix, and the in between string α is annotated with *rep*, we first find all decompositions of α of the form $\alpha_1\alpha_2\alpha_3$ where $\alpha 2 \neq \epsilon$. We then generate annotated strings $P\alpha_1([\alpha_2]_{alt}) * [\alpha_3]_{rep}Q$ for every such decomposition of α. These along with the string $P\alpha Q$ becomes candidates for generalization.

- Next, for any annotated string $P[\alpha]_{alt}Q$, for any decomposition of α of the form $\alpha_1\alpha_2$ where neither of the string is empty, generate $P([\alpha_1]_{rep} + [\alpha_2]_{alt})Q$. These along with $P\alpha Q$ becomes generalization candidates.

Shorter α_1 constructions are preferred for generalization, followed by longer α_2 for repetitions. For alternations, shorter α_1 constructions are preferred. The construction $P\alpha Q$ is last.

3.2 Phase II: Infer Recursive Properties

The idea here is to infer recursive properties and transform the expression into a context-free grammar. The regular expression that was synthesized in Phase I is first translated into a context-free grammar. Next, each pair of nonterminals that were synthesized during Phase I corresponding to repetition is equated and checked whether the resulting language represents a valid generalization.

4 Evaluation

For evaluation, we wanted to ensure that the procedure we followed was the same as Bastani et al. except for the new implementation, and using the same grammar for precision and recall. Hence, the first set of subjects are the original four programs used by Bastani et al. for evaluation: URL, Lisp, Grep, and XML. Out of these, we used the URL, and Grep handwritten grammars as given by Bastani et al.[2]. We tried to check the accuracy, but could only evaluate that of the handwritten Grep grammar as this was the only binary available[3]. The XML and Lisp grammars were written as Java programs[4]. Since these are well known standard formats, we used external grammars for these.

Next, we wanted to extend our evaluation to a few simpler grammars so that we can understand the characteristics of the algorithm in detail. Hence, the second set includes a few simple grammars: Ints, Decimals, Floats, and JSON.

We then investigated GLADE behavior on a few parenthesis variants: Palindrome, Paren, Bool Add, TwoParen, TwoParenD, TwoAnyParenD, BinParen, and BinAnyParen.

Finally, we wanted to find the performance of GLADE on real-world complex grammars. Hence, the third set contained ANTLR grammars obtained from the ANTLR repository [3]: Lua, MySQL, Pascal, XPath, C, TinyC, Tiny, and Basic.

For the first, second, and third set of grammars, we produced 50 random inputs using the F1 fuzzer [13]. The random exploration depth was set to 100. For ANTLR grammars, the GLADE algorithm took an extremely large amount of time to learn. Hence, we limited both the seed set and the individual size. That is, for these, we only generated 10 seed inputs with a maximum random exploration depth of 20. Note that GLADE claims not to require seed inputs that exercise all interesting behaviors. For ANTLR grammars, we used Grammarinator [15] as the input generator.

We use the same definitions of precision, recall, and the F1 score. We generate 1000 inputs from the synthesized grammar and check how many of them were recognized by the handwritten grammar for precision (P), and we generate 1000 inputs from the handwritten grammar and check how many were recognized by the synthesized grammar for recall (R). The *F1* score is calculated as $\text{F1} = \frac{2\times\text{P}\times\text{R}}{\text{P+R}}$

During the check for accuracy, we produced inputs from the Grep handwritten grammar and checked how many of these were accepted by the Grep binary. We found that only 33% of inputs were accepted. Given that the Grep grammar is far from the actual input grammar of Grep binary, we use ~~Grep~~ from now on to indicate that it is not the true grammar. Similarly, GLADE-I uses a relaxed definition for XML, allowing any number of root elements compared to the XML specification as claimed in the GLADE paper [6]. since XML is not the true XML grammar, we use ~~XML~~ to indicate that it is not the true grammar either. Unlike Grep, however, we also check whether GLADE can learn the XML grammar with the single root node constraint. We mark this grammar as XML.

We first used GLADE-II to synthesize grammars corresponding to each of these grammars. The details of the GLADE-II run are given in Table 1. *LTime* measures the total amount of time in seconds of learning a grammar. *Seeds Len* refers to the average length of the random seeds used in learning a grammar. σ refers to the standard deviation of the seed lengths. *Checks* lists the number of checks the algorithm performed in the learning process.

Table 2 contains the precision, accuracy, and the corresponding F1 score obtained by GLADE-II on each subject. It also shows the size of the synthesized grammars in Kilobytes. A few cells are empty (_) because the score is unavailable, This is due to the large size of these synthesized grammars which made parsing infeasible in a reasonable amount of time and memory.

Table 3 describes the statistics of handwritten grammars[5]. These were used as the black-box program (grammar+parser) whose input grammar Glade-II was expected to learn.

Table 4 describes the synthesized grammar statistics. The synthesized grammars are much larger and more complex than the actual input grammars of the black-box programs. The GLADE paper [6] does not report grammar sizes.

Why did we not evaluate our subjects with GLADE-I? As we hinted in the introduction, the GLADE-I source is exceedingly entangled, and it is hard to include new programs for evaluation easily.

[2]https://github.com/obastani/glade-full/blob/master/data/handwritten/
[3]https://github.com/obastani/glade-full/tree/master/data/prog
[4]https://github.com/obastani/glade-full/blob/master/src/glade/constants/SyntheticGrammars.java

[5]Only ASCII symbols were considered.

Table 1. Glade-II Execution

	Grammar	LTime (s)	Seeds Len	σ	Checks
Original	URL	193	13.64	3.66	91,885
	Lisp	1,463	13.93	13.20	81,769
	~~XML~~	5,840	16.44	13.15	129,362
	XML	618	15.22	9.64	73,272
	~~Grep~~	1,891	20.24	14.88	99,843
Simple	Ints	1	2.08	1.18	3,216
	Decimals	15	3.72	2.20	21,292
	Floats	16	5	2.45	22,827
	JSON	7,398	24.04	45.78	172,163
Parenthesis	Palindrome	45	11.18	6.50	66,208
	Paren	6,636	42.73	47.54	179,626
	Bool Add	3,917	40.86	63.52	160,017
	TwoParenD	2,976	28.8	45.42	100,696
	TwoAnyParenD	2,469	28.22	36.07	162,034
	BinParen	3,048	33.61	33.89	165,827
	BinAnyParen	30,568	78.02	61.12	513,414
Programming	Tiny	3,935	39.7	15.35	60,574
	Lua	59,006	111.7	135.29	247,296
	Pascal	68,605	182	113.32	317,801
	MySQL	73,261	99.2	129.59	177,153
	XPath	5,866	67.3	41.04	89,341
	C	12,982	110.5	292.31	143,282
	TinyC	53,528	130.8	153.07	268,454
	Basic	6,408	80.7	99.95	114,612

Table 2. Glade-II Scores

	Grammar	Precision	Recall	F1	Size (KB)
Original	URL	0.687	1	0.81	296
	Lisp	0.378	1	0.55	638
	~~XML~~	0.55	0.96	0.7	976
	XML	0.579	0.759	0.66	635
	~~Grep~~	1	1	1	2,000
Simple	Ints	0.983	1	0.99	14
	Decimals	0.848	1	0.92	74
	Floats	0.914	0.984	0.95	71
	JSON	0.531	0.797	0.64	594
Parenthesis	Palindrome	1	0.19	0.32	13
	Paren	1	1	1	280
	Bool Add	1	0.891	0.94	173
	TwoParenD	0.838	1	0.91	143
	TwoAnyParenD	0.831	0.61	0.7	126
	BinParen	0.778	1	0.88	194
	BinAnyParen	0.623	1	0.77	575
Programming	Tiny	0.213	1	0.35	281
	Lua	0.203	_	_	1,400
	MySQL	0.331	_	_	4,600
	Pascal	0.024	_	_	1,800
	XPath	0.654	_	_	1,800
	C	0.718	_	_	3,700
	TinyC	0.472	_	_	1,300
	Basic	0.365	_	_	577

Table 3. Source Grammars

	Grammar	Non-terminals	Rules	Terminals
Original	URL	13	119	73
	Lisp	12	78	63
	~~XML~~	13	142	65
	XML	12	140	65
	~~Grep~~	12	155	91
Simple	Ints	5	16	10
	Decimals	7	19	11
	Floats	10	27	14
	JSON	27	159	101
Parenthesis	Palindrome	2	6	8
	Paren	3	5	3
	Bool Add	4	7	5
	TwoParenD	3	5	3
	TwoAnyParenD	3	8	7
	BinParen	4	7	3
	BinAnyParen	4	9	7
Programming	Tiny	36	111	73
	Lua	272	1,356	139
	Pascal	604	1,108	109
	MySQL	5,942	9,589	1,167
	XPath	162	623	245
	C	677	2,352	200
	TinyC	28	83	53
	Basic	628	1,174	286

Table 4. Synthesized GLADE-II Grammars

	Grammar	Non-terminals	Rules	Terminals
Original	URL	436	7,604	78
	Lisp	923	15,928	63
	~~XML~~	1,086	24,938	65
	XML	693	16,282	69
	~~Grep~~	993	54,756	91
Simple	Ints	49	297	10
	Decimals	194	1,252	19
	Floats	260	1,524	14
	JSON	1,418	10,727	114
Parenthesis	Palindrome	46	89	44
	Paren	1,635	3,549	3
	Bool Add	1,097	2,224	9
	TwoParenD	842	1,821	5
	TwoAnyParenD	830	1,554	106
	BinParen	1,188	2,625	12
	BinAnyParen	3,514	7,260	120
Programming	Tiny	571	6,652	75
	Lua	1,723	33,647	139
	Pascal	2,975	41,648	134
	MySQL	1,478	120,213	137
	XPath	1,180	46,760	136
	C	1,153	102,499	127
	TinyC	2,294	30,126	162
	Basic	1,298	13,683	135

The experiments were done on a machine with 8 Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz CPUs, with a memory of 16GB. The operating system was Ubuntu.

5 Discussion

There are several limitations with the GLADE paper.

5.1 Dependence on Seeds

In its discussion of relevant research, the GLADE paper [6] claims that some of the other grammar inference techniques rely on positive examples that exercise all interesting behaviors. One can wonder whether this implies that the seeds required by GLADE need not cover all interesting behaviors. In our evaluation, the performance of GLADE is strongly dependent on the features covered by seed inputs. A smaller number of seeds results in lower recall (less variety) but higher precision (less chance of making mistakes).

5.2 Reporting of Results

The F1 score as given in Figure 4 is never explicitly specified in figures. The most important information—precision and recall of the synthesized grammars—are never reported separate from F1.

5.3 Evaluation Results

The F1 score we obtain that is listed in Table 2 is much lower than expected from the GLADE paper. The comparison of scores for the original four programs—URL, Lisp, XML, and Grep—is reported in Table 5.[6] As we mentioned previously, the evaluation of Grep by the GLADE paper is unreliable. The problem is that it uses a handwritten grammar for computing precision, and this handwritten grammar for Grep accepts a much larger language than the actual Grep program. Hence, the reported F1 score is highly inflated.

The precision scores for Decimals (0.85), and JSON (0.53), and Tiny (0.21) are much lower than expected. These grammars were not part of the original GLADE paper but were added to investigate the capabilities of GLADE. While investigating the reason, we found that the merge strategy of GLADE fails to preserve the precision in certain instances. We also found that the number of checks used by GLADE is often insufficient to correctly identify repetition generalization. However, increasing the number of such checks will adversely impact the speed of grammar learning.

5.4 Practicality of the Inferred Grammars

One of the strong claims of GLADE is that the recovered grammar can be immediately used for fuzzing. However, we found that the size of the grammar generated is extremely large. For example, learning Grep resulted in a **2 MB** grammar. The problem with such large grammars is that it is essentially enumerative. It cannot be feasibly used for parsing existing seed files as the parsers we tried to use gave up on such large grammars. Even grammar-based generators tend to have trouble using such large grammars. This is especially noticeable when considering the original grammars. For example, Palindrome resulted in a **13 KB** grammar, while the actual grammar contains a single nonterminal and five rules.

The biggest surprise comes from the parenthesis languages. These are trivial languages with less than five nonterminal symbols. It should be trivial for GLADE to recover their grammar. However, GLADE fares poorly in most both in terms of accuracy (F1) and on the size of the grammars recovered, thousands of nonterminal symbols, and hundreds of kilobytes in size. On inspection, the GLADE recovered grammar was strongly enumerating rather than abstracting.

Table 5. GLADE Learning Accuracy (F1 Score)

	Grammar	Language	GLADE-I F1	Glade-II F1
Original	URL	Regular	0.92	0.81
	Lisp	Context-Free	0.97	0.55
	~~XML~~	Context-Free	0.98	0.7
	XML	Context-Free	_	0.66
	~~Grep~~	Regular	0.93	1.00[7]

5.5 Insights about the GLADE Algorithm

During our implementation of GLADE algorithm and the subsequent evaluation, we found a number of insights about the GLADE algorithm, and why it has problems with some of the languages. These we describe in detail below.

1. The GLADE algorithm cannot learn a valid XML representation. Even in the paper [6], the regular expression synthesized – (<a(><a/><)*/a>)* – does not always produce valid XML inputs as it lacks a root element.[8]
2. The heuristic checks specified by Bastani et al. is insufficient even for XML which drove their design [6, Section 8]. For example, given a seed <ab>, the first generalization is <a*b/> and the second generalization is <a*b*/> . The second generalization is imprecise because we can now construct </>. However, it is accepted because the two required checks (<a/> and <abb/>) pass [6, Section 4.3 Check Construction].
3. Merging can incur loss of precision. Consider, for example, TwoAnyParenD. We start with a seed input [()1], which is generalized to [()*1*] . During Phase II, ()* and 1* are hence checked for unification. To verify the new generalization, GLADE constructs two checks [6, Section 5.3 Check Construction] – [111]

[6]As the actual F1 score was not reported by Bastani et al. [6], we estimated it from the graph [6, Figure 4(b)].

[7]The handwritten grammar used for computing precision is much more permissive than the actual Grep grammar. Hence, the high F1 score.

[8]Bastani et al.[6, Section 7 Limitations] incorrectly claims that it is valid XML subset.

and [()()()]. Since both are valid, the new generalization is accepted. However, the resulting grammar can now produce [1()] which is invalid.

4. In Phase II, only repetitions are considered for unification. These are, however, insufficient in many cases. Consider XML, and seed input hi. GLADE never learns about hi because hi is not a repetition. We found the same issue in Palindrome where the grammar is *exclusively* made up of concrete enumerations.
5. The character generalization [6, Section 5.3 Check Construction] can produce generalizations that do not preserve precision. Consider the Grep grammar. We use a] as a seed input. GLADE now constructs the check – [] – which passes, producing [|a as a generalization for the first index. Next, GLADE constructs the check – aa – which passes, resulting in]|a as a generalization for the second index, and the new generalized language (a|[)(]|a) . However, this language loses precision because it can produce [a which is invalid.

6 Threats to Validity

We acknowledge that our evaluation of the GLADE algorithm is subject to the following threats.

Defects in the implementation. One of the largest threats to our evaluation is the possibility that (1) we misunderstood some parts of the GLADE paper and/or (2) we implemented the algorithm incorrectly. Given that this is a software program, this is a possibility that cannot be completely mitigated. We have tried to reduce the possible bugs as much as possible by carefully documenting our code, reviewing our code multiple times, investigating how simple grammars that exercised each feature of the GLADE algorithm behaved, and investigating a sample of inputs that reduced the precision or recall of the synthesized grammar.

GLADE algorithm vs. tool. For investigating GLADE, we also considered the GLADE tool supplied for replication. However, the GLADE tool does not support extraction or inspection of the inferred grammars, and we found it prohibitively hard to extract the GLADE algorithm code, as it is deeply entangled with code for custom serialization and custom input generation provided with GLADE. However, we note that our implementation achieves better F1 scores than Kulkarni et al. [18] (who used the GLADE tool) for JSON (0.64 > 0.59), XML (0.66 > 0.42), and Lisp (0.55 > 0.38). While the evaluation of Kulkarni achieved higher precision for TinyC (0.47 < 0.60), we note that the recall which is an indication of the amount of actual abstraction is surprisingly low (0.17).

Defects in the input generator. Another threat is that (1) our input generator is faulty (generates invalid strings) (2) or that it is biased (generates a skewed distribution of inputs). We have tried to mitigate it by using off-the-shelf fuzzers such as the F1 fuzzer [13] and Grammarinator [15]. Further, we have checked that the strings that the fuzzers generate are parsed by the same grammar.

Defects in the parser. Another threat is the possibility that our parser may be defective, rejecting valid inputs or accepting invalid inputs. We have mitigated this by using an off-the-shelf well tested textbook parser [22].

7 Questions and Answers

Given that this is a replication study, questions may arise about our procedure. Let us address the most important ones.

1) Why do we not use GLADE-I [2]?

Our focus is to replicate the GLADE paper, not its implementation, as we see the paper as the definite, archived, and cited reference. ACM defines [1] replicability as: "For computational experiments, this means that an independent group can obtain the same result using artifacts which they develop completely independently." Hence, we independently implemented the same algorithm. We note that our effort was prompted by the discovery that some of the GLADE F1 scores failed the *reproducibility* test for XML and Lisp when attempted by other researchers [18] using the same artifacts from GLADE-I (Section 6).

2) Is this a complete replication of the GLADE paper?

We attempt to replicate only what we consider to be the main result of the paper, which is the high accuracy achieved in F1 scores while learning different grammers. In particular, we do not replicate the evaluation of L-Star or RPNI. Secondly, we do not replicate the fuzzing experiments.

3) How do we know that the language $a^n b^n$ could not be learned by GLADE?

Note that $a^n b^n$ can equivalently be defined as:
⟨*S*⟩ ::= '(' <S> ')' | ϵ

We evaluated Palindrome (Figure 1) which is a trivial extension of the language $a^n b^n$. Palindrome is defined as:
⟨*S*⟩ ::= '(' <S> ')' | '[' <S> ']' | '' <S> '' | ϵ

That is, Palindrome contains three pairs of parenthesis rather just one pair. We analyzed different variations of the same language with different pairs, and found that the particular pattern—nesting nonterminals without repetition—is not learnable by GLADE.

4) How representative of real-world grammars are JSON and Decimals?

We argue that both grammars are *representative* and *simple:*

Representativeness. Decimals numbers are a common component in almost all programming languages. JSON is one of the most popular data-interchange formats, similar to XML and Lisp S-Expressions. Hence, we believe that both the Decimals and JSON grammars are representative of the real world.

Simplicity. The Decimals grammar is a *regular grammar* containing only 19 rules, and 7 nonterminals (Table 3). JSON is an LL(1) grammar that is a heavily reduced subset of the actual JSON specification. It contains only 27 nonterminals and 159 rules. We believe that regular grammars containing 11 rules should be considered simple by any definition, and LL(1) grammars are one of the simplest grammar classes under the Context-Free Grammar umbrella.

5) Does your implementation of GLADE include the optimizations from original GLADE implementation?

The only optimization mentioned in the GLADE paper [6] is *multiple-inputs* optimization, which enables GLADE to learn from multiple seed inputs. We have implemented that.

6) Is there a potential for non-determinism in the GLADE learning?

As far as we are aware, there is no potential for non-determinism in the GLADE implementation. We contacted the GLADE authors regarding our implementation. The only advice was to be careful about the order in which the alternatives were tried. We followed their advice and have implemented it exactly as the paper mentions. If there are any avenues of non-determinism influencing the grammar learning by GLADE, the GLADE paper does not mention it.

7) Why do you not evaluate *Learning Highly Recursive Input Grammars* [18] as well?

Our focus is on replication of GLADE. We do not claim that our research is much more than that. Hence, evaluation of *Learning Highly Recursive Input Grammars* is out of scope for this study.

8) How dependent is GLADE on the seed selection?

The GLADE paper uses ambiguous language in this regard. It says that *it can produce the context-free grammar even if the positive examples given do not cover all "interesting behaviors"*. However, the paper does not provide a definition of *behavior*. Hence, we would not know how to validate (or invalidate) this claim.

9) What is going wrong with GLADE and how can it be addressed?

This paper is a pure replication study of the GLADE paper. Hence, a detailed analysis of what is going wrong with GLADE, and how to overcome it is out of scope for this study.

8 Conclusion

Recovering input grammars for existing programs is an important, yet challenging problem. The GLADE algorithm by Bastani et al. is the first published approach that is set to recover general context-free grammars using membership queries alone. Having reimplemented the GLADE algorithm, we find that the accuracy of the inferred context-free grammars is much lower than originally reported, a discrepancy recently also reported for the original GLADE tool [18]. Our investigation details more issues with the GLADE algorithm; notably, we show that its inferred grammars can be extremely large and enumerative, indicating low usability for practical tasks such as parsing or producing inputs with general fuzzers. Prospective users should also evaluate other grammar mining approaches, such as the "blackbox" and "whitebox" approaches listed in Section 2.

Should the GLADE issues have been caught by the PLDI 2017 reviewers? In total, replicating and evaluating GLADE took us more than six person-months; we cannot expect from reviewers to spend all this time checking a paper. We hope, however, that future authors search for and report weaknesses just as they do for strengths, and that future reviewers appreciate honesty just as they appreciate success.

Replication studies are still rare in our field. Indeed, it is much more work to replicate a piece of research, especially from a paper, than to implement a new alternative from scratch (for which one may also get more credits). That extra effort comes from the required quality assurance: Does the reimplementation really exactly reflect the algorithm(s) as stated in the paper? Of course, such quality assurance would be expected from any piece of research; yet, it is the authors of the replication study that would be challenged with such questions, not so much the authors of the original paper. As a community, we need to further encourage replication and reuse of research results—by making tools and data available, usable, understandable, and extensible. Such standards must become the norm, not the exception.

Our annotated reimplementation GLADE-II and all experimental data is available at:

https://doi.org/10.5281/zenodo.6326396

A Appendix

$\langle S\rangle$ ::= '(' $\langle S\rangle$ ')' |'[' $\langle S\rangle$ ']' |'$\langle$' <$S\rangle$ '>' |'{' $\langle S\rangle$ '}' | ϵ

Figure 1. Palindrome

$\langle S\rangle$ ::= $\langle PS\rangle$
$\langle PS\rangle$ ::= $\langle P\rangle\langle PS\rangle$ | $\langle P\rangle$
$\langle P\rangle$ ::= '(' $\langle PS\rangle$ ')' | '(' ')'

Figure 2. Paren

$\langle S\rangle$ ::= $\langle S\rangle$ '+' $\langle S\rangle$ | '(' $\langle S\rangle$ ')' | $\langle D\rangle$
$\langle D\rangle$ ::= 1 | 0

Figure 3. Bool Add

$\langle S\rangle$ ::= '(' $\langle S\rangle$ ')' $\langle S\rangle$ | ϵ

Figure 4. TwoParen

$\langle S\rangle$ ::= '(' $\langle S\rangle$ ')' $\langle S\rangle$ | $\langle D\rangle$
$\langle D\rangle$ ::= 1 | 1$\langle D\rangle$

Figure 5. TwoParenD

$\langle S\rangle$ ::= $\langle D\rangle$
| '(' $\langle S\rangle$ ')' $\langle S\rangle$
| '[' $\langle S\rangle$ ']'
| '{' $\langle S\rangle$ '}'
$\langle D\rangle$ ::= ϵ | 1 | 1$\langle D\rangle$

Figure 6. TwoAnyParenD

$\langle S\rangle$ ::= $\langle PS\rangle$
$\langle PS\rangle$ ::= $\langle P\rangle\langle PS\rangle$ | $\langle P\rangle$
$\langle P\rangle$ ::= '(' $\langle PS\rangle$ ')' | $\langle OS\rangle$
$\langle OS\rangle$ ::= '1' | '1' $\langle OS\rangle$

Figure 7. BinParen

$\langle S\rangle$ ::= $\langle PS\rangle$
$\langle PS\rangle$ ::= $\langle P\rangle\langle PS\rangle$ | $\langle P\rangle$
$\langle P\rangle$ ::= '(' $\langle PS\rangle$ ')' | '[' $\langle PS\rangle$ ']' | '{' $\langle PS\rangle$ '}' | $\langle OS\rangle$
$\langle OS\rangle$ ::= '1' | '1' $\langle OS\rangle$

Figure 8. BinAnyParen

$\langle START\rangle$ ::= $\langle INTEGER\rangle$
$\langle INTEGER\rangle$ ::= $\langle DigitNZ\rangle$ $\langle DigitZs\rangle$ | '0'
$\langle DigitZs\rangle$ ::= ϵ | $\langle DigitZ\rangle\langle DigitZs\rangle$
$\langle DigitZ\rangle$ ::= '0' | $\langle DigitNZ\rangle$
$\langle DigitNZ\rangle$ ::= [1-9]

Figure 9. Integer grammar

$\langle START\rangle$ ::= $\langle DECNUM\rangle$
$\langle DECNUM\rangle$ ::= $\langle INT\rangle$ '.' $\langle DEC\rangle$
$\langle DEC\rangle$ ::= $\langle DigitZs\rangle\langle DigitNZ\rangle$ | '0'
$\langle INT\rangle$ ::= $\langle DigitNZ\rangle\langle DigitZs\rangle$ | '0'
$\langle DigitZs\rangle$ ::= ϵ | $\langle DigitZ\rangle\langle DigitZs\rangle$
$\langle DigitZ\rangle$ ::= '0' | $\langle DigitNZ\rangle$
$\langle DigitNZ\rangle$ ::= [1-9]

Figure 10. Decimal grammar

$\langle START\rangle$ ::= $\langle FLOAT\rangle$
$\langle FLOAT\rangle$ ::= $\langle INT\rangle$ '.' $\langle EXT\rangle$ | '.' $\langle EXT\rangle$ | $\langle INT\rangle$ '.'
$\langle EXT\rangle$ ::= $\langle DEC\rangle$ | $\langle DEC\rangle\langle LETTER\rangle\langle OP\rangle\langle INT\rangle$ | $\langle DEC\rangle\langle LETTER\rangle\langle INT\rangle$
$\langle DEC\rangle$ ::= $\langle DigitZs\rangle\langle DigitNZ\rangle$ | '0'
$\langle INT\rangle$ ::= $\langle DigitNZ\rangle\langle DigitZs\rangle$ | '0'
$\langle OP\rangle$ ::= '-'
$\langle LETTER\rangle$ ::= 'e' | 'E'
$\langle DigitZs\rangle$ ::= ϵ | $\langle DigitZ\rangle\langle DigitZs\rangle$
$\langle DigitNZ\rangle$ ::= [1-9]
$\langle DigitZ\rangle$ ::= '0' | $\langle DigitNZ\rangle$

Figure 11. Float grammar

References

[1] [n. d.]. Artifact Review and Badging – Version 1.1. https://www.acm.org/publications/policies/artifact-review-and-badging-current.

[2] [n. d.]. GLADE Implementation and experiments. https://github.com/obastani/glade-full.

[3] [n. d.]. Grammars written for ANTLR v4. https://github.com/antlr/grammars-v4.

[4] 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

[5] Dana Angluin and Michael Kharitonov. 1995. When Won't Membership Queries Help? *J. Comput. System Sci.* 50, 2 (1995), 336–355. https://doi.org/10.1145/103418.103420

[6] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain). ACM, New York, NY, USA, 95–110. https://doi.org/10.1145/3140587.3062349

[7] 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* (Alexandria, Virginia, USA). ACM, New York, NY, USA, 317–329. https://doi.org/10.1145/1315245.1315286

[8] Alexander Clark. 2010. Distributional learning of some context-free languages with a minimally adequate teacher. In *International Colloquium on Grammatical Inference*. Springer, 24–37. https://doi.org/10.1007/978-3-642-15488-1_4

[9] Jean-Christophe Deprez and Arun Lakhotia. 2000. A Formalism to Automate Mapping from Program Features to Code.. In *IWPC*. 69–78. https://doi.org/10.1109/WPC.2000.852481

[10] E Mark Gold. 1967. Language identification in the limit. *Information and control* 10, 5 (1967), 447–474. https://doi.org/10.1016/S0019-9958(67)91165-5

[11] E Mark Gold. 1978. Complexity of automaton identification from given data. *Information and control* 37, 3 (1978), 302–320. https://doi.org/10.1016/S0019-9958(78)90562-4

[12] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 172–183. https://doi.org/10.1145/3368089.3409679

[13] Rahul Gopinath and Andreas Zeller. 2019. Building Fast Fuzzers. *CoRR* (2019). arXiv:1911.07707 [cs.SE] https://arxiv.org/abs/1911.07707

[14] 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. https://doi.org/10.1109/ICPC.2015.20

[15] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ACM, 45–48. https://doi.org/10.1145/3278186.3278193

[16] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore). ACM, New York, NY, USA, 720–725. https://doi.org/10.1145/2970276.2970321

[17] C Jacobs and D Grune. 1990. *Parsing techniques: A practical guide*. Springer.

[18] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2021. Learning Highly Recursive Input Grammars. In *36th IEEE/ACM International Conference on Automated Software Engineering*. ACM New York, NY, USA. https://doi.org/10.1109/ASE51524.2021.9678879

[19] 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

[20] José Oncina and Pedro Garcia. 1992. Identifying regular languages in polynomial time. In *Advances in structural and syntactic pattern recognition*. World Scientific, 99–108. https://doi.org/10.1142/9789812797919_0007

[21] Václav Rajlich and Norman Wilde. 2002. The role of concepts in program comprehension. In *Proceedings 10th International Workshop on Program Comprehension*. IEEE, 271–278. https://doi.org/10.1109/WPC.2002.1021348

[22] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2021. Parsing Inputs. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/html/Parser.html Retrieved 2021-11-16 14:26:40+01:00.