V-Star: Learning Visibly Pushdown Grammars from Program Inputs

Xiaodong Jia and **Gang Tan**
Penn State University
PLDI 2024

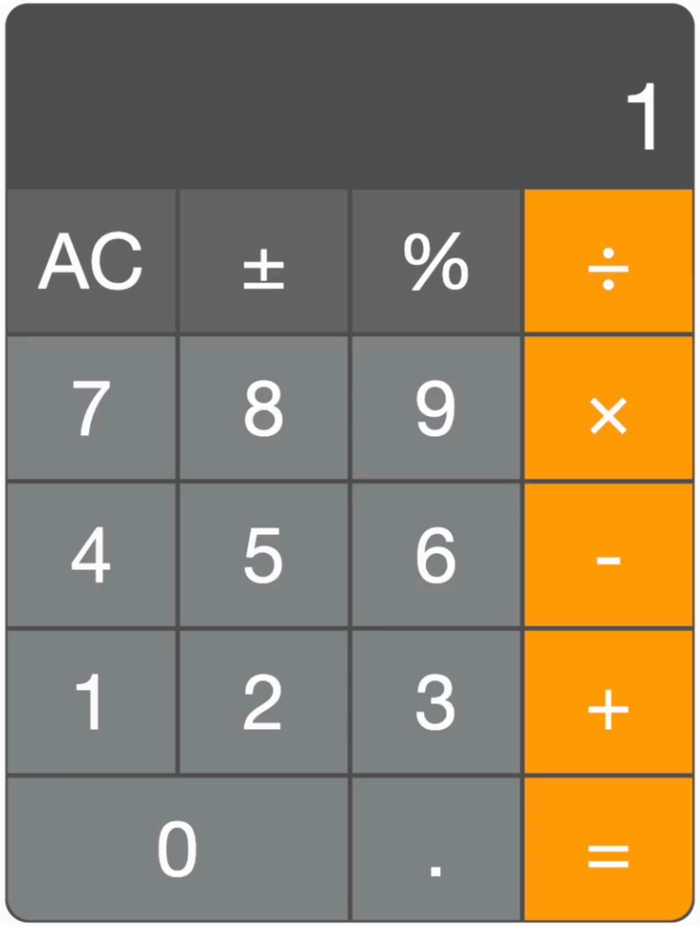
Overview

- Background
- Key Contributions of V-Star
- Running Example
- Evaluation
- Future Work

Program Input Learning

Learn formal grammars of inputs from **black-box programs** and **sample valid inputs**.

- **Oracle**: The black-box program
  - A calculator
  - JSON/XML parser binaries
  - Web server that accepts HTTP requests
- **Seed Strings**: Given valid inputs



Background

Importance

- Widely applicable
- Improves security and robustness

Application

- Grammar-based fuzzing
- Program validation
- Program comprehension
- Reverse engineering

Background

Related Work

- **Existing Tools**
  - Glade (PLDI '17)
  - Arvada (ASE '22)
  - Learn context-free grammars
- **Limitations:**
  - Limited accuracy for many practical grammars
  - Do not fully utilize the *nesting structures* in program inputs

V-Star (PLDI '24)

- **Observation**: Using nesting structures can significantly increase model accuracy.
- **Approach**: Exploits *nesting structures* in program inputs to improve accuracy.

Nesting Structures

```
Program input :  <p>         Hello     </p>         World!
Tokenization  :  OPEN_TAG    TEXT      CLOSE_TAG    TEXT
```

Recursion is delimited by special paired symbols, namely *call symbols* and *return symbols*.

```
     Recursion
  ┌───────────────┐
XML -> OPEN_TAG XML CLOSE_TAG XML | TEXT | ε
           ↑             ↑
          Call         Return
```

Nesting Structures

```
Program input :  <p>         Hello     </p>        World!
Tokenization  :  OPEN_TAG    TEXT      CLOSE_TAG   TEXT
```

Recursion is delimited by special paired symbols, namely *call symbols* and *return symbols*.

```
     Recursion                  Tail recursion does not count
  ┌───────────────┐ ─ ─ ─ ─ ─ ──┐
XML -> OPEN_TAG XML CLOSE_TAG XML | TEXT | ε
          ↑           ↑
         Call       Return
```

Nesting Structures

```
      Recursion                     Tail recursion does not count
  ┌────────────────┐ ─ ─ ─ ─ ─ ─ ──┐
XML -> OPEN_TAG XML CLOSE_TAG XML | TEXT | ε
           ↑              ↑
          Call          Return
```

Visibly Pushdown Grammars (VPGs) =

Regular Grammars + Nesting Structures

$$L \rightarrow \epsilon$$

$$L \rightarrow cA \qquad \text{Regular Grammar}$$

$$L \rightarrow aAbB \qquad \text{Models Nesting Structures}$$

Denote Sentences of VPGs

- Sentences in VPGs are normal strings with explicitly denoted call and return symbols, known as *tagging*.

```
<p> Hello </p> World!
```

- **V-Star** learns which substrings need to be colorized, also known as the *tagging function* for program inputs.

From Program Input Learning to Active Learning

- **Oracle**: Answers *membership queries*

$$\mathcal{O} : \text{string } s \mapsto \{\text{true}, \text{false}\}$$

- **Challenges**: With finite queries, exact learning is not guaranteed.
- **Minimally Adequate Teacher (MAT)**: Answers *equivalence queries*

$$\mathcal{E} : \text{grammar } G \mapsto \{\text{exact 🎉}, \text{counterexample string } s\}$$

- **Active Learning**: learn grammar from a MAT.

Exact Learning: Achievable?

	Regular	VPG	CFG
Positive Examples	Impossible		
Positive + Negative	NP-complete		
Membership Queries	Very Likely NP		
MQ + Seed Strings	Very Likely NP	(V-Star)	(Glade & Arvada)
MAT (MQ + EQ)	Polynomial (Angluin's L^*)	Polynomial	Very Likely NP

V-Star's Contribution

- Learns how to model program inputs as VPGs.
- Learns VPGs using active learning methods.

V-Star Workflow

1. **Identify Call and Return Symbols**:
   - Use oracle and seed strings to infer nesting structures.
   - Develop a *tagging function* to recognize call and return symbols.
2. **Learn VPA and Convert to VPG**:
   - Use an L*-like algorithm to learn a Visibly Pushdown Automaton (VPA).
   - Convert the VPA into a Visibly Pushdown Grammar (VPG).

V-Star Example: Arithmetic Formula

Seed String:

`(1+(2×3)/4)`

What are the Call and Return Symbols?

- Hypothesize recursion as:

```
expr -> "(" expr ")" expr | number | ...
```

- Therefore, `(` and `)` are the call and return symbols.

`( 1+ ( 2×3 ) /4 )`

V-Star Example: Arithmetic Formula

Seed String (Encrypted):

□▼●□▼●▼△●▼△

What are the Call and Return Symbols?

- **Nesting Patterns**: Two substrings (x, y) that can be repeated at the same time, and must be repeated at the same time.

□▼●□▼●▼△●▼△

□▼●□□▼●▼△△●▼△

□▼●□□□▼●▼△△△●▼△

...

□▼● $($ □ $)^k$ ▼●▼ $($ △ $)^k$ ●▼△

V-Star Example: Arithmetic Formula

- From valid strings □▼● (□)^k ▼●▼ (△)^k ●▼△ , the nesting pattern is (u, x, z, y, v) = (□▼● , □ , ▼●▼ , △ , ●▼△), or simply

$$(x, y) = (\square, \triangle)$$

- **Lemma (V-Star)**: each nesting pattern (x, y) must contain a call symbol in x, and a return symbol in y.
- Therefore, □ and △ are the call and return symbols.

□ ▼● □ ▼●▼ △ ●▼ △

- **The tagging function**: For any program input, tags □ as call, and △ as return symbols.

□ ▼●▼●▼ △ ●▼ , □ ▼●▼ , ...

Learn Finite State Automata and Visibly Pushdown Automata

Input
1
1×
1×1
1×1×1
1×1×1×1

- Conjecture: regular expression (`1×`)* `1` specifies valid inputs.
- You have learned a formal grammar 🎉

Learn Finite State Automata and Visibly Pushdown Automata

- Learn regular expression
  - Regular expression is equivalent to Finite State Automata (FSA).
- $\rightarrow$ Learn FSA
  - Each state in the automata is an equivalence class.
- $\rightarrow$ Learn Equivalence Classes
  - $\rightarrow$ Angluin's L^* (1987): There is a table-based method!

Fill the Prefix-Suffix Table

Prefix	Suffix		
	ϵ	1	× 1
ϵ	ϵ	1	× 1
1	1	11	1 × 1
1 ×	1 ×	1 × 1	1 × × 1
1 × 1	1 × 1	1 × 11	1 × 1 × 1
1 × 1 × 1	1 × 1 × 1	1 × 1 × 11	1 × 1 × 1 × 1

Fill the Prefix-Suffix Table

Prefix	Suffix		
	ϵ	1	$\times 1$
ϵ			
1			
$1 \times$			
1×1			
$1 \times 1 \times 1$			

- Each color sequence represents an equivalence class.

Fill the Prefix-Suffix Table

Prefix	Suffix ϵ	Suffix 1	Suffix $\times\ 1$
ϵ			
1			
$1\ \times$			
1×1			
$1 \times 1 \times 1$			

- `ε` is equivalent to `1 ×`.

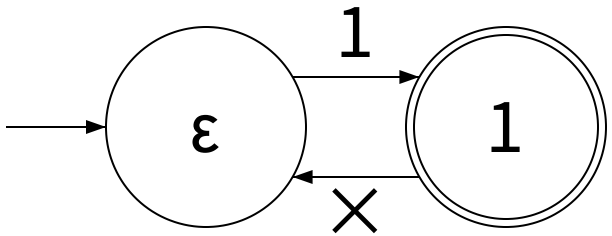
Fill the Prefix-Suffix Table

Prefix	Suffix ϵ	Suffix 1	Suffix × 1
ϵ			
1			
1 ×			
1 × 1			
1 × 1 × 1			

- `1` is equivalent to `1 × 1` and `1 × 1 × 1`.

Build an FSA

- The equivalence classes can be convert to an FSA directly.



- The FSA above is equivalent to regular expression (`1×`)* `1` .

Doesn't work for VPG

Prefix	Suffix ϵ	$\times 1)$	$\times 1) \times 1)$	...	$\times 1) \ldots \times 1)$	...
ϵ	ϵ	$\times 1)$	$\times 1) \times 1)$	...	$\times 1) \times 1) \times 1)$	...
$(1$	$(1$	(1×1)	$(1 \times 1) \times 1)$	...	$(1 \times 1) \times 1) \times 1)$	...
$((1$	$((1$	$((1 \times 1)$	$((1 \times 1) \times 1)$	...	$((1 \times 1) \times 1) \times 1)$	...
...	...	...	...	...	...	...
$(\ldots(1$	$(((1$	$(((1 \times 1)$	$(((1 \times 1) \times 1)$	...	$(\ldots(1 \times 1) \ldots \times 1)$	...
...	...	...	...	...	...	...

- Infinite number of equivalence classes.

Learn Finite State Automata and Visibly Pushdown Automata

Infix	Left Context, Right Context		
	ϵ, ϵ	$\epsilon, \times 1$	$(, \times 1)$
ϵ	ϵ	$\times 1$	$(\times 1)$
1	1	1×1	(1×1)
$1 \times$	$1 \times$	$1 \times \times 1$	$(1 \times \times 1)$
(1×1)	(1×1)	$(1 \times 1) \times 1$	$((1 \times 1) \times 1)$

- Each infix must be well-matched.
- The number of equivalence classes becomes finite.

More Info in the Paper

- **Hierarchy**:
  - Regular Grammar < VPGs < CFGs
- **Formal Definition of VPGs and VPA**
- **Conversions and Learning**:
  - From Finite State Automata (FSA) to Visibly Pushdown Automata (VPA)
  - How to identify nesting structures using nesting patterns
  - How to learn Visibly Pushdown Automata (VPA)
- **Partial Tokenizer**:
  - What is it?
  - How does it help?

Evaluation Overview

- **Evaluation Methodology**
  - Replicated the artifact of Arvada (ASE '22).
  - Includes oracle grammars, evaluation datasets, and seed strings.
- **Selected Grammars**
  - Five Grammars: JSON, LISP, XML, While, MathExpr
  - Chosen because they are VPGs

Further exploration of additional program inputs is reserved for future research.

Evaluation: Accuracy

- **Recall** $\frac{|L \cap L_{\mathcal{O}}|}{|L|}$: Probability that a string of the oracle is accepted by the learned grammar G.
- **Precision** $\frac{|L \cap L_{\mathcal{O}}|}{|L_{\mathcal{O}}|}$: Probability that a string in G is accepted by the oracle.
- **F-1 Score** $\frac{2}{1/\text{recall}+1/\text{prec}}$: Harmonic mean of precision and recall, indicating overall accuracy.

Grammar	GLADE			Arvada			V-Star		
	Recall	Prec	F-1	Recall	Prec	F-1	Recall	Prec	F-1
JSON	0.42	0.98	0.59	0.97 ± 0.09	0.92 ± 0.08	0.94 ± 0.05	1.00	1.00	1.00
Lisp	0.23	1.00	0.38	0.38 ± 0.26	0.95 ± 0.08	0.50 ± 0.18	1.00	1.00	1.00
XML	0.26	1.00	0.42	0.99 ± 0.02	1.00 ± 0.00	1.00 ± 0.01	1.00	1.00	1.00
While	0.01	1.00	0.02	0.91 ± 0.20	1.00 ± 0.00	0.94 ± 0.14	1.00	1.00	1.00
MathExpr	0.18	0.98	0.31	0.72 ± 0.24	0.96 ± 0.03	0.80 ± 0.16	1.00	1.00	1.00

Evaluation: Efficiency

- **#Queries**: The number of unique membership queries made during the learning process.
- **Time**: The overall running time for each tool.
- **#Seeds**: The number of seed strings (i.e., valid strings) shared by all tools.

Grammar		GLADE		Arvada		V-Star	
	#Seeds	#Queries	Time	#Queries	Time	#Queries	Time
JSON	71	11 K	21 s	6.8 K ± 394	25 s ± 2 s	541 K	33 min
Lisp	26	3.8 K	7 s	2.2 K ± 307	8 s ± 2 s	16 K	77 s
XML	62	15 K	21 s	12 K ± 1 K	61 s ± 5 s	208 K	16 min
While	10	9.2 K	13 s	5.4 K ± 563	15 s ± 1 s	1440 K	1.5 h
MathExpr	40	19 K	42 s	6.6 K ± 421	24 s ± 2 s	4738 K	6 h

Evaluation: More Statistics of V-Star

- **%Queries(Token)**: Percentage of membership queries made for token inference.
- **%Queries(VPA)**: Percentage of membership queries made for VPA learning.
- **#TestString**: The number of test strings sampled from the seed strings by V-Star.

Grammar	#Queries	%Q(Token)	%Q(VPA)	#TestString	Time
JSON	541 K	2.71%	97.29%	8043	33 min
Lisp	16 K	1.37%	98.63%	693	77 s
XML	208 K	94.93%	5.07%	682	16 min
While	1440 K	9.40%	90.60%	119	1.5 h
MathExpr	4738 K	0.11%	99.89%	2602	6 h

Most time is spent on VPA learning, not token inference, as short seed strings lead to shorter nesting patterns and a smaller search space.

Conclusion and Future Work

- **Conclusion**: V-Star is more accurate compared to other tools but requires more time due to VPA learning.
- **Future Work**:
  - **Performance**: Improve V-Star's efficiency to reduce VPA learning time.
  - **Evaluation**: Test V-Star on more practical grammars.
  - **Alternative Algorithms**: Explore other VPA learning methods.
  - **Readability**: Enhance the readability of the learned grammar.
  - **CFG Learning**: Use V-Star as a starting point for learning Context-Free Grammars.