Empowering learning-based vulnerability analysis via automated data augmentation

Haipeng Cai

Associate Professor

Washington State University

Vulnerability Analysis

- Trained/Evaluated: Synthetic Datasets.
- Real-world Datasets: Small.
- Results: Highly Inflated

Deep Learning-based Analysis

Static Analysis

Fuzzing

Dynamic Analysis

We urgently need datasets:

- Large-scale
- Realistic

Realistic Vulnerability Dataset

Automatic Generation:
- FixReverter:
  - Manual Designed Rules
  - Diversity Limited
- Neural Code Editors:
  - Chicken-Egg Problem
  - Semantic Aware

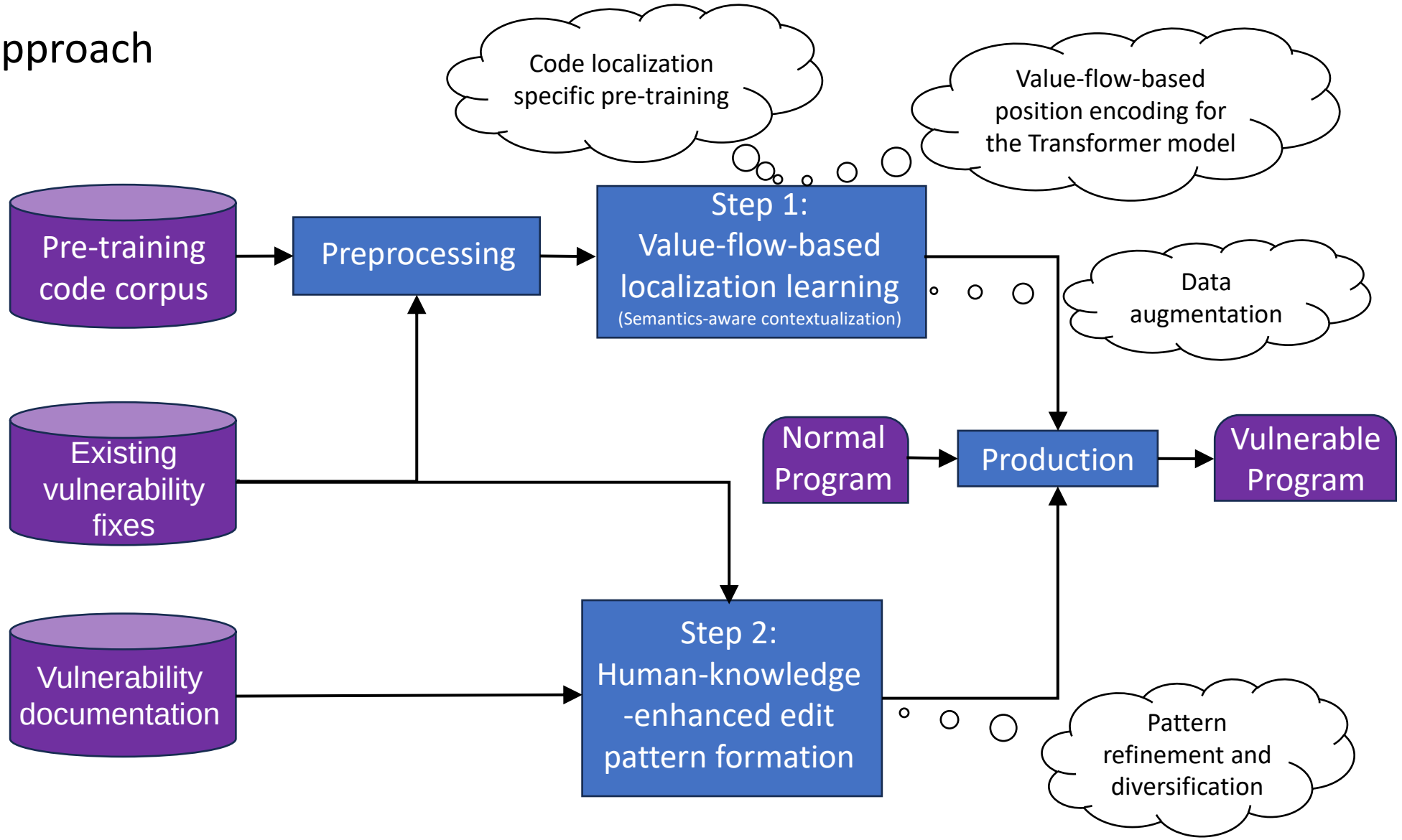
Wild Collection:
- Much human time
- Automatic Tool: Low Precision

- Deep Learning
  - Semantic Understanding
- Pattern Mining/Application
  - Small Training Set

VGX:
- **Human-knowledge-enhanced edit pattern formation**
- **Value-flow-based pre-trained Transformer for localization learning**
- **Large-scale vulnerability data generation with high accuracy.**

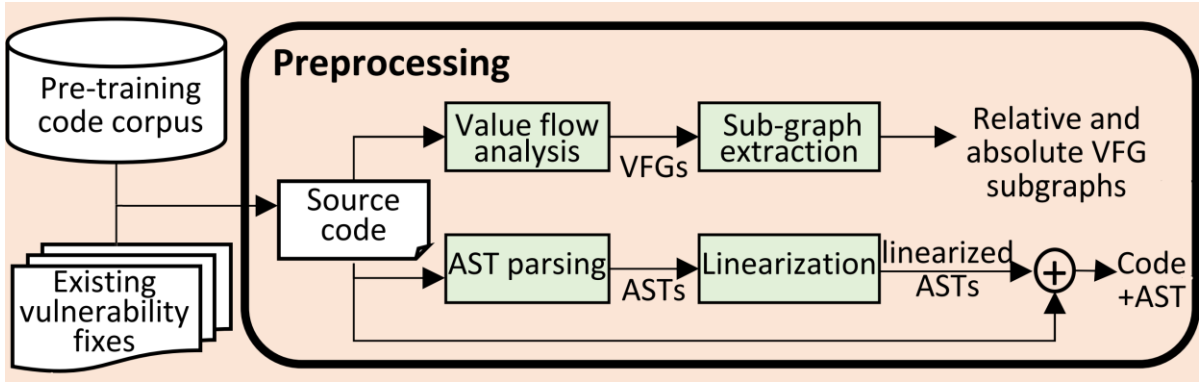
VGX: Large-Scale Sample Generation for Boosting Vulnerability Analyses

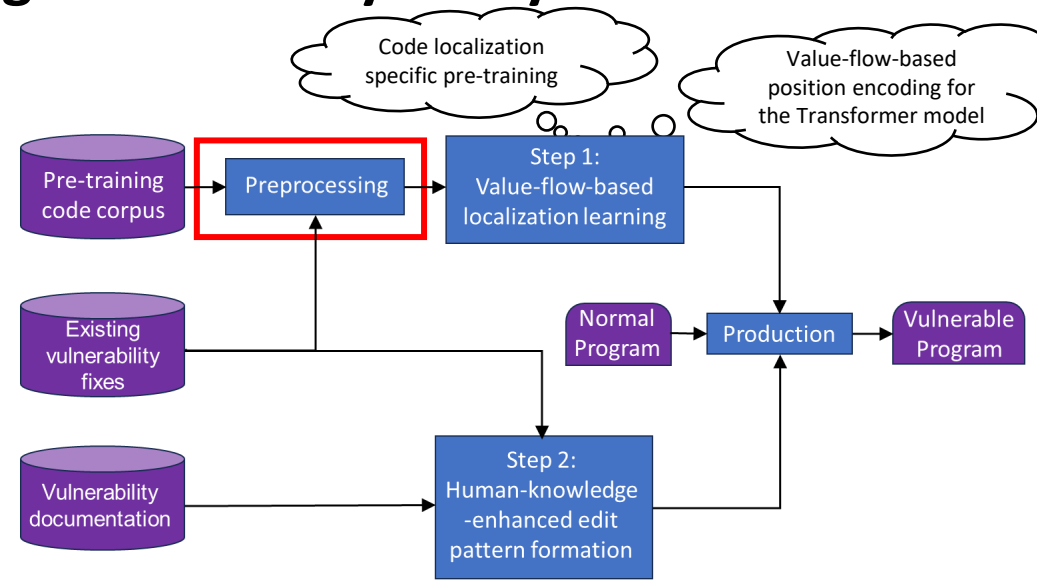
- Approach



VGX: Large-Scale Sample Generation for Boosting Vulnerability Analyses

- Approach
  - Preprocessing

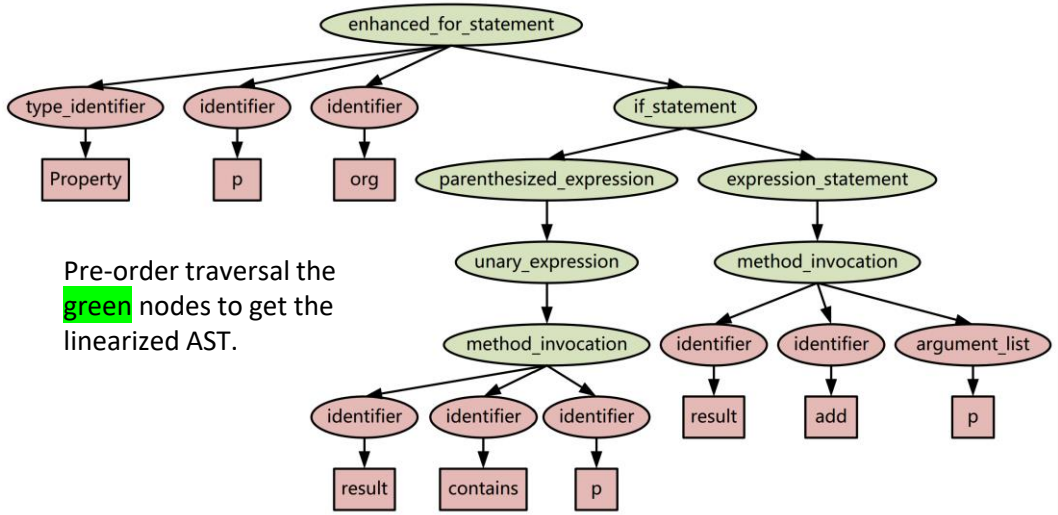





```
1  int foo(char *password)
2  {
3      char *buf=malloc(BUFSIZE*sizeof(char));
4      int res;
5      if(buf==NULL)
6          return -ENOMEM;
7      if(strlen(password)>=BUFSIZE)
8          return -EINVAL;
9      strcpy(buf, password);
10     res=check(buf);
11     if(res==1) {
12         printf("auth successful.\n");
13         free(buf);
14         return res;
15     } else {
16         printf("auth failed.\n");
17         free(buf);
18         return res;}
19 }
```

Arrowed lines indicate value flow, with different colors just differentiating the value flow of different variables (e.g., *`password'* and *`buf'*).

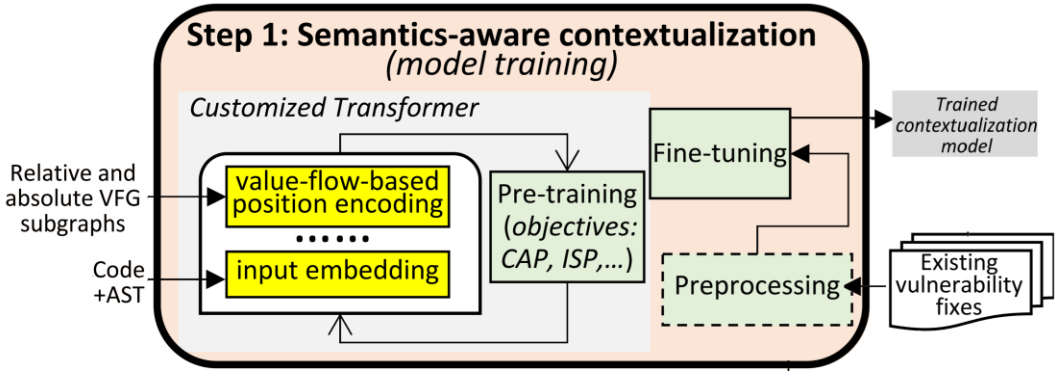
Grey background marks where vulnerabilities may be injected (e.g., Lines 5-8).

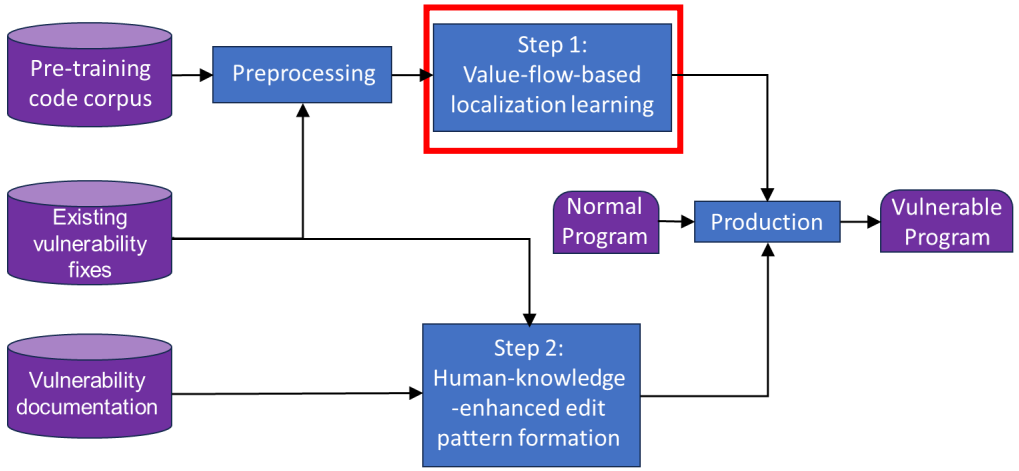


Pre-order traversal the green nodes to get the linearized AST.

VGX: Large-Scale Sample Generation for Boosting Vulnerability Analyses

- Approach
  - Step 1: Semantics-aware contextualization





  - Self-Attention block with value-flow-based position encoding

Output hidden representation of the i-th token.

Attention between i-th and j-th tokens.

$$z_i = \sum_{i=1}^{n} \frac{exp(a_{ij})}{\sum_{j'=1}^{n} exp(a_{ij'})}(x_j W^V + r_{ij}^V + r_{ij}^{V_{VFG}})$$

$$a_{ij} = \frac{1}{\sqrt{2d}}(x_i W^Q)(x_j W^K + r_{ij}^K + r_{ij}^{K_{VFG}})^T + \frac{1}{\sqrt{2d}}(a_i^Q)(a_j^K)^T + \frac{1}{\sqrt{2d}}(a_i^{Q_{VFG}})(a_j^{K_{VFG}})^T$$

VFG-based relative position encoding

VFG-based absolute position encoding

Traditional relative position encoding

Traditional absolute position encoding

Message passing aggregation

$$x'_v = GRU(x_v, \sum_{(u,v)\in E} g(x_u))$$

$$x_g = \sum_{v \in V} x'_v$$

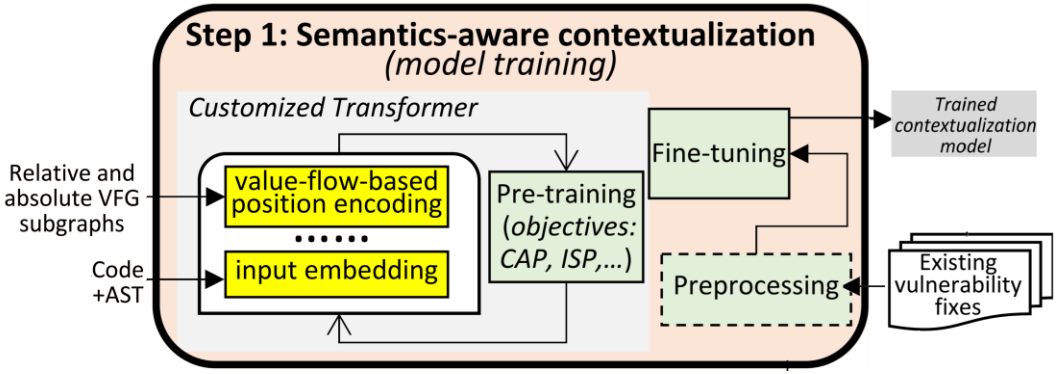
Sum up to get the graph representation

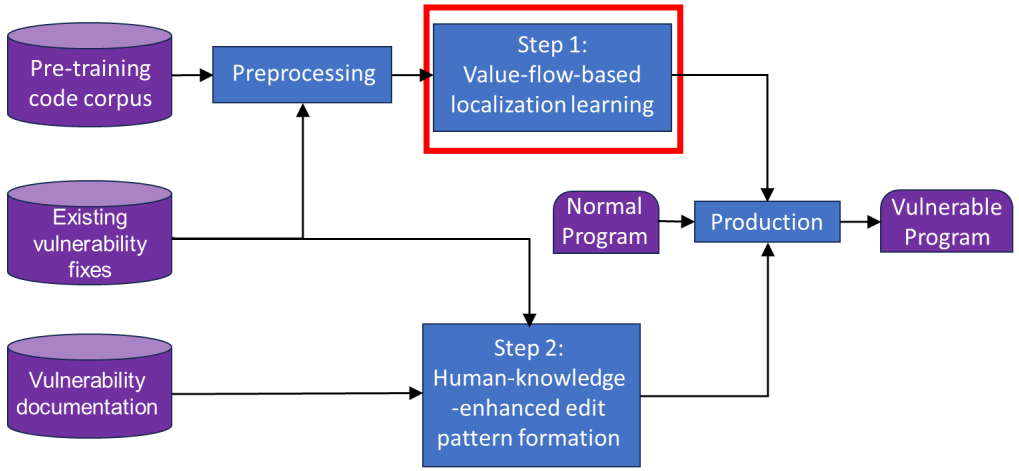
$$r_{ij}^{V_{VFG}} = x_g R_{VFG}^V$$
$$r_{ij}^{K_{VFG}} = x_g R_{VFG}^K$$
$$a_i^{Q_{VFG}} = x_g A_{VFG}^V$$
$$a_j^{K_{VFG}} = x_g A_{VFG}^K$$

multiplied with a weight matrix to get the value-flow-based position encoding

VGX: Large-Scale Sample Generation for Boosting Vulnerability Analyses

- Approach
  - Step 1: Semantics-aware contextualization

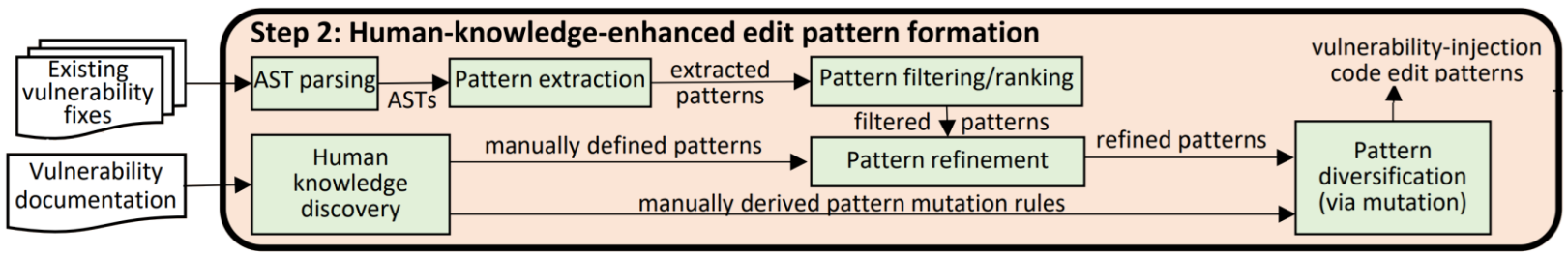


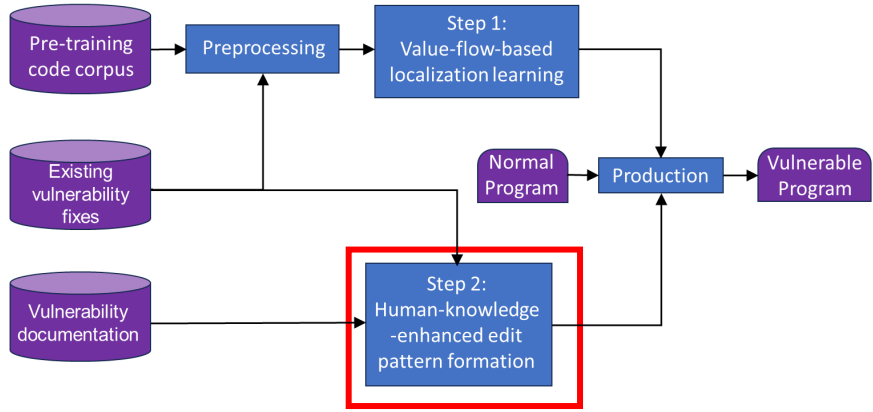


  - Pre-training tasks:
    - CodeT5 Tasks: mask span prediction (`MSP`), identifier tagging (`IT`), masked identifier prediction (`MIP`)
    - Code-AST Prediction (`CAP`)
    - Irrelevant Statement Prediction (`ISP`)
  - Data augmentation for fine-tuning

VGX: Large-Scale Sample Generation for Boosting Vulnerability Analyses

- Approach
  - Step 2: Human-knowledge-enhanced edit pattern formation

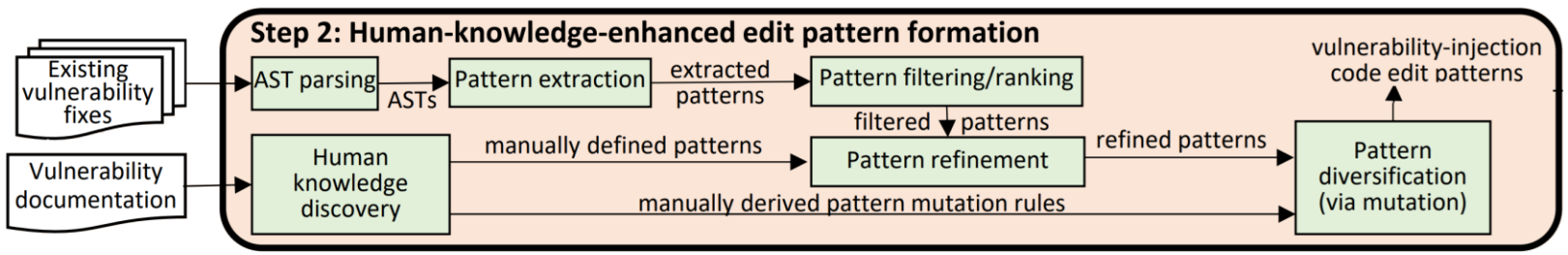


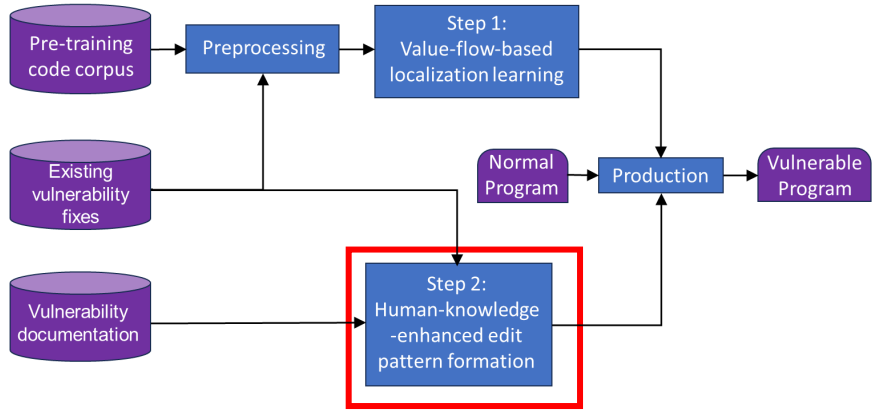


  - Pattern Extraction: Same as VULGEN
  - Pattern filtering/ranking:
    - Prevalence score (s_{preval})
    - Specialization score (s_{spec})
    - Identifier score (s_{ident})
    - $S_{rank} = S_{preval} \times S_{spec} \times S_{ident}$

VGX: Large-Scale Sample Generation for Boosting Vulnerability Analyses

- Approach
  - Step 2: Human-knowledge-enhanced edit pattern formation





  - Pattern refinement
    - Select the top-300 patterns in the ranking
    - Read the CWE documentation
    - Removed 21 patterns that are too general which cause many false positives
    - Manually derived 20 patterns which support regular expression
  - Pattern diversification
    - Manually derived 4 pattern mutation rules to diversify the patterns
    - Obtained 305 new patterns
  - Finally got 604 patterns for vulnerability injection

VGX: Large-Scale Sample Generation for Boosting Vulnerability Analyses

- Approach
  - Step 2: Human-knowledge-enhanced edit pattern formation

Table 10: Manually Defined Vulnerability-Injection Patterns

Patterns: *mutex*(h0); => EMPTY **Justification:** "Race Condition" mostly happens with a lack of mutex related statements, but there are many mutex related function [76]. Thus, once the located statement involve "mutex", we delete it.
Patterns: *TCHECK*(h0); => EMPTY *assert*(h0); => EMPTY **Justification:** There are many samples in the training set deleting statements involving "TCHECK" and "assert", but they usually use different function names[77]. Thus, once located statement involve "TCHECK" or "assert", we delete it.
Patterns: *free*(h0); => EMPTY *Free*(h0); => EMPTY *destruct*(h0); => EMPTY *destroy*(h0); => EMPTY *unref*(h0); => EMPTY *clear*(h0); => EMPTY **Justification:** "Memory Leak" mostly happens with not releasing assigned memory. However, there may be many different functions for releasing the memory [78]. Thus, once the the located statement involve memory release related functions, we delete it.
Patterns: unsigned h0; => h0; int64_t h0; => int h0; static h0 h1 = h2; => h0 h1 = h2; **Justification:** "Type Error" usually happens with not using static, unsigned, large-size types, but the current patterns specify too many details like identifier names or assigned values in the patterns [79]. Thus, we make these details holes so that they are more general.
Patterns: memset(h0); => EMPTY h0 = *ERR*; => EMPTY h0 = *NONE* => EMPTY h0 = 0; => EMPTY h0 = NULL; => EMPTY *buf* = h0; => EMPTY **Justification:** "Use of Uninitialized Variables" usually happens with not initializing declared variables, but current patterns specify too many details like identifier names and values in the patterns [80]. Thus, we make these details holes and use regular expression to represent the common initialized value, so that they are more general.
Patterns: h0 = kcalloc(hole1, hole2, hole3); => h0 = kzalloc(h1*h2, h3); h0 = calloc(hole0, hole1); => h0 = malloc(h1*h2); **Justification:** "Memory Allocation Vulnerability" usually happens when using unsafe memory allocation functions, but current patterns specify too many details like identifier names and values in the patterns [81]. Thus, we make them holes to make the patterns more general.

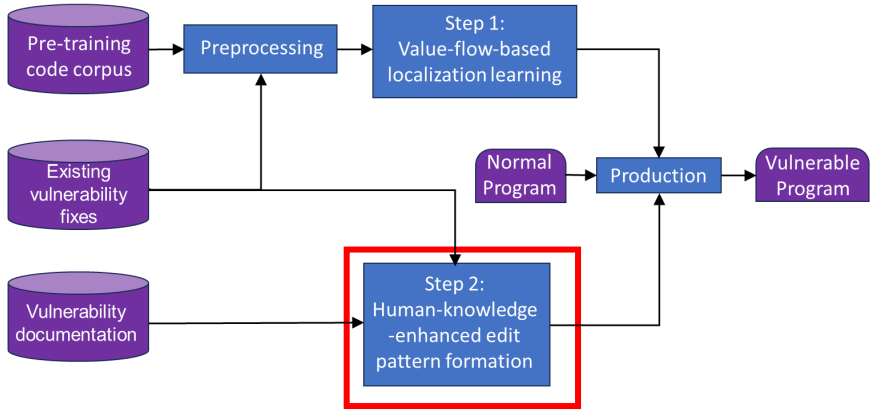
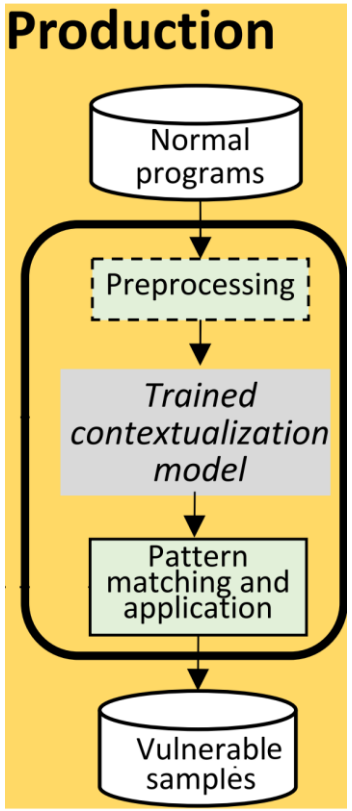


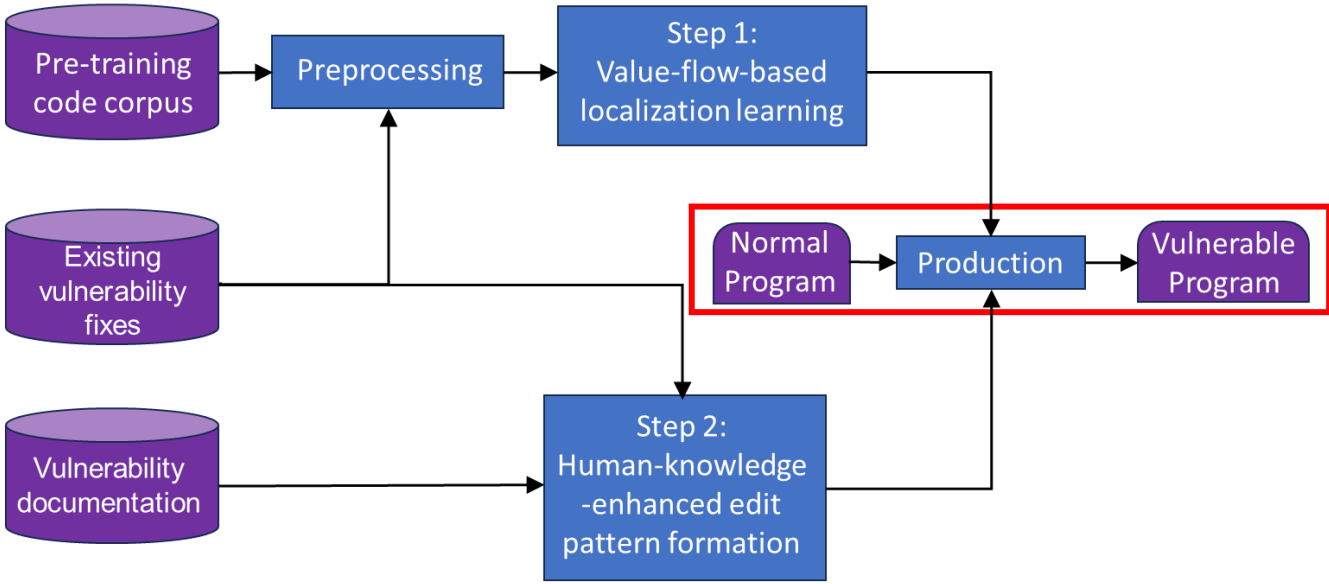
Table 11: Pattern Mutation Rules

Rule: function_name(parameters); =>new_function_name(new_parameters); ⇔ h0=function_name(parameters); => h0=new_function_name(new_parameters); **Justification:** Some function calls like strncpy may return some values but the return values do not always assign to a variable. Mutating such patterns so that they have or remove the return value assignments increases the generalizability of the pattern set.
Rule: if(condition) return NULL/0/-1; => EMPTY ⇔ if(condition) return -EINVAL/EBADFD/ENOTSOCK/EPERM/ENODEV/ENOMEM; => EMPTY **Justification:** The typical safety issue checks do some check in an if statement condition and then return an error code when found the issue. However, there are many possible returned error codes. We mutate these returned error codes to make the patterns more general.
Rule: if(specific_condition) return error_code; => EMPTY ⇒ if(hole) return error_code;=> EMPTY **Justification:** In our automatic pattern mining, the mined safety checks if statements may be too specific in the condition. However, if an if statement only has one return statement, it is very likely that it is a safety check if statement. Thus, we remove the specific conditions in these if statements and use a hole to make the patterns more general.
Rule: if(condition) return error_code; => EMPTY ⇔ if(condition) break/continue; => EMPTY **Justification:** When the safety checks if statements find issues, they may not always exit the function using a return. If the safety check is in a for/while/switch block, it may use a break or continue to exit. Thus, we mutate the exit statement to make the patterns more general.

VGX: Large-Scale Sample Generation for Boosting Vulnerability Analyses

- Approach
  - Step 3: Vulnerability Data Production





VGX: Large-Scale Sample Generation for Boosting Vulnerability Analyses

- VGX effectiveness

Technique	Precision	Recall	F1	Success Rate
VGX	59.46%	22.71%	**32.87%**	**93.02%**
VULGEN	17.50% (239.77%↑)	15.74% (44.28%↑)	**16.51% (99.09%↑)**	**75.96% (22.45%↑)**
CodeT5	12.65% (370.04%↑)	12.65% (79.53%↑)	**12.65% (159.84%↑)**	**24.81% (274.93%↑)**
Getafix	4.67% (1173.23%↑)	2.58% (780.23%↑)	**3.32% (890.06%↑)**	**57.75% (67.07%↑)**
Graph2Edit	13.97% (325.62%↑)	13.97% (65.56%↑)	**13.97% (135.29%↑)**	**21.71% (328.47%↑)**

Evaluation Dataset

- 7,764 samples
- Training: 6,989, Testing: 775

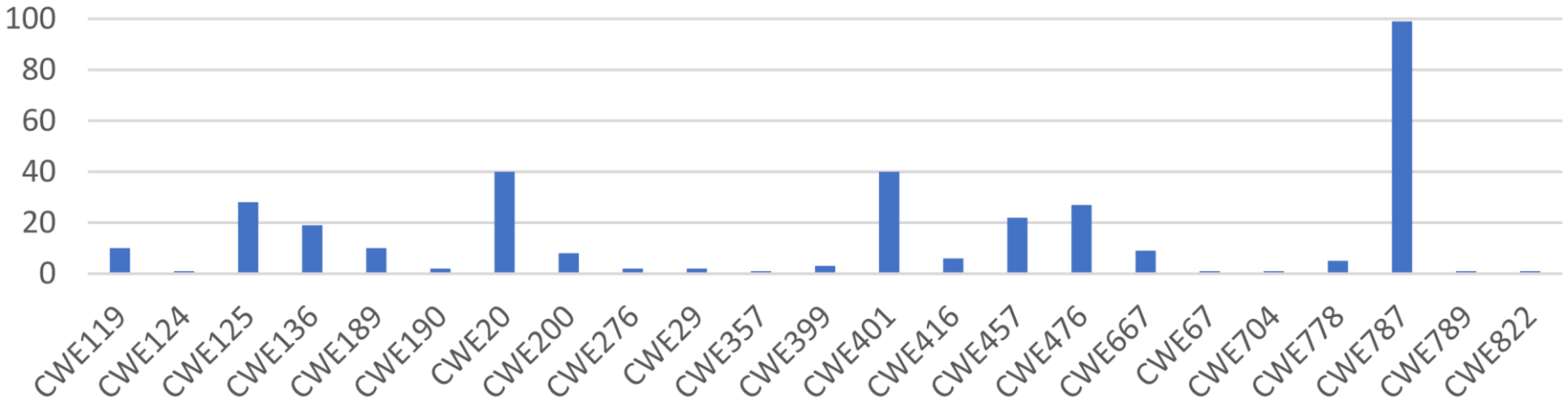
VGX: Large-Scale Sample Generation for Boosting Vulnerability Analyses

- Ablation Study

Experiment	Loc Acc	Precision	Recall	F1
VGX	55.35%	59.46%	22.71%	**32.87%**
No CAP and ISP	53.03% (4.37%↑)	54.36% (9.38%↑)	20.90% (8.66%↑)	**30.19% (8.87%↑)**
No AST	49.67% (11.44%↑)	53.90% (10.32%↑)	19.61% (15.81%↑)	**28.76% (14.29%↑)**
No VFG	52.13% (6.18%↑)	53.29% (11.58%↑)	21.93% (3.56%↑)	**31.07% (5.79%↑)**
No Augmentation	51.61% (7.25%↑)	53.33% (11.49%↑)	19.61% (15.81%↑)	**28.67% (14.65%↑)**
No Diversification	55.35% (0.00%↑)	62.45% (-4.78%↑)	20.38% (11.43%↑)	**30.73% (6.96%↑)**
No Refinement	55.35% (0.00%↑)	71.91% (-17.31%↑)	16.51% (37.55%↑)	**26.85% (22.42%↑)**

VGX: Large-Scale Sample Generation for Boosting Vulnerability Analyses

- Large-scale production
  - Based on 738,453 real-world normal samples
    - VGX generates 150,392 samples
    - Success rate: 90.13%
  - Vulnerabilities cover 23 CWE types



VGX: Large-Scale Sample Generation for Boosting Vulnerability Analyses

- Downstream analysis improvement
  - Sample 10% of the generated samples (15,039) for augmentation

- Vulnerability Detection

Model	Precision	Recall	F1
Devign-ori	9.82%	50.19%	**16.43%**
Devign-aug-VGX	12.37% (25.97%↑)	52.47% (4.54%↑)	**20.01% (21.79%↑)**
Devign-aug-VulGen	11.23% (14.35%↑)	30.03% (-40.17%↑)	**16.35% (-0.49%↑)**
Devign-aug-SARD	15.27% (55.49%↑)	15.21% (-69.69%↑)	**15.24% (-7.24%↑)**
LineVul-ori	26.42%	2.52%	**4.61%**
LineVul-aug-VGX	11.38% (-56.93%↑)	78.00% (2995%↑)	**19.86% (330.80%↑)**
LineVul-aug-VulGen	9.97% (-62.26%↑)	3.73% (48.01%↑)	**5.42% (17.57%↑)**
LineVul-aug-SARD	9.19% (-65.21%↑)	85.70% (3300%↑)	**16.60% (260.09%↑)**
IVDetect-ori	9.06%	75.52%	**16.18%**
IVDetect-aug-VGX	13.21% (45.81%↑)	35.66% (-52.78%↑)	**19.28% (19.15%↑)**
IVDetect-aug-VulGen	7.90% (-12.80%↑)	65.03% (-13.89%↑)	**14.09% (-12.92%↑)**
IVDetect-aug-SARD	10.04% (10.81%↑)	55.94% (-25.92%↑)	**17.02% (5.19%↑)**

- Vulnerability Localization

Model	Top-10 Accuracy	Model	Top-10 Accuracy
LineVul-ori	48.84%	**LineVD-ori**	59.25%
LineVul-aug-VGX	58.27% (19.31%↑)	**LineVD-aug-VGX**	66.87% (12.86%↑)
LineVul-aug-VulGen	53.43% (9.39%↑)	**LineVD-aug-VulGen**	52.68% (-11.09%↑)
LineVul-aug-SARD	49.85% (2.07%↑)	**LineVD-aug-SARD**	64.18% (8.32%↑)

- Vulnerability Repair

Model	Top-1 Accuracy	Top-5 Accuracy	Top-50 Accuracy
VulRepair-ori	8.55%	11.81%	16.29%
VulRepair-aug-VGX	21.05% (146.20%↑)	29.12% (146.57%↑)	30.14% (85.02%↑)
VulRepair-aug-VulGen	11.81% (38.13%↑)	16.77% (41.20%↑)	17.85% (9.85%↑)
VulRepair-aug-SARD	11.07% (29.47%↑)	13.92% (17.87%↑)	17.18% (5.46%↑)
VRepair-ori	2.58%	5.16%	8.62%
VRepair-aug-VGX	4.41% (70.93%↑)	10.59% (105.23%↑)	17.18% (99.30%↑)
VRepair-aug-VulGen	2.85% (10.46%↑)	7.26% (40.70%↑)	14.05% (62.99%↑)
VRepair-aug-SARD	1.36% (-47.28%↑)	3.46% (-32.94%↑)	4.96% (-42.45%↑)

VGX: Large-Scale Sample Generation for Boosting Vulnerability Analyses

- Real-World Vulnerability Discovery
  - CVEs Detected by Improved LineVul but missed by the original one among the 71 CVEs between 2021-2023

CVE ID	Project	CWE ID	CVE-ID	Project	CWE ID
2022-46149	Cap'n Proto	CWE-125	2021-3764	Linux Kernel	CWE-401
2023-27478	libmemcached-awesome	CWE-200	2022-47938	Linux Kernel	CWE-125
2022-28388	Linux Kernel	CWE-415	2023-23002	Linux Kernel	CWE-476
2023-22996	Linux Kernel	CWE-772	2022-42895	Linux Kernel	CWE-824
2021-3743	Linux Kernel	CWE-125	2022-34495	Linux Kernel	CWE-415
2022-24958	Linux Kernel	CWE-763	2022-47520	Linux Kernel	CWE-125
2022-30594	Linux Kernel	CWE-863			

Empowering learning-based vulnerability analysis via automated data augmentation

- Summary
  - Presented VGX, a state-of-the-art technique for large-scale vulnerability data generation.
  - Integrated human knowledge and semantics information into the pipeline.
  - Demonstrated that VGX is able to generate practical vulnerability dataset.
- Take-aways
  - Value flow information helps transformer models learn vulnerability injection location
  - Task specific pre-training (through respective objectives) facilitates model's learning of general knowledge specific for the task (of vulnerability injection).
  - Human knowledge about vulnerability overcomes overfitting to limited seed samples of vulnerability edit patterns.