Automatic code generation for malware detection based on MITRE ATT&CK techniques

Alexandru-Gabriel Sîrbu

WeADL 2023 Workshop

The workshop is organized under the umbrella of WeaMyL, project funded by the EEA and Norway Grants under the RO-NO-2019-0133.
Contract: No 26/2020

Table of contents

- Task presentation
- General translation overview
- Mundane code generation
- Improving code generation
- MITRE ATT&CK
- Solution details
- Original contribution
- Issues
- Conclusion
- Future work

Task presentation

Natural Language Text => Source Code

Input:

- Code Description – ambiguous
- Code Description + Test Case – less abstract

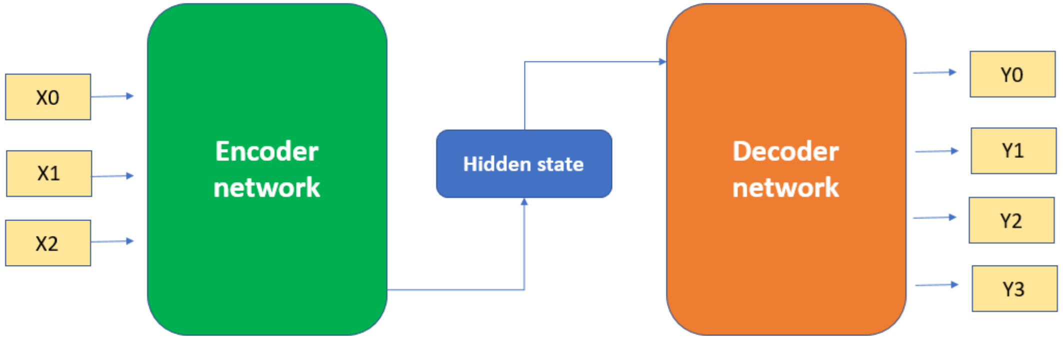
Output:

- All Program's Code – runnable
- Code Snippet – just handles the required logic

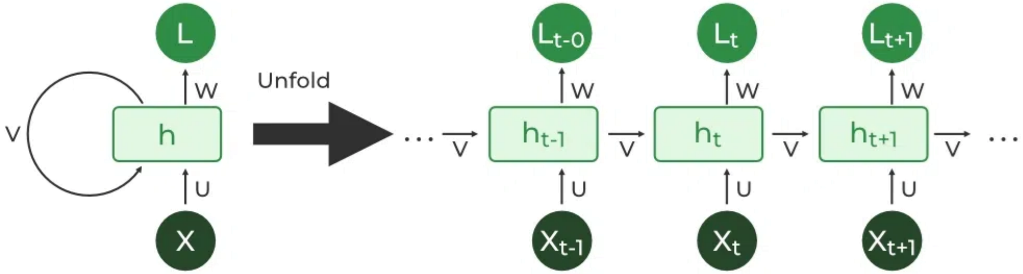
General translator overview

- Encoder-Decoder architecture
- Variable input and output
  - Input: solved by using padding
  - Output: solved by using Recurrent Neural Networks

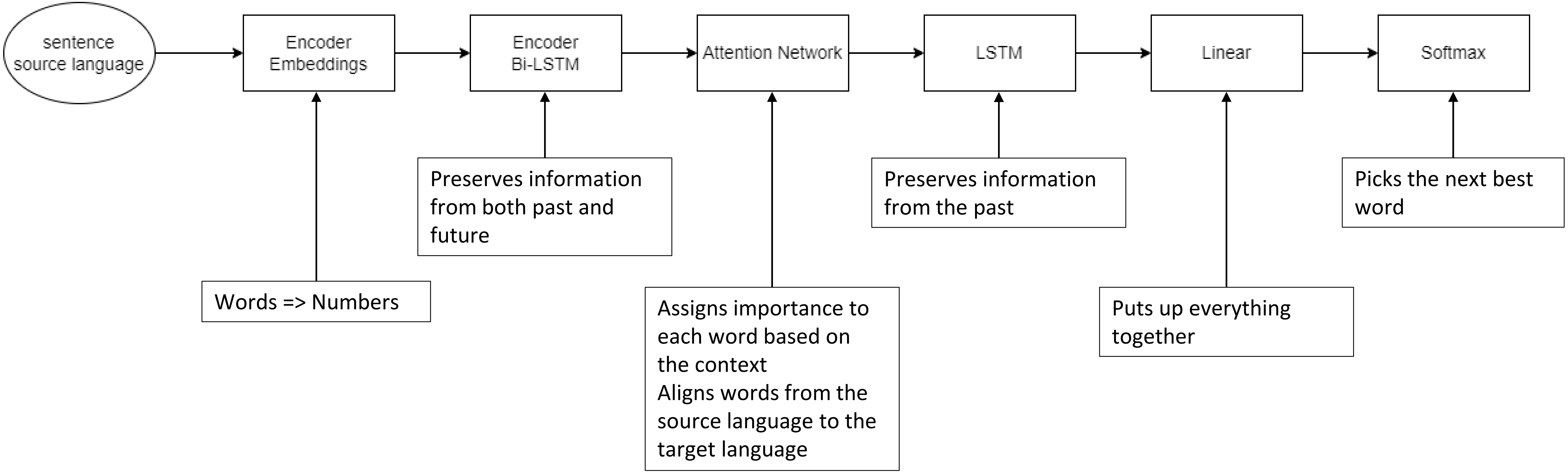
Encoder-Decoder architecture



Recurrent Neural Network



General translator architecture


sentence source language
Encoder Embeddings
Encoder Bi-LSTM
Attention Network
LSTM
Linear
Softmax
Words => Numbers
Preserves information from both past and future
Assigns importance to each word based on the context
Aligns words from the source language to the target language
Preserves information from the past
Puts up everything together
Picks the next best word

General translator evaluation

Metrics

- Accuracy
    - Standard metric
    - Does not work when there are multiple translations for the same sentence
- BLEU
    - Computes how much of our generated sentence is “syntactically valid” when compared to valid translations
    - e.g., “the cat is on the mat” vs “there is a cat on the mat”
    - {“the cat”, “cat is”, “is on”, “on the”, “the mat”} vs {“there is”, “is a”, “a cat”, “cat on”, “on the”, “the mat”}
    - Score: 2 / 5

Mundane code generation

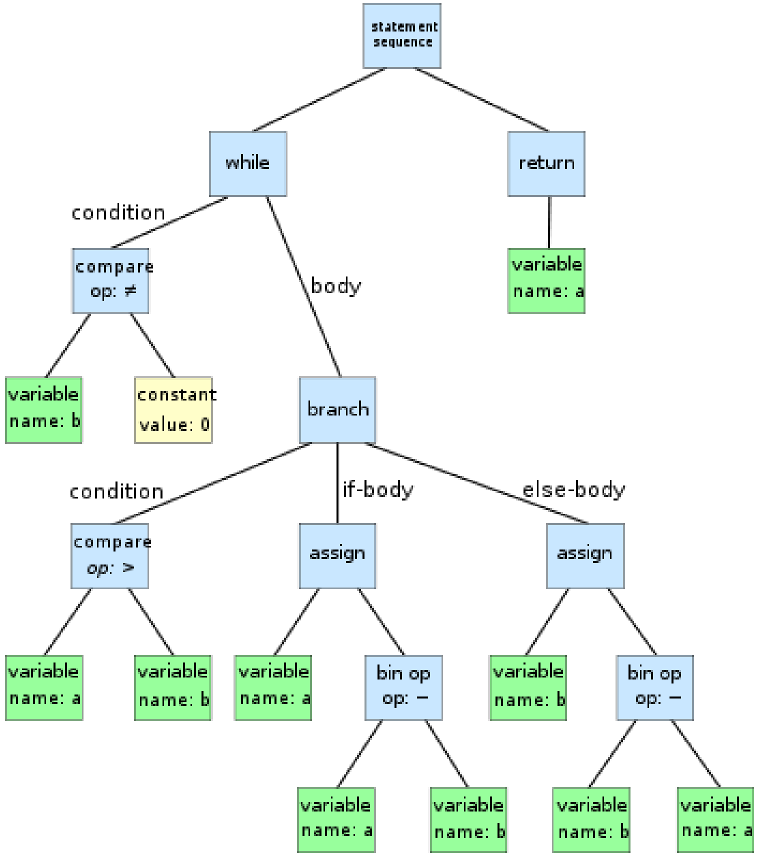
- Generate code word by word based on the input
- Prone to syntactic errors
  - Code not runnable
  - Requires manual work

Improving code generation

- Generate code as an Abstract Syntax Tree based on the input
    - Abstract Syntax Tree <=> Code

Abstract Syntax Tree example

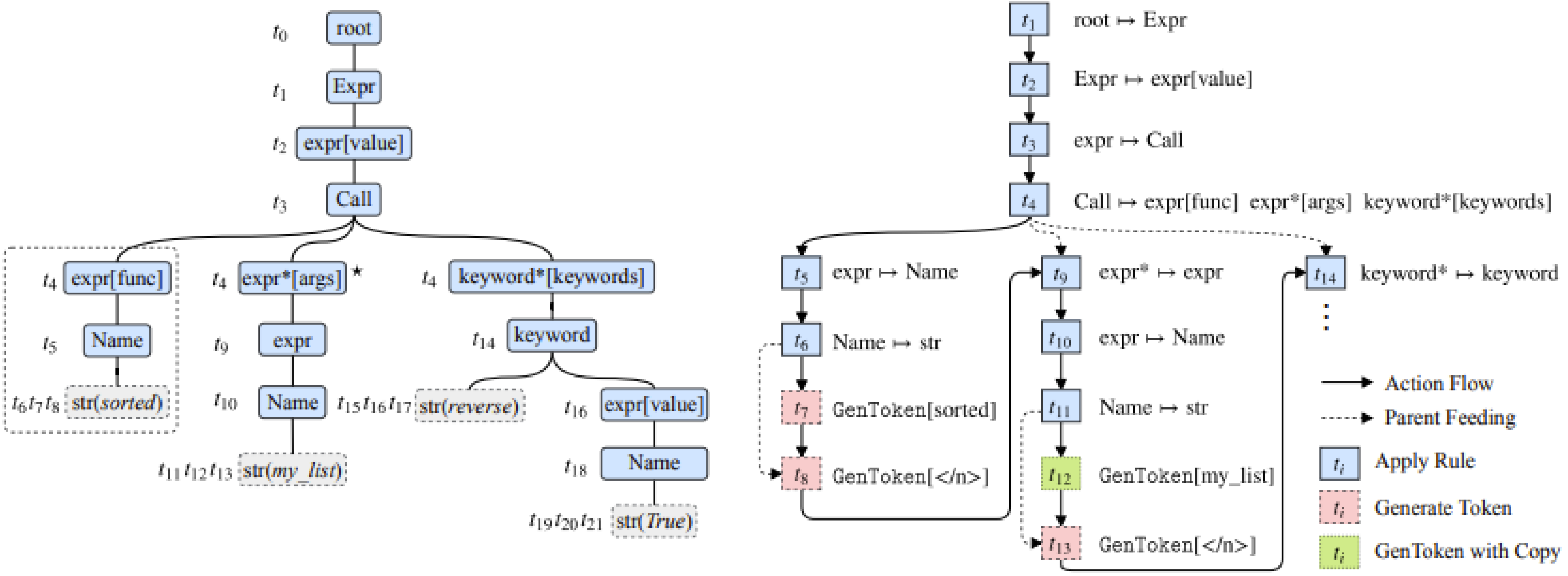
```
while b ≠ 0:
    if a > b:
        a := a - b
    else:
        b := b - a
return a
```



Improving code generation

- Generate one node at a time
- Input to model:
  - Description
  - State
  - Parent node
  - Right-most left sibling

Improving code generation



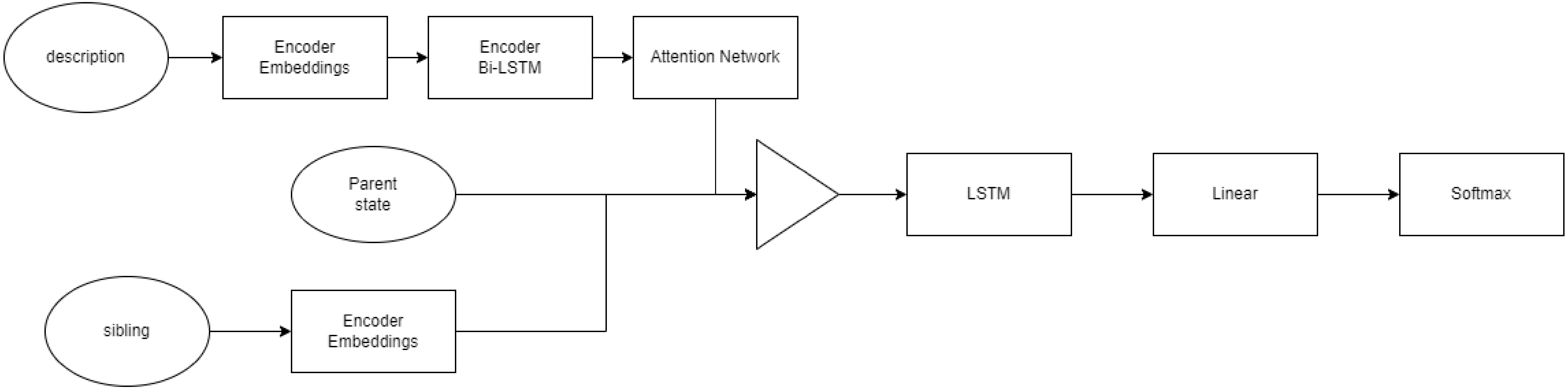
(a)

(b)

Input: `sort my_list in descending order`

Code: `sorted(my_list, reverse=True)`

Improving code generation



MITRE ATT&CK

- Open source
- Globally accessible knowledge base
- Contains tactics used by attackers during cyberattacks

MITRE ATT&CK tactics

- Describe specific methods or procedures
- Used to enhance security
- Used in international antivirus tests

Detection rule example

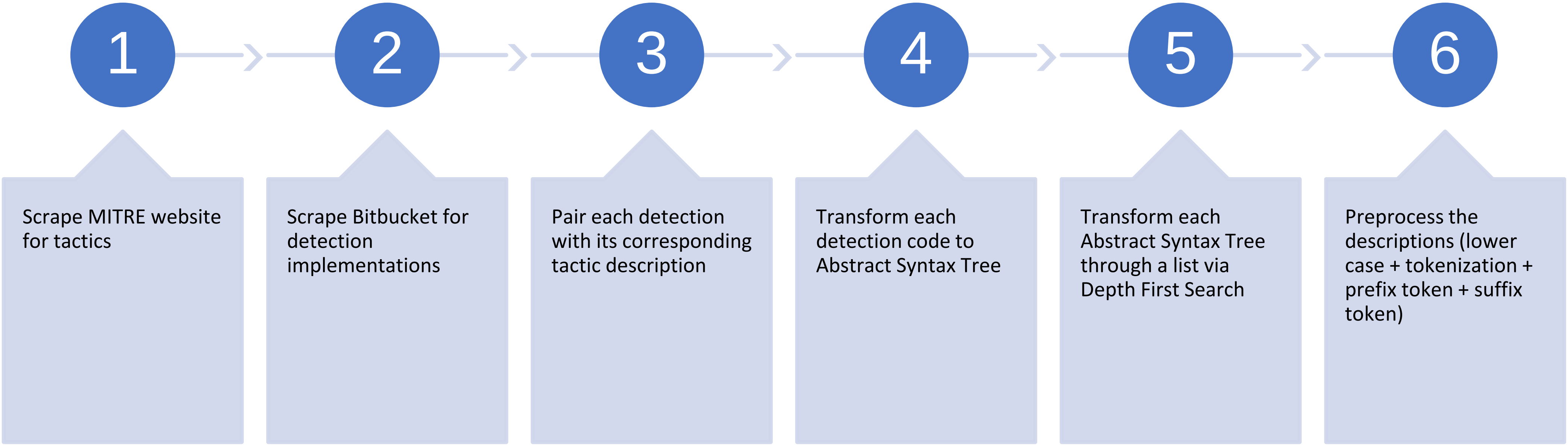
Adversaries may create a new process with an existing token to escalate privileges and bypass access controls. Processes can be created with the token and resulting security context of another user using features such as CreateProcessWithTokenW and runas.

```
[SIGNATURE]
Name = 'T1134.002'
[INFO]
Create Process with Token;
[RULES]
or {
    a = s.print('Case T1134.002 - Create Process with Token');
    forone actionProcess in ProcessAction.listFromAction(274) #
ACT_PROC_CREATE {
        n.and(actionProcess.flags, 3) == 0; # not excepted or
hidden
        actionProcess.process.path ==
"c:\windows\system32\runas.exe"
        actionProcess.process.cmdLine contains "/user";
    }
}
```

Dataset statistics

MITRE ATT&CK technique descriptions	411
MITRE ATT&CK technique implementations	102
Average number of description tokens	62.1
Average number of nodes in generated AST	462.3

Building the solution



1. Scrape MITRE website for tactics
2. Scrape Bitbucket for detection implementations
3. Pair each detection with its corresponding tactic description
4. Transform each detection code to Abstract Syntax Tree
5. Transform each Abstract Syntax Tree through a list via Depth First Search
6. Preprocess the descriptions (lower case + tokenization + prefix token + suffix token)

Original contribution

- Created directly nodes
  - Less nodes => more efficient
- Models for internal languages
  - High generalization and practical usage
- Dealt with low data entries
  - Encoder – pre-trained RoBERTa
- Using 2 components
  - Structure Generator + Dynamic Data Generator (variable names, strings)

Issues

Infinite loop when generating lists

- In literature: cap the elements in a list
- Our approach: generate AST Nodes on request

BLEU

- Evaluates syntax, not meaning
- No alternative at this moment

Results for structure generation

Dataset Model	Mitre LSTM	Mitre RoBERTa	Hearthstone Seq2Tree	Django Seq2Tree
BLEU-4	76.3	81.1	75.8	84.5
Dataset entries	102	102	665	18805

Conclusion

Automated code still requires human supervision

- Must be used in conjunction with good programming practices

Code generation

- Great for repetitive code
- Might struggle with new attack vectors

Future work

Generating dynamic data from code

- 100% complete runnable code
- Integrate the components together

Demo