SymLM: Predicting Function Names in Stripped Binaries via Context-Sensitive Execution-Aware Code Embeddings

SecLab
COMPUTER SECURITY LABORATORY

EDGE Institute

Xin Jin
The Ohio State University
jin.967@osu.edu

Kexin Pei
Columbia University
kpei@cs.columbia.edu

Jun Yeon Won
The Ohio State University
won.126@osu.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Overview

Function name prediction in stripped binaries is a very useful but extremely challenging task. For this,

- We design a **function symbol name prediction** and **binary language modeling** framework, SymLM.
- We propose a **novel neural architecture** to jointly learn function semantics preserved in **execution behavior** and **calling context**.
- We evaluate SymLM with **1.4M** binary functions and show that it outperforms the state-of-the-art works by **up to 35% in F1 score** with better generalizability and obfuscation resistance.
- We show SymLM's component effectiveness and **practical use cases** with IoT firmware images.

Background and Motivation

1. Commercial software (e.g., IoT firmware, browsers, and pdf readers) is usually **closed-source** and shipped in **stripped** binaries, whose **semantic information** (e.g., function names) is **missing**. Predicting function names helps reverse engineers **understand code semantics, identify malware/vulnerabilities**, etc.

2. Predicting function names is **very challenging**, because:

a. Semantic similar code can appear differently, e.g.,

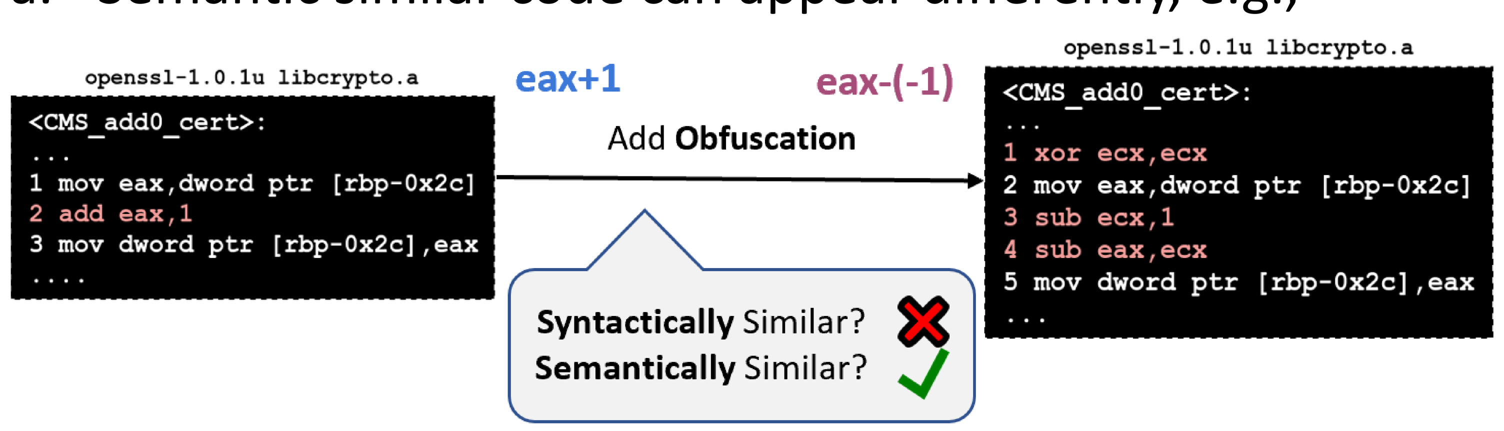


Fig. 1: Semantically Similar but Syntactically Different Code

b. Function names are noisy, e.g., the probability that 2 developers give the same name for a function is 6.9%.

c. Modeling calling context is necessary, e.g.,

```
YY_BUFFER_STATE yy_scan_string (char *yystr) {
    size_t _yybytes_len;
    YY_BUFFER_STATE pyVar1;
    _yybytes_len = strlen(yystr);
    pyVar1 = yy_scan_bytes(yystr,_yybytes_len);
    return pyVar1;
}
```

Fig. 2: Partial Semantics Preserved in Its Callees

Our key observations: predicting function names requires (1) learning semantics from execution behavior, (2) resolving NLP issues, and (3) modeling calling context.

Methodology

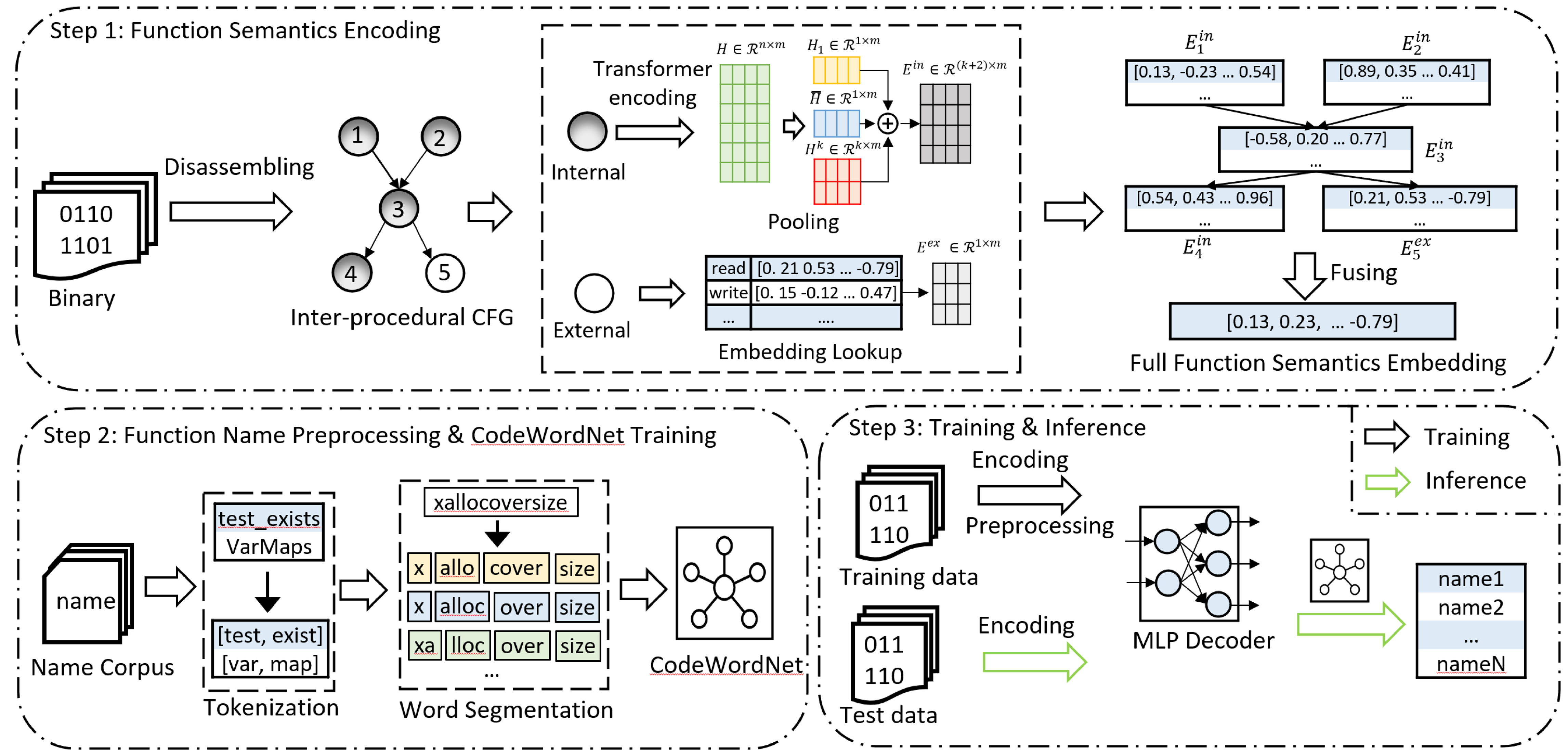


Fig. 3: System Workflow with Three Major Steps

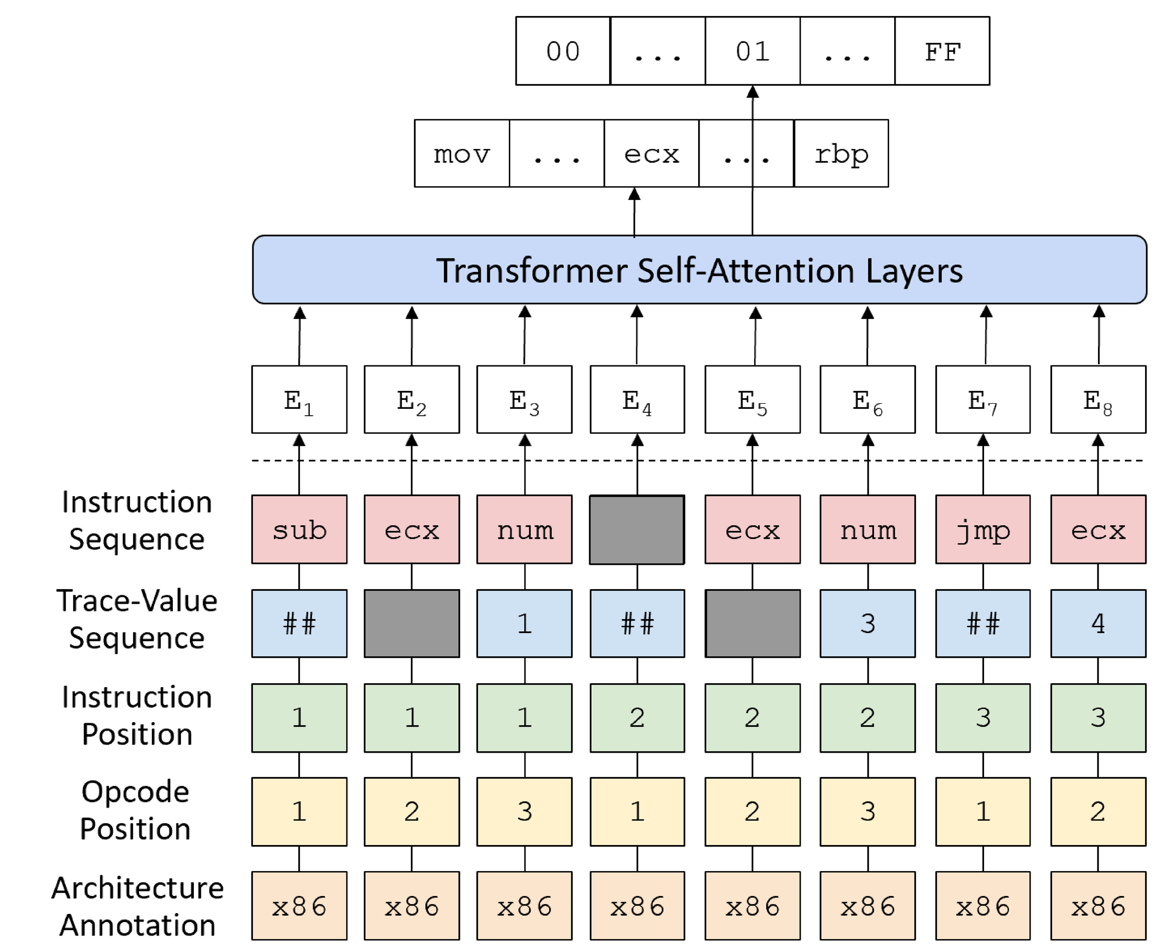


Fig. 4: The Pretrained Model Used for **Transformer Encoding**. The pretraining tasks force the model to learn **execution behavior**.

Step 1: SymLM generates embeddings by fusing semantics of calling context and function instructions. It encodes internal functions by a pretrained model and external functions by an embedding lookup table.

Step 2: SymLM resolves NLP issues by tokenizing names into words, segmenting words by a unigram language model, and embedding words with CodeWordNet (consisting of 3 word embedding models).

Step 3: SymLM updates weights of the pretrained model, embedding lookup layer, and MLP decoder in training, and predicts names of stripped binaries that semantically match ground truth with CodeWordNet.

Evaluation and Results

1. Evaluation Setup: we built our datasets with 27 open-source projects, compiled into **16K binaries and 1.4 M functions** in 4 architectures, 4 optimizations, and 4 obfuscation options.

2. Overall Performance: 0.634 precision, 0.677 recall, and 0.655 F1 score on average.

3. Comparison to the state-of-the-art works:

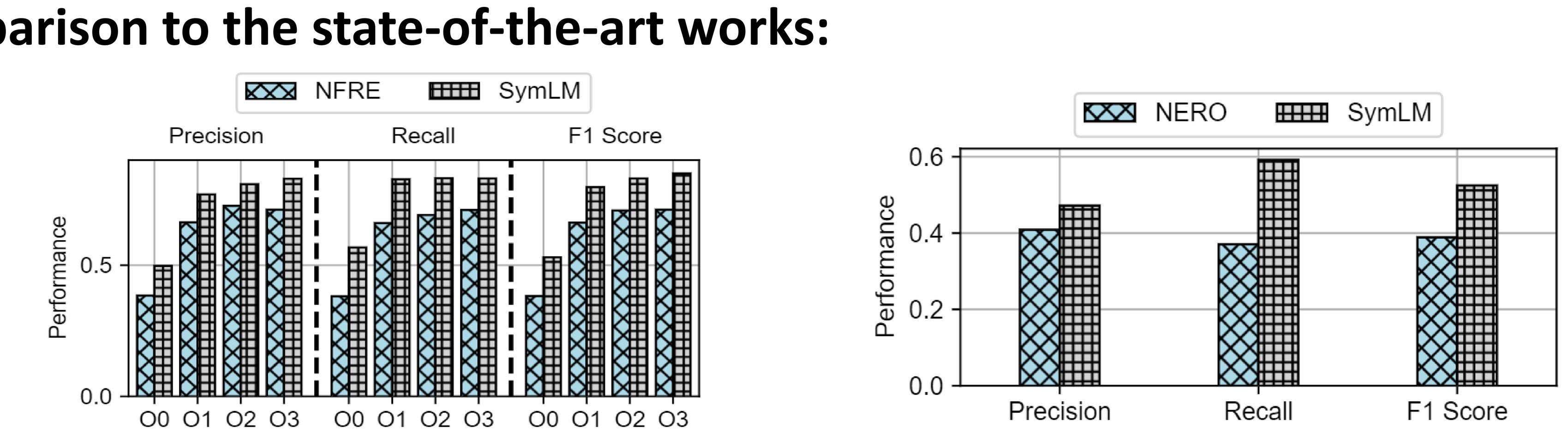
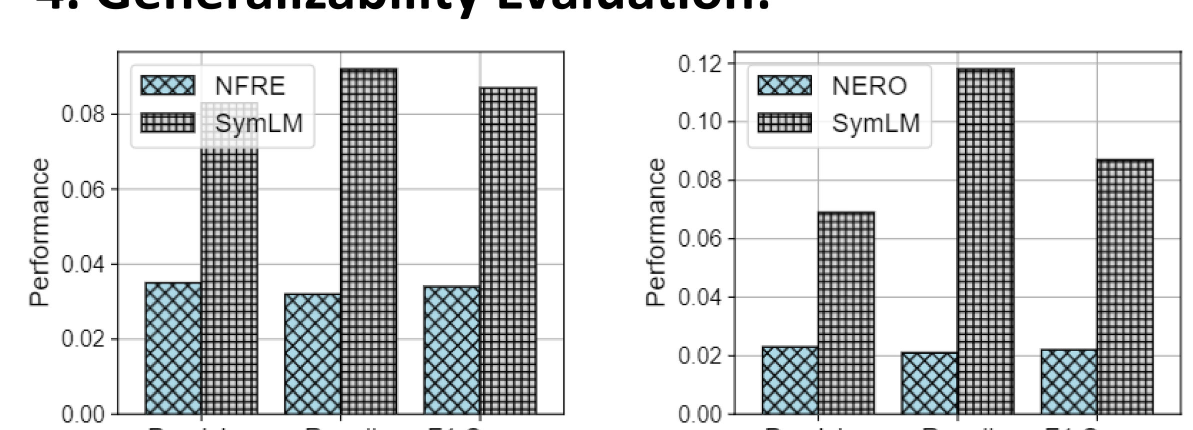


Fig. 5: Baseline Comparison

4. Generalizability Evaluation:



(a) SymLM v.s. NFRE (b) SymLM v.s. NERO

Fig. 6: Generalizability Comparison

5. Obfuscation Resistance:

Tab. 1: Performance (F1 Score) on Obfuscated (OBF) Binaries

OBF	SymLM	NFRE
None	0.806	0.595
bcf	0.757 (-6.1%)	0.491 (-17.5%)
cff	0.768 (-4.7%)	0.445 (-25.2%)
sub	0.726 (-9.9%)	0.505 (-15.2%)
split	0.788 (-2.2%)	0.496 (-16.6%)

Use Case Study

We show the practical use case of SymLM with **8 32-bit ARM IoT firmware images** with 18% unseen IoT-specific name words (e.g., analog).
Result: 172/1062 names correctly predicted.

```
uint32_t read(uint32_t ulPin){
  ...
  if (pin == NC) uVar3 = 0;
  else {
    uVar2 = read_value(pin);
    uVar3 = (uint32_t)uVar2;
    if (uVar4 != 0xc) {
      if ((uint) uVar4 < 0xc)
        return (uint)(uVar2 >> (0xcU - uVar4 & 0xff));
      return uVar3 << (uVar4 - 0xcU & 0xff);
    }
  }
  return uVar3;
}
```

Fig. 7 Example Prediction. The ground truth names are *analogRead* and *abc_read_value*.

Scan the QR code or visit https://github.com/OSUSecLab/SymLM to find our code.