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

Xin Jin[1] Kexin Pei[2] Jun Yeon Won[1] Zhiqiang Lin[1]

[1]The Ohio State University

[2]Columbia University

CCS 2022

The Need for Stripped Binary Analysis

- **Closed-source**: Commercial software shipped in stripped binaries.

The Need for Stripped Binary Analysis

- **Closed-source**: Commercial software shipped in stripped binaries.

```
void FUN_001092f3(byte *param_1) {
    byte *local_48;
    ulong local_40;
    ...
    if (*local_48 == 10) {
        local_48 = local_48 + 1;
    }
    else if (*local_48 == 0x3a) {
        DAT_00119470 = '\0';
        bVar3 = *local_48;
    }
    ...
}
```

The Need for Stripped Binary Analysis

- **Closed-source**: Commercial software shipped in stripped binaries.
- **Insecure**: Impactful vulnerabilities found in software binaries.

Critical Vulnerability Affects Millions of IoT Devices

CISA, Mandiant, and ThroughTek share the details of a vulnerability that could allow attackers to observe camera feeds and remotely control devices.

Kelly Sheridan
Senior Editor

August 17, 2021

Metamorworks via Adobe Stock

Existing Approaches

Binary Analysis

1. Control flow analysis [GH19].
2. Decompilation [BPKV22].

Existing Approaches

Binary Analysis

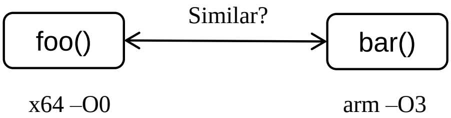
1. Control flow analysis [GH19].
2. Decompilation [BPKV22].
3. Taint analysis [CLZ21].
4. Symbolic execution [DBR20].

Existing Approaches

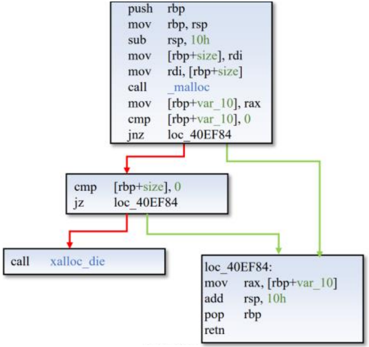
- **Key objectives**: understanding, analyzing, and answering questions about program behavior and semantics.

Existing Approaches

- **Key objectives**: understanding, analyzing, and answering questions about program behavior and semantics.



buf overflow?



Malicious?

Predicting binary function names is extremely useful

- Function names: summary of <u>function behavior and semantics</u>.

Predicting binary function names is extremely useful

- Function names: summary of function behavior and semantics.

```
void FUN_001092f3(byte *param_1) {
    byte *local_48;
    ulong local_40;
    ...
    if (*local_48 == 10) {
        local_48 = local_48 + 1;
    }
    else if (*local_48 == 0x3a) {
        DAT_00119470 = '\0';
        bVar3 = *local_48;
    }
    ...
}
```

Predicting binary function names is extremely useful

- Function names: summary of function behavior and semantics.

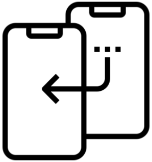
```
void FUN_001092f3(byte *param_1) {
    byte *local_48;
    ulong local_40;
    ...
    if (*local_48 == 10) {
        local_48 = local_48 + 1;
    }
    else if (*local_48 == 0x3a) {
        DAT_00119470 = '\0';
        bVar3 = *local_48;
    }
    ...
}
```

Function Name Prediction

```
void DNS_flood(byte *param_1) {
    byte *local_48;
    ulong local_40;
    ...
    if (*local_48 == 10) {
        local_48 = local_48 + 1;
    }
    else if (*local_48 == 0x3a) {
        DAT_00119470 = '\0';
        bVar3 = *local_48;
    }
    ...
}
```

Predicting binary function names is extremely useful

- Fundamental applications

Malware Analysis	Vulnerability Detection	Clone Identification	Program Comprehension
			

Predicting binary function names is very challenging

Challenges

1. **Missing Semantics**. Very limited semantic information.

Predicting binary function names is very challenging

Challenges

1. **Missing Semantics**. Very limited semantic information.

Missing Semantics

Names of **identifiers** and **function parameters** use the same words as function names [LWN21].

Predicting binary function names is very challenging

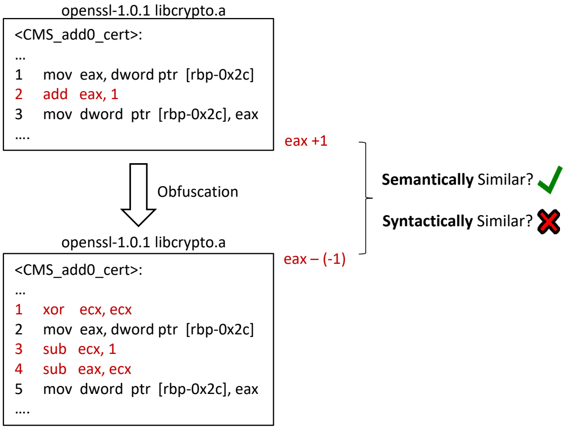
Challenges

1. **Missing Semantics**. Very limited semantic information.
2. **Binary Variation**. Semantically similar code appearing differently.

Predicting binary function names is very challenging

Challenges

1. **Missing Semantics**. Very limited semantic information.
2. **Binary Variation**. Semantically similar code appearing differently.



Predicting binary function names is very challenging

Challenges

1. **Missing Semantics**. Very limited semantic information.
2. **Binary Variation**. Semantically similar code appearing differently.
3. **Noisy Function Names**. Different developers naming functions differently.

Predicting binary function names is very challenging

Challenges

1. **Missing Semantics**. Very limited semantic information.
2. **Binary Variation**. Semantically similar code appearing differently.
3. **Noisy Function Names**. Different developers naming functions differently.

Reason

- Synonyms and abbreviations are ubiquitous in function names.
- Even single letters can be meaningful when probably used [BGOF17].
- Probability (two developers select same names for the same function) = **6.9%** [FMN+20].

Predicting binary function names is very challenging

Challenges

1. **Missing Semantics**. Very limited semantic information.
2. **Binary Variation**. Semantically similar code appearing differently.
3. **Noisy Function Names**. Different developers naming functions differently.
4. **OOV Issues**. Out-of-vocabulary words widely used.

Predicting binary function names is very challenging

Challenges

1. **Missing Semantics**. Very limited semantic information.
2. **Binary Variation**. Semantically similar code appearing differently.
3. **Noisy Function Names**. Different developers naming functions differently.
4. **OOV Issues**. Out-of-vocabulary words widely used.

OOV Words

	Category	**Ratio**	**Examples**
1	Abbreviation concatenation	29.9%	statinfo, streq
2	Clean word concatenation	22.3%	sharefile, startpoints
3	Misspelling	14.6%	anewer, tac, sb
4	Clean word	12.1%	dependent, specifer
5	Abbreviation	7.0%	utils, pred
6	Inflection	9.6%	addresses, using
7	Digits in word	4.5%	add32, merge2

Predicting binary function names is very challenging

Challenges

1. **Missing Semantics**. Very limited semantic information.
2. **Binary Variation**. Semantically similar code appearing differently.
3. **Noisy Function Names**. Different developers naming functions differently.
4. **OOV Issues**. Out-of-vocabulary words widely used.
5. **Comprehensive semantic modeling**. Semantics preserved in calling context.

Predicting binary function names is very challenging

Challenges

1. **Missing Semantics**. Very limited semantic information.
2. **Binary Variation**. Semantically similar code appearing differently.
3. **Noisy Function Names**. Different developers naming functions differently.
4. **OOV Issues**. Out-of-vocabulary words widely used.
5. **Comprehensive semantic modeling**. Semantics preserved in calling context.

```
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;
}
```

Prior Works and Limitations

- Machine learning is promising.

OpenAI DALLE-2

GitHub Copilot

test.js > findHighestNumber

```
function findHighestNumber(array) {
    var highestNumber = 0;
    for (var i = 0; i < array.length; i++) {
        if (array[i] > highestNumber) {
            highestNumber = array[i];
        }
    }
    return highestNumber;
}
```

Prior Works and Limitations

- Existing works treat binary code as natural language sequence.

Prior Works and Limitations

- Existing works treat binary code as natural language sequence.
- Basis of natural language processing:

"My pet is a cat."

Context effect: P("cat"|"pet") >> P("a"|"pet").

Prior Works and Limitations

- Existing works treat binary code as natural language sequence.
- Basis of natural language processing:

"My pet is a cat."

Context effect: P("cat"|"pet") >> P("a"|"pet").

- Context effect not holding in binary code:

"mov eax, ebx"

P("ebx"|"eax") == P("mov"|"eax").

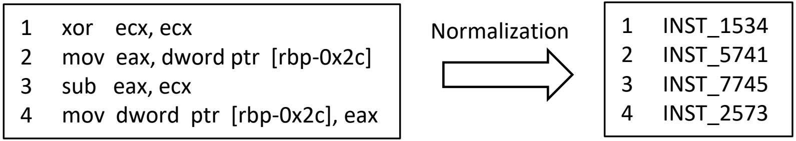
Prior Works and Limitations

- Limited features, e.g.,
  1. **handcrafted features**: Debin [HIT+18] and Punstrip [PECK20].

Feature	Type	Description
	Static features	
Size	Scalar	Size of the symbol in bytes.
Hash	Binary	SHA-256 hash of the binary data.
Opcode Hash	Binary	SHA-256 hash of the opcodes.
VEX instructions	Scalar	Number of VEX IR instructions.
VEX jumpkinds	Vector(8)	VEX IR jumps inside a function e.g. *fall-through*, *call*, *ret* and *jump*
VEX ordered jumpkinds	Vector(8)	A ordered list of VEX jumpkinds.
VEX temporary variables	Scalar	Number of temporary variables used in the VEX IR.
VEX IR Statements, Expressions and Operations	Vector(54)	Categorized VEX IR Statements, Expressions and Operations.
Callers	Vector(N)	Vector one-hot encoding representation of symbol callers.
Callees	Vector(N)	Vector one-hot encoding representation of symbol callees.
Transitive Closure	Vector(N)	Symbols reachable under this function.
Basic Block ICFG	Vector(300)	Graph2Vec vector representation of labeled ICFG.
VEX IR constants types and values	Dict	Number of type of VEX IR constants used.

Prior Works and Limitations

- Limited features, e.g.,
  1. **handcrafted features**: Debin [HIT+18] and Punstrip [PECK20].
  2. **partial function semantics**: NERO [DAY20] and NFRE [GCXZ21].



Preprocessing Step of NFRE

Key Observations and Insights

Predicting function names

- Program semantics is manifested in execution behavior.

Key Observations and Insights

openssl-1.0.1 libcrypto.a

```
<CMS_add0_cert>:
...
mov  eax, dword ptr  [rbp-0x2c]
add   eax, 1
mov  dword  ptr  [rbp-0x2c], eax
....
```

Syntax Different

openssl-1.0.1 libcrypto.a (obfuscated)

```
<CMS_add0_cert>:
...
xor    ecx, ecx
mov  eax, dword ptr  [rbp-0x2c]
sub   ecx, 1
sub   eax, ecx
mov  dword  ptr  [rbp-0x2c], eax
....
```

Key Observations and Insights

openssl-1.0.1 libcrypto.a

```
<CMS_add0_cert>:
...
mov  eax, dword ptr  [rbp-0x2c]
add   eax, 1
mov  dword  ptr  [rbp-0x2c], eax
....
```

Execution

Before: eax = 0
After: eax = 1

Syntax Different

Identical

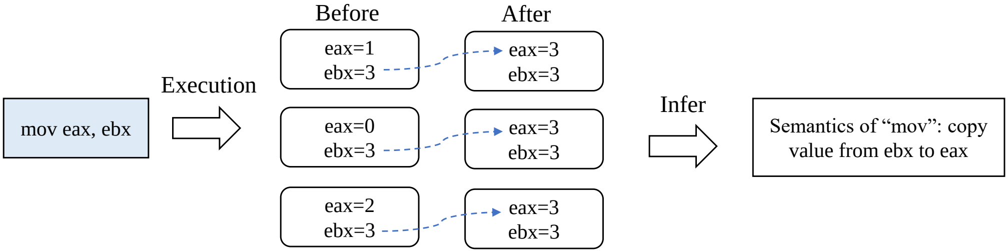
openssl-1.0.1 libcrypto.a (obfuscated)

```
<CMS_add0_cert>:
...
xor    ecx, ecx
mov  eax, dword ptr  [rbp-0x2c]
sub   ecx, 1
sub   eax, ecx
mov  dword  ptr  [rbp-0x2c], eax
....
```

Execution

Before: eax = 0
After: eax = 1

Key Observations and Insights



Key Observations and Insights

Predicting function names

- Program semantics is manifested in execution behavior.
- Learning semantics requires understanding both function instructions and calling context.

Key Observations and Insights

Predicting function names

- Program semantics is manifested in execution behavior.
- Learning semantics requires understanding both function instructions and calling context.
- Measuring function name semantics will be very helpful.

Problem Definition

- Problem formalization: multi-label multi-class classification task.

Problem Definition

- Problem formalization: multi-label multi-class classification task.
- Given function semantics ($\mathcal{E}$), instruction sequence ($\mathcal{I}$), and calling context ($\mathcal{C}$), we define composition function $\phi(\cdot)$:

$$E_{\mathcal{E}} = \phi(E_{\mathcal{C}}, E_{\mathcal{I}})$$

to map semantics to embedding space (E).

Problem Definition

- Problem formalization: multi-label multi-class classification task.
- Given function semantics ($\mathcal{E}$), instruction sequence ($\mathcal{I}$), and calling context ($\mathcal{C}$), we define composition function $\phi(\cdot)$:

$$E_{\mathcal{E}} = \phi(E_{\mathcal{C}}, E_{\mathcal{I}})$$

to map semantics to embedding space (E).

- Learning objective:

$$W = \Gamma(E_{\mathcal{E}})$$

where function name $W = \{w_1, w_2..., w_n\}$.

System Workflow: Step 1

- Problem formalization: multi-label multi-class classification task.
- Given function semantics ($\mathcal{E}$), instruction sequence ($\mathcal{I}$), and calling context ($\mathcal{C}$), we define composition function $\phi(\cdot)$:

$$E_{\mathcal{E}} = \phi(E_{\mathcal{C}}, E_{\mathcal{I}})$$

to map semantics to embedding space (E).

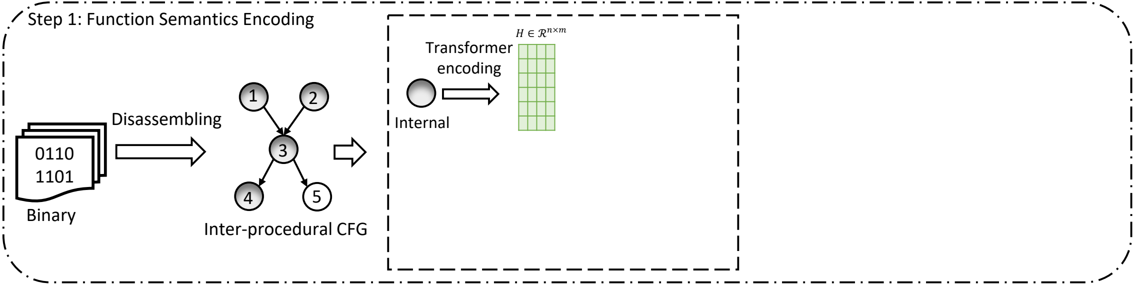
- Learning objective:

$$W = \Gamma(E_{\mathcal{E}})$$

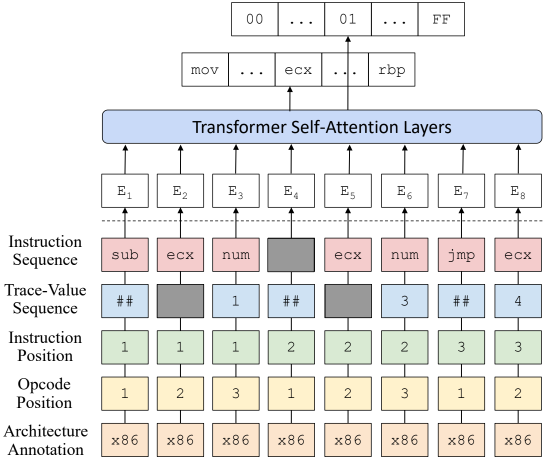
where function name $W = \{w_1, w_2 ..., w_n\}$.

System Workflow: Step 1

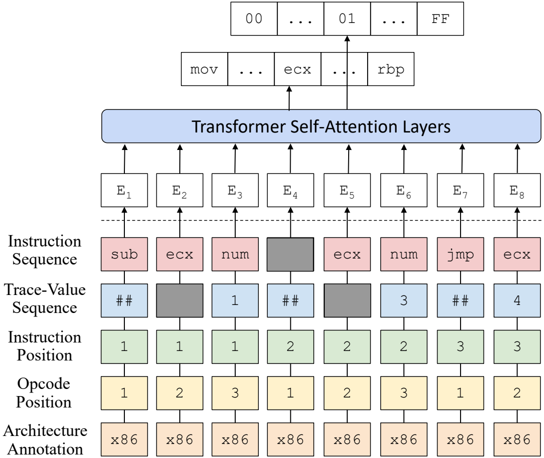
System Workflow: Step 1

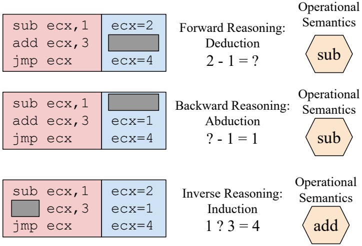


System Workflow: Step 1



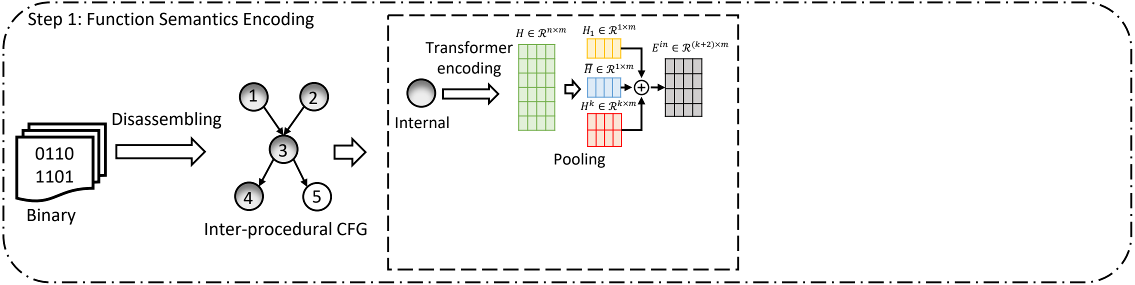
System Workflow: Step 1



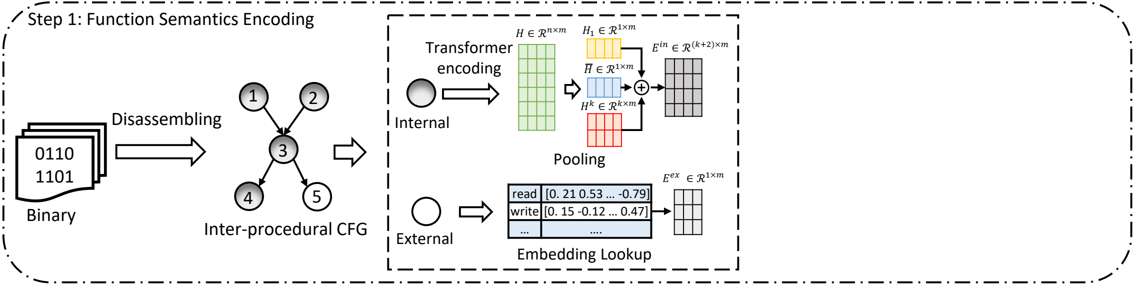


Pretraining Intuition

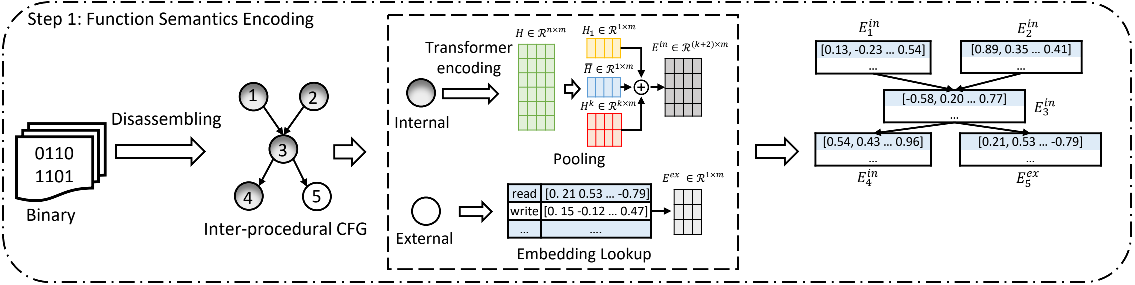
System Workflow: Step 1



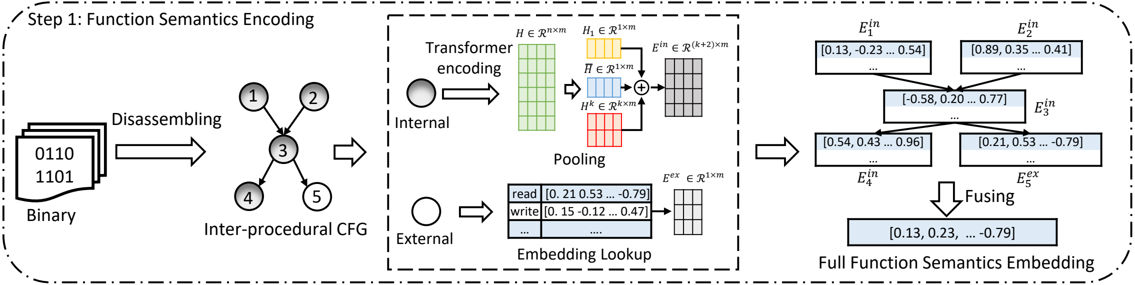
System Workflow: Step 1



System Workflow: Step 1



System Workflow: Step 1



System Workflow: Step 2

- Problem formalization: multi-label multi-class classification task.
- Given function semantics ($\mathcal{E}$), instruction sequence ($\mathcal{I}$), and calling context ($\mathcal{C}$), we define composition function $\phi(\cdot)$:

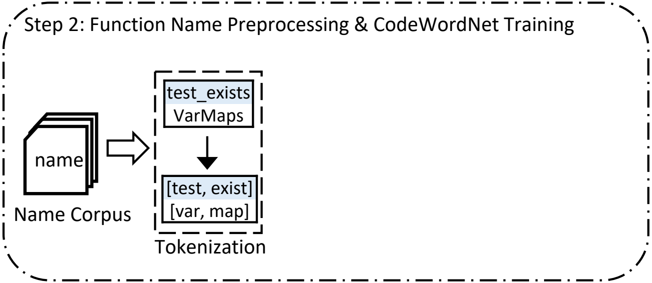
$$E_{\mathcal{E}} = \phi(E_{\mathcal{C}}, E_{\mathcal{I}})$$

  to map semantics to embedding space (E).
- Learning objective:

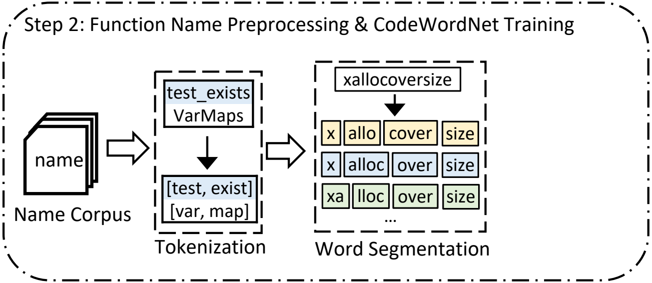
$$W = \Gamma(E_{\mathcal{E}})$$

  where function name $W = \{w_1, w_2 ..., w_n\}$.

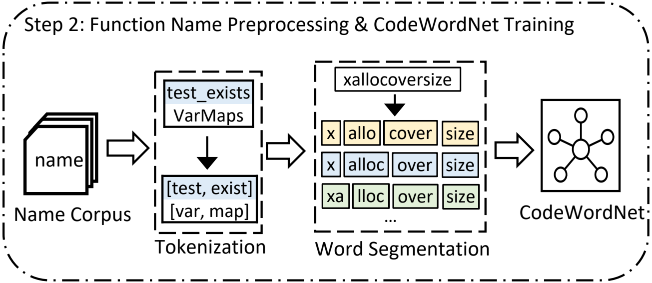
System Workflow: Step 2



System Workflow: Step 2

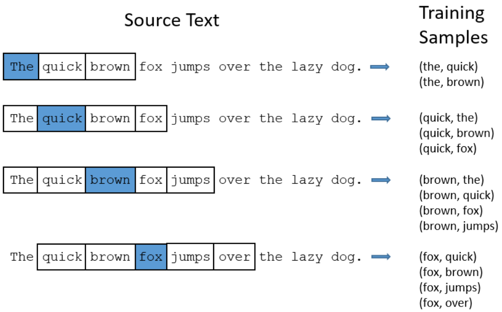


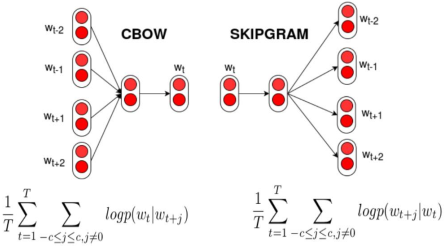
System Workflow: Step 2



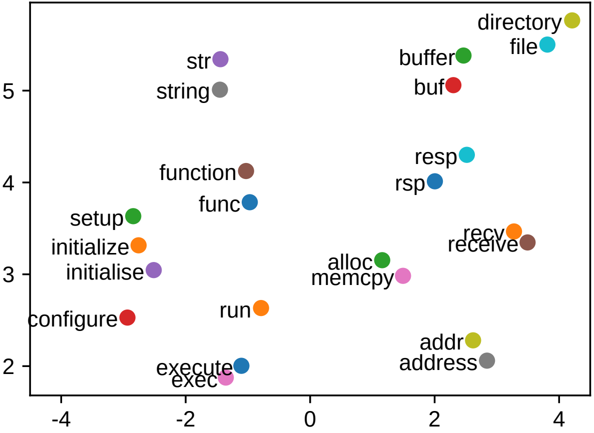
System Workflow: Step 2

- CodeWordNet: word embeddings [MSC+13]





System Workflow: Step 2



System Workflow: Step 3

- Problem formalization: multi-label multi-class classification task.
- Given function semantics ($\mathcal{E}$), instruction sequence ($\mathcal{I}$), and calling context ($\mathcal{C}$), we define composition function $\phi(\cdot)$:

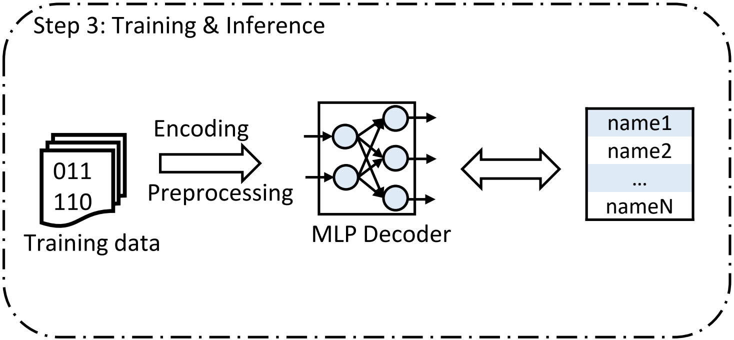
$$E_{\mathcal{E}} = \phi(E_{\mathcal{C}}, E_{\mathcal{I}})$$

to map semantics to embedding space (E).
- Learning objective:

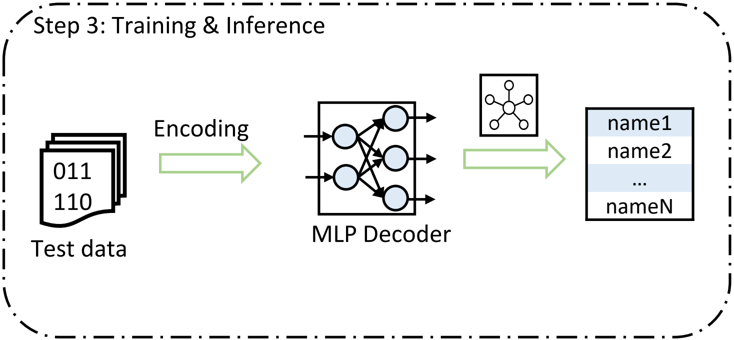
$$W = \Gamma(E_{\mathcal{E}})$$

where function name $W = \{w_1, w_2..., w_n\}$.

System Workflow: Step 3



System Workflow: Step 3



Evaluation Setup

1. **Dataset**: 16,027 different binaries and 1,431,169 functions

Evaluation Setup

1. **Dataset**: 16,027 different binaries and 1,431,169 functions
   - 27 open-source projects.

Evaluation Setup

1. **Dataset**: 16,027 different binaries and 1,431,169 functions
   - 27 open-source projects.
   - 4 architectures: x86, x64, ARM, and MIPS.

Evaluation Setup

1. **Dataset**: 16,027 different binaries and 1,431,169 functions
   - 27 open-source projects.
   - 4 architectures: x86, x64, ARM, and MIPS.
   - 4 optimization levels: O0, O1, O2, and O3 (GCC-7.5).

Evaluation Setup

1. **Dataset**: 16,027 different binaries and 1,431,169 functions
   - 27 open-source projects.
   - 4 architectures: x86, x64, ARM, and MIPS.
   - 4 optimization levels: O0, O1, O2, and O3 (GCC-7.5).
   - 4 obfuscation options: bcf, cff, sub, and split (LLVM obfuscator).

Evaluation Setup

1. **Dataset**: 16,027 different binaries and 1,431,169 functions
   - 27 open-source projects.
   - 4 architectures: x86, x64, ARM, and MIPS.
   - 4 optimization levels: O0, O1, O2, and O3 (GCC-7.5).
   - 4 obfuscation options: bcf, cff, sub, and split (LLVM obfuscator).
2. Baselines: NERO [DAY20] and NFRE [GCXZ21].

Evaluation Setup

1. **Dataset**: 16,027 different binaries and 1,431,169 functions
   - 27 open-source projects.
   - 4 architectures: x86, x64, ARM, and MIPS.
   - 4 optimization levels: O0, O1, O2, and O3 (GCC-7.5).
   - 4 obfuscation options: bcf, cff, sub, and split (LLVM obfuscator).
2. Baselines: NERO [DAY20] and NFRE [GCXZ21].
3. Metrics: precision, recall, and F1-score.

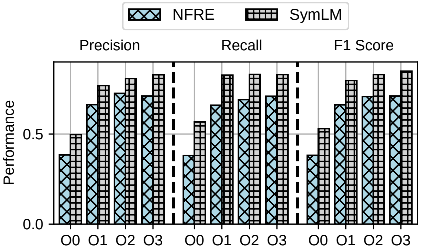
Overall Performance

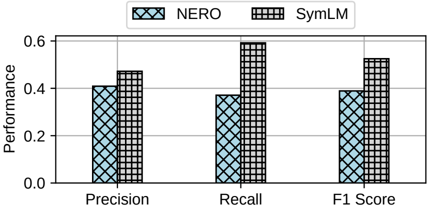
- SymLM achieves 0.634 precision, 0.677 recall, and 0.655 F1 score on average.

ARCH	OPT	Precision	Recall	F1 Score
x86	O0	0.637	0.646	0.642
	O1	0.682	0.702	0.692
	O2	0.744	0.829	0.784
	O3	0.783	0.833	0.807
x64	O0	0.497	0.567	0.530
	O1	0.769	0.827	0.797
	O2	0.808	0.831	0.830
	O3	0.829	0.830	0.849
arm	O0	0.446	0.494	0.469
	O1	0.611	0.681	0.644
	O2	0.672	0.717	0.694
	O3	0.646	0.689	0.667
mips	O0	0.453	0.511	0.480
	O1	0.507	0.529	0.518
	O2	0.724	0.790	0.755
	O3	0.563	0.588	0.575

Baseline Comparison

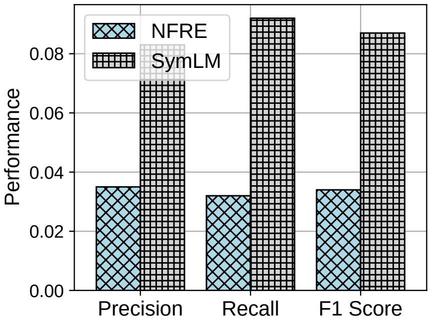
- SymLM outperforms the state-of-the-art works (up to 35% improvement on F1 score).

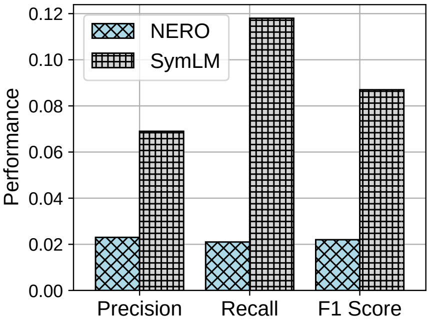




Generalizability

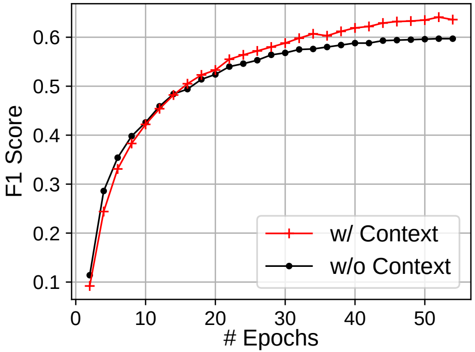
- SymLM is more generalizable to unseen binary functions (295.5% better F1 score).





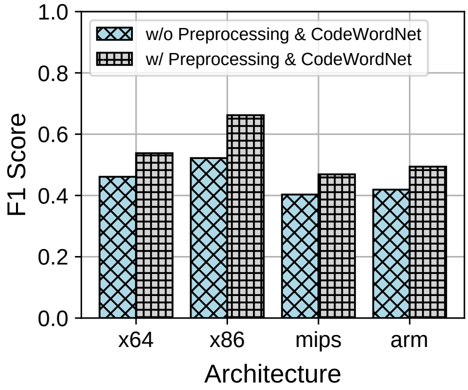
Component Effectiveness: Calling Context Modeling

- Learning calling context semantics improves SymLM's performance by 7.9%.



Component Effectiveness: Preprocessing and CodeWordNet

- Preprocessing and CodeWordNet boost SymLM's performance by 16.7%.



Use Case

- ▶ SymLM successfully infers function semantics of IoT firmware image [Gat].

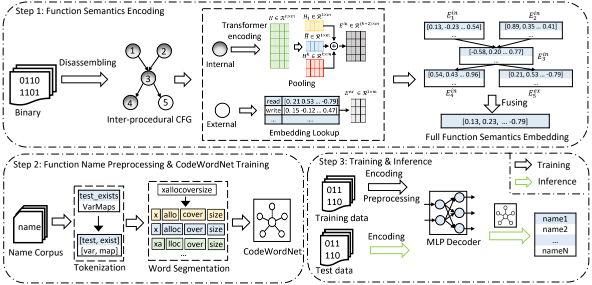
```c
uint32_t analogRead(uint32_t ulPin){
  ...
  if (pin == NC) uVar3 = 0;
  else {
    uVar2 = adc_read_value(pin);
    uVar3 = (uint32_t)uVar2;
    if (uVar4 != 0xc) {
      if ((uint) uVar4 < 0xc)
        return (uint)(uVar2 >> (0xcU - uVar4 & 0xff));
      return uVar3 << (uVar4 - 0xcU & 0xff);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

Ground Truth

```c
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;
}
```

Prediction

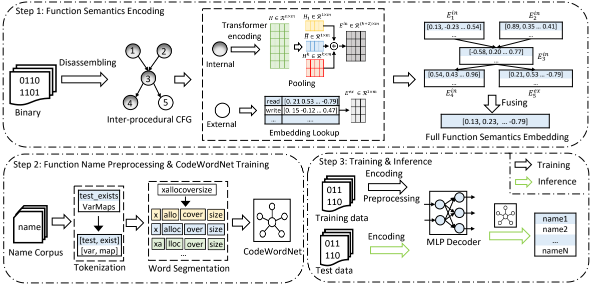
Takeaway



SymLM

- A novel neural architecture that generates execution-aware context-sensitive code embeddings.
- Effective modules, function name preprocessing and CodeWordNet, to calculate function name similarity.
- Advancing the state-of-the-art and practical use cases.

Takeaway



SymLM

- A novel neural architecture that generates execution-aware context-sensitive code embeddings.
- Effective modules, function name preprocessing and CodeWordNet, to calculate function name similarity.
- Advancing the state-of-the-art and practical use cases.

The source code is available at https://github.com/OSUSecLab/SymLM.

References I

- Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach, and Dror G Feitelson, *Meaningful identifier names: the case of single-letter variables*, 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), IEEE, 2017, pp. 45–54.
- Kevin Burk, Fabio Pagani, Christopher Kruegel, and Giovanni Vigna, *Decomperson: How humans decompile and what we can learn from it*, 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 2765–2782.
- Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang, *Selectivetaint: Efficient data flow tracking with static binary rewriting*, 30th USENIX Security Symposium (USENIX Security 21), USENIX Association, August 2021.
- Yaniv David, Uri Alon, and Eran Yahav, *Neural reverse engineering of stripped binaries using augmented control flow graphs*, Proceedings of the ACM on Programming Languages **4** (2020), no. OOPSLA, 1–28.
- Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk, *Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level*, 2020 IEEE Symposium on Security and Privacy (SP), IEEE, 2020, pp. 1021–1038.
- Dror Feitelson, Ayelet Mizrahi, Nofar Noy, Aviad Ben Shabat, Or Eliyahu, and Roy Sheffer, *How developers choose names*, IEEE Transactions on Software Engineering (2020).
- *Gateway*, `https://github.com/RiS3-Lab/p2im-real_firmware/blob/master/binary/Gateway`, Accessed: 2022-04-26.
- Han Gao, Shaoyin Cheng, Yinxing Xue, and Weiming Zhang, *A lightweight framework for function name reassignment based on large-scale stripped binaries*, Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2021, pp. 607–619.

References II

- Masoud Ghaffarinia and Kevin W Hamlen, *Binary control-flow trimming*, Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, 2019, pp. 1009–1022.
- Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev, *Debin: Predicting debug information in stripped binaries*, Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 1667–1680.
- Yi Li, Shaohua Wang, and Tien N Nguyen, *A context-based automated approach for method name consistency checking and suggestion*, 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp. 574–586.
- Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean, *Distributed representations of words and phrases and their compositionality*, Advances in neural information processing systems **26** (2013).
- James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder, *Probabilistic naming of functions in stripped binaries*, Annual Computer Security Applications Conference, 2020, pp. 373–385.