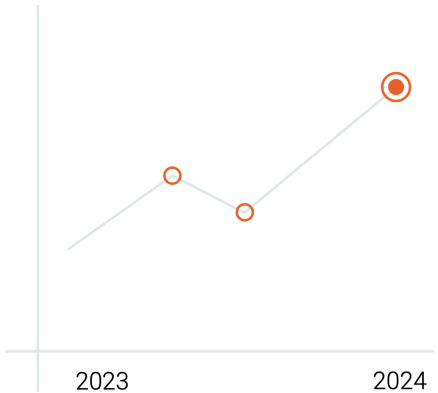
Unleashing the Power of Generative Model in Recovering Variable Names from Stripped Binary

Xiangzhe Xu, Zhuo Zhang, Zian Su, Ziyang Huang, Shiwei Feng, Yapeng Ye, Nan Jiang, Danning Xie, Siyuan Cheng, Lin Tan, Xiangyu Zhang

PURDUE
UNIVERSITY

Reverse Engineering are Important



Rapid7 Labs tracked

5,600+ ransomware incidents between January 2023 and February 2024*

*This number doesn't reflect incidents that go unreported

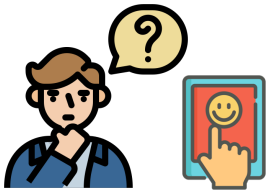
Motivation: Reverse Engineering for All

- **Democratize reverse engineering:** Unlock insights for everyone.

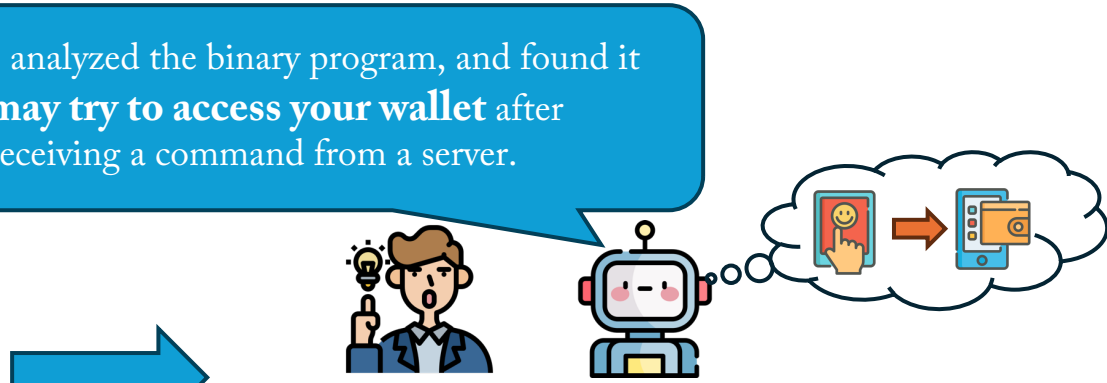
Current state: A normal user relies on **third-party certification** to understand the intention of a program.

Motivation: Reverse Engineering for All

- **Democratize reverse engineering:** Unlock insights for everyone.



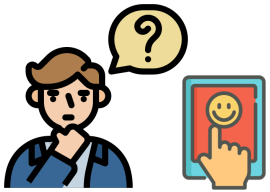
Current state: A normal user relies on **third-party certification** to understand the intention of a program.



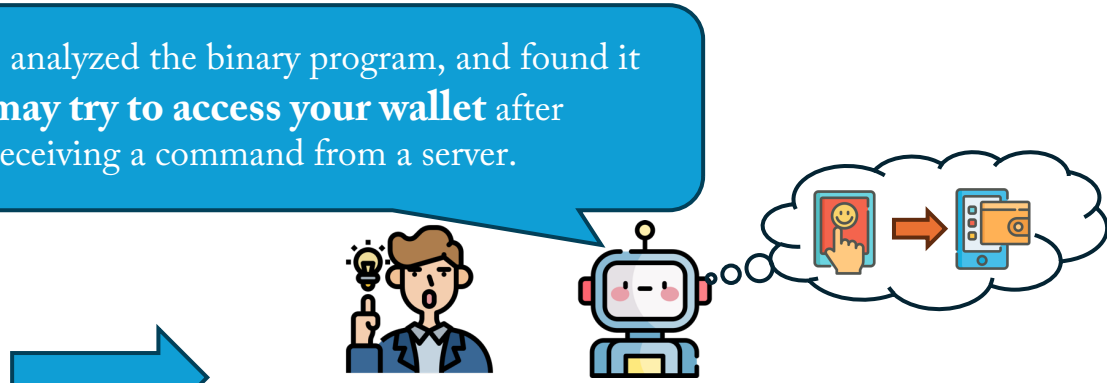
Democratized reverse engineering: **Everyone** has the right to know what is running on their machine, presented in **accessible natural language.**

Motivation: Reverse Engineering for All

- **Democratize reverse engineering:** Unlock insights for everyone.
- **Empower experts:** Analyze unknown binaries with less efforts.
- **Our approach:** Reconstruct high-quality symbol information.



Current state: A normal user relies on **third-party certification** to understand the intention of a program.



Democratized reverse engineering: **Everyone** has the right to know what is running on their machine, presented in **accessible natural language.**

RE for All – Democratize RE

- *Everyone* should be able to understand programs executed on their machine.
- Recent advances in AI agents enable accessible insights for executable programs.
- Symbol information is essential for AI-driven binary program analysis.

RE for All – Democratize RE

- *Everyone* should be able to understand programs executed on their machine.
- Recent advances in AI agents enable accessible insights for executable programs.
- Symbol information is essential for AI-driven binary program analysis.

```
int64 sub_401580(int64 img, int w, int h) {
 int64 result;
 *(int *)img = abs32(w);
 *(int *)(img + 4) = abs32(h);
 *(int *)(img + 8) = 4 *
      (3 * (*(int *)img + 1) / 4);
 *(int64 *)(img + 16) =
      malloc(*(int *)(img + 8) * (int64)h);
 if (*(int64 *)(img + 16)) result = 0;
 else result = -1;
 return result;
}
```

A decompiled binary function with reconstructed variable names highlighted

1 This function, ***bmp_create***, initializes an image structure with specified width ***w*** and height ***h***.
2 It sets the width and height of the image structure to the absolute values of ***w*** and ***h***,
3 calculates the stride (which is the aligned width considering a 4-byte boundary) …

GPT4 summary for the corresponding source code function.

RE for All – Democratize RE

- *Everyone* should be able to understand programs executed on their machine.
- Recent advances in AI agents enable accessible insights for executable programs.
- Symbol information is essential for AI-driven binary program analysis.

No context about "image" in 1.

No high-level denotations "width and height" in 2.

No key constraints on "4-byte aligned" boundary.

```
}
```

A decompiled binary function with reconstructed variable names highlighted

1 This function, ***bmp_create***, initializes an image structure with specified width ***w*** and height ***h***.
2 It sets the width and height of the image structure to the absolute values of ***w*** and ***h***,
3 calculates the stride (which is the aligned width considering a 4-byte boundary) …

GPT4 summary for the corresponding source code function.

1 This function, ***sub_401580***, takes three args and performs a series of operations on them.
2 It calculates the absolute values of a2 and a3 and stores them at specific memory locations relative to a1.
3 It then calculates a value based on these absolute values and stores it at another location.

GPT4 summary for the decompiled function without reconstructed names.

RE for All – Democratize RE

- *Everyone* should be able to understand programs executed on their machine.
- Recent advances in AI agents enable accessible insights for executable programs.
- Symbol information is essential for AI-driven binary program analysis.

No context about "image" in 1.

No high-level denotations "width and height" in 2.

No key constraints on "4-byte aligned" boundary.

```
}
```

1 This function, ***bmp_create***, initializes an image structure with specified width ***w*** and height ***h***.
2 It sets the width and height of the image structure to the absolute values of ***w*** and ***h***,
3 calculates the stride (which is the aligned width considering a 4-byte boundary) …

GPT4 summary for the corresponding source code function.

1 This function, ***sub_401580***, takes three args and performs a series of operations on them.
2 It calculates the absolute values of a2 and a3 and stores them at specific memory locations relative to a1.
3 It then calculates a value based on these absolute values and stores it at another location.

GPT4 summary for the decompiled function without reconstructed names.

Similar quality to source code summary.

With high-level contexts and abstractions.

1 This function, ***img_init***, initializes an image structure. It takes three parameters: a pointer to the image structure ***img***, and two integers ***w*** and ***h*** for width and height.
2 The width and height are stored as absolute values in the image structure.
3 It then calculates and stores the row size, aligning it to a 4-byte boundary.

GPT4 summary for the decompiled function with reconstructed names.

RE for All – Empower Experts

- Symbol information can help reverse engineer identify relevant functions and understand code with less efforts.
- A concrete example: reconstructed symbols in a C2 client.

```
1  for (j = (const char **)&unk_60A500;
2        *j; j = v66 + 2) {
3    v73 = j;
4    v65 = strcasecmp(*j, &s2);
5    v66 = v73;
6    if (!v65) {
7     ((void(*)(int64, char *, char *))v73[1])
                                (fd, &dest, i);
8  }}}
```

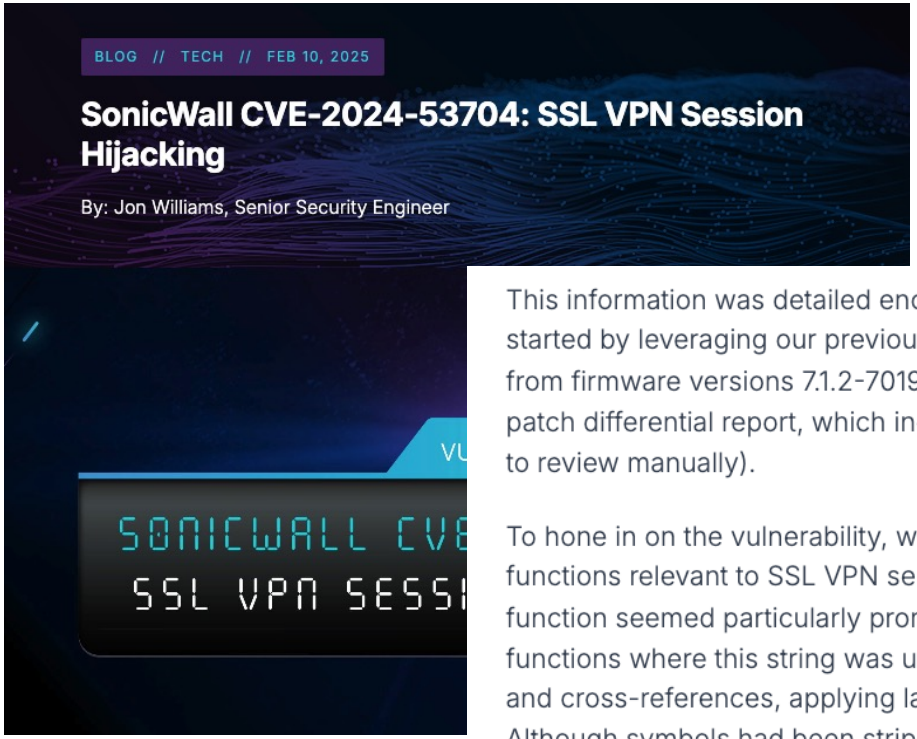
Decompiled code snippet in a C2 client.

```
1  for (cmd_ptr = (const char **) &cmd_table;
2          *cmd_ptr; cmd_ptr = cmd_tmp + 2) {
3    matched_cmd = cmd_ptr;
4    cmp = strcasecmp(*cmd_ptr, &cmd);
5    cmd_tmp = matched_cmd;
6    if (!cmp) {
7      ((void(*)(int64, char *, char *))
            matched_cmd[1])(fd, &dest, tok);
8  }}}
```

The same code snippets with reconstructed names.

RE for All – Empower Experts

RE for All – Empower Experts



This information was detailed enough to give us a solid lead on finding the bug. We started by leveraging our previous research to decrypt and extract the **sonicosv** binary from firmware versions 7.1.2-7019 and 7.1.3-7015. We then used BinDiff to generate a patch differential report, which included a large number of changed functions (too many to review manually).

To hone in on the vulnerability, we searched strings within the unpatched binary to find functions relevant to SSL VPN session cookies. The **getSslvpnSessionFromCookie** function seemed particularly promising, so we traced cross-references to identify the functions where this string was used. We slowly dug through the web of function calls and cross-references, applying labels as we went to help us make sense of the code. Although symbols had been stripped from the binary, log messages were often used to

Challenge1: Generalize to Unseen Names

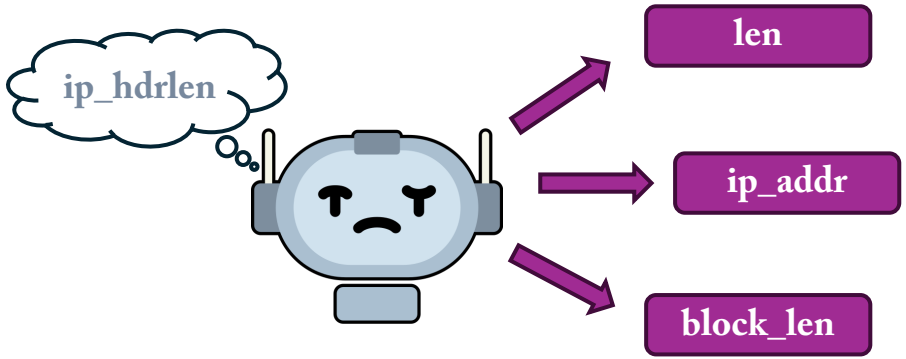
- Classification-based name recovery fails to generalize to unseen names.
- A classifier essentially selects the closest match from its training data.

Challenge1: Generalize to Unseen Names

- Classification-based name recovery fails to generalize to unseen names.
- A classifier essentially selects the closest match from its training data.
- A concrete example: Predicting `ip_hdrlen`.
  - Let's suppose a well-trained classification model.
  - It attempts to choose name for a variable with ground truth name `ip_hdrlen`.
  - But this name is not in the training dataset.

Challenge1: Generalize to Unseen Names

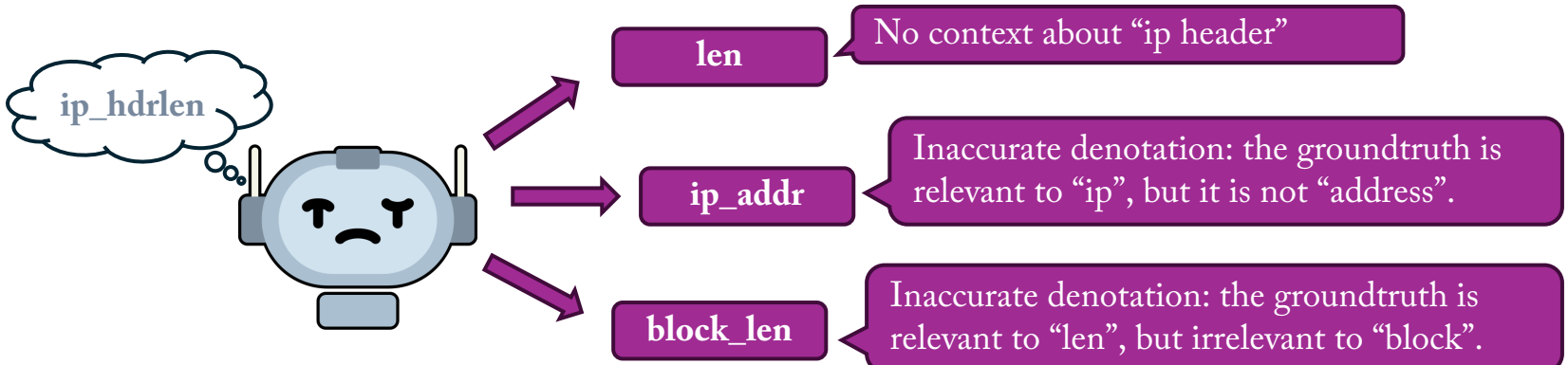
- Classification-based name recovery fails to generalize to unseen names.
- A classifier essentially selects the closest match from its training data.
- A concrete example: Predicting `ip_hdrlen`.
  - Let's suppose a well-trained classification model.
  - It attempts to choose name for a variable with ground truth name `ip_hdrlen`.
  - But this name is not in the training dataset.



A classification model fails to find the best fit name from the training dataset.

Challenge1: Generalize to Unseen Names

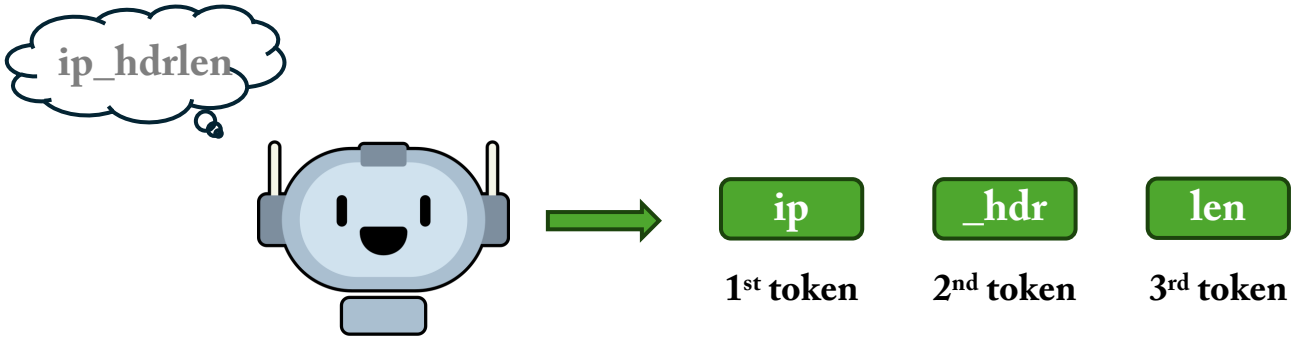
- Classification-based name recovery fails to generalize to unseen names.
- A classifier essentially selects the closest match from its training data.
- A concrete example: Predicting `ip_hdrlen`.
  - Let's suppose a well-trained classification model.
  - It attempts to choose name for a variable with ground truth name `ip_hdrlen`.
  - But this name is not in the training dataset.



A classification model fails to find the best fit name from the training dataset.

Solution1: Compose Names from Tokens

- Insight: Most unseen names are composed from common names.
- We leverage generative models to compose names from tokens.



A generative model composes an unseen name from common tokens.

Challenge2: Reason Programs with Limited Contextual Information

- Binary programs has limited contextual information due to the lack of meaningful names from calling contexts.

```
0 int send_packet(...) {
  ...
1 memset(packet, 0, sizeof(packet));
2 struct iphdr* ip_hdr =(struct iphdr*)packet;
  ...
3 // Fill in the IP Header
4 memset(ip_hdr, 0, 0x3c);
5 ip_hdr->ihl = ip_hdrlen >> 2;
6 ip_hdr->tos = 0;
7 ip_hdr->id = htonl(...);
  ...
8 // Compute checksums
9 tcp_checksum(ip_hdr,ip_hdrlen,...);
  ...}
```

Source code of a function "send_packet".

```
A0int sub_401430(...){
  ...
A1 memset(s, 0, 0x400);
A2 v29 = s;
  ...

A3 memset(v29, 0, 0x3c);
A4 *v29 = (n >> 2) & 0xF;
A5 *((char*)v29+1) = 0;
A6 *((uint16*)v29+2) = htonl(...);
  ...

A8 sub_40197A(v29, n,...);
  ...}
```

The corresponding decompiled code.

Challenge2: Reason Programs with Limited Contextual Information

- Binary programs has limited contextual information due to the lack of meaningful names from calling contexts.

```
0 int send_packet(...) {
   ...
1  memset(packet, 0, sizeof(packet));
2  struct iphdr* ip_hdr =(struct iphdr*)packet;
   ...
3  // Fill in the IP Header
4  memset(ip_hdr, 0, 0x3c);
5  ip_hdr->ihl = ip_hdrlen >> 2;
6  ip_hdr->tos = 0;
7  ip_hdr->id = htonl(...);
   ...
8  // Compute checksums
9  tcp_checksum(ip_hdr,ip_hdrlen,...);
   ...}
```

Source code of a function "send_packet".

```
A0int sub_401430(...){
   ...
A1 memset(s, 0, 0x400);
A2 v29 = s;
   ...

A3 memset(v29, 0, 0x3c);
A4 *v29 = (n >> 2) & 0xF;
                          l(...);

A8 sub_40197A(v29, n,...);
   ...}
```

Information from calling contexts is missing

The corresponding decompiled code.

Challenge2: Reason Programs with Limited Contextual Information

- Binary programs has limited contextual information due to the lack of meaningful names from calling contexts.
- Directly prompting an LLM or a naïve SFT are sub-optimal.

```
0 int send_packet(...) {
  ...
1 memset(packet, 0, sizeof(packet));
2 struct iphdr* ip_hdr =(struct iphdr*)packet;
  ...
3 // Fill in the IP Header
4 memset(ip_hdr, 0, 0x3c);
5 ip_hdr->ihl = ip_hdrlen >> 2;
6 ip_hdr->tos = 0;
7 ip_hdr->id = htonl(...);
  ...
8 // Compute checksums
9 tcp_checksum(ip_hdr,ip_hdrlen,...);
  ...}
```

Source code of a function "send_packet".

```
A0int sub_401430(...){
  ...
A1 memset(s, 0, 0x400);
A2 v29 = s;
  ...

A3 memset(v29, 0, 0x3c);
A4 *v29 = (n >> 2) & 0xF;
                    l(...);

A8 sub_40197A(v29, n,...);
  ...}
```

Information from calling contexts is missing

The corresponding decompiled code.

Input to an LLM

```
int sub_401430(...){
 ...
 memset(s, 0, 0x400);
 v29 = s;
 ...
 memset(v29, 0, 0x3c);
 *v29 = (n >> 2) & 0xF;
 *((char*)v29+1) = 0;
 *((uint16*)v29+2)
            = htonl(...);
 sub_40197A(v29, n,...);
 ...}

Q: v29 , s , n
```

Expected output

```
A: v29 -> ip_hdr
   s -> packet
   n -> ip_hdrlen
```

A naïve data sample to SFT an LLM.

Challenge2: Reason Programs with Limited Contextual Information

- Binary programs has limited contextual information due to the lack of meaningful names from calling contexts.
- Directly prompting an LLM or a naïve SFT are sub-optimal.

```
0 int send_packet(...) {
  ...
1 memset(packet, 0, sizeof(packet));
2 struct iphdr* ip_hdr =(struct iphdr*)packet;
  ...
3 // Fill in the IP Header
4 memset(ip_hdr, 0, 0x3c);
5 ip_hdr->ihl = ip_hdrlen >> 2;
6 ip_hdr->tos = 0;
7 ip_hdr->id = htonl(...);
  ...
8 // Compute checksums
9 tcp_checksum(ip_hdr,ip_hdrlen,...);
  ...}
```

Source code of a function "send_packet".

```
A0int sub_401430(...){
  ...
A1 memset(s, 0, 0x400);
A2 v29 = s;
  ...

A3 memset(v29, 0, 0x3c);
A4 *v29 = (n >> 2) & 0xF;

A8 sub_40197A(v29, n,...);
  ...}
```

The corresponding decompiled code.

Information from calling contexts is missing

A model can hardly associate v29 with ip_hdr.

Input to an LLM

```
int sub_401430(...){
 ...
 memset(s, 0, 0x400);
 v29 = s;
 ...
 memset(v29, 0, 0x3c);
 *v29 = (n >> 2) & 0xF;
 *((char*)v29+1) = 0;
 *((uint16*)v29+2)
          = htonl(...);
 sub_40197A(v29, n,...);
 ...}

Q: v29 , s , n
```

Expected output

```
A: v29 -> ip_hdr
   s -> packet
   n -> ip_hdrlen
```

A naïve data sample to SFT an LLM.

Solution2: Fine-tune with Contextual Info

- Context-aware fine-tuning: use the name hints from the calling context.

```
0 int send_packet(...) {
  ...
1 memset(packet, 0, sizeof(packet));
2 struct iphdr* ip_hdr =(struct iphdr*)packet;
  ...
3 // Fill in the IP Header
4 memset(ip_hdr, 0, 0x3c);
5 ip_hdr->ihl = ip_hdrlen >> 2;
6 ip_hdr->tos = 0;
7 ip_hdr->id = htonl(...);
  ...
8 // Compute checksums
9 tcp_checksum(ip_hdr,ip_hdrlen,...);
  ...}
```

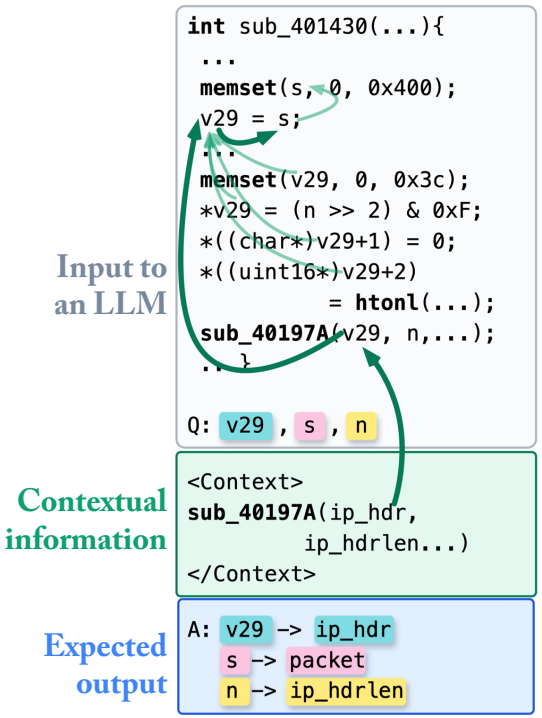
Source code of a function "send_packet".

```
A0int sub_401430(...){
  ...
A1 memset(s, 0, 0x400);
A2 v29 = s;
  ...

A3 memset(v29, 0, 0x3c);
A4 *v29 = (n >> 2) & 0xF;
A5 *((char*)v29+1) = 0;
A6 *((uint16*)v29+2) = htonl(...);
  ...

A8 sub_40197A(v29, n,...);
  ...}
```

The corresponding decompiled code.



A data sample for context-aware fine-tuning.

Solution2: Fine-tune with Contextual Info

- Context-aware fine-tuning: use the name hints from the calling context.

```
0 int send_packet(...) {
  ...
1 memset(packet, 0, sizeof(packet));
2 struct iphdr* ip_hdr =(struct iphdr*)packet;
  ...
3 // Fill in the IP Header
4 memset(ip_hdr, 0, 0x3c);
5 ip_hdr->ihl = ip_hdrlen >> 2;
6 ip_hdr->tos = 0;
7 ip_hdr->id = htonl(...);
  ...
8 // Compute checksums
9 tcp_checksum(ip_hdr,ip_hdrlen,...);
  ...}
```

Source code of a function "send_packet".

```
A0int sub_401430(...){
  ...
A1 memset(s, 0, 0x400);
A2 v29 = s;
  ...
A3 memset(
A4 *v29
A5 *((ch
A6 *((ui
  ...
A8 sub_40197A(v29, n,...);
  ...}
```

The corresponding decompiled code.

The name of v29 can be inferred from the calling context.

Input to an LLM

```
int sub_401430(...){
 ...
 memset(s, 0, 0x400);
 v29 = s;
 ...
 memset(v29, 0, 0x3c);
 *v29 = (n >> 2) & 0xF;
 *((char*)v29+1) = 0;
 *((uint16*)v29+2)
          = htonl(...);
 sub_40197A(v29, n,...);
 ...}

Q: v29 , s , n
```

Contextual information

```
<Context>
sub_40197A(ip_hdr,
          ip_hdrlen...)
</Context>
```

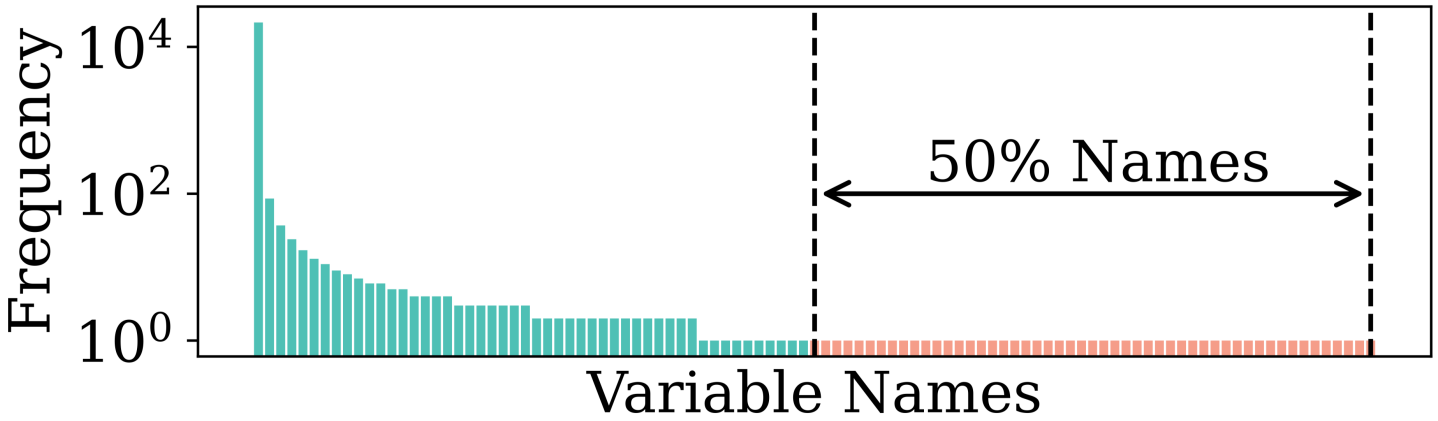
Expected output

```
A: v29 -> ip_hdr
   s -> packet
   n -> ip_hdrlen
```

A data sample for context-aware fine-tuning.

Challenge3: Bias Toward Frequent Names

- The name distribution follows a long-tailed distribution.
- Over half of all names appear only once.



Challenge3: Bias Toward Frequent Names (Cont'd)

- For example, the statistics of a widely-used high-quality training dataset show that the first argument to `memset` is frequently labeled as `buffer` than other names.

memset	buffer		file	
	T	F	T	F
T	19	700	2	717
F	292	206504	165	206631
χ^2	1.6e-63		0.22	

Challenge3: Bias Toward Frequent Names (Cont'd)

- For example, the statistics of a widely-used high-quality training dataset show that the first argument to `memset` is frequently labeled as `buffer` than other names.
- A model trained on this dataset is likely to mis-predict `s` as `buffer`, while its ground truth name is `packet`.

memset	buffer		file	
	T	F	T	F
T	19	700	2	717
F	292	206504	165	206631
χ^2	1.6e-63		0.22	

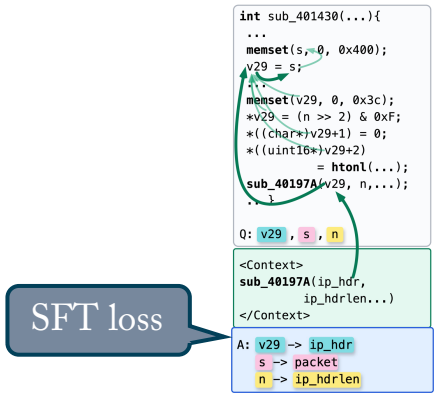
```
A0int sub_401430(...){
   ...
A1 memset(s, 0, 0x400);
A2 v29 = s;
   ...

A3 memset(v29, 0, 0x3c);
A4 *v29 = (n >> 2) & 0xF;
A5 *((char*)v29+1) = 0;
A6 *((uint16*)v29+2) = htonl(...);
   ...

A8 sub_40197A(v29, n,...);
   ...}
```

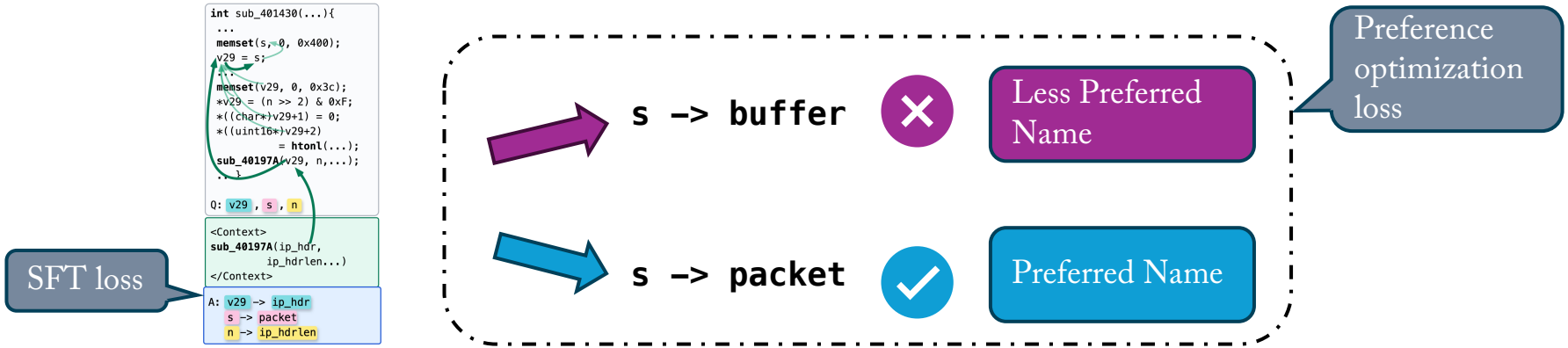
Solution3: Mitigate Bias via Preference Optimization

- Misassigning a frequent name to a variable may not be entirely “wrong”. It indicates developers’ **preference** about choosing names.
- Developers **prefer** not to use frequent names when they don’t fit the program context.
- We teach models the developers’ preference via preference optimization.



Solution3: Mitigate Bias via Preference Optimization

- Misassigning a frequent name to a variable may not be entirely "wrong". It indicates developers' **preference** about choosing names.
- Developers **prefer** not to use frequent names when they don't fit the program context.
- We teach models the developers' preference via preference optimization.



In addition to the SFT loss, our training requires the model to increase the probability for preferred names while suppressing less-preferred ones. This is achieved by preference optimization.

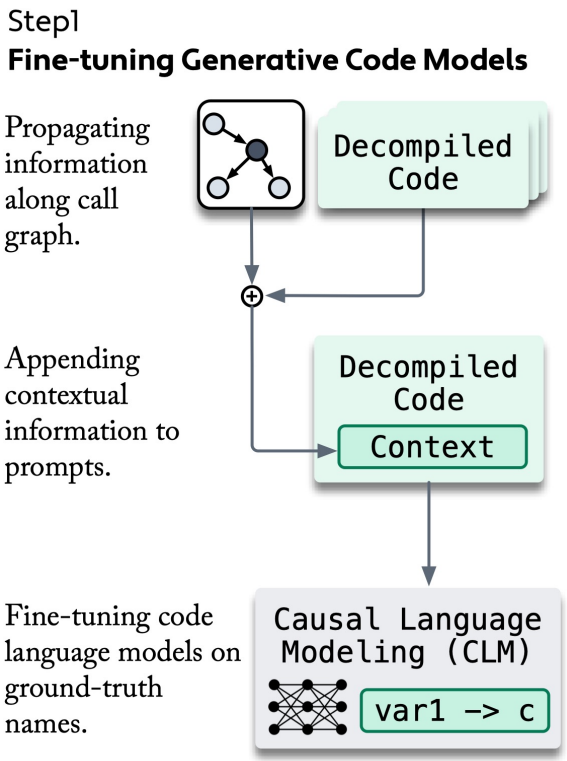
Overall: 3-Stage Training Pipeline

Step 1: Fine-tuning with contextual information.

Step 2: Identifying model's bias towards frequent names.

Step 3: Mitigating bias via preference optimization.

Overall: 3-Stage Training Pipeline



Step 2: Identifying model's bias towards frequent names.

Step 3: Mitigating bias via preference optimization.

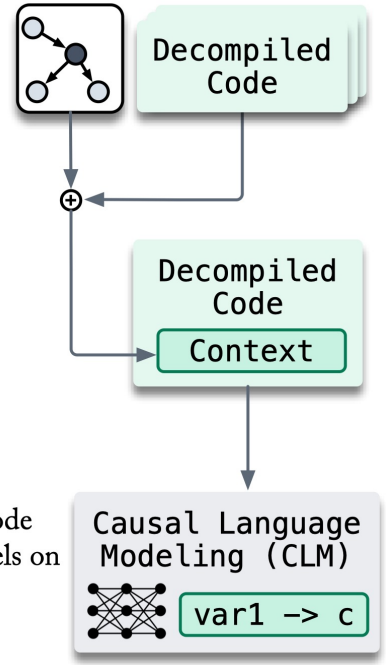
Overall: 3-Stage Training Pipeline

Step1
Fine-tuning Generative Code Models

Propagating information along call graph.

Appending contextual information to prompts.

Fine-tuning code language models on ground-truth names.

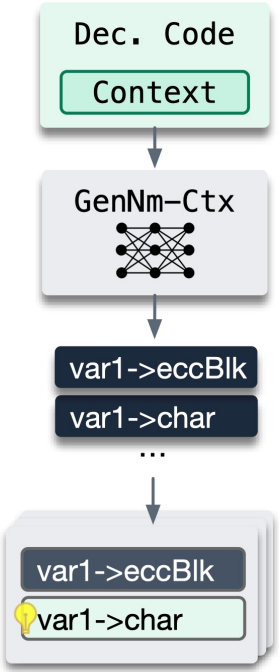


Step2
Building a Symbol Preference Dataset

Inferencing on the training dataset with the fine-tuned model.

Sampling multiple names for each variable.

Constructing pairwise dataset reflecting symbol preference.



Step 3: Mitigating bias via preference optimization.

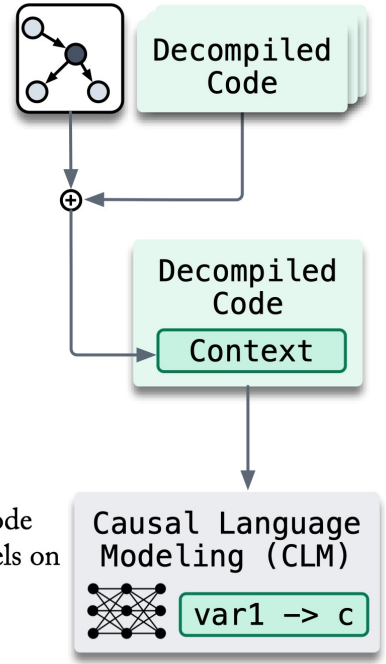
Overall: 3-Stage Training Pipeline

Step1
Fine-tuning Generative Code Models

Propagating information along call graph.

Appending contextual information to prompts.

Fine-tuning code language models on ground-truth names.

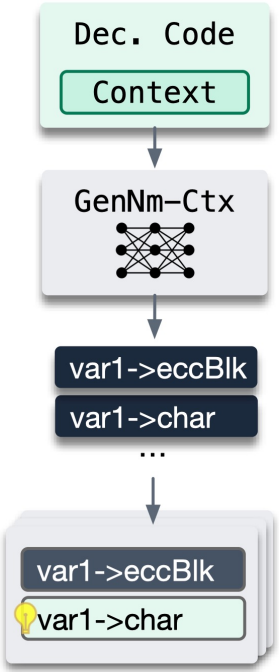


Step2
Building a Symbol Preference Dataset

Inferencing on the training dataset with the fine-tuned model.

Sampling multiple names for each variable.

Constructing pairwise dataset reflecting symbol preference.

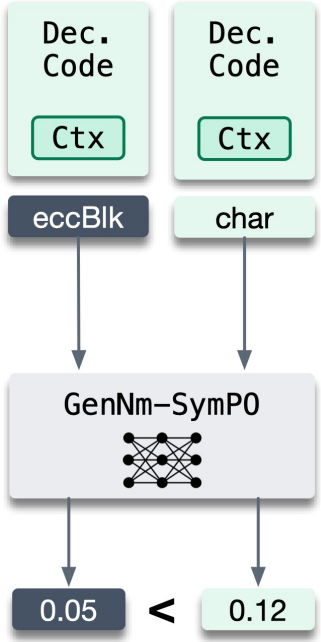


Step3
Symbol Preference Optimization

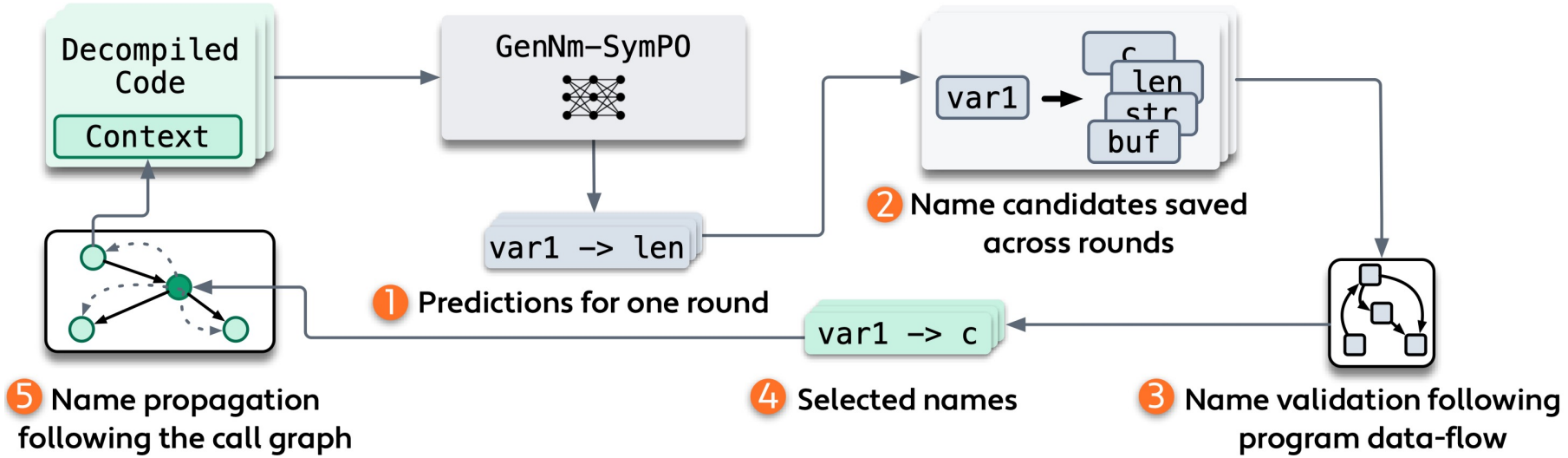
Composing two queries with the same code, for better and worse names, repsectively.

Calculating the probabilities for generating both names.

Requiring the probability for the better name to be larger.



Overall: Iterative Inference



Evaluation: Overall Performance

- Our prototype GenNM achieves better performance than both the classification-based model (VarBERT) and the generative model trained with a naïve SFT loss (ReSym).
- The performance is measured as the token-wise precision and recall between the predicted name and the ground truth name for a variable.

Dataset	Model	Proj. NIT		Proj. IT		Overall	
		Precision	Recall	Precision	Recall	Precision	Recall
DIRTY	VarBERT	23.6	21.7	31.4	29.6	27.2	25.5
	ReSym	25.3	24.9	35.6	34.3	30.2	29.3
	GENNM	**30.5**	**28.8**	**41.7**	**39.6**	**35.8**	**33.9**
VarCorpus	VarBERT	20.9	19.3	32.5	31.0	29.8	28.3
	ReSym	23.5	24.1	34.2	35.8	31.7	33.1
	GENNM	**29.5**	**27.4**	**44.7**	**42.8**	**41.2**	**39.3**

Evaluation: Overall Performance

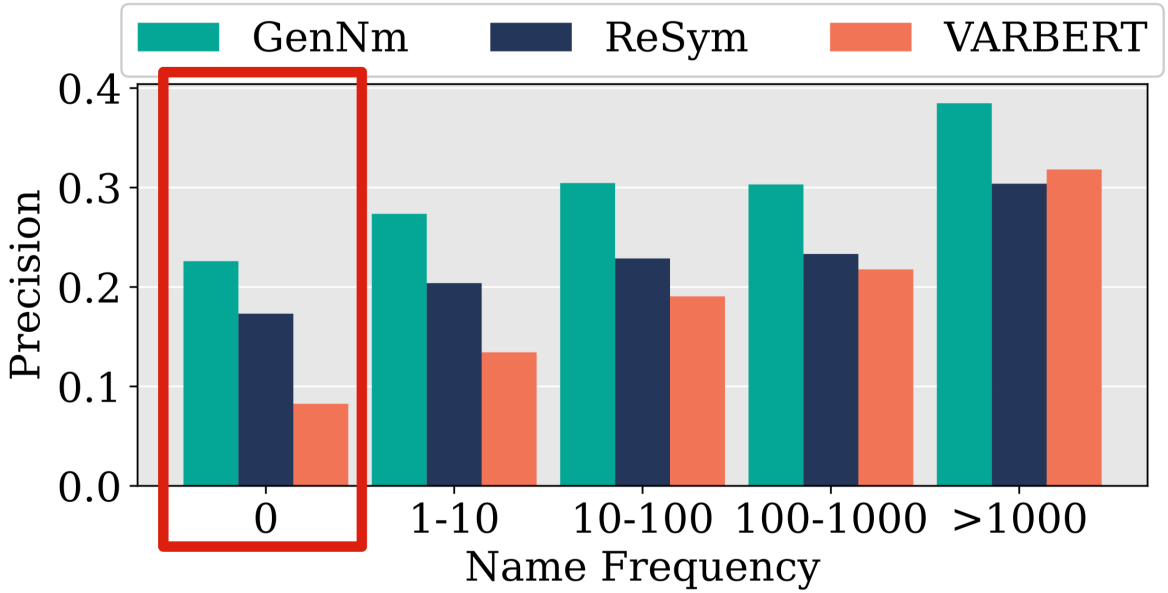
- GenNM achieves better performance than prompting black-box LLMs.

Model	Prompt	Precision	Recall
GPT-3.5	zero-shot	26.2	27.7
	3-shot	29.7	28.9
GPT-4	zero-shot	30.3	33.3
	3-shot	31.4	32.6
CodeLlama-70B	zero-shot	-	-
	3-shot	27.4	26.9
GENNM	-	**42.7**	**39.7**

Only 2B parameters

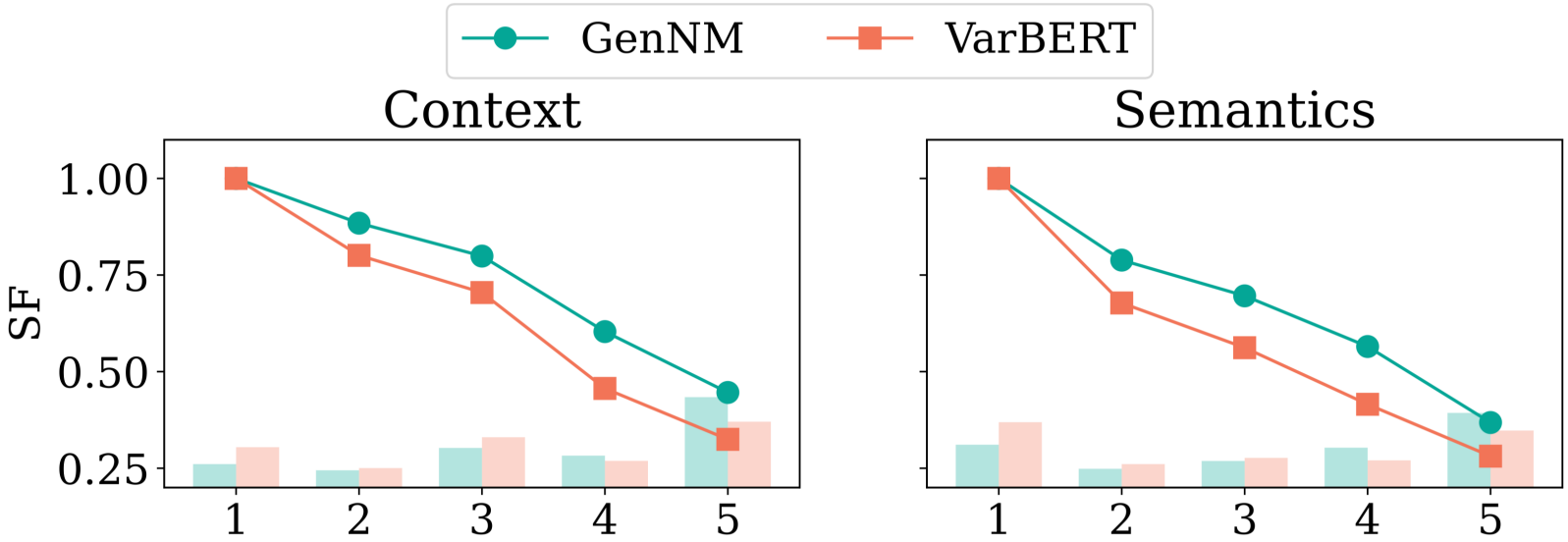
Evaluation: Generalization to Rare Name

- GenNm achieves significantly better performance on unseen and rare names (i.e., names with low frequencies in the training dataset).



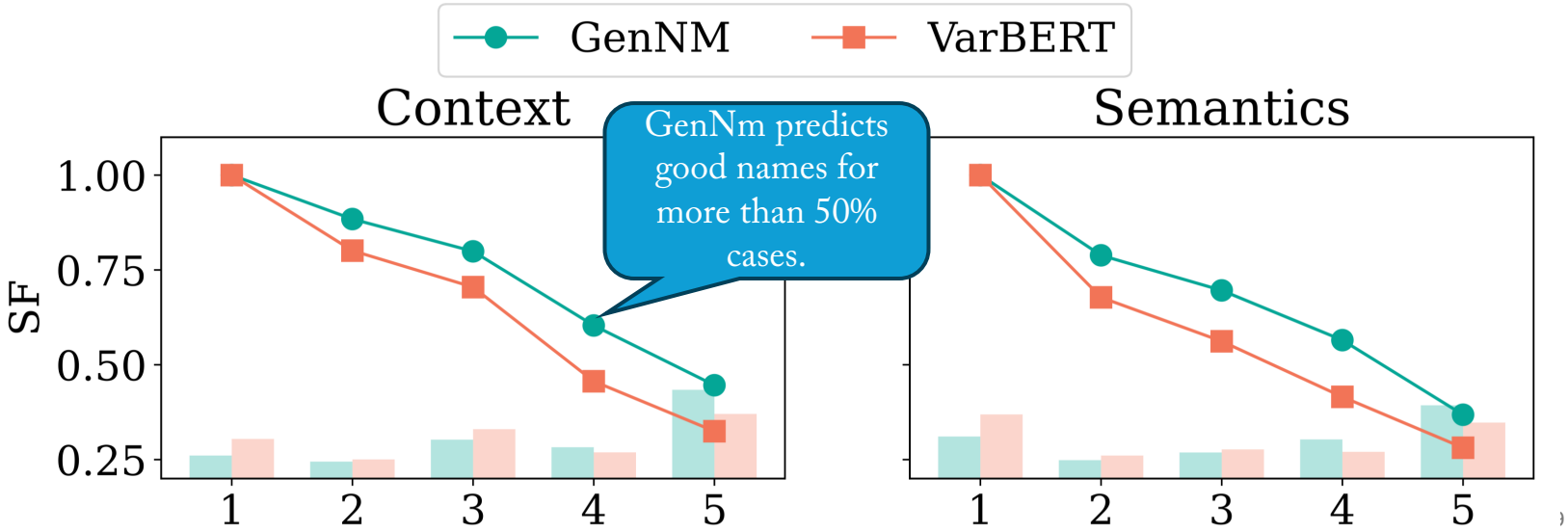
Evaluation: GPT4Evaluator

- We instruct GPT4 to judge the quality of reconstructed names, scoring from 1 (worst) to 5 (best).
- This metric approximates how a human reverse engineer would evaluate the reconstructed names.



Evaluation: GPT4Evaluator

- We instruct GPT4 to judge the quality of reconstructed names, scoring from 1 (worst) to 5 (best).
- This metric approximates how a human reverse engineer would evaluate the reconstructed names.



Takeaways

- **Context-aware fine-tuning:** We propose a domain-specific design for SFT LLMs.
- **SymPO:** We design SYMbol Preference Optimization to mitigate data bias.
- **GenNM:** We implement a prototype that predicts good names for more than 50% variables.

Takeaways

- **Context-aware fine-tuning:** We propose a domain-specific design for SFT LLMs.
- **SymPO:** We design SYMbol Preference Optimization to mitigate data bias.
- **GenNM:** We implement a prototype that predicts good names for more than 50% variables.

We actively work on

democratized reverse engineering,

trustworthy coding agent, and

agentic code reasoning systems.

Please connect with us!

Thank you!

GPT4Evaluator Example - 1

```
__int64 process_wait(__pid_t *proc,
                     unsigned int wait_secs){
 ...
 memset(&sigact, 0, sizeof(sigact));
 sigact.sa_handler = (__sighandler_t)sighandler;
 sigaction(14, &sigact, 0LL);
 alarm(wait_secs);
 if(waitpid(*proc, &status, 0) == *proc)
   ...
 return result;
}

Prediction: timeout
```

(a) Context:5, Semantics:5. The predicted name has exactly the same semantics and context with the ground-truth name.

GPT4Evaluator Example - 2

```
__int64 randn(double mu, double sigma){

  if ( ... ){
   v2 = (double)qword_4138D0 * sigma + mu;
  }}...

Prediction: variance
```

(b) Context:4, Semantics:2. The predicted name has almost the same context as the ground truth (both are related to statistics). However, the semantics is misleading since variance is typically the square of sigma.

GPT4Evaluator Example - 3

```
__int64 file_exists(const char *filename)
{
    ...
    fd = open(filename, 0);
    if (fd < 0)
        return (unsigned int)fd;
    close(fd);
    return 1LL;
}

Prediction: path
```

(c) Context:5, Semantics:4. The predicted name is consistent with the program context. However, the semantics of the predicted name does not imply the variable refers to a file. (`path` may also point to a directory.)