HexT5: Unified Pre-training for Stripped Binary Code Information Inference

Jiaqi Xiong[*], Guoqiang Chen[*], Kejiang Chen[*], Han Gao[*], Shaoyin Cheng[*†‡], Weiming Zhang[*‡]
{jqxiong, ch3nye, gh2018}@mail.ustc.edu.cn, {chenkj, sycheng, zhangwm}@ustc.edu.cn
[*]University of Science and Technology of China, Hefei, China
[†] Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation, Hefei, China

***Abstract*—Decompilation is a widely used process for reverse engineers to significantly enhance code readability by lifting assembly code to a higher-level C-like language, pseudo-code. Nevertheless, the process of compilation and stripping irreversibly discards *high-level semantic information* that is crucial to code comprehension, such as comments, identifier names, and types. Existing approaches typically recover only one type of information, making them suboptimal for semantic inference. In this paper, we treat pseudo-code as a special programming language, then present a unified pre-trained model, HexT5, that is trained on vast amounts of natural language comments, source identifiers, and pseudo-code using novel pseudo-code-based pre-training objectives. We fine-tune HexT5 on various downstream tasks, including code summarization, variable name recovery, function name recovery, and similarity detection. Comprehensive experiments show that HexT5 achieves state-of-the-art performance on four downstream tasks, and it demonstrates the robust effectiveness and generalizability of HexT5 for binary-related tasks.**

***Index Terms*—Reverse Engineering, Deep Learning, Binary Diffing, Information Inference, Programming Language Model**

I. Introduction

Most commercial software in the wild is closed-source and released as stripped binaries that contain no debug information for the purpose of easy distribution or copyright protection. In many security scenarios, e.g., malware detection, vulnerability discovery, or legacy code maintenance. Reverse engineering is essential for verifying the logic and comprehending the inner mechanisms of binary files. The absence of useful information in stripped binary significantly hinders the comprehension and readability of assembly code, making reverse engineering a laborious and challenging task. In recent years, reverse engineering tools decompilers, such as Ghidra [1], Hex-Rays [2] and Binary Ninja [3], have employed sophisticated program analysis and heuristics to reconstruct much of information about a program's variables, types, structured control flow, complex data structures, and function signatures. This information is organized into pseudo-code, a C-like approximation of the binary code.

Although these tools can lift assembly into C-like pseudo-code, thus reducing the cognitive burden of understanding assembly code, they are weak at recovering *high-level semantic information*, e.g., comments, identifiers, user-defined types, and idiomatic structure. These pieces of information are all lost during compilation and therefore absent in the generated pseudo-code. For example, as shown in Figure 1, function names and variable names, which are highly important for pseudo-code comprehension, are replaced with meaningless placeholders (sub_40B012F and v1). To improve the decompilation, researchers have developed a suite of deep-learning techniques to uncover information obscured by the compiler and present it in a human-understandable manner. Take a concrete example, NFRE [4], SymLM [5], and Nero [6] deduce the natural names of binary functions in stripped binaries. Likewise, DIRE [7], DIRECT [8], and DIRTY [9] reconstruct the original variable names in functions. StateFormer [10], Debin [11], and DIRTY [9] attempt to recover source-level data types. Overall, they all fill in the missing *blank* in binary code.

However, information inference in binaries should not be limited to recovering missing details. In practice, analysts may not be concerned about the minor blanks of binary code but require a high-level overview of the code. Typically, summarizing a binary code snippet into natural language descriptions is the most straightforward way to understand the functionalities of programs behind the binary code. By analogy with source code summarization, we call it *binary code summarization*.

Furthermore, tracing binary code back to its source code is another feasible way for information inference, since all required information can be found directly from the source files. Consider the following scenario: when an unknown binary function is given, analysts try to find similar or homologous functions in a massive binary function codebase, where each function can be traced. Once successfully retrieved, the decompilation will be completed. This task, commonly known as called Binary Code Similarity Detection (**BCSD**) or Binary Code Clone Search, is currently an active research focus for binary analysis with numerous studies [12,13,14,15,16,17,18, 19,20,21]. In this paper, BCSD is viewed as an indirect method for information inference.

Binary code summarization is currently a novel and natural demand but lacks corresponding research. For instance, Gepetto [22], which is an IDA Pro plugin built on Chat-GPT [23], was posted on GitHub in December 2022 and

[‡]Corresponding author

has got 2.1K stars in two months. It asks OpenAI's gpt-3.5-turbo model to explain what a binary function does. Notably, the first binary code summarization model BinT5 [24] has been proposed. The authors fine-tuned the source code model CodeT5 [25] on binaries. To the best of our knowledge, they are the *only* two to investigate this task.

Besides, most approaches we mentioned mostly concentrate on *only one specific* task, i.e., the study for information inference on binaries is separated. For better accuracy, the latest research methods tended to prioritize predicting a single type of information [6,4,5,7,8,10,24]. Especially, high-level semantic information inference seems to be orthogonal to BCSD. Nonetheless, Some studies in deep learning [26,27] have demonstrated that single-task learning might reduce the model generalization capability.

Programming language models with *pre-train and fine-tune* paradigm, such as CodeBERT [28], CodeT5 [25], PLBART [29], UnixCoder [30], CoditT5 [31], TypeT5 [32] have displayed remarkable competency across several software-related tasks, including summarization, generation, clone detection, bug fix, etc. To improve model suitability for code-related tasks, they do not substantially modify the model architecture but instead rely on well-designed, target-driven, domain-specific pre-training objectives.

On top of the popular CodeT5 model, we develop Hexadecimal Code T5, **HexT5**, a unified pre-trained model to seamlessly support both various information inference tasks and similarity detection and allows for multi-task learning. To guide a model in recovering lost semantic information, we design a novel pre-training objective that *explicitly* models the lost identifiers. Furthermore, we propose to utilize denoising objectives to capture the token-level semantic information and contrastive learning to capture the similarity between binary functions and their variants, which are compiled from the same source code with different optimization flags, compilers, or architectures.

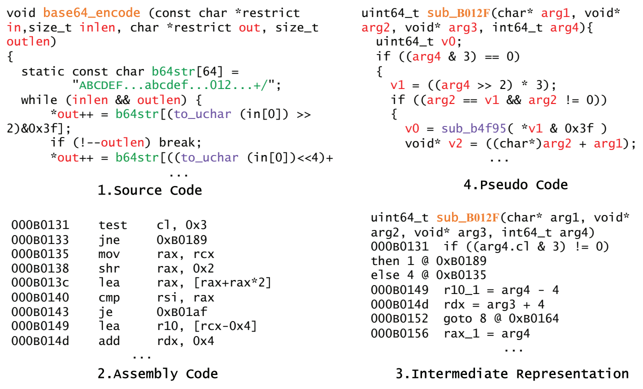
To pre-train the model, a new large-scale dataset is essential. The CAPYBARA dataset, provided by BinT5 [24], is insufficient (14,245 stripped functions) for pre-training. So we collect Natural Language comments from source files (**NL**), extract Source Identifiers from DWARF information (**SI**), and align them with Pseudo-Code (**PC**). The new dataset, dubbed **NSP** , is built upon various architectures, optimization flags, and compilers. And it contains over 5 million `Hex-Ray` pseudo-code functions and 1.98 million comments, which could be applied in pre-training and several inference tasks.

We evaluate HexT5 on four downstream inference tasks including three conditional sequence generation tasks: summarization, variable name recovery, and function name recovery, along with one similarity detection task. Experimental results show that our model achieves state-of-the-art performance. Further ablation analysis reveals that four pre-training objectives and transfer learning from source to binary code can benefit HexT5.

In summary, we have made the following contributions:

- We create a new large-scale, well-formatted dataset, NSP, for our pre-training and downstream tasks, with an automated approach for collecting decompiled code's comments.
- We formulate various innovative pre-training objectives that capture the semantics of both pseudo-code and high-level natural language and align them.
- We implement HexT5 on NSP Dataset. Upon fine-tuning on four downstream tasks, proposed HexT5 achieves improved performance over existing models.

We release the dataset, code, and pre-trained model of HexT5 at https://github.com/USTC-TTCN/HexT5 to facilitate the follow-up research.



```
void base64_encode (const char *restrict
in,size_t inlen, char *restrict out, size_t
outlen)
{
  static const char b64str[64] =
         "ABCDEF...abcdef...012...+/";
  while (inlen && outlen) {
      *out++ = b64str[(to_uchar (in[0]) >>
2)&0x3f];
      if (!--outlen) break;
      *out++ = b64str[((to_uchar (in[0])<<4)+
                    ...
```

```
000B0131    test    cl, 0x3
000B0133    jne     0xB0189
000B0135    mov     rax, rcx
000B0138    shr     rax, 0x2
000B013c    lea     rax, [rax+rax*2]
000B0140    cmp     rsi, rax
000B0143    je      0xB01af
000B0149    lea     r10, [rcx-0x4]
000B014d    add     rdx, 0x4
                 ...
```

```
uint64_t sub_B012F(char* arg1, void*
arg2, void* arg3, int64_t arg4)
000B0131  if ((arg4.cl & 3) != 0)
then 1 @ 0xB0189
else 4 @ 0xB0135
000B0149  r10_1 = arg4 - 4
000B014d  rdx = arg3 + 4
000B0152  goto 8 @ 0xB0164
000B0156  rax_1 = arg4
                  ...
```

```
uint64_t sub_B012F(char* arg1, void*
arg2, void* arg3, int64_t arg4){
  uint64_t v0;
  if ((arg4 & 3) == 0)
  {
    v1 = ((arg4 >> 2) * 3);
    if ((arg2 == v1 && arg2 != 0))
    {
      v0 = sub_b4f95( *v1 & 0x3f )
      void* v2 = ((char*)arg2 + arg1);
                 ...
```

Fig. 1: Different decompilation stages for function *base64_encode* in *libgnutls*[1].

II. Background and Related Work

We start with some background on decompilation, strip, and pre-trained programming language models. Then introduce some related downstream work.

A. Decompilation

Figure 1 depicts an example of decompilation. It typically undergoes three stages during the decompilation process in a counterclockwise direction: Firstly, a binary file is disassembled by the platform-specific tool and converted into assembly code. Then, the assembly code is transformed into a platform-independent intermediate representation (IR) using binary analytical techniques, e.g., static single assignment (SSA). In the subsequent stage, program analyses are employed to reconstruct an Abstract Syntax Tree (AST) that mirrors the structure of the original program. Ultimately, this AST is transformed into decompiled code [7].

Contrasted with assembly code, pseudo-code (PC) offers several prominent advantages. (1) PC is platform-independent and exhibits a higher degree of abstraction. For instance, depending on optimization, a local int-type variable may be placed on a stack or a register, which can significantly alter the assembly code. On the contrary, PC is unaffected and has a more uniform style than binary code, thus mitigating

[1]https://www.gnutls.org/

discrepancies arising from diverse architectures, optimization flags, and compilers. (2) PC contains richer syntactic information: Through the identification and restoration of key features (e.g., return variable, logical structure, and function arguments), PC becomes similar to source code and eliminates the interference of hard-coding addresses. (3) PC incorporates richer natural semantic information: as shown in Figure 1, the string "`ABC...+/`" stored in `.rodata` segment is back into pseudo-code by the decompiler. This particular string aids in the identification of `base64`.

B. Strip

Stripping denotes the process of removing the symbol table and other unnecessary metadata from a binary file. It can significantly complicate the reverse engineering process and hinder the accuracy and effectiveness of reverse engineering tools. For example, except for lost identifiers shown in Figure 1.4, function boundaries preserved in debugging information are also lost. Determining the list of functions within stripped binaries is a research topic [33,34] beyond the scope of this paper.

C. Pre-trained Programming Language Model

Text-To-Text Transfer Transformer (T5 [35]) is a pre-training and fine-tuning model proposed by Google in 2019, whose architecture, Transformer [36], has led to many state-of-the-art models such as ChatGPT [23] and GPT-4 [37]. An important and attractive feature of T5 is that it reframes all NLP tasks into a unified text-to-text format where the input and output are text strings. Inspired by T5, Yue et al. [25] built a large pre-trained transformer-based model, namely CodeT5, to solve software-related problems such as code summarization, code generation, clone detection, and code refinement. CodeT5 is pre-trained on six programming languages and natural language comments from open-source repositories.

In our work, we formulate name prediction as a text-to-text task, leading us to select the CodeT5 model as our starting point. Moreover, compared with other pre-training models (e.g., UnixCoder[30], PLBART[29]), CodeT5 is the only one pre-trained on the C programming language. Considering the syntactic similarity between pseudo-code and C code, we have grounds to believe that transfer learning from source code to pseudo-code can be successfully accomplished.

D. Related Dowstream tasks

Task1: Summarization. Source code summarization is to generate useful comments given the code and has long been of interest. Yue et al. [25] and Guo et al. [30] treat it as a bimodal generation task and have achieved state-of-the-art performances. Al-Kaswan et al. [24] is the first to apply transfer learning to binary code, by fine-tuning CodeT5 [25] model on pseudo-code to summarize decompiled binaries.

Task2: Variable Name Recovery. Debin [11] attempts to recover function names and other debug information, employing Conditional Random Fields (CRFs) to predict the properties of extracted memory cells and relationships between code and data. DIRE [7], supporting variable renaming, leverage LSTM and GGNN as encoders for embedding the tokenized code and abstract syntax trees (ASTs) from Hey-Ray pseudo-code. Nitin et al. [8] extend the BERT [38] as an encoder-decoder architecture to predict the variable name in pseudo-code. DIRTY [9] is the follow-up work of DIRE[7], the authors focus on both variable types and names and apply *type mask* and *multi-task encoder* to find optimal results.

Task3: Function Name Recovery. David et al. [6] represent assembly code as an augmented control flow graph and extend the CFG with call sites. It opts for a 4-layer Graph Convolutional Network as the encoder to utilize the structural information brought by CFG. Gao et al. [4] present a SentencePiece [39] model to clean the noisy function labels and a Bidirectional LSTM to capture the association of instruction and function name. Considering the context of the function, Jin et al. [5] design a semantic fusing encoder to get behavior-aware code embeddings.

Task4: BCSD. Based on statistical features and CFG, Gemini [14] and Genius [15] leverage the graph embedding network to encode functions to vectors for similarity detection. DeepBinDiff [17] and Asm2Vec [16] adapt Word2Vec [40] to generate a distributed representation for opcodes and operands of assembly instructions. SAFE [12] uses a self-attentive neural network. Asteria [19] decompiles binary code and uses a Tree-LSTM network based on ASTs to calculate function similarities. Graph Matching Network [41] (GMN) cannot produce embeddings. It computes a similarity score through a cross-graph attention mechanism to associate nodes across graphs and identify differences. Recently, Trex [42], PalmTree [20], JTrans [13], and Binary AI [43] opt for Transformer-based architecture as their backbones. They also follow *pre-training and fine-tuning* paradigm and get state-of-the-art results.

TABLE I: Comparison with related works on downstream tasks

Model	**Year**	**Input**	T1	T2	T3	T4
Debin [11]	2018	IR	✗	✓	✓	✗
DIRE [7]	2019	PC	✗	✓	✗	✗
Nero [6]	2020	ASM	✗	✗	✓	✗
NFRE [4]	2021	ASM	✗	✗	✓	✗
DIRECT [8]	2021	PC	✗	✓	✗	✗
DIRTY [9]	2022	PC	✗	✓	✗	✗
SymLM [5]	2022	ASM	✗	✗	✓	✗
BinT5 [24]	2023	PC	✓	✗	✗	✗
BCSDs	#	#	✗	✗	✗	✓
HexT5	2023	PC	✓	✓	✓	✓

Deep learning for binaries. Apart from the tasks our study focuses on, deep learning has been widely used in many other binary reverse engineering tasks, including type inference [10, 9], boundary detection [33,34], disassembling [44,45], etc.

Table I shows the downstream tasks we concentrate on and related approaches. T1 is **Task1**. Input is the representation of binary code, as depicted in Figure 1, which includes assembly code (ASM), intermediate representation (IR), and pseudo-code (PC).

III. The Design

In this section, We start with an overview of our approach, then dive into the technical details of each component.

A. Overview of our framework

Figure 2 provides an overview of HexT5. First, we collect the source code with respect to binary files from code repositories. Next, these raw data will be parsed and decompiled without human involvement. The Natural Language - Source Identifier - Pseudo Code (**NL-SI-PC**) data pairs will be constructed.

Designing pre-training objectives is the core of our approach. Based on the pseudo-code and comment characteristics, we define a series of pre-training tasks to learn practical knowledge from both the PC-only data and PC-NL bimodal data in the NSP dataset. Then, fine-tune the model to adapt to downstream tasks.

B. Generation of Training Data

Like other LMs, training HexT5 requires a large corpus of labeled or unlabeled data. Fortunately, there exist many open-source tools to help us to build the dataset automatically.

Data Collection. In our framework, DWARF [46], a widely used, standardized debugging data format, plays a *bridge* role between the source code and the binary code. DWARF debugging information could record the functions, variables inside binary functions and the *source code location* (the line number, the column number and the source file name) of the binary function. Through these records, we transform binary code comment collection into source code task.

In C and C++ projects, there are no standard documentation syntax specifications or documentation tools. The only strategy to collect the comments is that we collect the single-line and multi-line comments above the location of function declarations and definitions, from the location to the first not comment type element we meet. Notice that the comments in the same line as the right curly braces are not collected, which might be the comment in the tail of the last block. Moreover, the comment at the beginning of the file would be dropped, for it might be `COPYRIGHT NOTICE`. We utilize the srcML [47] to analyze and parse the source files.

Additionally, both the binary file with DWARF information and stripped file are required to generate the pseudo-code (**PC**) and source identifiers (**SI**), The latter are the names that developers provide for variables, types, functions, and labels in source files. Following the setup and scripts of DIRE [7], we generate the pseudo-code from stripped files and extract symbols from unstrippped binaries, Subsequently, we map these symbols to stripped pseudo-code via their addresses. All the pseudo-code is generated from stripped binaries to avoid the assistance of symbols during decompilation.

Pseudo-code Normalization. The variable names and function names in the pseudo-code are meaningless tokens generated by the decompiler, e.g., sub_40B012F and v1, which is constructed by the specific prefix along with the address or the order number. These types of tokens could not provide rich training signals but aggravate the OOV (out of vocabulary) problem. In our approach, as shown in Figure 3, these meaningless names are replaced by meaningless tokens. These tokens are normal tokens that stand for variables (<VAR1>) or callees (<FUNC1>), the identical functions or variables that appear in different locations would be replaced with identical tokens. Meanwhile, they are also special sentinel tokens to be predicted in a pre-training objective. In our approach, we retain max 50 function tokens and 100 variable tokens. What deserves to be mentioned is that the first function sentinel token (<FUNC>) is special and stands for the *only* identifier name for each function.

Not all the identifiers in pseudo-code have meaningful names given by the programmers, especially for variables. Some might be the middle ones generated by the compiler or decompiler, some might not be analyzed properly by the decompiler. In our training dataset, *only* 17.18% of variables and 99.42% of functions have developer assigned name. These entities without corresponding true names would be marked as **<UKN>**, standing for **Unknown Name**.

Besides, another important operation for pseudo-code is truncation, Pseudo-code is much longer than the C source code. A crucial reason is that function inlining and C macro. We delete all the comments in pseudo-code body which is generated by the decompiler. Besides, the *string* inside the function that exceeds 40 tokens would be truncated.

Comment Clean. Not all the sentences in the comments could be the descriptive summary of functions. And noisy data is inevitable, for example, non-English comments, *TODO* comments, and incomplete comments [48,49]. We leverage the following rules to clean the comment.

- Delete special tokens, e.g., #, /*, */, //, ///, //!, URLs, EMAILs, FILE PATHs, etc.
- Delete non-English sentences through FastText [50] language identification algorithm.
- Delete sentences containing developer tokens like " `FIXME`, `TODO` ".
- Retain the sentences containing descriptive symbol, e.g., " **@brief, \brief, @purpose, description, @param, return** ". They are Doxygen[2] style comments. Part of the projects supports this documentation tool.

The sentence we retain would be set at the beginning of the comment to better understand and avoid truncation by the tokenizer.

C. HexT5

1) Pre-training objectives: In this section, we describe the pre-training tasks used in HexT5.

[2]https://github.com/doxygen/doxygen

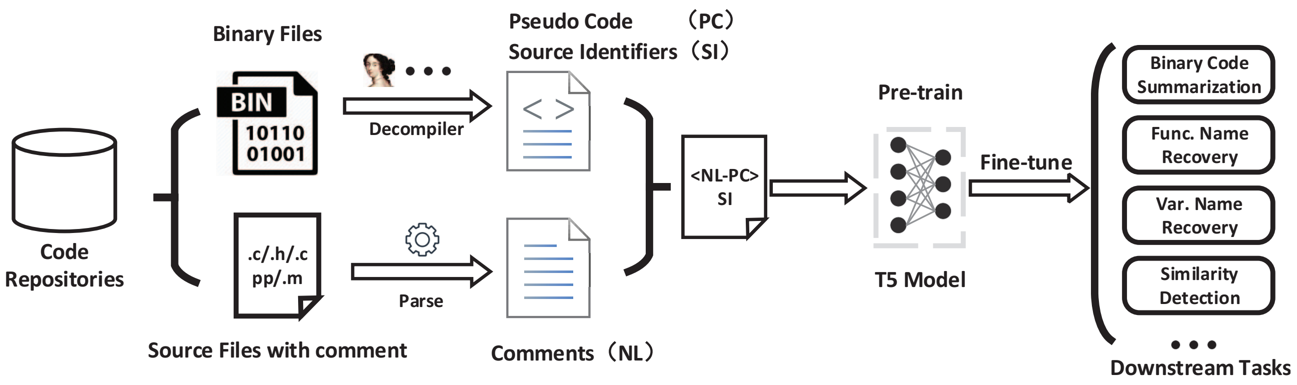
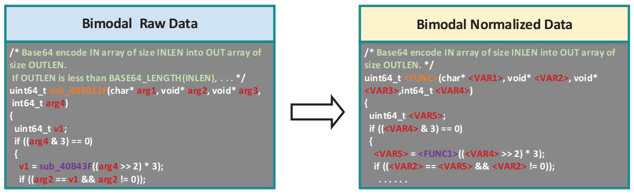


Fig. 2: The overview of our framework, including two main steps, the generation of training data, HexT5 model.

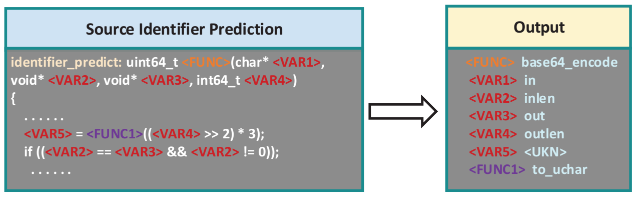

Bimodal Raw Data

```
/* Base64 encode IN array of size INLEN into OUT array of
size OUTLEN.
 If OUTLEN is less than BASE64_LENGTH(INLEN), . . . */
uint64_t sub_40B012F(char* arg1, void* arg2, void* arg3,
 int64_t arg4)
{
  uint64_t v1;
  if ((arg4 & 3) == 0)
  {
   v1 = sub_40B43F((arg4 >> 2) * 3);
   if ((arg2 == v1 && arg2 != 0));
```

Bimodal Normalized Data

```
/* Base64 encode IN array of size INLEN into OUT array of
size OUTLEN. */
uint64_t <FUNC>(char* <VAR1>, void* <VAR2>, void*
<VAR3>,int64_t <VAR4>)
{
  uint64_t <VAR5>;
  if ((<VAR4> & 3) == 0)
  {
   <VAR5> = <FUNC1>((<VAR4> >> 2) * 3);
   if ((<VAR2> == <VAR5> && <VAR2> != 0));
     . . . . . .
```

Fig. 3: Normalization of Pseudo-code and Comment.


Source Identifier Prediction

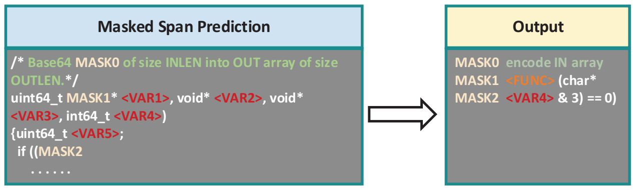
```
identifier_predict: uint64_t <FUNC>(char* <VAR1>,
void* <VAR2>, void* <VAR3>, int64_t <VAR4>)
{
  . . . . . .
  <VAR5> = <FUNC1>((<VAR4> >> 2) * 3);
  if ((<VAR2> == <VAR3> && <VAR2> != 0));
  . . . . . .
```

Output

```
<FUNC>  base64_encode
<VAR1>  in
<VAR2>  inlen
<VAR3>  out
<VAR4>  outlen
<VAR5>  <UKN>
<FUNC1>  to_uchar
```

Fig. 5: Source Identifier Prediction: Predict the lost identifiers.

Masked Span Prediction (MSP). Inspired by CodeT5 [25], T5 [35], we utilize this basic unsupervised denoising method to learn the semantics of both pseudo-code and comments. As Figure 4 shows: First this objective splits the input sequence into chunks and corrupts the chunks by randomly masking a span of tokens for each. Then the decoder recovers the original texts by predicting these masked spans.


Masked Span Prediction

```
/* Base64 MASK0 of size INLEN into OUT array of size
OUTLEN.*/
uint64_t MASK1* <VAR1>, void* <VAR2>, void*
<VAR3>, int64_t <VAR4>)
{uint64_t <VAR5>;
 if ((MASK2
  . . . . . .
```

Output

```
MASK0  encode IN array
MASK1  <FUNC> (char*
MASK2  <VAR4> & 3) == 0)
```

Fig. 4: Masked Span Prediction: Recover the corrupt texts.

Source Identifier Prediction (SIP). This pre-training objective originates from DOBF [51] (*deobfuscation*) in the source code domain. DOBF proposes to revert the obfuscation source code function, where names of functions and variables have been replaced by uninformative names, back to their original forms. Dispensing with the first obfuscation step, we directly use T5 to predict the sentinel tokens.

Dissimilar with **MSP**, only the names of special sentinel tokens, including function, variable, and callee, would be predicted. Text to be predicted is a dictionary composed of *sentinel tokens* • *original identifier name* pairs. As shown on the right side of Figure 5: "<FUNC> base64_encode <VAR1> in ... <FUNC1> to_char ...".

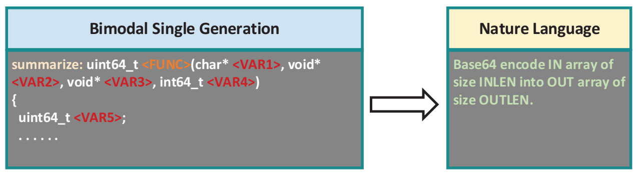
It is a compromise. In DIRE [7], DIRECT [8] and DIRTY [9], the authors viewed variable renaming as a multi-classification task and applied *embedding pooling* (DIRE, DIRTY) or *maximum probability* (DIRECT) to ensure variable consistency, because a variable can appear multiple times in the code tokens of a function. In our objective, it is a code infilling task, and T5 *just generates a special-formatted string*. Our scheme is more succinct and supports all types of identifier renaming, but DIRE [7] and DIRECT [8] can rename longer functions.

In DOBF, all the tokens to be predicted have a certain ground truth because all the obfuscated tokens are masked from original source code. However, as pointed out in Section III-B, *only* a part of variables have developer-given answers. During pre-training, <UKN> tokens would be masked, making the loss function to be indifferent to them.

Bimodal Single Generation (BSG). To close the gap between the pre-training and downstream tasks, we propose to leverage the NL-PC bimodal data to train the model for a bimodal conversion as shown in Figure 6. SIP and MSP work on NL-PC and PC-only data, but this task is only deployed on NL-PC pairs. The input and output are the pseudo-code and corresponding comment, respectively.

Many source code pre-trained models [30,25] usually deploy bimodal dual generation tasks to generate fluent NL texts or syntactically correct code snippets. However, in our tasks, of even greater concern is the understanding of the PC. So NL → PC generation would not take part in our pre-training.

Akin with Vanilla T5 [35], CodeT5 [25], the three tasks above, are all converted into text-to-text (T2T) format. During



```
summarize: uint64_t <FUNC>(char* <VAR1>, void*
<VAR2>, void* <VAR3>, int64_t <VAR4>)
{
  uint64_t <VAR5>;
  ......
```

Base64 encode IN array of size INLEN into OUT array of size OUTLEN.

Fig. 6: Bimodal Single Generation: Put another way, Binary Code Summarization.

pre-training, these task data could be sampled from original dataset and mixed into a mini-batch. However, SIP and BSG share with same inputs. To discriminate between them, we add a prefix task description, dubbed **hard prompt**, at the beginning of the input sequence: identifier_predict and summarize, respectively in Figure 5 and Figure 6. And all T2T tasks in T5 [35] are trained by *cross-entropy loss*. As Equation 1 illustrates, with known input and the first j-1 tokens of the output string, T5 would predict the jth tokens in an auto-regressive manner with maximum probability $\mathbf{P}$, where $\mathbf{O}$ is the output, $\hat{\mathbf{I}}$ is the masked input.

$$\mathcal{L}_{T2T} = \sum_{j=1}^{|\mathbf{O}|} -\log \mathbf{P}\left(\mathbf{O}_j \mid \hat{\mathbf{I}}, \mathbf{O}_{<j}\right) \tag{1}$$

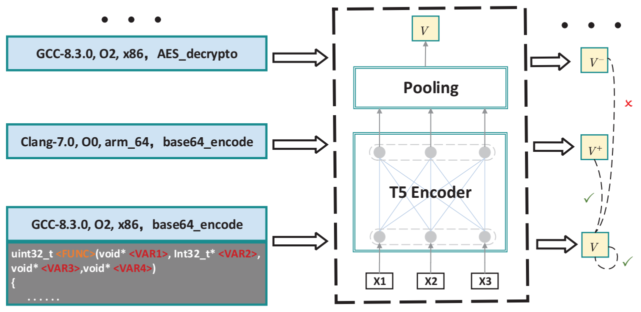
As different tasks have different dataset sizes, to mitigate the bias towards high-resource tasks, we follow the balanced sampling strategy in CodeT5 [25]. For N number of datasets (or tasks), with probabilities $\{q_i\}_{i=1}^{N}$, we define the following multinomial distribution to sample from:

$$q_i = \frac{r_i^{\alpha}}{\sum_{j=1}^{N} r_j^{\alpha}}, \text{ where } r_i = \frac{n_i}{\sum_{k=1}^{N} n_k} \tag{2}$$

where n_i is number of examples for ith task and α is set to 0.7.

Contrastive Learning (CL). Unlike token-level text-to-text pre-training tasks above, this is a segment (function)-level text-to-embedding task. The whole pseudo-code is the input, and the output is the corresponding semantic embedding of the function. We propose to utilize the semantic embedding V to measure the similarity of two binary functions. As illustrated in Figure 7, we apply contrastive learning to learn the function semantics and improve semantic embedding performance. However, T5 model cannot directly generate the semantic embedding. To extract function representation, pooling is an essential step to aggregate the hidden states of the input to form an embedding. According to the finding of Sentence-T5 (ST5 [52]), we choose the *Encoder-only mean pooling*. In this method, the T5 decoder is frozen, and the average of all token representations from the encoder forms the embedding V.

As expressed in the Equation 3, for V, it indicates a semantic embedding of a sample in mini-batch samples $\mathcal{B}$. InfoNCE [53] is a common contrastive loss by pulling semantically close neighbors nearby and pushing apart non-neighbors, where τ is a temperature parameter, and sim is the similarity


GCC-8.3.0, O2, x86， AES_decrypto
Clang-7.0, O0, arm_64， base64_encode
GCC-8.3.0, O2, x86， base64_encode
V
Pooling
T5 Encoder
X1
X2
X3
V⁻
V⁺
V
×
✓
✓

```
uint32_t <FUNC>(void* <VAR1>, Int32_t* <VAR2>,
void* <VAR3>,void* <VAR4>)
{
    ......
```

Fig. 7: Contrastive Learning: Contrast samples against each other.

function of embeddings. V^+ is a semantically close neighbor, and V^- are the non-neighbors. In mini-batch $\mathcal{B}$, other samples V_j except V are viewed as negative samples.

$$\mathcal{L}_{CL} = -\log \frac{e^{sim\left(V,V^+\right)/\tau}}{\sum\limits_{j\in\mathcal{B}} e^{sim\left(V,V_j^-\right)/\tau}} \tag{3}$$

Besides, positive sample V^+ should be explicitly provided. In our framework, we provided two approaches to sample the positive one. **supervised**: the variants. **self-supervised**: we follow the SimCSE [54] and forward the same input using a different hidden dropout mask as a positive example [30].

2) Fine-tune: In this paper, we focus on four downstream tasks: Summarization, Function Name Recovery, Variable Name Recovery, and Similarity Detection (BCSD). In Summarization, to align with *BSG*, we insert the "summarize:" prompt at the beginning of input sequence. In Function Name Recovery and Variable Name Recovery, like SIP, the "identifier_predict:" prompt is appended. Besides, we split the output string of SIP to get the results of different tasks. In BCSD, we get the semantic embedding of pseudo-code function by *pooling*, then use the Cosine Similarity $\cos(V_1, V_2) = \frac{\mathbf{V_1}\cdot\mathbf{V_2}}{\|\mathbf{V_1}\|\|\mathbf{V_2}\|}$ to compute the similarity scores.

IV. Evaluation

We organize our evaluation around two main research questions and question 1 has four sub-questions:

RQ1: How does our pre-trained model, HexT5, compare to baselines on downstream tasks?

- **1.1:** on *Binary Code Summarization* task
- **1.2:** on *Variable Name Recovery* task
- **1.3:** on *Function Name Recovery* task
- **1.4:** o *BCSD* task

RQ2: Do our proposed pre-training objectives and the weights of CodeT5 [25] help a model in better reasoning about and performing information inference?

A. Experiment Setup

Pre-training Setting The architecture of HexT5 is same as CodeT5-base[3], consisting 12 encoder and decoder layers, 12

[3]https://huggingface.co/Salesforce/codet5-base

attention heads, and a hidden dimension size of 768. The total number of parameters is 223M. HexT5 is initialized from the CodeT5-base model. Then, we follow Liu et al. [55] to train a byte-pair encoding (BPE [56]) tokenizer with 32K subword units and add 150 special tokens (see Section III-B) into the tokenizer. We set the maximum source and target sequence lengths to 512 and 256, respectively.

We implement HexT5 using IDA Pro [2] 7.5, Transformers [57] 4.24.0, PyTorch [58] 1.11.0, CUDA 11.3. Moreover, we leverage the pycparser, FastText, srcML to collect and clean the data. Meanwhile, we conduct all experiments on a Linux server equipped with Intel Xeon Gold 5220R processors, 256GB RAM and 10 NVIDIA 3090 GPUs. To accelerate the pre-training, we use the mixed precision of `FP16`, and DeepSpeed ZeRO-2[59] optimization.

We pre-train the model with the denoising objective for 30 epochs and other tasks for 20 epochs, it takes us about 25 days. In MSP, we set the corruption rate as 15% and the average length as 3, respectively. In CL, the dropout rate of the encoder is set to 0.15.

Datasets. We follow the BinKit [60] to build a cross-compiling environment. First select 51 projects from the GNU repository [4], then compile them with four architectures (`x86`, `x86_64`, `arm_32`, `arm_64`), four compiler optimization flags (`-O0`, `-O1`, `-O2`, `-O3`), and four compilers (`clang-7.0`, `clang-9.0`, `gcc-7.2.0`, `gcc-8.3.0`). Finally, we obtain the datasets with 15,847 different binaries, 1,981,423 natural language comments, and 5,393,646 functions. In NSP, about 15% pseudo-code functions are longer than 512 tokens, we do not remove them to simulate the real-world environment.

Dataset Partition. Researches [7,4,24] have revealed that large binary code corpora may contain near-duplicate code across training and testing sets, which can cause evaluation metrics to be artificially inflated. A common cause is the reuse of the libraries and underlying code, for example, in `Binutils`, `nm` and `ar` share the same binary file descriptor (BFD) processing code. Inspired by Lacomis et al. [7] and Jin et al. [5] and David et al. [6], we utilize two manners to split the dataset: *cross-project* and *cross-binary*. Figure 8 illustrates these: In *cross-project*, an entire project is allocated to one of the sets, ensuring that functions from the same project are not dispersed across different sets. In *cross-binary*, the binary file serves as the smallest indivisible unit. Consequently, a binary and its variants may be distributed across different sets. Existing methods do not *explicitly* stipulate the rules of variant assignment. Our approach guarantees adherence to the following constraint: variants differing only in their compilers are not allocated to different sets, while other variants are assigned randomly.

In our experiments, we first split the entire dataset into a train, validation, and test set in *cross-project* manner. These sets comprise approximately 80%, 10%, and 10% of the total dataset, respectively. After pre-trained on the training set, to assess the efficacy and generalizability of the models, we further subdivide the test set in two distinct manners. In *cross-project*, we fine-tune the model on the part of the train set and evaluate models on the total test set. In *cross-binary*, the test set is additionally divided into approximately 10% additional fine-tuning sets and 90% final testing sets using *cross-binary* method. We fine-tune the model on the fine-tuning set, then conduct tests on the remaining set. Depending on the test set partitioning strategy, the models can forecast the outcomes of unseen functions or the variants of previously observed functions.

[4]http://ftp.gnu.org/gnu/

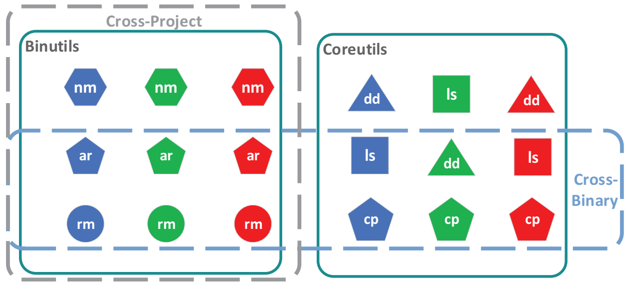


Fig. 8: Split the dataset by *cross-project* or *cross-binary*, same name with different color means the variants.

B. RQ1: Comparison to Prior Works

Different Tasks have different baselines and metrics, we introduce them at the starting place of experiments.

RQ1.1: Binary Code Summarization:

Baselines: We use BinT5 [24], and Gepetto[5] [22] as baselines. Besides, we also adapt PLBART [29] and RoBERTa [55] to binary code summarization. PLBART is based on three token-level denoising pre-training strategies: token masking, deletion, and infilling. And RoBERTa is the backbone of UnixCoder [30]. However, Guo et al. [30] represents code as ASTs and flattens them as the input of RoBERTa. This method increases the input length (70% longer [30]), so is not suitable for pseudo-code. We leverage the five pre-training tasks in UnixCoder to pre-train RoBERTa. All models are initialized from corresponding program language models with `base` model size.

Metrics: We use the `smoothed BLEU-4, Rouge-L, and METEOR` to evaluate the performance. In accordance with BinT5 [24], CodeT5[25], UnixCoder[30], we directly use the scripts provided with CodeXGLUE[6] to calculate the smoothed BLEU-4 scores.

Comparison on NSP dataset. We present results in Table II. For BinT5, it is built on CAPYBARA, a Ghidra [1] dataset, and we contacted BinT5's authors to request the raw binary data and received their assistance. However, the raw data is not organized as a dataset. For a fair comparison, we add two models, *CodeT5 fine-tuned on the pseudo-code* generated by

[5]https://github.com/JusticeRage/Gepetto

[6]https://github.com/microsoft/CodeXGLUE/tree/main/Code-Text/code-to-text/evaluator

TABLE II: Results for binary code summarization on NSP dataset in cross-project manner

Model	Metrics		
	BLEU-4	Rouge-L	METEOR
PLBART [29]	14.72	18.16	17.70
RoBERTa [55]	15.05	**20.43**	18.97
BinT5♣ [24]	11.26	17.25	14.50
Gepetto [22]	3.77	6.10	5.42
CodeT5 *ft* Hex-Ray	9.45	13.45	12.17
CodeT5 *ft* Ghidra	10.03	13.66	12.30
HexT5	**16.60**	**20.45**	**20.19**

♣ : denotes these data are *directly* from their paper.
ft: fine-tuned on pseudo-code

Hex-Ray or Ghidra, to verify the influence of the decompilers. Gepetto is a ChatGPT-based method and feeds the prompt, “Analyze the following C function: Suggest better ...”, along with pseudo-code into OpenAI’s API. For token length limitations, we randomly test 1,000 functions to evaluate it.

Benefiting from the pre-training tasks, pre-training based approaches significantly get higher scores than fine-tuning only approaches. Specifically, PLBART achieves the lowest scores in pre-trained models, a possible reason is that it only leverages token-level pre-training tasks. And HexT5 achieves an increase of 1.6 percentage points over RoBERTa, which is a relative increase of 10.29%. The only different pre-training task between HexT5 and RoBERTa is SIP, modeling lost identifiers could assist HexT5 to better capture the natural language semantics. For fine-tuning only models, HexT5 outperforms BinT5 by a relative improvement of 47.42%, 18.55%, and 39.24% in Smoothed BLEU-4, Rouge-L, and METEOR. The performances of *CodeT5 fine-tuned on the pseudo-code* approaches are close, with an absolute difference of less than 1% in all scores. For Gepetto, ChatGPT typically generates sentences that describe the behavior of variables. They do not conform to the style of the comments written by human engineers, therefore resulting in low scores.

Answer to RQ1.1

By combining pre-training and fine-tuning, we can further boost performance substantially across all metrics for binary code summarization. For example, HexT5 surpasses BinT5 by 47.42%, 18.55%, and 39.24% in Smoothed BLEU-4, Rouge-L, and METEOR.

RQ1.2: Variable Name Recovery:

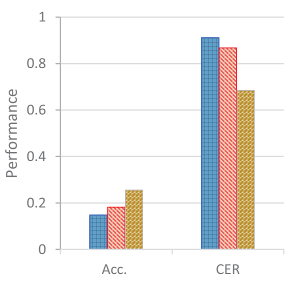
Baselines. DIRTY [9] and DIRECT [8] should be state-of-the-art tools for this task. However, DIRTY focuses more on type inference. And according to the reported results in its paper, the results of DIRTY and DIRECT are close. So, in this task, we evaluate the performance of DIRE, DIRECT, and HexT5.

Metrics. Variable name usually is brief, meanwhile, a given name can have multiple, equally acceptable names (e.g., fp, file_point, fpoint). So we examine Acuarcy (`Acc.`) and character error rate (`CER`) between predicted names and true names. Accuracy is defined as an exact match between names. For CER value, the lower the value, the better the performance of prediction with a CER of 0 being a perfect score. Notably, some variables do not have a developer assigned name (<UKN>), they are ignored from all metrics.

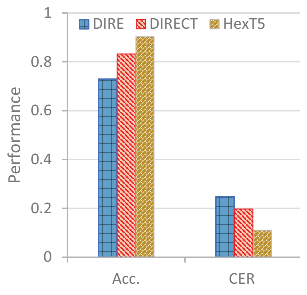
Comparison on NSP dataset. We train DIRE, DIRECT, and HexT5 in NSP dataset. They are both pseudo-code based models without additional preprocess. Figure 9 shows the results of variable name recovery on NSP test set. In both two metrics, HexT5 gets the best scores. In *cross-project* partition, HexT5 achieves an increase of 7.33 percentage points in accuracy over DIRECT and a decrease of 18.36 points in CER. And in *cross-binary*, these two values are 7.01 and 8.64. We observe that the performance of cross-project is significantly lower than cross-binary. It reveals the poor overall generalizability of all models. However, Figure 9a clearly shows the performance gap between HexT5 and DIRECT, in which HexT5 outperforms DIRECT by relatively 38.55%, 21.15% on accuracy and CER score.

Comparison on DIRE dataset. The DIRE dataset and code are available on GitHub[7], and both DIRE and DIRECT evaluate their models on this dataset. It comprises 1,011,049 functions. Notably, DIRE’s test set is divided into two subsets: ”Body in Train” and ”Body not in Train,” which aligns with the goal of our partition. We fine-tune DIRE, DIRECT, and HexT5 on the DIRE dataset.

The results are presented in Figure 10. The performance on *Body not in Train* is much better than on *cross-project* NSP in Figure 9a. This is due to the fact that DIRE does not *specifically* concentrate on stripped binaries, and assumes the pseudo-code has the ground-truth function names. Similar to the NSP’s results, HexT5 achieves higher overall accuracy and lower CER scores than DIRECT on both subsets. Specifically, in *Body not in Train* subset, HexT5 achieves 22.72% and 20.33% better performance regarding the accuracy and CER score, which indicates better generalizability.



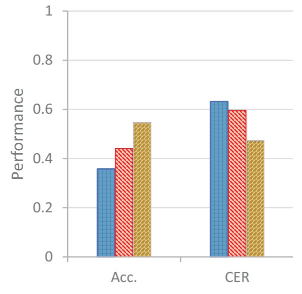
(a) In *cross-project* Manner



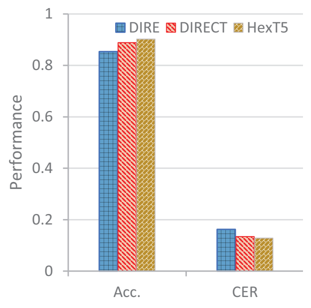
(b) In *cross-binary* Manner

Fig. 9: Results for variable name recovery on NSP dataset

[7]https://github.com/CMUSTRUDEL/DIRE



(a) Body not in Train



(b) Body in Train

Fig. 10: Results for variable name recovery on DIRE dataset

Answer to RQ1.2

HexT5 has better generalizability and effectiveness than the state-of-the-art works for variable name recovery. For example, it outperforms DIRECT by 22.73% and 20.33% on accuracy and CER in predicting unseen functions in DIRE dataset.

RQ1.3: Function Name Recovery:

Baselines. We use NFRE [4] and SymLM [5] as our baselines. According to their reported results, both tools outperform Debin [11] and Nero [6]. Both of them convert programs to assembly code. NFRE supports x86 and x86_64 architectures. SymLM is released in their GitHub repository[8], it can work on the x86, x86_64, arm_32, and MIPS architectures.

Metrics. In accordance with SymLM and NFRE, we calculate word-level `Precision, Recall, and F1-score` to evaluate the performance of function name recovery. Both NFRE and SymLM utilize SentencePiece and an embedding based neural network to denoise the ambiguous function names. Their approaches make ground-truth function name tokenization different from standard. For convenience, we directly leverage the script provided by NFRE to calculate the metrics.

Comparison on NSP dataset. We train and evaluate the NFRE and SymLM on their support architectures. Meanwhile, we retrain their SentencePiece and function name embedding model. The results are summarized in Figure 11a and Figure 11b. Note that the y-axis tick values of different figures are different. It clearly shows that HexT5 outperforms SymLM by 43.90%, 7.17%, and 21.05% on precision, recall, and F1-score in *cross-project* manner. And In *cross-binary*, HexT5 outperforms SymLM by a relative improvement of 7.59%, 2.43%, and 3.33% . Similar to variable name recovery, *cross-project* is much hard than *cross-bianry*. We give a practical example to explain this phenomenon: token "`xorriso`" is the prefix of functions in *Xorriso* project, and around 2% functions in test set have this token. However, in *cross-project*, none of the models could generate it, because it is apart from the train set totally.

Comparison on Nero dataset. Nero [6] releases its open dataset on GitHub[9], and both NFRE and SymLM report their performance on Nero. We evaluate NFRE, SymLM, and HexT5 on this dataset. Because Nero's test set contains part of our train set (e.g. `Coreutils`), we replace them with zlib and compile it with the same configurations as Nero. Nero only supports x86_64 architecture. The results are summarized in Figure 11c. Nero splits it dataset by package, however, as Gao et al. [4] illustrated, in NERO's test set, some functions exist in both train and evaluation sets. This duplication situation is considered to be a realistic use case [7,9]. So the results of Nero is between *cross-project* NSP and *cross-binary* NSP. HexT5 achieves a relative increase of 17.02% and 11.63% over SymLM on precision and F1-score. Meanwhile, the recall score of SymLM is higher than HexT5 with a relative increase of 14.43%. By overall comprehensive comparison, HexT5 is still more efficient than SymLM and has better generalization capability.

Answer to RQ1.3

Overall, HexT5 is a more accurate and more scalable technique for function name recovery than other state-of-the-art approaches. For example, HexT5 outperforms SymLM by 21.05% on F1-score in *cross-project* NSP.

RQ1.4: Binary Code Similarity Detection:

Baselines. We select Asteria [19], SAFE [12], PalmTree [20], and Graph Matching Networks (GMN [41]) as baseline techniques. They apply different neural networks and features to generate semantic embeddings. Asteria is the *only* one that lifts assembly code to Hex-Ray pseudo-code as input, while others utilize assembly code. PalmTree does not support cross-architecture tasks.

Metrics. In this paper, we evaluate the performance of the one-to-many search scenario. Following the study [18], we utilize the Recall (Recall@K) and Mean reciprocal rank (MRR@K) at different K thresholds as the metrics, which are widely used ranking metrics.

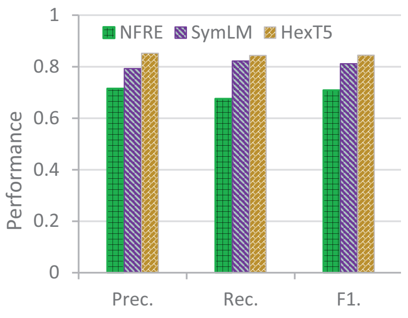
TABLE III: Results for binary code similarity detection on *cross-project* NSP dataset

Model	XO		XA+XO+XC	
	Recall@1	MRR@10	Recall@1	MRR@10
SAFE [12]	79.49	87.15	17.37	21.69
Asteria [19]	76.30	80.91	58.42	68.32
GMN [41]	50.12	62.30	42.13	50.32
PalmTree [20]	80.17	86.32	#	#
HexT5	**88.61**	**92.24**	**75.85**	**82.16**

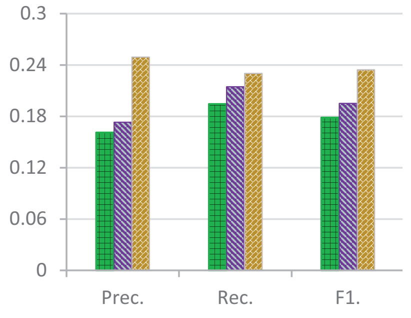
Comparison on NSP dataset. Unlike previous tasks, we must build the additional evaluate set from NSP test. We apply two

[8]https://github.com/OSUSecLab/SymLM

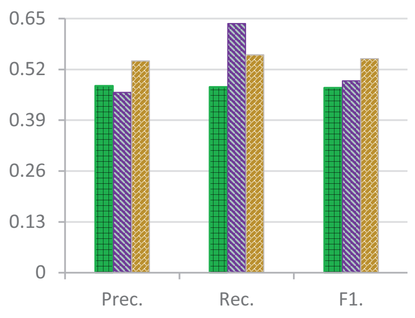
[9]https://github.com/tech-srl/Nero



(a) On NSP in *cross-binary* manner



(b) On NSP in *cross-porject* manner



(c) On Nero [6]

Fig. 11: Results for function name recovery on NSP and Nero dataset.

different sub-tasks defined by Marcelli et al. [18] to evaluate models: (1) XO (Cross-optimization): functions in evaluate set have different optimization flags but have the same compiler and architecture. (2)XO+XC+XA: functions come from arbitrary architectures, compiler versions, and optimization levels. First, we randomly select 500 functions and 10,000 functions as anchors and pooling, which are dissimilar to each other. Then we get the similar functions of the anchors from NSP as the ground truth. Finally, we mix the anchors, their ground truth, and pooling to form the evaluation set. Through the DWARF information, we guarantee that the anchor function in evaluate dataset has definite and fixed similar functions. In the XO+XC+XA task, we extend the numbers 500 and 10,000 to 2,000 and 40,000, respectively.

The results are shown in Table III, HexT5 outperforms the state-of-the-art approaches and achieves the best recall@1 of 88.61 in the XO task and 75.85 in the XC+XO+XA task. Notably, SAFE is not robust on cross-architecture tasks. It views an opcode and hard-coding operand (e.g., `call_-0xf90`) as a single token, thus maintaining a tokenizer with an overlarge vocabulary (527682 words). This weakness results in severe performance degradation. Instead, HexT5's capability of cross-architecture comes from the Hex-Ray decompiler. The address jump is replaced by C-like logic control flow or function call, enabling HexT5 to focus more on control flow rather than uninformative addresses.

Answer to RQ1.4

In one-to-many scenario, HexT5 can retrieve the best candidates accurately in a large function codebase. For example, HexT5 outperforms PalmTree by 10.52% and 6.85% on Recall@1 and MRR@10 scores.

C. RQ2: Ablation Study

We analyze the contributions of each pre-training objective and the weights of CodeT5 in HexT5. Owing to the limitations of computational resources, we choose HexT5-small as the test model. `Small` model consists of 6 encoder and decoder layers, 8 attention heads, and a hidden dimension size of 512. We compare the performance of our HexT5-small on downstream tasks by ablating each of the four objectives and the weights of CodeT5-small in *cross-project* NSP test set, and we leverage BLEU4 in summarization, Accuracy in variable name recovery, F1-score in function name recovery, and Recall@1 in BCSD to evaluate the performance.

As shown in Table IV, we observe that generally removing the parameters of CodeT5-small or MSP reduces the performances across all tasks. This suggests that these two objectives contribute to a better code understanding of our HexT5. The weights appear to be particularly crucial. Without them, all the scores decrease significantly. Put another way, training C code and pseudo-code together can enhance the ability of HexT5. Moreover, The effect of other objectives varies across tasks, with each one benefiting its related task. Furthermore, SIP and BSG can both increase the performance of conditional sequence generation tasks. We anticipate this is because comments and source identifiers have significant semantic overlap. As a side effect, there seems to be an adversarial relationship between CL and two conditional generation objectives. Adding CL would decrease BLEU4, Accuracy, and F1-scores, but this performance penalty is acceptable (relative decrease of 1.10%). Overall, HexT5-small with all objectives gets the highest scores in all the tasks.

Answer to RQ2

Overall, the ablation study demonstrates the effectiveness of the four pre-training tasks. We also verify the necessity of transfer learning from CodeT5 to HexT5.

V. Discussion

In the previous section, we found that HexT5 performs considerably well on *cross-binary* dataset. However, in more common scenarios (*cross-project*), the performance is less than satisfactory. Similar situations have been reported in studies such as NFRE [4] and SymLM [5]. This section will discuss the limitations of our work and potential avenues for future improvements.

Callee Inconsistency. For variables in the function body, we utilize a simple strategy to constrain the variable name prediction. But, another important semantic feature, *callee name*, has no explicit constraints. For instance, there exists

TABLE IV: Ablation Study with HexT5-small

Model	Summ. BLEU4	Var. Acc.	Func. F1.	BCSD Recall@1
HexT5-small	**14.45**	**22.17**	**18.34**	**84.13**
-w/o weights	11.94	15.89	15.67	83.45
-w/o MSP	14.22	17.46	18.10	83.28
-w/o SIP	14.22	16.83	16.53	84.08
-w/o BSG	12.41	20.94	17.45	**84.21**
-w/o CL	**14.61**	21.69	**18.29**	75.01

-w/o: without.
weights: parameters of CodeT5 [25]

a binary file with some functions A(), B($\ldots A$()), C($\ldots A$()). The predictions of A are uncertain in different functions.

In previous code language models and Transformer-based binary code models, a function is usually viewed as the sole input sequence. Therefore, these models commonly overlook the function calling context. However, in stripped binary codes, most of the names of internal callings are eliminated. This limitation prevents the models from understanding the behaviors of the callees through their names, a process that is commonplace in the source code domain. Consequently, incorporating calling context as part of the input is necessary. In other words, we reconsider binary code summarization (name prediction) as a *multi-document summarization* task. SymLM [5] aggregate context embedding by concatenating the embeddings of callers and callees. However, this strategy does not meet our requirements. For its semantic embedding model, Trex [42] cannot capture the calling context during pre-training. Meanwhile, the quadratic complexity of the Transformer [36] poses challenges in extending the input length. Despite these challenges, with the advent and explosion of the large language model (LLM), open-source code LLMs with extensive input lengths have achieved prominent progress in code intelligence. By fine-tuning or instruction tuning on code LLM, we may develop a file-level or call-graph-level pseudo-code LLM.

Obfuscation. In this paper, we evaluate HexT5 against architecture, compiler, and optimization flags, we do not consider any obfuscation, e.g., encryption, packing, compiler-based obfuscation (ollvm [61]), etc. Our work is built on Hex-Ray pseudo-code. Any progress in handling deobfuscation on pseudo-code is complementary to our approach.

Lack of Semantic Diversity. Although the NSP contains over 5M functions, we compiler the dataset with 4 architectures, 4 compilers, and 4 optimization flags. As a rough estimate, one source code function has 64 variants of binary code. Compared with the source code dataset [62] (6.1M), the diversity of NSP is too small. The approach proposed by JTrans [13] might be a feasible way, it utilizes PKGBUILD in ArchLinux to build a massive dataset on x86 and x86_64 architecture.

Besides, Distribution bias between the NSP and the wild binaries is another weakness. "*Anomaly*" detection tasks (e.g., malware detection, vulnerability search) are common in real-world scenarios. Although our method performs better than previous works in benign binary code, it has not yet guaranteed satisfactory predictions for malicious or buggy code.

VI. CONCLUSION

In this work, we show HexT5, a system integrating CodeT5 with previous decompilation tools, is effective at predicting the missing information for stripped binary code. To give out the overview of the whole function, we focus on a new type of high-level semantic information, *code summarization*, and we further propose an automated approach to extract the comment from the source file and pseudo-code from the binary file, respectively. Furthermore, we design four pre-training objectives to fully learn the semantics of PC and NL. Our evaluation has shown that HexT5 is up to 47.42%, 22.73%, 21.05%, and 10.52% more accurate than state-of-art tools in summarization, variable recovery, function recovery, and similarity detection respectively; while demonstrating its effectiveness, generalizability, and component necessity.

ACKNOWLEDGMENT

We would like to express our sincere gratitude to the anonymous reviewers for their insightful comments. We also thank Ali Al-Kaswan for his selfless assistance. This work was supported in part by the Natural Sci- ence Foundation of China under Grant U20B2047, 62072421, 62002334, 62102386 and 62121002, and by Open Fund of Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation under Grant CSSAE-2021-007, and by Cybersecurity Education Innovation Grant and Topsec Technologies Group.

REFERENCES

[1] NationalSecurityAgency, "Ghidra," https://ghidra-sre.org/, 2021.
[2] Hex-Rays SA, "IDA Pro," https://www.hex-rays.com/products/ida, 2021.
[3] Vector 35, "Binary Ninja," https://binary.ninja, 2022.
[4] H. Gao, S. Cheng, Y. Xue, and W. Zhang, "A lightweight framework for function name reassignment based on large-scale stripped binaries," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 607–619.
[5] X. Jin, K. Pei, J. Y. Won, and Z. Lin, "Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22, New York, NY, USA, 2022, p. 1631–1645.
[6] Y. David, U. Alon, and E. Yahav, "Neural reverse engineering of stripped binaries using augmented control flow graphs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, 2020.
[7] J. Lacomis, P. Yin, E. J. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, "Dire: A neural approach to decompiled identifier naming," in *34th IEEE/ACM International Conference on Automated Software Engineering*, San Diego, CA, 2019, pp. 628–639.
[8] V. Nitin, A. Saieva, B. Ray, and G. Kaiser, "Direct: A transformer-based model for decompiled variable name recovery," *NLP4Prog 2021*, p. 48, 2021.
[9] Q. Chen, J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig, and B. Vasilescu, "Augmenting decompiler output with learned

variable names and types," in *31st USENIX Security Symposium*, Boston, MA, Aug. 2022.

[10] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray, and S. Jana, "Stateformer: Fine-grained type recovery from binaries using generative state modeling," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[11] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1667–1680.

[12] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Safe: Self-attentive function embeddings for binary similarity," in *Proceedings of 16th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2019.

[13] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "jtrans: jump-aware transformer for binary code similarity detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 1–13.

[14] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 363–376.

[15] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 480–491.

[16] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 472–489.

[17] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *Network and distributed system security symposium*, 2020.

[18] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2099–2116.

[19] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, and Z. Shi, "Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021.

[20] X. Li, Y. Qu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, p. 3236–3251.

[21] D. Peng, S. Zheng, Y. Li, G. Ke, D. He, and T.-Y. Liu, "How could neural networks understand programs?" in *International Conference on Machine Learning*. PMLR, 2021, pp. 8476–8486.

[22] JusticeRage, "Gepetto," https://github.com/JusticeRage/Gepetto, 2021.

[23] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," *CoRR*, vol. abs/2203.02155, 2022.

[24] A. Al-Kaswan, T. Ahmed, M. Izadi, A. A. Sawant, P. Devanbu, and A. van Deursen, "Extending source code pre-trained language models to summarise decompiled binarie," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 260–271.

[25] W. Yue, W. Weishi, J. Shafiq, and C. H. Steven, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*, 2021.

[26] X. Liu, P. He, W. Chen, and J. Gao, "Multi-task deep neural networks for natural language understanding," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Jul. 2019.

[27] Y. Wang, C. Zhai, and H. Hassan, "Multi-task learning for multilingual neural machine translation," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Nov. 2020, pp. 1022–1034.

[28] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[29] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Online, Jun. 2021, pp. 2655–2668.

[30] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 7212–7225.

[31] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "CoditT5: Pretraining for source code and natural language editing," in *International Conference on Automated Software Engineering*, 2022.

[32] J. Wei, G. Durrett, and I. Dillig, "Typet5: Seq2seq type inference using static analysis," *arXiv preprint arXiv:2303.09564*, 2023.

[33] J. Alves-Foss and J. Song, "Function boundary detection in stripped binaries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19, New York, NY, USA, 2019, p. 84–96.

[34] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 611–626.

[35] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, pp. 140:1–140:67, 2020.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, Red Hook, NY, USA, 2017, p. 6000–6010.

[37] OpenAI, "Gpt-4 technical report," 2023.

[38] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019, pp. 4171–4186.

[39] T. Kudo and J. Richardson, "SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2018, pp. 66–71.

[40] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean,

"Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[41] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *International conference on machine learning*. PMLR, 2019, pp. 3835–3845.

[42] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Learning approximate execution semantics from traces for binary function similarity," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2776–2790, 2023.

[43] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 1145–1152.

[44] S. Yu, Y. Qu, X. Hu, and H. Yin, "{DeepDi}: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2709–2725.

[45] A. Flores-Montoya and E. Schulte, "Datalog disassembly," in *29th USENIX Security Symposium (USENIX Security 20)*, Aug. 2020, pp. 1075–1092.

[46] I. UNIX International, "Dwarf debugging information format version 4," https://dwarfstd.org/doc/DWARF4.pdf, 2010.

[47] J. I. Maletic and M. L. Collard, "Exploration, analysis, and manipulation of source code using srcml," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15, 2015, p. 951–952.

[48] L. Shi, F. Mu, X. Chen, S. Wang, J. Wang, Y. Yang, G. Li, X. Xia, and Q. Wang, "Are we building on the rock? on the importance of data preprocessing for code summarization," ser. ESEC/FSE 2022, New York, NY, USA, 2022.

[49] Q. Chen, X. Xia, H. Hu, D. Lo, and S. Li, "Why my code summarization model does not work: Code comment improvement with category prediction," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, 2021.

[50] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext.zip: Compressing text classification models," *arXiv preprint arXiv:1612.03651*, 2016.

[51] M. Lachaux, B. Rozière, M. Szafraniec, and G. Lample, "DOBF: A deobfuscation pre-training objective for programming languages," in *NeurIPS 2021, December 6-14, 2021, virtual*, 2021, pp. 14 967–14 979.

[52] J. Ni, G. Hernandez Abrego, N. Constant, J. Ma, K. Hall, D. Cer, and Y. Yang, "Sentence-t5: Scalable sentence encoders from pre-trained text-to-text models," in *Findings of the Association for Computational Linguistics: ACL 2022*, Dublin, Ireland, May 2022, pp. 1864–1874.

[53] S. Chopra, R. Hadsell, and Y. LeCun, "Learning a similarity metric discriminatively, with application to face verification," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1. IEEE, 2005, pp. 539–546.

[54] T. Gao, X. Yao, and D. Chen, "Simcse: Simple contrastive learning of sentence embeddings," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, 2021.

[55] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019.

[56] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Berlin, Germany, 2016, pp. 1715–1725.

[57] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, and R. Louf, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Oct. 2020, pp. 38–45.

[58] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[59] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery &; Data Mining*, ser. KDD '20, New York, NY, USA, 2020, p. 3505–3506.

[60] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," *IEEE Transactions on Software Engineering*, pp. 1–23, 2022.

[61] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, B. Wyseur, Ed., 2015, pp. 3–9.

[62] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, and D. Jiang, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.