BinAdapter: Leveraging Continual Learning for Inferring Function Symbol Names in a Binary

Nozima Murodova
nozima29@g.skku.edu
Sungkyunkwan University
Suwon, South Korea

Hyungjoon Koo*
kevin.koo@skku.edu
Sungkyunkwan University
Suwon, South Korea

ABSTRACT

Binary reverse engineering is crucial to gaining insights into the inner workings of a stripped binary. Yet, it is challenging to read the original semantics from a binary code snippet because of the unavailability of high-level information in the source, such as function names, variable names, and types. Recent advancements in deep learning show the possibility of recovering such vanished information with a well-trained model from a pre-defined dataset. Albeit a static model's notable performance, it can hardly cope with an ever-increasing data stream (*e.g.*, compiled binaries) by nature. The two viable approaches for ceaseless learning are retraining the whole dataset from scratch and fine-tuning a pre-trained model; however, retraining suffers from large computational overheads and fine-tuning from performance degradation (*i.e.*, catastrophic forgetting). Lately, continual learning (CL) tackles the problem of handling incremental data in security domains (*e.g.*, network intrusion detection, malware detection) using reasonable resources while maintaining performance in practice.

In this paper, we focus on how CL assists in the improvement of a generative model that predicts a function symbol name from a series of machine instructions. To this end, we introduce BinAdapter, a system that can infer function names from an incremental dataset without performance degradation from an original dataset by leveraging CL techniques. Our major finding shows that incremental tokens in the source (*i.e.*, machine instructions) or the target (*i.e.*, function names) largely affect the overall performance of a CL-enabled model. Accordingly, BinAdapter adopts three built-in approaches: ① inserting adapters in case of no incremental tokens in both the source and target, ② harnessing multilingual neural machine translation (M-NMT) and fine-tuning the source embeddings with ① in case of incremental tokens in the source, and ③ fine-tuning target embeddings with ② in case of incremental tokens in both. To demonstrate the effectiveness of BinAdapter, we evaluate the above three scenarios using incremental datasets with or without a set of new tokens (*e.g.*, unseen machine instructions or function names), spanning across different architectures and optimization levels. Our empirical results show that BinAdapter outperforms the state-of-the-art CL techniques for an F1 of up to 24.3% or a Rouge-l of 21.5% in performance.

*Corresponding author

This work is licensed under a Creative Commons Attribution International 4.0 License.
ASIA CCS '24, July 1–5, 2024, Singapore, Singapore
© 2024 Association for Computing Machinery.
ACM ISBN 979-8-4007-0482-6/24/07...$15.00
https://doi.org/10.1145/3634737.3645006

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**;

KEYWORDS

Binary analysis, Software security, Reverse engineering, Continual learning

ACM Reference Format:
Nozima Murodova and Hyungjoon Koo. 2024. BinAdapter: Leveraging Continual Learning for Inferring Function Symbol Names in a Binary . In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3634737.3645006

1 INTRODUCTION

A program is typically distributed in the form of a stripped executable binary (*i.e.*, removing useful information like debugging symbols) after compilation. Binary reverse engineering (*i.e.*, binary reversing) becomes essential to gain insights into the inner workings of the binary when one cannot access its source code. Binary reversing offers valuable assistance not only in diagnosing and resolving issues when encountering application bugs or crashes but also in malware analysis. It allows security researchers to understand the behavior of malware, identify vulnerabilities, develop effective countermeasures, and devise robust mitigation strategies.

However, understanding the functionality of an executable is quite challenging even to experienced experts because the original code semantics (*e.g.*, function names, function parameters, variable names, variable types, structures, class hierarchies) has disappeared with numerous transformation processes by modern compilers. Another reason that makes a reversing task difficult is the dynamic nature of binary code generation: *e.g.*, a particular platform has its own instruction set architecture and its specific intricacies; different compilers and their versions, different optimization levels, different compiler options, or their combination could generate a completely different assembly code from an identical source. Furthermore, it is common to deliberately employ obfuscation techniques such as packing, anti-debugging, and anti-analysis mechanisms, for obscuring the codes' structure and logic.

Recent advancements in deep learning techniques have introduced new possibilities for gaining insights into a binary's operation, surpassing traditional approaches that solely rely on static, dynamic, or a combination of both analyses. In particular, partial recovery of vanished information (*e.g.*, function name [12, 29, 58, 59], variable name [2, 9, 24, 52], decompiled code [43], variable type [9, 24])

in a stripped binary may greatly enhance the comprehension of an unknown binary's behavior. In this work, we focus on one of such efforts that predict a function name (*i.e.*, function symbol name as part of debugging information) [12, 17, 24, 29] from a given sequence of machine instructions. As with the previous techniques, the inference of an original function symbol name from machine instructions is analogous to a translation task from one language to another in the field of natural language processing (NLP).

Despite noticeable performance using prior approaches [12, 17, 24, 29], a static model that is learned from the fixed volume of a pre-collected dataset can hardly cope with an ever-incremental data stream. By nature, the number of source code and the functions within is unlimited (*i.e.*, infinitive), so is the number of compiled binaries accordingly. To update a pre-trained model with an incremental dataset, a naïve approach would be re-training the model. However, it is not only computationally expensive, but also may not be possible due to the unavailability of a previous dataset. Another approach is fine-tuning with the arrival of a new dataset. However, it is well known to suffer from a notorious catastrophic forgetting (CF) issue [41] (*i.e.*, performance degradation), which does not retain previously learned information well.

Lately, continual learning (CL; a.k.a lifelong learning) [56] tackles the overall problems of handling incrementally provided data, demonstrating its feasibility as a promising solution in many areas including (but not limited to) object detection [50, 70], data segmentation [48, 51], and medical imaging [3, 60]. Recent advances demonstrate the applicability of CL in security domains such as network intrusion detection [1, 54, 61] and malware detection [42, 63]. This work expands a CL application to the binary analysis domain, which tackles a function name inference problem (from machine instructions). To the best of our knowledge, we first investigate the impact of CL on the generative model's capability for a certain task, which has yet to be well explored in the security community.

In this paper, we propose BinAdapter, a system that is able to infer function symbol names from a stripped binary by leveraging CL. We build BinAdapter atop AsmDepictor [39], the state-of-the-art Transformer-based architecture tailored to a function name prediction task, inserting adapters [26] in the model during training. In essence, our findings provide two significant insights into the application of CL techniques for function symbol name inference. First, regularization-based CL techniques exhibit less effectiveness when applied to a generative model compared to a discriminative model: we empirically discover that an architecture-based CL model (*e.g.*, adapter-based tuning [26]) outperforms others. Second, the performance of the model is predominately affected by the *incremental tokens* in the source (*i.e.*, machine instructions) or the target (*i.e.*, function names): ① In case of no incremental tokens in either the source or target, *merely inserting adapters* into AsmDepictor turns out to be effective. ② In case of incremental tokens in the source, the best performance is achieved by *adopting multilingual neural machine translation (M-NMT) and fine-tuning source embeddings with adapters*. ③ With incremental tokens in both the source and the target, *adopting M-NMT and fine-tuning both embeddings with adapters* yields the best results.

To demonstrate the effectiveness of BinAdapter, we define three scenarios depending on which a new dataset introduces new tokens (*i.e.*, vocabularies) in either the source or the target. The scenarios encompass an incremental dataset with or without a set of new tokens (such as previously unseen machine instructions or function names), spanning across the known `x86_64` architecture or incorporating additional architectures like `ARM` or `MIPS`. Our experiments demonstrate that BinAdapter surpasses the state-of-the-art CL techniques ([41], [72]) by 18.9% in case ① and by 24.3% in case of ② and ③. These results indicate that BinAdapter can successfully learn new tokens from both the source and the target without any CF from a previous dataset. It is noteworthy to mention that BinAdapter shows slightly better performance than even full-retraining a separate model by utilizing up to 53% of all parameters. Finally, our experiments with multiple optimization levels demonstrate that BinAdapter can effectively adapt to varying code transformations, reaching the performance of around 70% of the F1-score. The following summarizes our contributions:

- We propose BinAdapter, to the best of our knowledge, the first function name inference approach that is equipped with a lifelong learning scheme for a generative model (without the well-known catastrophic forgetting problem).
- We introduce three efficient CL approaches for predicting function symbol names. Our experiments reveal that incremental tokens in both the source and target affect the model's performance.
- We implement the prototype of BinAdapter, and demonstrate the efficacy and efficiency of BinAdapter with 3 different scenarios on various incremental datasets (*e.g.*, new binary architectures and function names).

We open-source BinAdapter [1] to foster further research in the area of CL for binary reversing.

2 BACKGROUND

Continual Learning. An ordinary approach to machine learning typically encompasses data collection, data cleansing, and labeling, training a model, evaluating the model, and its deployment. However, such a static model that is sampled from a pre-collected dataset and/or a fixed set of classes is incapable of coping with the ever-increasing data stream and a new task in the real world. For example, a model that learns three-label classification would fail to recognize an additional label. CL tackles this problem while handling any incrementally provided data, which allows one to update an existing model while minimizing its CF on previously learned information. For effective CL, three promising directions to overcome CF have been introduced: *regularization-based* [41, 72], *replay-based* [8, 34, 45, 66, 67, 69], and *architecture-based* [14, 46, 73] CL. Note that we employ an *architecture-based* CL technique that incorporates a parameter isolation method, freezing the old parameters and utilizing them for learning a new task. Our research findings indicate that this approach is highly effective in preventing CF specifically in the context of sequence generation tasks.

Adapter-based Tuning for CL. Houlsby et al. [26] first introduce adapters for Transformer models as an alternative to fine-tuning with minimal trainable parameters. Simply put, the adapters are lightweight modules inserted into pre-trained Transformer [68] layers, which differs from naïve fine-tuning. Suppose a network

[1] https://github.com/SecAI-Lab/BinAdapter

with (pre-)trained parameters $\boldsymbol{w}$. While fine-tuning merely adjusts $\boldsymbol{w}$ according to a new task (*i.e.*, dataset), adapter-based tuning introduces new modules with randomly initialized parameters $\boldsymbol{v}$, followed by updating only $\boldsymbol{v}$ without touching $\boldsymbol{w}$. The main benefit of adapter-based tuning is that a model is free from a notorious forgetting problem while CL is possible from an incremental dataset because ① there is no interaction between tasks, and ② frozen parameters preserve performance from an original dataset. Each adapter [26] consists of two feed-forward multi-layer perceptron (MLP) layers that adaptively transfer previous knowledge to a new task. A training step updates those compact adapters alone while pre-trained parameters stay intact, incorporating a new dataset without worrying about a catastrophic forgetting problem. Our work leverages such adapters into a function name prediction task that assists in binary reversing.

Multilingual Neural Machine Translation (M-NMT). The Transformer [68]'s encoder-decoder structure has achieved remarkable success in the NMT field. However, existing models often focus on a limited number of languages, such as English, necessitating an efficient means to add new source or target languages without retraining the models from scratch. A recent advancement [4] introduces multilingual NMT incremental training, which involves replacing the shared embedding matrix with a language-specific embedding matrix. The new language embeddings are then fine-tuned independently, while freezing other parameters. This approach relies on the availability of shared tokens across different languages. In our study, we tackle the task of function name prediction in executable binaries using a CL approach within the context of M-NMT. Our finding shows that a naïve M-NMT approach is ineffective due to the scarcity of common tokens in the case of different hardware architectures (*e.g.*, less than 3% in Figure 2). Consequently, we devise the scheme of inserting a single adapter [26] module in each encoder layer and directly *fine-tuning the shared embeddings*, which yields improved performance in practice.

Machine Code Representation. Unlike vocabularies (*i.e.*, tokens) in NLP, disassembled machine instructions can have a massive number of tokens (*e.g.*, instruction, opcode, operand). The sparsity of each instruction not only requires too much computational resources to update its own embedding during backpropagation, but also suffers from an out-of-vocabulary (OOV) problem. Karampatsis et al. [30] discover that byte-pair-encoding (BPE) [16] can efficiently handle the vast number of arbitrary tokens for a large language model tokenizer. Note that BPE is a data compression algorithm by successively replacing the most common pair of consecutive characters with a new token. Later, AsmDepictor [39] demonstrates the effectiveness of BPE in a function name prediction task, which we adopt in this paper.

Function Symbol Name Inference Task with Deep Learning. The task of deducing a function name from a sequence of machine instructions (*i.e.*, binary code snippets) is useful for reverse engineers to gain the insights of a binary. However, it poses a significant challenge when auxiliary debugging symbols have been stripped off. Recent advancements [12, 24, 29, 39] demonstrate that a deep neural network is able to infer the symbol name of a function (*i.e.*, as one of debugging information from an executable's symbol table). In this work, we apply CL techniques on top of AsmDepictor [39], a Transformer-based function name inference system due to its superior performance among others [12, 24, 29].

3 LEVERAGING CONTINUAL LEARNING FOR FUNCTION NAME INFERENCE

3.1 Motivation

A stripped executable binary retains a very limited amount of high-level information that has been present in the source code after complex transformations by a compiler. On the other hand, a non-stripped binary includes symbol information such as function names, variable names, and structures in a symbol table primarily for debugging purposes. With that information, recent studies [2, 9, 12, 29, 38, 43, 52, 58, 59] have revealed that it is possible to partially recover (infer) symbol names (*i.e.*, labels provided by a programmer) from machine instructions. While ML-based models are beneficial for such deductions, the current learning process limits adaptive performance with an additional corpus (*e.g.*, unlearned function names). Besides, the dynamic nature of binary generation leads to enriching machine code representations that hold the same semantics in accordance with the evolution of compilers and hardware architectures, necessitating a systematic approach to continual learning with ever-increasing binary datasets.

3.2 Problem Definition and Goal

Problem Description. We focus on the task of predicting function symbol names, assuming that the original function naming effectively describes the function's behavior. The inference process is akin to language translation tasks in NLP, where we translate a sequence of machine code into a text (function name) using an encoder-decoder model like Transformers [68]. However, the traditional static model with a pre-defined dataset faces limitations when confronted with unseen machine codes or function names from an incremental dataset. In general, it is a demanding yet promising task to maintain an up-to-date model without degrading its performance with the introduction of new data. CL is an active research area [5, 33] to address this problem. Likewise, in this work, we delve into the realm of a continuously learnable and generative model that can infer function names from a sequence of machine code by harnessing lifelong learning techniques.

Scope. A wide spectrum of CL approaches encompasses different targets such as discriminative or generative models, and diverse datasets that involve incremental labels, data distribution changes, and increasing volume of data. In our research, we specifically focus on the CL for a Transformer-based generative model for deducing function symbol names when provided with an incremental binary dataset. Our investigation explores various CL approaches and their applications in various domains to achieve our objective, including regularizers [41, 72], adapters [35, 73], and M-NMT [4]. We initially train a static model with a binary dataset for the `x86_64` architecture. Subsequently, we train the model using an incremental dataset where there are unknown tokens in the original dataset.

Goal. We aim to develop a system that is capable of handling ever-incremental binary data streams for a generative inference model without CF, which can predict function names using CL techniques.

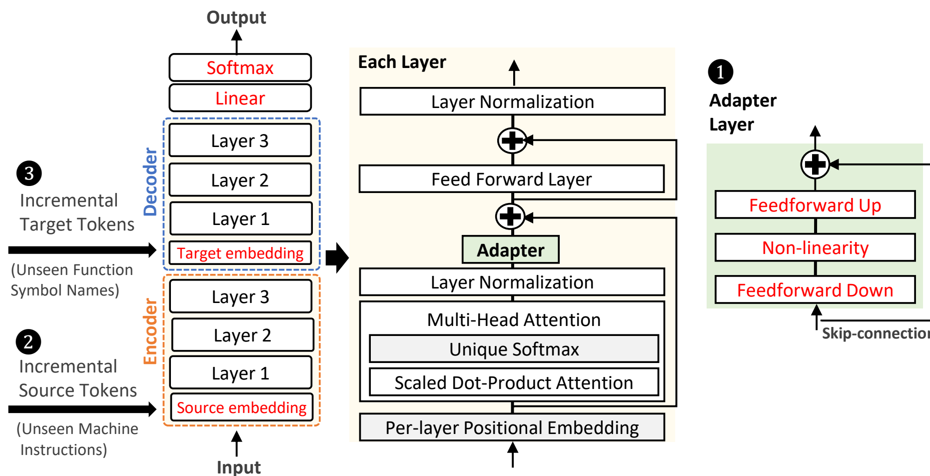

Figure 1: Overview of the CL engine architecture in BinAdapter. We adopt the original AsmDepictor [39] model tailored to a function symbol name prediction task. For CL, we insert the adapters [26] (*i.e.*, green box) after the multi-head attention layer in every encoder/decoder of Transformer. It is worth noting that the square boxes in red fonts represent trainable downstream data during adapter tuning (*i.e.*, adapters, source/target embeddings, and the final output layer). BinAdapter is designed to take different approaches depending on the presence of common tokens: ❶ inserting adapters by default in case that no token is introduced in the source and the target, ❷ adding fine-tuning source embeddings later with adapters in case incremental tokens are present in the source, and ❸ adding fine-tuning both source and target embedding layer with adapters in case incremental tokens are present in both the source and the target.

3.3 Conceivable Scenarios

Our empirical experiments demonstrate that the whole performance of a model using CL techniques largely relies on the set of (un)seen tokens in the source and the target. Suppose that $\mathcal{S}$ and $\mathcal{T}$ represent a set of tokens of the source (*i.e.*, assembly language) and that of the target (*i.e.*, function name), respectively, which are collected from the original (initial) dataset. Similarly, $\mathcal{S}^{+}$ and $\mathcal{T}^{+}$ denote a set of additional tokens in the source and the target from an incremental dataset. The right arrow ($\rightarrow$) means re-training a (continuously learnable) model based on the set of given tokens between the source and the target. We set up the following three conceivable scenarios.

- **Scenario 1**: $(\mathcal{S}, \mathcal{T}) \rightarrow (\mathcal{S}, \mathcal{T})$
  An incremental dataset does not introduce any new tokens in the source or the target. In other words, all tokens in the target is a subset of those in the source. (Leftmost Venn diagram in Figure 2).
- **Scenario 2**: $(\mathcal{S}, \mathcal{T}) \rightarrow (\mathcal{S}^{+}, \mathcal{T})$
  An incremental dataset introduces new tokens in the source alone. (*e.g.*, compiling the same source in a different architecture). (Centered Venn diagram in Figure 2).
- **Scenario 3**: $(\mathcal{S}, \mathcal{T}) \rightarrow (\mathcal{S}^{+}, \mathcal{T}^{+})$
  An incremental dataset introduces new tokens in both the source and the target, which is the most realistic scenario (*e.g.*, compiling the different source in a different architecture). (Rightmost Venn diagram in Figure 2).

Note that we intentionally exclude the case of $(\mathcal{S}, \mathcal{T}) \rightarrow (\mathcal{S}, \mathcal{T}^{+})$ because our observation shows that additional (unseen) function names in the target can commonly introduce additional tokens in the source. For example, a different function routine typically has a different function body, introducing additional tokens.

4 DESIGN

Preprocessing Engine. The purpose of the preprocessing step in BinAdapter is to obtain a sanitized dataset for efficient learning. For a given binary, we collect source tokens with the following steps: ① disassembling the binary, ② identifying the boundary of each function to extract the machine instructions within, ③ refining the whole dataset for better performance (See data refinement in Section 5), and ④ applying BPE [65] to all instructions, which assists in generating a reasonable number of input tokens. For function symbol names, we collect target tokens by splitting an original function name into a single token (*i.e.*, vocabulary) with the delimiter of a capital letter (in a camel-case) or an underscore.

Continual Learning Engine. Figure 1 illustrates the overview of the CL components in BinAdapter. We follow the special design of the Transformer-based function symbol name prediction model from AsmDepictor [38] including a three-layer structure in both encoder and decoder (rather than six layers in Transformer [68]), per-layer positional embedding, and the unique softmax in the multi-head attention. To handle each scenario in Section 3.3, BinAdapter introduces a corresponding approach. First, Scenario 1 [$(\mathcal{S}, \mathcal{T}) \rightarrow (\mathcal{S}, \mathcal{T})$] is a typical CL configuration where an incremental dataset is given. As this case assumes that neither the source nor the target tokens are present (*i.e.*, no OOV) in an incremental dataset, we simply adopt the adapters [26] structure, one of the architecture-based approaches, by inserting them between the Attention layers and

Algorithm 1 Pseudocode for Training Strategies in BinAdapter

```
Input: S+, T+, pretrained_model
Output: State_dict of (new embedding and) adapter
  adapter ← newAdapter(pretrained_model)
  if S+ ⊆ S and T+ ⊆ T then                                  ▷ Scenario 1
      train(adapter)
      save(adapter)
  else if S+ ⊈ S and T+ ⊆ T then                             ▷ Scenario 2
      fineTune(pretrained_model.src_embed, adapter)
      save(pretrained_model.src_embed, adapter)
  else if S+ ⊈ S and T+ ⊈ T then                             ▷ Scenario 3
      fineTune(pretrained_model.src_embed,
      pretrained_model.tgt_embed, adapter)
      save(pretrained_model.src_embed,
      pretrained_model.tgt_embed, adapter)
  end if
```

feed-forward layers of AsmDepictor. In essence, one can insert the adapters (❶ in Figure 1) dynamically and only train them while preserving other pre-trained parameters to prevent CF. While adapters are efficient in handling model adaptivity for an incremental dataset without compromising its performance from a previous dataset, our experiment indicates that training only adapter modules does not scale for a new vocabulary.

Secondly, Scenario 2 [$(\mathcal{S}, \mathcal{T}) \rightarrow (\mathcal{S}^{+}, \mathcal{T})$] assumes that an incremental dataset introduces source tokens alone. As is common to compile an executable binary with the same source code but a different environment or setting (*e.g.*, architecture, compiler, compiler version, optimization level), the source tokens may vary accordingly. We adopt the M-NMT technique [4] that is capable of coping with new tokens without requiring to separately training a new model. On top of that, we fine-tune source embeddings solely in the encoder (❷ in Figure 1) because our finding shows performance degradation in case that the common source tokens between an initial dataset and an incremental dataset are extremely rare, like a cross-architecture binary.

Third, Scenario 3 [$(\mathcal{S}, \mathcal{T}) \rightarrow (\mathcal{S}^{+}, \mathcal{T}^{+})$] assumes that an incremental dataset introduces both source tokens and target tokens. We apply a similar approach to Scenario 2 by fine-tuning both source embeddings in the encoder (❷ in Figure 1) and target embeddings in the decoder (❸ in Figure 1). Specifically, we include the Transformer decoder's embedding matrix and output layer during training, known as the *vocabulary projection matrix*.

Algorithm 1 shows the pseudocode that describes a different training strategy depending on the three aforementioned scenarios. In a nutshell, given an incremental dataset, BinAdapter chooses the corresponding training module: inserting adapters for Scenario 1, M-NMT and fine-tuning source embeddings with adapters for Scenario 2, and M-NMT and fine-tuning both source and destination embeddings with adapters for Scenario 3.

Inference Engine. Once the model has been trained with the CL engine in Section 4, we now have an incremental module produced by adapters (*i.e.*, adapter modules) and a set of incremental tokens accordingly (from an incremental dataset). Note that BinAdapter separately stores the adapter module, vocabulary, and corresponding embeddings for each incremental dataset. During the inference, BinAdapter first identifies the correct embeddings by matching vocabulary from given instructions, followed by choosing a respective adapter module(s) to be loaded.

5 IMPLEMENTATION

Data Refinement. We wrote the preprocessing script in Python that is applicable for all architectures in Ghidra [53], handling both machine instructions and function names. We refine our dataset with the following strategies for better performance by: ① excluding linker-inserted functions such as `deregister_tm_clones`, `frame_dummy`, and `__do_global_dtors_aux`, ② removing functions that contain one instruction; *e.g.*, `jump` or `ret`, which is mostly a wrapper that calls another function within because it does not imply meaningful (contextual) semantics corresponding to a function name, and ③ eliminating duplicate functions from our final corpus that has the same function body but different function names because such a sample could degrade the overall model performance during learning. For BPE, we use `subword-nmt` [64] for subword-based tokenization by setting the vocabulary threshold of $10,000$. For the programs written in C++, we apply a simple rule for (mangled) function names, which removes special letters other than a tilda (~) that represents a destructor.

Model Implementation. We employ the publicly available AsmDepictor Transformer [37] model using Pytorch [57]. We follow the original implementation, discarding all tokens that are longer than 300 in each sequence. We use a scheduled Adam optimizer with $\beta 1 = 0.9$, $\beta 2 = 0.98$ and $\epsilon = 10^{-9}$ as proposed by the original Transformer. However, we modify a warmup step and a multiplication factor differently according to a new dataset size and CL techniques in our experiments (mainly up to 10,000 and 1.0, respectively). For Adapters, we adopt the original implementation provided by Zhang et al. [73] with the GELU (Gaussian Error Linear Unit) [25] activation function as a nonlinear part between the two fully connected layers. We initialize Adapters randomly for each incremental data and train it for 20-50 epochs (until reaching a convergence). Lastly, Adapters are significantly affected by the learning rate. Hence, for effective regulation, we establish warmup steps and a multiplication factor in a scheduler with the values of 4,000 and 1.0, respectively.

Hyperparameter Tuning for Baselines. We observe that the techniques based on regularization such as EWC [41] and SI [72] are highly sensitive to hyperparameters. EWC and SI as our baselines utilize a single hyperparameter for tuning (*i.e.*, regularization coefficient), dubbed a lambda value (λ), which defines the penalty applied to important parameters for each task. We empirically set up the coefficient (λ) to 25 for both EWC and SI. As a final note, in LoRA [28], we set a ranking (`r`) parameter to 16 that records the best performance.

6 EVALUATION

In this section, we evaluate BinAdapter with three research questions. We run our experiments on a 64-bit Ubuntu 20.04 system equipped with Intel(R) Xeon(R) Gold 5218R CPU 2.10GHz, 256GB RAM, and two RTX A6000 GPUs.

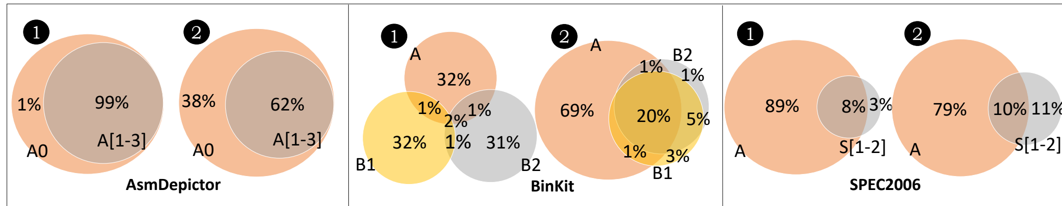

Figure 2: A Venn diagram to illustrate the distribution of common tokens (vocabularies) between the initial dataset and the incremental datasets (*i.e.*, AsmDepictor, BinKit, SPEC2006) at a glance. Note that ❶ represents the token distribution for the source (*i.e.*, machine instructions with BPE), and ❷ represents that for the target (*i.e.*, function names). For example, the AsmDepictor dataset shows that the entire tokens in the incremental datasets (*e.g.*, $A[1-3]$) are the subset of the tokens from the initial dataset (*e.g.*, $A0$). On the contrary, the BinKit dataset depicts that the source tokens are rarely shared between different instruction set architectures (ISAs). The notation that represents each dataset follows Table 2.

Table 1: We set up three experiments (Section 3.3) where BinAdapter takes different approaches according to each scenario. Note that the source tokens from an incremental dataset with a different architecture retains little common (source) tokens in the initial dataset due to different ISAs.

Scenario	Incremental Data	Approach
$(\mathcal{S},\mathcal{T}) \rightarrow (\mathcal{S},\mathcal{T})$	AsmDepictor (x64)	Adapters
$(\mathcal{S},\mathcal{T}) \rightarrow (\mathcal{S}^+,\mathcal{T})$	BinKit (ARM, MIPS)	Adapters + Src Emb
$(\mathcal{S},\mathcal{T}) \rightarrow (\mathcal{S}^+,\mathcal{T}^+)$	SPEC2006 (x64), BinKit (ARM, MIPS)	Adapters + Src + Tgt Emb

6.1 Dataset

Corpus Generation. We prepare our binary corpus from three different datasets including AsmDepictor [38], BinKit [36], and SPEC2006 [10]. First, we utilize the whole dataset provided by AsmDepictor [39], which consists of 3,063 binaries. Second, by leveraging BinKit [36], we create 1,166 binaries for `ARM-64` (big-endian) and `MIPS-64` (little endian) architectures that are compiled with Crosstool-NG [11] that supports cross-compiling in Linux. Binkit includes popular C libraries and programs such as `binutils` [19], `coreutils` [20], `findutils` [21], `BusyBox` [7], `OpenSSL` [55], etc. For our experiment, we filter out unique binaries compiled with `gcc` v6.4 and `clang` v5.0 with the -O2 optimization level. Third, we generate 24 C-written binaries with the default build script (using both gcc and clang) provided by SPEC2006 [10]. Lastly, we create 87 C++-written binaries from BinKit [36] (*e.g.*, file extensions with `.cc` and `.cpp`). Table 2 summarizes the whole binary corpus for our CL experiments.

6.2 Experimental Setup

6.2.1 Incremental Scenarios. To demonstrate the effectiveness of BinAdapter, we carefully set up our experiments according to each scenario (Section 3.3). Table 1 summarizes the overall settings of each BinAdapter approach with its corresponding dataset.

Scenario 1: $(\mathcal{S},\mathcal{T}) \rightarrow (\mathcal{S},\mathcal{T})$. The first scenario occurs when an incremental dataset introduces neither new source tokens nor new target tokens. For this case, we use the AsmDepictor [38] dataset alone. We split the whole functions in the dataset into four parts: $A0$, $A1$, $A2$, and $A3$. $A0$ is the initial dataset (*i.e.*, 258,692 functions) for training, and all others ($A[1-3]$) are three incremental datasets with around 12%, 8%, and 4% more unseen functions, respectively (Table 2).

Scenario 2: $(\mathcal{S},\mathcal{T}) \rightarrow (\mathcal{S}^+,\mathcal{T})$. The second scenario is where an incremental dataset introduces new source tokens alone. This case is common because an executable binary may be diverse with the same source code when compiled with a different compiler, the compiler version, optimization level, or different architecture. For this scenario, we adopt the dataset from BinKit [6] where B1 and B2 binaries are compiled for different architectures (*e.g.*, `ARM-64` and `MIPS-64`). We define `ARM` and `MIPS` binaries as two incremental datasets ($B1$ and $B2$) and C++ programs as $B3$ from `x86_64` as in Table 2 where A is the initially trained dataset. As expected, the distribution of the source tokens because the instructions in another architecture largely differs from those in the initial dataset (*e.g.*, `x86_64`). Unlike in Scenario 1, here the entire AsmDepictor dataset (A: 323,364 functions) is the initial dataset (Table 2) and we record its original performance as an initial performance. Note that we include part of the functions in BinKit whose target tokens are the subset of the initial dataset ($\mathcal{T}$). The statistics to meet such criteria is 142,814 functions (*i.e.*, 6,929 target tokens) from 500 binaries on `MIPS`, and 103,897 functions (*i.e.*, 9,949 target tokens) from 494 binaries on `ARM` (This is not shown in Table 2).

Scenario 3: $(\mathcal{S},\mathcal{T}) \rightarrow (\mathcal{S}^+,\mathcal{T}^+)$. We define the third scenario where an incremental dataset introduces both new source tokens and new target tokens. For `x86_64` architecture, we utilize the SPEC2006 [10] dataset as two incremental datasets ($S1$ and $S2$ in Table 2). We set up the entire AsmDepictor dataset (A: 323,364 functions) as the initial dataset (Table 2). Likewise, for `ARM` and `MIPS` binaries, we harness two incremental datasets ($B1$ and $B2$) from BinKit [6] (Table 2).

6.2.2 Common Token Analysis in the Corpus. Figure 2 illustrates the distribution of common vocabularies between the dataset for initial training and the incremental datasets. ❶ and ❷ represent the token distribution for the source (*i.e.*, assembly code with BPE) and the target (*i.e.*, function name), respectively. For example, the

Table 2: Summary of our binary corpus for continual learning experiments. We use three datasets from (A)smDepictor [39], (B)inKit [6, 36], and (S)PEC2006 [10]. We split the AsmDepictor dataset (A) into four parts where $A0$ is the initial dataset, and $A[1-3]$ are incremental. Similarly, we utilize the BinKit and SPEC2006 datasets as part of incremental datasets ($B[1-3], S[1-2]$) where the whole A dataset is the initial one. The B3 dataset represents C++ programs from BinKit. We elaborate on preparing the datasets in Section 6.1.

Corpus		(A)smDepictor [38]				(B)inkit [36]			(S)PEC2006 [10]	
# of Functions	A	A0	A1	A2	A3	B1	B2	B3	S1	S2
Architecture	x86_64					ARM	MIPS	x86_64	x86_64	
Train	323,364	258,692	30,672	20,000	10,000	95,656	130,473	78,761	32,336	16,168
Valid/Test	80,842	64,674	7,168	6,000	5,000	23,915	32,619	19,690	6,006	2,000
Inc. Rate	-	-	+11.8%	+7.7%	+3.8%	+29.5%	+40.3%	+24.3%	+9.9%	+4.9%
# of Tokens w/ BPE (Instruction)	9,923	9,923	6,876	3,654	2,453	9,951	9,674	9,183	1,037	1,032
# of Tokens (Function Name)	23,663	23,663	14,969	7,786	3,941	7,168	7,886	4,108	5,910	5,098

rate of common source tokens (*i.e.*, overlapping area) in the middle Ven diagram of Figure 2 is indeed at most 2% between the set of binaries on `x86_64` (A from AsmDepictor), the one on `ARM` ($B1$ from BinKit), and the one on `MIPS` ($B2$ from BinKit) as defined in Table 2.

6.2.3 Evaluation Metrics. We assess our approach with two popular evaluation metrics, namely, F1 and Rouge-l. We compute the F1-score as the harmonic means of a precision (P) and a recall (R) where TP, TN, FP, and FN represent the number of true positives, true negatives, false positives, and false negatives, respectively.

$$P = \frac{TP}{TP + FP},\ R = \frac{TP}{TP + FN},\ F1 = \frac{2 \cdot P \cdot R}{P + R} \quad (1)$$

We also adopt Rouge-l [44], an automatic evaluation metric for the quality of a machine translation task using the longest common subsequence (LCS) and skip-gram statistics.

6.2.4 Research Questions. We raise the following four research questions to assess BinAdapter for each scenario in terms of ① effectiveness, ② efficiency, ③ ablation studies, and ④ capability of learning varying code semantics. Besides, we demonstrate a case study of BinAdapter with notable examples.

- **RQ1.** How effective is BinAdapter in different scenarios (Table 1) compared to other CL baseline techniques for a function name prediction task (Section 6.3)?
- **RQ2.** How efficient is BinAdapter in terms of training parameters, storage overheads, and inference time (Section 6.4)?
- **RQ3.** How appropriate is the current design of BinAdapter with ablation studies (*e.g.*, number of adapters, fine-tuning layers) (Section 6.5)?
- **RQ4.** How does BinAdapter learn transformed code semantics like optimization levels (Section 6.6)?

6.3 Effectiveness (RQ1)

6.3.1 Scenario 1. **Baselines.** We set up the five baselines for the comparison with BinAdapter: two fine-tuning methods (FT and FT + Enc in Figure 3), EWC [41], SI [72], which are standard CL techniques and LoRA [28] adaptive modules with CL, when no new token is introduced. One of well well-known means to train a model on a new dataset would be fine-tuning the trained model. We perform two types of fine-tuning on the model trained with $A0$ in Table 2): ① fine-tuning all encoder and decoder layers on a new dataset with the pre-trained weights (FT), and ② fine-tuning only top encoder layers while freezing the rest modules (FT + Enc). Next, we examine both EWC [41] and SI [72] techniques from the open-source project that is maintained by Hsu et al. [27]. Lastly, we utilize LoRA [28] for CL by freezing pre-trained model parameters and training LoRA modules alone (LoRA + CL). We directly insert LoRA to Asmdepictor [39] into each attention layer of both encoders and decoders (*i.e.*, query (Q), key (K), and value (V) vectors), followed by training those modules for each dataset.

Table 3: Performance comparison results between BinAdapter and five baseline CL techniques under Scenario 1 with incremental datasets from AsmDepictor (*i.e.*, $A[1-3]$ in Table 2). Each metric is averaged over three incremental learning datasets.

Technique	Precision	Recall	F1	Rouge-l
FT (Encoder + Decoder)	21.88	19.89	20.25	21.76
FT (Encoder only)	36.32	33.99	34.38	36.15
EWC [41]	41.76	40.65	41.04	45.75
SI [72]	39.22	38.05	38.32	41.56
LoRA+CL [28]	59.53	57.76	58.89	60.31
BinAdapter	**60.56**	**59.32**	**59.89**	**61.44**

Results. Table 3 displays the average precision, recall, F1, and Rouge-l of BinAdapter compared with the five baseline techniques for all data splits (A[0-3]). BinAdapter surpasses all baselines on average. Figure 3 visualizes the breakdown of the inferences from an incremental dataset (upper) and the ones from the initial dataset of A0 (lower). Because we leverage LoRA as a parameter isolation technique to preserve pre-trained parameters like Adapters, the performance of BinAdapter does not drop as in Figure 3 (lower). On the other hand, strong regularization algorithms in EWC and SI exhibit even better performance, however, they suffer from CF (*e.g.*, drastically dropping Rouge-l values: SI: 0.71 → 0.20, EWC: 0.67 → 0.30) for the previous dataset (Figure 3 - lower).

6.3.2 Scenario 2. **Baselines.** In this scenario, the regularization-based baselines like SI and EWC are excluded because they fail to extend new vocabularies from an incremental dataset. Instead, we include full retraining by retraining an existing model with the incremental dataset for comparison. We compare BinAdapter

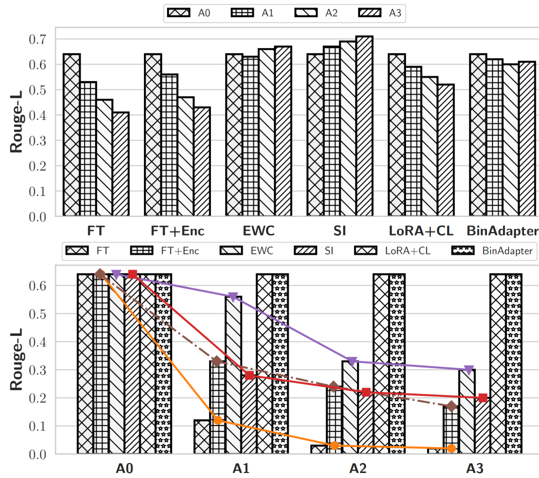

Figure 3: Performance comparison between BinAdapter and five CL baselines under Scenario 1. The upper plot shows the Rouge-l values for newly learned function names from incremental datasets whereas the lower plot shows the performance for the initial dataset (A0) after learning each incremental data (A[1-3]). The lines indicate the forgetting rates of each baseline. While both EWC and SI exhibit marginal improvements (upper), they are susceptible to a catastrophic forgetting problem (lower). On average, BinAdapter demonstrates superior performance, achieving an F1 score of 59.9 or a Rouge-l score of 61.4.

with the M-NMT approach [4] that includes adapters by default. Besides, we set up the baselines of BinAdapter with LoRA, M-NMT with LoRA, BinAdapter without adapters, and N-NMT without adapters to confirm the evident effectiveness of LoRA and Adapters.

Results. Table 4 summarizes the comparison results using the same metrics (*i.e.*, precision, recall, F1, Rouge-l) according to each incremental dataset (*e.g.*, *B*1 (`ARM`), *B*2 (`MIPS`), and *B*3 (`x86_64; C++`) from BinKit). The results show that the performance of BinAdapter and the baseline techniques are agnostic to language-specific dataset (*i.e.*, different ISAs). While BinAdapter outperforms other baselines (F1 of 69.67 and Rouge-l of 71.13) on average, the experimental performances between the techniques with LoRA and the ones without adapters are different. Namely, the performances of BinAdapter and M-NMT between eliminating adapters and adding LoRA modules are hard to rank each other. Interestingly, although we observe similar performance between LoRA + CL and BinAdapter in Scenario 1, LoRA fails to effectively adapt to which new instructions have been introduced. Figure 4 depicts a learning process with F1 per epoch between BinAdapter with LoRA and BinAdapter (adapter modules only). Finally, the lower plot in Figure 5 shows that the overall performance of BinAdapter with incremental datasets does not fluctuate.

6.3.3 Scenario 3. **Baselines.** We utilize the same baselines with Scenario 2 because Scenario 3 requires to handle new vocabularies

Table 4: Performance comparison of BinAdapter with retraining, and five CL baseline techniques under Scenario 2 with incremental datasets from BinKit ($B[1-3]$ in Table 2). Our experiments include both BinAdapter and M-NMT with LoRA and without Adapters. BinAdapter surpasses all other techniques, achieving an F1 of 70.47 on average.

Technique	Dataset	Precision	Recall	F1	Rouge-L
None (Original Model)	A	71.52	71.53	71.52	73.73
Retraining	B1	66.21	64.34	65.47	69.23
	B2	67.83	66.01	66.87	71.08
	B3	**79.29**	**76.91**	**76.87**	**77.82**
M-NMT w/o Adapters [4]	B1	11.35	8.54	9.08	10.02
	B2	15.12	14.12	14.96	14.45
	B3	14.35	14.35	13.91	16.42
M-NMT w/ LoRA [4, 28]	B1	8.43	6.01	6.52	12.34
	B2	13.02	13.99	13.59	23.06
	B3	15.09	19.35	15.48	20.89
M-NMT [26]	B1	32.65	29.88	30.22	35.72
	B2	33.56	34.98	33.91	40.38
	B3	52.99	53.33	51.78	52.61
BinAdapter w/o Adapters	B1	66.03	64.32	64.67	68.56
	B2	66.99	64.97	65.78	65.87
	B3	66.44	62.15	62.46	63.32
BinAdapter w/ LoRA	B1	55.78	54.32	54.08	53.60
	B2	61.33	58.44	59.57	60.60
	B3	67.59	61.91	62.51	62.93
BinAdapter	B1	**68.97**	**66.42**	**67.09**	**67.75**
	B2	**72.87**	**70.23**	**70.96**	**71.74**
	B3	75.65	73.91	73.38	74.67

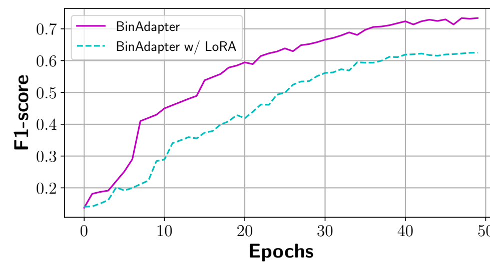

Figure 4: Illustration of a learning process per epoch between BinAdapter w/ LoRA and BinAdapter adaptation process using the $B3$ dataset under Scenario 2.

as well. Note that we evaluate BinAdapter with both SPEC2006 and BinKit datasets from Table 2.

Results with SPEC2006. Table 5 reports the precision, recall, F1, and Rouge-l of BinAdapter compared with full retraining and the five baseline techniques for S1 and S2 datasets. M-NMT and BinAdapter are comparable F1 (60.55 VS 61.37) and Rouge-l values (61.45 VS 62.03) with a marginal improvement of BinAdapter. Additionally, the techniques that detach adapters show noticeable performance degradation whereas the ones that insert LoRA modules demonstrate slightly lower performance than the original M-NMT and BinAdapter. A final note is that, even with full retraining, the performance is no better than BinAdapter on average (F1 of

Table 5: Performance comparison results between BinAdapter, five baseline CL techniques, and full-retraining under Scenario 3 with incremental datasets from SPEC2006 ($S[1-2]$ in Table 2). Our experiments include both BinAdapter and the M-NMT approaches with LoRA and without adapters.

Technique	Precision	Recall	F1	Rouge-l
None (Original Model)	71.52	71.53	71.52	73.75
Retraining	61.32	60.02	60.29	61.91
M-NMT w/o Adapters	42.06	39.78	40.13	43.04
M-NMT w/ LoRA	57.11	56.32	56.23	56.69
M-NMT	61.67	59.85	60.55	61.45
BinAdapter w/o Adapters	44.67	42.34	43.47	45.38
BinAdapter w/ LoRA	55.38	53.84	54.12	54.38
BinAdapter	**62.93**	**61.12**	**61.37**	**62.03**

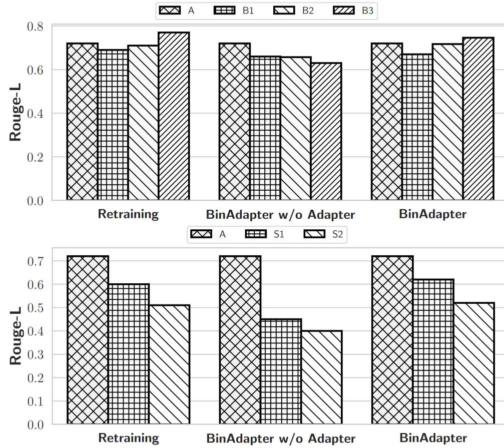

Figure 5: Performance comparison between BinAdapter and four baseline CL techniques with incremental datasets under Scenario 2 (upper) and Scenario 3 (lower). The Rouge-l with A indicates the initial (original) performance in AsmDepictor [39]. The performance with BinAdapter is comparable to retraining cases.

61.37 and Rouge-l of 62.03). The upper plot in Figure 5 depicts the decreasing Rouge-l values with incremental datasets, which we hypothesize that the incremental size of the dataset (*i.e.*, SPEC2006) is relatively small (*e.g.*, < 10%).

Results with BinKit. Similarly, Table 6 briefly describes the performance comparison with the BinKit dataset (*e.g.*, $B[1-3]$). Similar to others, BinAdapter shows comparable performance to M-NMT with a marginal advance (F1 of 71.24 and Rouge-l of 72.53) on average.

6.4 Efficiency (RQ2)

Training Parameters. BinAdapter incorporates three distinct approaches to accommodate different types of incremental datasets. Each approach is tailored to handle datasets that contain unseen

Table 6: Performance comparison between BinAdapter and M-NMT as a baseline under Scenario 3 with B[1-3] dataset). BinAdapter outperforms M-NMT by a small margin.

Technique	Dataset	Precision	Recall	F1	Rouge-l
M-NMT	B1	70.12	69.06	68.34	69.46
	B2	70.34	68.57	69.01	69.78
	B3	71.99	70.10	69.52	70.67
BinAdapter	B1	**71.85**	**69.38**	**69.91**	**70.67**
	B2	**73.97**	**71.63**	**72.84**	**73.44**
	B3	**74.11**	**72.02**	**71.51**	**72.91**

Table 7: Training layers per scenario and the number of training parameters. BinAdapter takes three approaches into account, relying on unseen tokens in the source, target, or both. Our findings show that inserting adapters is sufficient in case of no new tokens in an incremental dataset, source and target embedding layers are required to achieve a desirable performance.

Scenario	Training Params	Training Layers
1	1.5M (3.1%)	Adapters
2	17.2M (43.1%)	Src Emb + Adapters
3	21.2M (53.1%)	Tgt Emb + Src Emb + Adapters

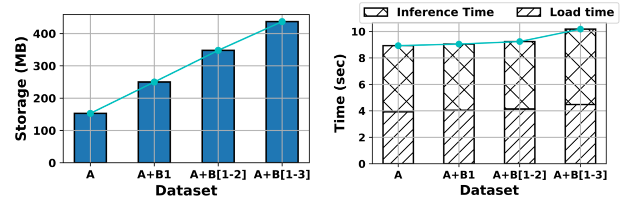

Figure 6: Illustration of storage and inference time overheads with the new dataset arrivals. As the number of modules increases in BinAdapter, the whole model size grows proportionally (left). Similarly, both loading adapters/embeddings and inference (*e.g.*, 100 samples) rise (right).

tokens in the source, target, or both. While the last approach (Scenario 3; $(\mathcal{S}, \mathcal{T}) \rightarrow (\mathcal{S}^{+}, \mathcal{T}^{+})$) is capable of covering all scenarios, we deliberately separate each case to be handled with a different approach for the sake of efficiency. Table 7 outlines the target layers to be trained for each scenario and the corresponding number of training parameters. In the absence of unseen tokens in an incremental dataset, a lightweight model that solely inserts adapters demonstrates minimal performance degradation, with 3.1% of the number of parameters for initial training. However, our findings show that introducing either source or target embedding layers achieves a desirable performance improvement, even with the increase in parameters (*e.g.*, 43.1% and 53.1%, respectively).

Storage Overheads and Inference Time. As the number of incremental datasets increases, the number of adapter modules increases

Table 8: Ablation study to determine the optimal layer(s) for the source tokens on an incremental dataset. The experimental results reveal that tuning the source embedding layer alone produces comparable performance to others, having a relatively small number of trainable parameters (RQ3 in Section 6.5).

Fine-tuned Layer(s)	F1-score	Rouge-l	Trained Params
Src Emb Layer	64.42	63.22	**17M (42.9%)**
Src Emb + 1st Encoder	62.89	66.54	20M (51.1%)
Src Emb + 1st/2nd Encoder	64.67	67.34	23M (59.4%)
Src Emb + All Encoder	63.79	63.78	26M (67.7%)

Table 9: Analysis on the impact of the number of adapters in a layer (RQ3 in Section 6.5). Unlike the original design that introduces two adapters in a single layer, we insert a single adapter per layer because increasing the number of adapters does not lead performance advancement. Note that we assume Scenario 1 (Section 6.3) using the $A1$ dataset (Table 2).

# of Adapters Per Layer	F1	Rouge-l	Trained Params
1	61.17	61.87	**1.5M (3.1%)**
2	**61.31**	**61.88**	3.2M (6.1%)

because BinAdapter utilizes M-NMT with adapters, which incurs more storage overheads by design. Figure 6 (left) illustrates the storage overhead where the whole size of adapters grows (closely) proportional to the arrival of new datasets. In a similar vein, Figure 6 (right) depicts the breakdown of the inference overheads into loading and inference time, being affected by additional datasets.

6.5 Design Appropriateness (RQ3)

Ablation Study for Fine-tuning Layers. We conduct an ablation study to determine the optimal layer(s) for the (source) vocabularies on an incremental dataset. In this study, we train a Transformer model by incrementally unfreezing (*i.e.*, training) one encoder layer at a time, while comparing performance metrics and the number of trainable parameters. As shown in Table 8, our finding indicates that introducing additional layers does not improve the overall performance whereas the number of parameters increases accordingly, for example, from 17M to 26M. Finally, we make the decision to incorporate a source embedding layer for source tokens and a target embedding layer for target tokens from an incremental dataset.

Number of Adapters. We investigate the impact of inserting adapters on the number of trainable parameters (*i.e.*, weights), aiming to strike a balance between efficacy and efficiency. We conduct two separate experiments, namely Scenario 1 (Section 6.3) using the $A1$ dataset (Table 2) as followings. First, we assess the number of adapters in a single layer. The original design of adapters [26] harnesses two adapters at each Transformer layer; *i.e.*, between two feed forward layers and layer normalization. However, as in Table 9, our results indicate that inserting two adapters does not significantly increase overall performance, despite doubling the number of parameters (*i.e.*, 1.5M → 3.2M). Second, we examine the impact of the number of adapters across the whole Transformer structure (*i.e.*, encoders and decoders). Table 10 presents the results for four cases depending on the adapter insertion locations: *e.g.*, the first encoder layer (1 adapter), the first encoder and decoder layers (2 adapters), all three encoder layers (3 adapters), and all three encoder and decoder layers (6 adapters). In this scenario, we observe a notable performance improvement, with F1 scores from 34.4% to 61.2%, which scales proportionally with the number of adapters. Based on these findings, we choose to insert a total of six adapters, with one adapter for each encoder and decoder for BinAdapter.

6.6 Learning Code Semantics (RQ4)

This section explores the BinAdapter's capability of learning transformed code semantics in practice. To this end, we generate an additional dataset with different optimization levels including O0, O1, and O3 for both C and C++ written programs in BinKit [6]. Table 11 summarizes the extended corpus statistics: the number of functions (1,444,781 in total) and the number of tokens for instructions and function names. Because the original Asmdepictor [39] has been trained with O2 alone, we train our own model with the dataset, followed by explicit training BinAdapter under Scenario 2 (Section 3.3). Table 12 reports that BinAdapter can successfully learn a new dataset with a smaller size of model parameters (40%), demonstrating comparable performance with retraining.

6.7 Case Study

With the incremental dataset from the binaries compiled for `MIPS` (*i.e.*, $B2$ in Table 2) BinAdapter quickly learns both new source and target tokens. We observe that, in some cases, our model with CL assists to make a better inference (*i.e.*, closer to the ground truth) than a full training model. For example, BinAdapter predicts the function symbol name of `elfcore_write_ppc_vmx` as `elfcore_write_ppc_tm_cvmx` whereas `stab_class_method_var` from the full training model. Table 13 showcases the examples of successfully inferring function symbol names with our CL techniques. Meanwhile, the prediction has been failed with the original model due to the lack of new vocabulary knowledge (*e.g.*, MIPS instructions).

7 DISCUSSIONS AND LIMITATIONS

Alternative Solutions to CL. One of the ultimate goals is to build an artificial agent that could autonomously and constantly learn complex knowledge from ever-growing information like humans. One straightforward approach for knowledge accumulation would be re-training the whole dataset from scratch. Although re-training every time could preserve model performance, it inevitably suffers from an immense computational overhead. Another direction for learning new knowledge is a fine-tuning technique (*i.e.*, transfer learning) on a new dataset based on a pre-trained model. While consuming relatively lower resources, the downside of fine-tuning is the significant loss for existing knowledge (*i.e.*, CF) when learning new information. Note that CL techniques come into play to retain performance at a reasonable computational cost.

Robustness against Code Transformations. A binary can be diversified through aggressive transformations with various code

Table 10: Analysis on the impact of the number of adapters across the entire Transformer layers (both encoder and decoder). We observe a remarkable performance enhancement by increasing the number of adapters. Based on the results, the current design of BinAdapter has a total of six adapters for all layers (RQ3 in Section 6.5).

# of Adapters	F1	Rouge-l	Trained Params	Adapter inserted Layers
1	34.36	36.57	0.2M (0.5%)	1 Encoder Layer
2	35.23	37.02	0.5M (1.1%)	1 Encoder Layer, 1 Decoder Layer
3	57.35	57.96	0.8M (1.6%)	3 Encoder Layers
6	**61.17**	**61.87**	1.5M (3.1%)	3 Encoder Layers, 3 Decoder Layers

Table 11: A new dataset compiled with multiple optimization levels from BinKit [36] (RQ4 in Section 6.6).

Corpus	B1 (ARM)			B2 (MIPS)			B3 (x86_64)		
# of Functions	O0	O1	O3	O0	O1	O3	O0	O1	O3
Train	104,787	113,379	87,285	188,176	167,630	126,664	84,825	81,464	77,753
Valid/Test	24,845	20,008	15,404	33,208	29,582	22,353	21,206	20,365	19,438
# of Tokens w/ BPE (Instruction)	8,032	9,084	9,151	6,661	9,149	9,144	9,495	9,174	9,160
# of Tokens (Function Name)	7,627	7,407	6,229	8,580	8,153	9,613	4,237	4,096	4,034

Table 12: We confirm that BinAdapter is capable of learning new knowledge with the additional dataset (Table 11). With 40% parameters of the retrained model, BinAdapter shows comparable F1 values (RQ4 in Section 6.6).

		BinAdapter			Base (Retraining)		
Dataset	Opt.	Precision	Recall	F1	Precision	Recall	F1
B1	**O0**	65.84	63.66	64.73	64.07	62.97	63.51
	O1	66.39	64.13	65.24	65.95	64.57	65.25
	O3	75.04	72.53	72.63	76.22	74.94	74.83
B2	**O0**	71.45	70.72	70.15	72.96	72.79	72.35
	O1	69.73	70.06	68.99	71.15	70.31	69.93
	O3	71.62	71.1	70.43	71.84	70.92	70.59
B3	**O0**	84.65	83.96	84.31	85.17	84.37	84.76
	O1	82.96	81.01	80.85	81.89	80.29	80.17
	O3	72.09	71.62	71.85	72.15	70.63	70.16

obfuscation techniques (*e.g.*, dead code insertion, opaque predicates, loop unrolling, polymorphism). Such disturbing cases that include deliberately aggressive transformations are orthogonal to our work, however, we hypothesize that BinAdapter could additionally learn some code structures from an incremental dataset. Conversely, BinAdapter could potentially acquire a deeper understanding of code semantics from a dynamically evolving function, such as a version that has undergone bug patches.

Generalizability of Continual Learning to Other Models. Recent advances in NLP and CL have demonstrated the successful utilization of adapters for language models while preserving previous knowledge. The key aspect of an adapter lies in a module replacement during inference. Likewise, BinAdapter incorporates adapter modules into the Transformer-based architecture like BERT [13] and GPT [62]. However, the optimal insertion location(s) within a model remains an open problem (but agnostic to our approach) for adapter-based CL techniques.

Storage Overheads and Inference Time. BinAdapter unavoidably requires more resources as the number of modules increases. Accordingly, seeking and loading the appropriate adapter modules may take a while, impeding prompt inferences. Modotto et al. [46] propose an Entropy-based classifier to select the correct adapter during test time, but their focus is primarily on Task-Oriented Dialogue Systems, which deviates from our data incremental settings. Hence, handling multiple adapters to augment similar embeddings and adapters efficiently is part of our future work.

Continual Learning Techniques Requiring a Previous Dataset. Replay-based CL approaches (*e.g.*, [45]) show superior performance over regularizers [41] without extra modules to be trained. Recently, Gao et al. demonstrate an advanced replay-based technique, REPEAT [18] for managing ever-incremental source code datasets with comparably less forgetting. However, these techniques require pre-selected exemplars from a previous dataset, which we put aside from our baselines. Meanwhile, BinAdapter addresses CF in exchange for an additional space and training time along with an incremental dataset, which does not need prior exemplars.

8 RELATED WORK

We survey the literature on CL in NLP, categorizing varying techniques into three main types: regularization, rehearsal, and architectural strategies. Meanwhile, ML-assisted binary reversing has been extensively explored in the field.

Regularization-based CL. One direction to mitigate CF for CL is to penalize changes to the model parameters that would negatively impact the performance on previously learned tasks. Elastic Weight Consolidation (EWC) [41] places a penalty on the change in model parameters by estimating the importance of each parameter from previous tasks. Similarly, Synaptic Intelligence (SI) [72] introduces a computationally efficient technique, which computes the importance of each parameter based on the sensitivity of a loss function upon parameter changes. We compare BinAdapter with the two above regularization-based CL techniques as a baseline.

Table 13: Examples of successful inferences with BinAdapter with the $B2$ (MIPS) dataset. MR denotes "Matching Words Ratio" by taking the ratio of correctly predicted ones out of all tokens. We mark the ratio in bold in case that BinAdapter's prediction shows a higher MR. The rightmost column demonstrates the prediction failures from the original model that does not learn a new vocabulary (*e.g.*, MIPS instructions).

Ground Truth	Prediction w/ CL	MR	Prediction w/ Retraining	MR	Prediction w/ Original Model
ctf_dedup_mark_conflicting_hash	ctf_add_align	0.20	ctf_add_cu_mapping	0.20	ungettoken_ungettoken
ctf_hash_eq_integer	gnu_hash	0.25	ctf_hash_eq_string	0.75	mov_r11_r9_ungettoken
debug_make_static_member	debug_make_new	0.50	debug_make_typed_constant	0.50	ungettoken_ungettoken
dwg_add_style	dwg_add_ltype	0.67	dwg_add_layer	0.67	mov_rax_qword_ptr_[rsp+0xc0]
dwg_get_cellstylemap	dwg_get_verte_d	**0.67**	dwg_set_acs_con_class	0.33	ungettoken_ungettoken
elfcore_write_ppc_vmx	elfcore_write_ppc_tm_cvmx	**0.75**	stab_class_method_var	0.00	zn5clang4ento19
generate_abstrmethod	generate_given	0.50	generate_free_list	0.50	ungettoken_ungettoken
hash_free_items	hash_free	**0.67**	hash_flush	0.33	ungettoken_ungettoken
htab_create	htab_try_create	0.67	htab_create	1.00	ungettoken_hfsplus
make_relative_prefix_ignore_links	make_relative_prefix	**0.60**	find_in_path	0.00	zn5clang4ento19
md_process_bytes	sha_process_bytes	0.67	sha_process_bytes	0.67	mov_rax_qword_ptr_[rsp+0xc0]

Rehearsal (Replay)-based CL. Another direction [8, 45, 69] utilizes explicit model training on the previously learned data to alleviate catastrophic forgetting during the learning of new tasks. One prominent version of this method is a generative replay [34, 66, 67] technique which involves generating synthetic data that resembles previously learned data, which is then used to train the model. On the other hand, a distillation-based technique uses a teacher-student approach, where the previously learned model acts as a teacher to transfer its knowledge of the new model (*i.e.*, student). Note that we exclude this technique from our baselines due to two factors: ① obtaining exemplars requires the original dataset where we assume not to have access to it in our 2nd and 3rd scenarios and ② creating a separate generator model for assembly instructions poses challenges as assembly instructions are more sensitive compared to a natural language.

Architecture-based CL. Houlsby et al. [26] introduce an architecture that is equipped with adapter layers (within Transformers) that allow for parameter-efficient fine-tuning. Training the adapter modules helps to avoid catastrophic forgetting by freezing the pre-trained parameters of a model. Recent advances improve an adapter-assisted technique [14, 35, 46, 73] by storing the adapter layer tailored to a certain task and plugging it in during inference on the fly. LoRA (Low-Rank Adaptation) [28] proposes an adaptation to a new task efficiently (*e.g.*, number of parameters, memory utilization). In particular, LoRA inserts a module into Transformer's attention layers, achieving much less resource-intensive than inserting MLP adapters [26]. BinAdapter adopts the adapters.

ML-assisted Binary Reversing. A vast number of prior work leverages machine learning techniques to assist binary reversing [23, 47, 49, 71], decompilation [15, 31, 32], variable and function name prediction [9, 12, 24, 29, 43, 58]. Dire [43] proposes a variable name prediction model based on LSTM (Long Short-Term Memory) [22] and GNN [40] with an abstract syntax tree. Debin [24] introduces a prediction system for debug information such as function and variable symbols, which uses an Extremely randomized Tree classifier and a linear probabilistic graphical model. NERO [12], similarly, uses augmented representations of call sites with GNN to infer a function name. This paper mainly focuses on the feasibility of CL for function name prediction (*e.g.*, incremental tokens on both the source and the target) on top of AsmDepictor [39].

9 CONCLUSION

Binary reversing plays an essential role in addressing bugs or crashes in the absence of source code, understanding the internal behavior of binary code. Recent advances in deep neural networks have focused on recovering (disappeared) high-level information in the source, however, traditional static models struggle to handle the continuous growth of binary datasets. In this work, we aim to infer function symbol names using an incremental dataset that consists of previously unseen assembly code and function names, by leveraging the cutting-edge CL techniques without CF. We introduce BinAdapter, a system that can predict function names (*i.e.*, target) from a series of machine instructions (*i.e.*, source), which incorporates both adapters and the M-NMT approach. We have developed a prototype of BinAdapter, and conducted comprehensive experiments to demonstrate its effectiveness and efficiency. Our empirical results with several scenarios (*e.g.*, new tokens either in the source or the target) achieve an average performance improvement of up to 24.3% compared to baselines.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-00421; AI Graduate School Support Program (Sungkyunkwan University), No. 2022-0-01199; Graduate School of Convergence Security (Sungkyunkwan university), No. 2022-0-00688; AI Platform to Fully Adapt and Reflect Privacy-Policy Changes), and the Basic Science Research Program through NRF grant funded by the Ministry of Education of the Government of South Korea (No. NRF-2022R1F1A1074373). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

REFERENCES

[1] Suresh Kumar Amalapuram, Akash Tadwai, Reethu Vinta, Sumohana S Channappayya, and Bheemarjuna Reddy Tamma. 2022. Continual learning for anomaly based network intrusion detection. In *Proceedings of the 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*.

[2] Pratyay Banerjee, Kuntal Kumar Pal, Fish Wang, and Chitta Baral. 2021. Variable Name Recovery in Decompiled Binary Code using Constrained Masked Language Modeling. *CoRR* (2021).

[3] Chaitanya Baweja, Ben Glocker, and Konstantinos Kamnitsas. 2018. Towards continual learning in medical imaging. *CoRR* (2018).

[4] Alexandre Berard. 2021. Continual learning in multilingual NMT via language-specific embeddings. In *Proceedings of the 6th Conference on Machine Translation (WMT@EMNLP)*.

[5] Magdalena Biesialska, Katarzyna Biesialska, and Marta R. Costa-jussà. 2020. Continual Lifelong Learning in Natural Language Processing: A Survey. In *Proceedings of the 28th International Conference on Computational Linguistics (COLING)*.

[6] BinKit. 2022. Binary Code Similarity Analysis (BCSA) Benchmark. https://github.com/SoftSec-KAIST/BinKit.

[7] BusyBox. 2022. The Swiss Army Knife of Embedded Linux. https://busybox.net.

[8] Arslan Chaudhry, Marc'Aurelio Ranzato, Marcus Rohrbach, and Mohamed Elhoseiny. 2018. Efficient Lifelong Learning with A-GEM. In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*.

[9] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022. Augmenting decompiler output with learned variable names and types. In *Proceedings of the 31st USENIX Security Symposium (Security)*.

[10] Standard Performance Evaluation Corporation. 2023. SPEC2006. https://www.spec.org/cpu2006/.

[11] crosstool NG. 2023. A Versatile (cross) Toolchain Generator. https://github.com/crosstool-ng/crosstool-ng.

[12] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural Reverse Engineering of Stripped Binaries Using Augmented Control Flow Graphs. *Proceedings of the ACM on Programming Languages* (2020).

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. 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 (NAACL)*.

[14] Beyza Ermis, Giovanni Zappella, Martin Wistuba, Aditya Rawal, and Cedric Archambeau. 2022. Memory efficient continual learning with transformers. In *Proceedings of the 36th Annual Conference on Neural Information Processing Systems (NeurIPS)*.

[15] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An end-to-end neural program decompiler. In *Proceedings of the 32th Annual Conference on Neural Information Processing Systems (NeurIPS)*.

[16] Philip Gage. 1994. A new algorithm for data compression. *C Users Journal* (1994).

[17] Han Gao, Shaoyin Cheng, Yinxing Xue, and Weiming Zhang. 2021. 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 (ISSTA)*.

[18] Shuzheng Gao, Hongyu Zhang, Cuiyun Gao, and Chaozheng Wang. 2023. Keeping Pace with Ever-Increasing Data: Towards Continual Learning of Code Intelligence Models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*.

[19] GNU. 2023. GNU Binutils. https://www.gnu.org/software/binutils/.

[20] GNU. 2023. GNU Coreutils. https://www.gnu.org/software/coreutils/.

[21] GNU. 2023. GNU Findutils. https://www.gnu.org/software/findutils/.

[22] Alex Graves and Alex Graves. 2012. Long short-term memory. *Supervised sequence labelling with recurrent neural networks* (2012).

[23] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. 2019. DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis.. In *Proceedings of the 28th USENIX Security Symposium (Security)*.

[24] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[25] Dan Hendrycks and Kevin Gimpel. 2016. Gaussian Error Linear Units (GELUs). *arXiv e-prints* (2016).

[26] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*.

[27] Yen-Chang Hsu, Yen-Cheng Liu, Anita Ramasamy, and Zsolt Kira. 2018. Reevaluating Continual Learning Scenarios: A Categorization and Case for Strong Baselines. In *Proceedings of the 2nd NeurIPS Continual learning Workshop (NeurIPS)*.

[28] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *Proceedings of the 10th International Conference on Learning Representations (ICLR)*.

[29] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. 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 (CCS)*.

[30] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*.

[31] Deborah S Katz, Jason Ruchti, and Eric Schulte. 2018. Using recurrent neural networks for decompilation. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

[32] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. 2019. Towards Neural Decompilation. *CoRR* (2019).

[33] Zixuan Ke and Bing Liu. 2022. Continual Learning of Natural Language Processing Tasks: A Survey. *CoRR* (2022).

[34] Zixuan Ke, Hu Xu, and Bing Liu. 2021. Adapting BERT for Continual Learning of a Sequence of Aspect Sentiment Classification Tasks. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

[35] Zixuan Ke, Hu Xu, and Bing Liu. 2021. Adapting BERT for Continual Learning of a Sequence of Aspect Sentiment Classification Tasks. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

[36] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2022. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering* (2022).

[37] Hyunjin Kim. 2023. Pre-trained Model for the Official Implementation of AsmDepictor. https://zenodo.org/record/7978756.

[38] Hyunjin Kim. 2023. A Transformer-based Function Symbol Name Inference Model from an Assembly Language for Binary Reversing. https://github.com/agwaBom/AsmDepictor.

[39] Hyunjin Kim, Jinyeong Bak, Kyunghyun Cho, and Hyungjoon Koo. 2023. A Transformer-based Function Symbol Name Inference Model from an Assembly Language for Binary Reversing. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security (AsiaCCS)*.

[40] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*.

[41] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences* (2017).

[42] Liang Kou, Donghui Zhao, Hui Han, Xiong Xu, Shuaige Gong, and Liandong Wang. 2023. SSCL-TransMD: Semi-Supervised Continual Learning Transformer for Malicious Software Detection. *Applied Sciences* (2023).

[43] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. Dire: A neural approach to decompiled identifier naming. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[44] Chin-Yew Lin and Franz Josef Och. 2004. Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL)*.

[45] David Lopez-Paz and Marc'Aurelio Ranzato. 2017. Gradient episodic memory for continual learning. In *Proceedings of the 30th Annual Conference on Neural Information Processing Systems (NeurIPS)*.

[46] Andrea Madotto, Zhaojiang Lin, Zhenpeng Zhou, Seungwhan Moon, Paul A. Crook, Bing Liu, Zhou Yu, Eunjoon Cho, Pascale Fung, and Zhiguang Wang. 2021. Continual Learning in Task-Oriented Dialogue Systems. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[47] Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. 2022. {RE-Mind}: a First Look Inside the Mind of a Reverse Engineer. In *Proceedings of the 31st USENIX Security Symposium (Security)*.

[48] Andrea Maracani, Umberto Michieli, Marco Toldo, and Pietro Zanuttigh. 2021. Recall: Replay-based continual learning in semantic segmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (CVF)*.

[49] Luca Massarelli, Giuseppe A Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2019. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*.

[50] Angelo G Menezes, Gustavo de Moura, Cézanne Alves, and André CPLF de Carvalho. 2023. Continual object detection: a review of definitions, strategies, and challenges. *Neural Networks* (2023).

[51] Amir Nazemi, Zeyad Moustafa, and Paul W. Fieguth. 2023. CLVOS23: A Long Video Object Segmentation Dataset for Continual Learning. *CoRR* (2023).

[52] Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. 2021. DIRECT: A Transformer-based Model for Decompiled Variable Name Recovery. *NLP4Prog 2021* (2021).
[53] National Security Agency (NSA). 2019. Software Reverse Engineering (SRE) Suite of Tools. https://ghidra-sre.org/.
[54] Beny Nugraha, Krishna Yadav, Parag Patil, and Thomas Bauschert. 2023. Improving the Detection of Unknown DDoS Attacks through Continual Learning. In *Proceedings of the IEEE International Conference on Cyber Security and Resilience (CSR)*.
[55] OpenSSL. 2023. Cryptography and SSL/TLS Toolkit. https://www.openssl.org.
[56] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. 2019. Continual lifelong learning with neural networks: A review. *Neural networks* (2019).
[57] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Proceedings of the 32th Annual Conference on Neural Information Processing Systems (NeurIPS)*.
[58] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. 2020. Probabilistic naming of functions in stripped binaries. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.
[59] J. Patrick-Evans, M. Dannehl, and J. Kinder. 2023. XFL: Naming Functions in Binaries with Extreme Multi-label Learning. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP)*.
[60] Matthias Perkonigg, Johannes Hofmanninger, Christian J Herold, James A Brink, Oleg Pianykh, Helmut Prosch, and Georg Langs. 2021. Dynamic memory to alleviate catastrophic forgetting in continual learning with medical imaging. *Nature communications* (2021).
[61] Sai Prasath, Kamalakanta Sethi, Dinesh Mohanty, Padmalochan Bera, and Subhransu Ranjan Samantaray. 2022. Analysis of continual learning models for intrusion detection system. *IEEE Access* (2022).
[62] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).
[63] Mohammad Saidur Rahman, Scott Coull, and Matthew Wright. 2022. On the Limitations of Continual Learning for Malware Classification. In *Proceedings of the Conference on Lifelong Learning Agents (CLLA*.
[64] Rico Sennrich. 2023. Subword Neural Machine Translation. https://github.com/rsennrich/subword-nmt.
[65] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*.
[66] Hanul Shin, Jung Kwon Lee, Jaehong Kim, and Jiwon Kim. 2017. Continual learning with deep generative replay. *Advances in neural information processing systems* (2017).
[67] Michiel van der Ven and Andreas S. Tolias. 2018. Generative replay with feedback connections as a general strategy for continual learning. *CoRR* (2018).
[68] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Proceedings of the 30th Annual Conference on Neural Information Processing Systems (NeurIPS)*.
[69] Hong Wang, Wenhan Xiong, Mo Yu, Xiaoxiao Guo, Shiyu Chang, and William Yang Wang. 2019. Sentence Embedding Alignment for Lifelong Relation Extraction. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.
[70] Jianren Wang, Xin Wang, Yue Shang-Guan, and Abhinav Gupta. 2021. Wanderlust: Online continual object detection in the real world. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (CVF)*.
[71] Hongfa Xue, Shaowen Sun, Guru Venkataramani, and Tian Lan. 2019. Machine learning-based analysis of program binaries: A comprehensive study. *IEEE Access* (2019).
[72] Friedemann Zenke, Ben Poole, and Surya Ganguli. 2017. Continual learning through synaptic intelligence. In *Proceedings of the 34th International conference on machine learning (ICML)*.
[73] Yanzhe Zhang, Xuezhi Wang, and Diyi Yang. 2022. Continual Sequence Generation with Adaptive Compositional Modules. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*.