FlowMalTrans: Unsupervised Binary Code Translation for Malware Detection Using Flow-Adapter Architecture

Minghao Hu
George Mason University
mhu20@gmu.edu

Junzhe Wang
George Mason University
jwang69@gmu.edu

Weisen Zhao
George Mason University
wzhao9@gmu.edu

Qiang Zeng
George Mason University
zeng@gmu.edu

Lannan Luo ✉
George Mason University
lluo4@gmu.edu

Abstract

Applying deep learning to malware detection has drawn great attention due to its notable performance. As malware spreads across a wide range of Instruction Set Architectures (ISAs), it is important to extend malware detection capacity to multiple ISAs. However, training a deep learning-based malware detection model usually requires a large number of labeled malware samples. The process of collecting and labeling sufficient malware samples to build datasets for each ISA is labor-intensive and time-consuming. To reduce the burden of data collection, we propose to leverage the ideas of Neural Machine Translation (NMT) and Normalizing Flows (NFs) for malware detection. Specifically, when dealing with malware in a certain ISA, we translate it to an ISA with sufficient malware samples (like X86-64). This allows us to apply a model trained on one ISA to analyze malware from another ISA. We have implemented and evaluated the model on *seven* ISAs. The results demonstrate the high translation capability of our model, enabling superior malware detection across ISAs.

1 Introduction

Malware is defined as software intended to damage computers or associated systems (Preda et al., 2008; Li et al., 2025). In recent years, many malware detection tools have been developed (Xie et al., 2024), and their success largely depends on the underlying techniques. Signature-based detection identifies malware by matching patterns from known malware families (Sathyanarayan et al., 2008), but it often fails to detect altered or novel malware. In contrast, behavior-based detection analyzes program execution to identify suspicious behavior (Liu et al., 2011), yet this method lacks scalability.

The use of deep learning methods like LSTM and CNNs in malware detection has garnered significant interest because of their strong performance (Sewak et al., 2018; Gopinath and Sethuraman, 2023). However, there are two major challenges that hinder the broader adoption of deep learning in this domain.

Challenges. Malware infects a broad range of Instruction Set Architectures (ISAs). With the rise in cyberattacks on IoT devices and computer systems (Wu et al., 2022b,a; Chen et al., 2025; Ma et al., 2023; Tsai et al., 2024; Luo et al., 2021; Li et al., 2020; Zhu et al., 2025; Wu and Arafin, 2025), malware has begun targeting an increasing number of ISAs (Davanian and Faloutsos, 2022; Caviglione et al., 2020). Hence, it is essential to enable malware detection across ISAs. Currently, there are numerous ISAs available (Wang et al., 2023a). Due to this wide variety, training one model for each ISA demands substantial time and effort. In addition, deep learning models rely heavily on having enough malware samples during training, which is problematic for *low-resource* ISAs where such data is limited.

Our Approach. Malware is generally closed-source whose original source code is typically inaccessible. What is available is the binary form of the malware. Once disassembled, this binary can be represented in assembly language. Based on this observation, we propose leveraging Neural Machine Translation (NMT) techniques (Artetxe et al., 2018) to facilitate malware analysis.

When handling a binary in a given ISA (referred to as the *source* ISA), we translate it to an ISA with rich malware samples, such as X86-64, which we refer to as the *target* ISA. Once translated, we use a model trained on the *target* ISA to test the translated code. This approach facilitates malware detection across multiple ISAs using a model trained exclusively on the target ISA, eliminating the need for extensive malware samples in other ISAs.

In order to capture ISA-specific syntax and semantic information, we leverage normalizing flows (NFs), a class of invertible transformations that models complex distributions while maintaining

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 3251–3272

tractability. Traditional translation models often struggle with capturing complex dependencies and handling the variation in data distributions. NFs address this challenge by modeling complex, high-dimensional distributions through invertible transformations, enabling more efficient learning of intricate relationships between source and target data. By doing so, we ensure a smooth and structured mapping between ISAs, leading to more precise and reliable translations. Furthermore, normalizing flows enable efficient sampling and density estimation, which improves the adaptability of our model to unseen binaries and enhances generalization across different architectures.

We design `FlowMalTrans`, which is a flow-adapter-based unsupervised binary code translation architecture. This design allows for robust feature alignment between different ISAs, improving translation fidelity and ultimately enhancing malware detection across ISAs. To bridge the gap between the latent space of the source and target ISA, `FlowMalTrans` learns an invertible transformation of binary representations from one ISA to another, facilitating robust feature alignment across different ISAs and enhancing translation fidelity. `FlowMalTrans` operates in a completely unsupervised manner, eliminating the need for parallel datasets. Notably, the training of `FlowMalTrans` does not require any malware samples and relies only on binaries compiled from the abundance of opensource programs. *Despite never encountering any malware samples during training,* `FlowMalTrans` *is still capable of translating malware across ISAs with high translation quality*.

Results. We have implemented `FlowMalTrans`, and evaluated its performance on *seven* ISAs, including X86-64, i386, ARM32, ARM64, MIPS32, PPC32 and M68K. `FlowMalTrans` achieves higher BLEU score than the baseline method `UnsuperBinTrans` (Ahmad and Luo, 2023). Furthermore, we apply `FlowMalTrans` to the malware detection task and achieve exceptional results, highlighting its superior translation quality, which enables highly effective malware detection across ISAs. Below we highlight our contributions:

- We propose `FlowMalTrans`, a novel unsupervised approach to translate binaries across different ISAs. `FlowMalTrans` addresses the data scarcity issue by enabling the detection of malware in low-resource ISAs using a model trained on the high-resource ISA.
- We leverage normalizing flows (NFs) to capture ISA-specific features. With NFs, `FlowMalTrans` captures code semantics and properly decodes instruction sequences.
- The training of `FlowMalTrans` does not require any malware samples, yet the model is still capable of translating malware across ISAs and achieves high translation quality.
- We expand the evaluation to cover a *broader range of ISAs* compared to prior work, showcasing `FlowMalTrans`'s wider applicability and improved performance.

2 Related Work

2.1 Malware Detection

Signature-Based Detection. Conventional malware detection methods primarily use signature-based approaches, working by recognizing malicious code patterns (Sebastio et al., 2020; Rohith and Kaur, 2021; Sathyanarayan et al., 2008; Behal et al., 2010). However, these methods are ineffective against newly developed or altered malware that modifies its code to avoid being identified.

Behavior-Based Detection. This approach detects malware by examining program behavior during execution (Liu et al., 2011; Burguera et al., 2011; Aslan et al., 2021; Saracino et al., 2016). However, it may produce false negatives when malicious actions are not activated during observation.

Deep Learning-Based Detection. Deep learning has been widely used for detecting malware. A range of models, including CNNs and LSTMs, have been utilized (Sewak et al., 2018; Gopinath and Sethuraman, 2023; Wang et al., 2024; He et al., 2023; Lei et al., 2022; Zhang et al., 2024; Zhao et al., 2025). Nonetheless, these models generally demand substantial datasets of malware samples for training. Consequently, most existing methods concentrate on widely used ISAs like X86 and ARM, leaving *low-resource* ISAs relatively unaddressed.

2.2 Normalizing Flows

Normalizing flows (NFs) are a family of generative models that transform a simple base distribution into a complex target distribution via a series of invertible transformations (Ho et al., 2019). The key idea is to apply a sequence of differentiable mappings to a latent variable. Different types of NFs include coupling layers (Dinh et al., 2014), autoregressive flows (Papamakarios et al., 2017), and continuous normalizing flows (Chen et al., 2018).

The flow adapter architecture has been explored in neural machine translation (NMT), such as vari-

ational NMT (Calixto et al., 2019; Ma et al., 2019; Eikema and Aziz, 2019). Flow adapters leverage NFs to model complex posterior distributions in sequence-to-sequence models, allowing for more flexible latent representations and enhancing the robustness of translation (Shu et al., 2020).

2.3 Source Code Translation

Source code translation converts source code in one language to another. Early work uses *transpilers* or *transcompilers* (Andrés and Pérez, 2017; Tripuramallu et al., 2024), which rely on handcrafted rules. However, they produce unidiomatic translations that prove hard for human programmers to read. Another issue is incomplete feature support, resulting in improper translations.

Deep learning has introduced new approaches to source code translation (Roziere et al., 2020; Weisz et al., 2021; Lachaux et al., 2020). However, since malware is closed-source, *source code translation is **inapplicable** for malware*.

2.4 Program Analysis-Based Binary Code Translation

Several approaches leverage program analysis techniques to convert binary code from one ISA to another. They can be broadly categorized into *static analysis-based* and *dynamic analysis-based*. Static analysis-based translation analyzes and translates the binary code before execution (Shen et al., 2012; Cifuentes and Van Emmerik, 2000). But ensuring all paths are accurately translated is challenging.

Dynamic analysis-based translation performs translation during execution (Chernoff et al., 1998; Ebcioglu et al., 2001), but it requires sophisticated runtime environments. Plus, both approaches face challenges related to *architecture-specific features* and encounter difficulties in achieving accuracy, performance, and compatibility.

Summary. We propose an unsupervised binary code translation approach leveraging ***deep learning techniques***. The state of the art, `UnsuperBinTrans` (Ahmad and Luo, 2023), is the first and only existing work *applying deep learning to binary code translation*. However, it has several limitations, which our model `FlowMalTrans` aims to overcome (Section 3.2). Our evaluation demonstrates that `FlowMalTrans` outperforms `UnsuperBinTrans` in binary code translation and achieves superior malware detection performance.

3 Overview

3.1 Motivation

Malware propagates over various instruction set architectures (ISAs), highlighting the importance of enabling malware detection across multiple ISAs. Nonetheless, training deep learning models generally depends on extensive datasets of malware samples. This requirement becomes more difficult to fulfill with the diversity of ISAs, as *low-resource* ISAs frequently lack adequate malware samples.

To alleviate the challenges of data collection and tackle the problem of limited data, we propose translating binaries from a *low-resource* ISA to a *high-resource* ISA. Thus malware detection model trained on the *high-resource* ISA can be applied to the translated binaries, helping to overcome the data scarcity faced by *low-resource* ISAs.

3.2 Limitations of the State of the Art

`UnsuperBinTrans` (Ahmad and Luo, 2023) is the first and only existing work that applies deep learning techniques to binary code translation. While InnerEye (Zuo et al., 2018) applies NMT techniques for binary code similarity comparison, it *does not perform binary code translation across ISAs*. Instead, it employs two encoders from NMT models to generate embeddings for binary pairs. `UnsuperBinTrans`, however, has several limitations, which our model aims to overcome.

Shared Encoder *vs.* Separated Encoders. `UnsuperBinTrans` employs a shared encoder for ISAs, which limits the model's ability to capture ISA-specific syntax and semantics, leading to suboptimal translation. In contrast, `FlowMalTrans` employs a separate encoder for each ISA, allowing each to learn specialized representations.

RNN *vs.* Transformer. `UnsuperBinTrans` relies on RNNs, which suffer from vanishing gradients and difficulties in capturing long-range dependencies. In contrast, `FlowMalTrans` leverages Transformer, which employ self-attention mechanisms to model long-range dependencies.

ISA-Agnostic *vs.* ISA-Specific Latent Representations. `UnsuperBinTrans` learns latent representations for binaries that are ISA-agnostic by abstracting away ISA-specific details. In contrast, `FlowMalTrans` learns ISA-specific latent representations that preserve crucial architectural distinctions while maintaining alignment with a shared cross-ISA space. This is achieved by employing NFs. By transforming the source-ISA representa-

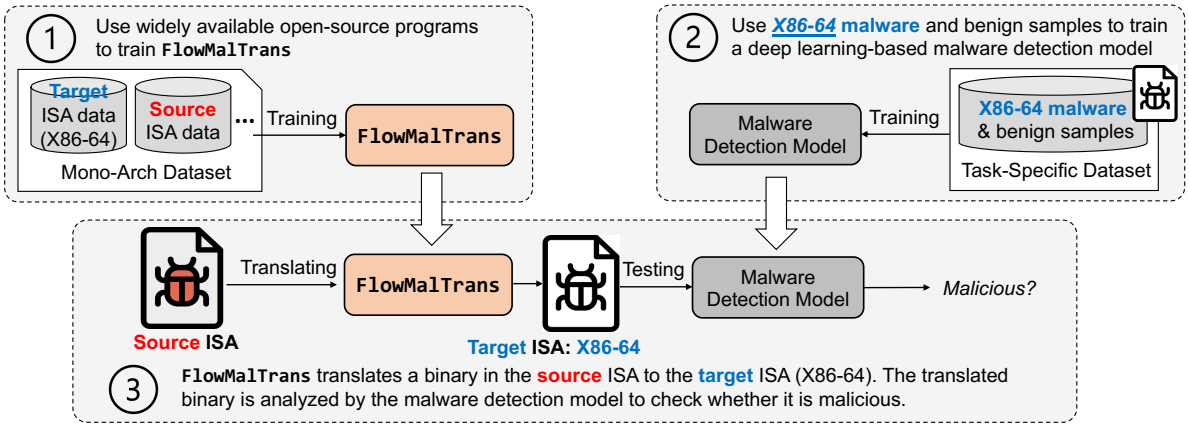

Figure 1: Applying FlowMalTrans to detect malware in a source ISA using a malware detection model trained on the target ISA (e.g., X86-64).

tions into the target-ISA representations via NFs, the decoder can leverage the representations to generate better aligned target-ISA binary code.

Two ISAs *vs.* Broader ISAs. UnsuperBinTrans is limited to only two ISAs (X86-64 and ARM32), whereas FlowMalTrans extends its coverage, supporting seven ISAs.

3.3 Model Applications

Figure 1 shows the workflow of applying FlowMalTrans to detect malware across ISAs. In step ①, we train FlowMalTrans to translate binaries from the source ISA to the target ISA (such as X86-64). The training dataset can be constructed using various available *opensource* programs. *The training of* FlowMalTrans *does not require any malware samples*. It should be noted that malware is typically a closed-source program; thus *cross-compilation that generates binaries across ISAs from source code does not apply to malware*.

In step ②, we train a deep learning model using a *task-specific dataset* containing malware and benign samples in the target ISA.

Finally, in step ③, when dealing with a binary in the source ISA, we use FlowMalTrans to translate the binary to the target ISA and reuse the malware detection model trained on the target ISA to test the translated code for detecting malware.

Note that due to the scarcity of malware samples in a low-resource ISA, directly training a robust malware detection model on such an ISA is challenging. Our approach overcomes this limitation through code translation, making robust malware detection feasible for low-resource ISAs.

4 Model Design

We present the design and training of FlowMalTrans. Notably, training FlowMalTrans requires only mono-architecture datasets for each ISA, *without the need for any malware samples*.

4.1 Instruction Normalization

A binary, after being disassembled, is represented as a sequence of instructions. An instruction includes an opcode and zero or more operands. We regard *opcodes/operands as words* and *basic blocks as sentences*. A basic block is a straight-line sequence of instructions with no branches inside it.

In Natural Language Processing (NLP), the out-of-vocabulary (OOV) issue is a well-known problem. To mitigate the OOV problem, we employ the normalization strategy.

- (R1): We use IDA Pro (IDA, 2023) to disassemble binaries, which generates dummy names (Dummy name, 2023). We replace dummy names with their respective prefixes. For example, off_ and seg_ represent offset pointer value and segment address value. They are replaced with <OFF> and <SEG>.
- (R2): Function names are replaced by <FUNC>.
- (R3): Number and hexadecimal constants are replaced by <VALUE> and <HEX>, respectively.

We provide examples in Appendix A.2 that illustrate how these normalization rules are applied to assembly code across different ISAs.

4.2 Normalizing Flows

We consider basic blocks as sentences, and instructions as words. The prior state-of-the-art,

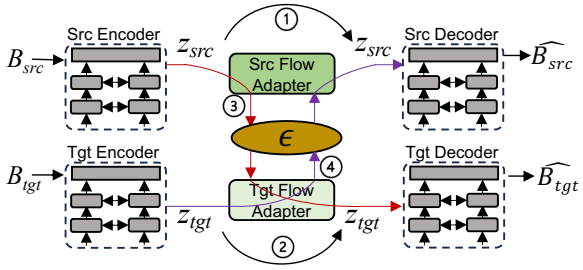

Figure 2: Model architecture of FlowMalTrans, which contains a pair of flow adapters for modeling the distributions of basic block representations in the source ISA and target ISA, respectively.

UnsuperBinTrans, assumes that the latent representations of source and target basic blocks share a common semantic space, and learns ISA-agnostic latent representations, which abstract away ISA-specific details. However, this can be overly restrictive, as different ISAs exhibit distinct syntactic and structural characteristics that are not always easily aligned in a shared latent space. By enforcing ISA-agnostic representations, UnsuperBinTrans may lose critical ISA-specific features necessary for accurate translation, leading to suboptimal performance.

Our model FlowMalTrans assumes the representations are different due to ISA-specific characteristics. Thus, they are modeled separately for each ISA, which makes it possible to better capture basic-block semantics in an ISA-specific manner. Specifically, FlowMalTrans uses a pair of NFs (i.e., source flow and target flow) to model the distributions of basic-block latent representations in the source and target ISA. During translation, a latent code transformation is performed to transform the source representation z_{src} into the target representation z_{tgt}, which is fed to the decoder to generate a better basic block in the target ISA. Figure 2 shows how NFs are employed in our model.

Modeling Latent Representations by NFs. We use a pair of NFs to model the distribution of the basic block latent representations in different ISAs, i.e., $p_{z_{src}}(z_{src})$ and $p_{z_{tgt}}(z_{tgt})$, where z_{src} and z_{tgt} are the latent representation of the source basic block and target basic block, respectively. Through NFs, we transform the distributions of the source and target representations to the base distribution ϵ (e.g., standard normal distribution), which can be viewed as the "true" underlying semantic space, abstracting away from ISA specifics. We denote such mappings as $\mathbf{G}_{(z_{src}\to\epsilon)}$ and $\mathbf{G}_{(z_{tgt}\to\epsilon)}$.

Designing NFs. We use Real-valued Non-volume Preserving (RealNVP), a type of NFs designed for efficient density estimation and sampling (Dinh et al., 2022), to build the source and target flow.

Two key components of RealNVP are: *Multi-scale Architecture* and *Affine Coupling Layer*. With them, RealNVP can: (1) capture ISA-specific representations for each ISA at both local and global scales; (2) transform these encoded latent representations from one ISA to another invertibly; and (3) train using maximum likelihood estimation (MLE) with a tractable Jacobian determinant.

We construct FlowMalTrans with two RealNVP models for the source flow and target flow. Each RealNVP model consists of three sequential flows to perform the latent code transformation. We explicitly model the source and target blocks with K sequential flows (where $K = 3$ in our case):

$$p_{z_{src}}(z_{src}) = p_\epsilon(\epsilon)\prod_{i=1}^{K}\left|\det\frac{\partial f_{src}^{(i)}(z^{(i)})}{\partial z^{(i)}}\right|^{-1} \quad (1)$$

$$p_{z_{tgt}}(z_{tgt}) = p_\epsilon(\epsilon)\prod_{i=1}^{K}\left|\det\frac{\partial f_{tgt}^{(i)}(z^{(i)})}{\partial z^{(i)}}\right|^{-1} \quad (2)$$

where $p_\epsilon(\epsilon)$ is a base distribution. We select standard Gaussian distribution $\mathcal{N}(0,1)$ as our base distribution here. $f_{src}^{(i)}$ is the i-th transformations for the source blocks. $z^{(i)}$ is the intermediate variable where $z^{(1)} = \epsilon$ and $z^{(K)} = z_{src}$.

We denote the mapping process in Equation (1) as $\mathbf{G}_{(\epsilon\to z_{src})}$ and Equation (2) as $\mathbf{G}_{(\epsilon\to z_{tgt})}$. Given the invertibility feature of NFs, the mappings are also invertible. Thus, we have: $\mathbf{G}_{(\epsilon\to z_{src})} = \mathbf{G}^{-1}_{(z_{src}\to\epsilon)}$ and $\mathbf{G}_{(\epsilon\to z_{tgt})} = \mathbf{G}^{-1}_{(z_{tgt}\to\epsilon)}$. To achieve latent code transformation from source to target, we can formalize the transformation process as the composition of $\mathbf{G}_{(z_{src}\to\epsilon)}$ and $\mathbf{G}_{(\epsilon\to z_{tgt})}$:

$$\mathbf{G}_{(z_{src}\to z_{tgt})} = \mathbf{G}_{(z_{src}\to\epsilon)} \circ \mathbf{G}_{(\epsilon\to z_{tgt})} \quad (3)$$

Therefore, we can learn a transformation $\mathbf{G}_{(z_{src}\to z_{tgt})}$ by learning the transformation $\mathbf{G}_{(z_{src}\to\epsilon)}$ and $\mathbf{G}_{(\epsilon\to z_{tgt})}$. We also notice that $\mathbf{G}_{(z_{src}\to z_{tgt})}$ and $\mathbf{G}_{(z_{tgt}\to z_{src})}$ are invertible because they are compositions of two invertible mappings. Thus, We can follow a similar procedure and learn $\mathbf{G}_{(z_{tgt}\to z_{src})}$.

4.3 Encoder & Decoder for Code Translation

For binary code translation, different ISAs exhibit significant syntactic and semantic differences. To capture and align these variations, we propose a

separated-encoder design, where each ISA is assigned a dedicated encoder. The source encoder and decoder are responsible for encoding and generating basic blocks in the source ISA, while the target encoder and decoder handle basic blocks in the target ISA. `FlowMalTrans` employs a multi-layer bi-directional Transformer (Vaswani, 2017) to construct the encoders and decoders for the source and target ISA, as shown in Figure 2.

Encoding. Here we describe how the source encoder encodes the source basic block and generates the source basic block representation z_{src}. A similar procedure is applied to the target basic block representation generation.

The source encoder processes the source basic block $\mathbf{B}_{src} = \{\boldsymbol{b}_0, \cdots, \boldsymbol{b}_S\}$, and generates the hidden representations $\{\boldsymbol{h}_0, \cdots \boldsymbol{h}_S\}$. The basic block-level representation z_{src} is computed using Equation (4). The target encoder follows a similar process to encode the target basic block.

$$\begin{aligned} z_{src} = \boldsymbol{W}(& \text{ max-pool}([\boldsymbol{h}_0, \ldots, \boldsymbol{h}_S]) \\ & + \text{mean-pool}([\boldsymbol{h}_0, \ldots, \boldsymbol{h}_S]) \\ & + \boldsymbol{h}_0) \end{aligned} \tag{4}$$

Cross-lingual Translation. To enable the decoder to better leverage ISA-specific latent representations, we perform a latent code transformation. For example, as shown in Figure 2, after calculating z_{src} using Equation (4), we employ the source flow to transform z_{src} to ϵ, and then the target flow to transform ϵ to z_{tgt}, which is in the target latent representation space. Then, the target decoder can decode z_{tgt} to generate the target basic block.

In contrast, the prior work `UnsuperBinTrans` indiscriminately uses the same representational space for both source and target, without performing latent code transformation.

Decoding. To capture the semantics and mitigate improper alignments between the source and target ISA, we follow the procedure in (Setiawan et al., 2020) to generate the decoded representation:

$$\boldsymbol{o}_i = (1 - \boldsymbol{g}_i) \odot \boldsymbol{s}_i + \boldsymbol{g}_i \odot \boldsymbol{z} \tag{5}$$

where $\boldsymbol{g}_i = \sigma([\boldsymbol{s}_i; \boldsymbol{z}])$, $\sigma(\cdot)$ is the sigmoid function. $\boldsymbol{o}_i$ denotes Hadamard product between two tensors. The values in $\boldsymbol{g}_i$ control the contribution of $\boldsymbol{z}$ to $\boldsymbol{o}_i$.

4.4 Model Training

We first employ the causal language modeling (CLM) and masked language modeling (MLM) objectives to pretrain the encoders and decoders. We then train `FlowMalTrans` in an unsupervised manner using three objectives: *denoising auto-encoding (DAE)*, *back translation (BT)*, and *maximum likelihood estimation (MLE)*. The DAE reconstructs a basic block from its noised version, as illustrated in steps ① and ② in Figure 2. The BT process, shown in steps ③ and ④, involves performing the latent code transformation twice while jointly training the encoders and decoders. For example, in BT for the source ISA, the transformation first occurs in the source-to-target direction, followed by the target-to-source direction. Appendix A.1 provides further details.

Maximum Likelihood Estimation (MLE). The NFs are directly trained by maximum likelihood estimation (MLE) of basic-block level latent representations. The objective can be formulated as:

$$\mathcal{L}_{MLE}(\mathbf{G}_{(z_{src} \to \epsilon)}) = E_{z \sim p_{z_{src}}}[\log p_{z_{src}}(z)] \tag{6}$$

where $E_{z \sim p_{z_{src}}}$ is approximated by sampling mini-batches of latent representations generated by the encoder during training. By minimizing the loss, we can construct $\mathbf{G}_{(z_{src} \to z_{tgt})}$ and $\mathbf{G}_{(z_{tgt} \to z_{src})}$.

Trained with the DAE, BT, and MLE objectives, `FlowMalTrans` effectively translates basic blocks across ISAs. During testing, it translates each basic block of a given binary from the source ISA and then concatenates the translated blocks to form a translated binary for the target ISA.

5 Evaluation

5.1 Experimental Settings

We implement `FlowMalTrans` using Transformer with 64 hidden units, 4 heads, ReLU activations, a dropout rate of 0.1, and learned position embeddings. Appendix A.4 presents the details. All the experiments were conducted on a computer with a 64-bit 3.6 GHz Intel Core i9-CPU, a Nvidia GeForce RTX 4090 and 64GB RAM.

Model Comparison. We consider four *baseline models* and one *Same-ISA model* for comparison.

- *Baseline Model 1:* `UnsuperBinTrans` (Ahmad and Luo, 2023) is the first and only existing work that focuses on binary code translation leveraging deep learning (Section 3.2).
- *Baseline Model 2:* `UniMap` (Wang et al., 2023a) also aims to resolve the data scarcity issue in malware detection. `UniMap` learns transfer knowledge to enable the reuse of a model across ISAs. In contrast, we focus on binary code translation to enable model reuse.

- *Baseline Model 3:* `CrossIns2Vec` (Wang et al., 2024), similar to `UniMap`, aims to learn transfer knowledge to enable model reuse.
- *Baseline Model 4: IR-based malware detection model*. Intermediate representation (IR) can abstract away ISA differences and represent binaries in a uniform style (angr, 2024).
- *Same-ISA model.* We consider a model trained and tested on the *same* ISA, *without employing any translation*, as the Same-ISA model. As expected, this model, *if trained with sufficient data*, is likely to outperform a model trained on one ISA and tested on another, representing the best-case scenario.

5.2 Training `FlowMalTrans`

We consider seven ISAs: X86-64, i386, ARM64, ARM32, MIPS32, PPC32 and M68K. We translate binaries from other ISAs to X86-64.

Training Datasets. We collect various open-source programs, including *openssl*, *binutils*, *findutils*, and *libgpg-error*, which are widely used in prior binary analysis works (Ding et al., 2019; Li et al., 2021). We compile them for each ISA using `GCC` with different optimization levels (O0-O3) and disassemble the binaries using IDA Pro (IDA, 2023) to extract basic blocks, which are normalized and deduplicated. Finally, we create a training dataset for each ISA. The adequacy of these datasets is evaluated in Appendix A.3.

Byte Pair Encoding. We use byte pair encoding (BPE) to process the datasets. We set the BPE merge times based on the principles derived from empirical experiments and theoretical insights:

- The vocabulary size discrepancy between the source and target ISA should $< 15\%$. A large vocabulary discrepancy can lead to an imbalanced learning problem, resulting in inefficiencies and overfitting (Gowda and May, 2020).
- The vocabulary size of each ISA should $< 12k$, to balance capturing word semantics with computational resource constraints. A large vocabulary size can negatively impact model performance due to increased complexity and sparse token distributions (Jean et al., 2014).

As shown in Table 1, we can see that the vocabulary discrepancy within each pair is small, making them well-suited for training. Note that these principles are tailored to our specific scenarios.

Model Training. We first pre-train `FlowMalTrans` and then train it on the DAE, BT and MLE tasks until the loss drops < 0.3. The training time takes around $23h$, $22h$, $25h$, $23h$, $22h$ and $22h$, for i386↔X86-64, ARM32↔X86-64, ARM64↔X86-64, MIPS32↔X86-64, PPC32↔X86-64, and M68K↔X86-64, respectively.

Table 1: Vocabulary size, BPE merge times, and joint vocabulary size for each pair of ISAs.

ISA Pair (src ↔ tgt)	Vocab. Size (src)	Vocab. Size (tgt)	Merge Time	Joint Vocab. Size
i386 ↔ X86-64	7,135	7,104	10,000	9,688
ARM32 ↔ X86-64	9,416	9,236	22,000	17,262
ARM64 ↔ X86-64	5,142	4,455	9,000	7,104
MIPS32 ↔ X86-64	10,995	11,620	14,000	12,685
PPC32 ↔ X86-64	8,691	9,504	17,000	13,032
M68K ↔ X86-64	7,148	6,484	9,000	8,386

Table 2: BLEU scores of `FlowMalTrans` & baseline.

ISA Pair (src → tgt)	`FlowMalTrans` (*our work*)	`UnsuperBinTrans` (*baseline*)
i386 → X86-64	0.41	0.35
ARM32 → X86-64	0.40	0.28
ARM64 → X86-64	0.37	0.32
MIPS32 → X86-64	0.36	0.27
PPC32 → X86-64	0.31	0.25
M68K → X86-64	0.39	0.31

5.3 Testing on Binary Code Translation

We use the Bilingual Evaluation Understudy (BLEU) (Papineni et al., 2002) to evaluate the translation performance, which is often used to evaluate the quality of machine-generated translations by measuring the n-gram overlap between the translation and reference. We set the tokenization of `SacreBLEU` (Post, 2018) to `None`, apply add-one smoothing, and use the default settings

Testing Datasets. We use three packages, *zlib-1.2.11*, *coreutils-9.0*, and *diffutils-3.7*, to test `FlowMalTrans`. Note that *these packages are not included in the training dataset of* `FlowMalTrans`. We compile them on the six ISAs using `GCC-11.4.0` with different optimization levels (O0-O3) and use IDA Pro to disassemble them.

Translation Results. We consider six ISAs, i386, ARM32, ARM64, MIPS32, PPC32 and M68K, as the source ISAs, and X86-64 as the target ISA. For each binary B_1 in the source ISA, there exists a semantically equivalent binary B_2 in X86-64. We use `FlowMalTrans` to translate B_1 into X86-64, resulting in a translated binary B_3 in X86-64. The BLEU score is computed between B_3 and B_2. We report the average BLEU score for all binaries. The results are shown in Table 2. Appendix A.11 presents some translation examples.

We compare `FlowMalTrans` to the baseline `UnsuperBinTrans`. We use the open-source trained model of `UnsuperBinTrans` for this comparison. ***Note that* `UnsuperBinTrans` *fo-***

cuses solely on two ISAs (X86-64 and ARM32). To ensure a comprehensive comparison, we train* `UnsuperBinTrans` *on the same mono-architecture training datasets for the additional ISA pairs. We can see that `FlowMalTrans` outperforms `UnsuperBinTrans` across all ISA pairs and demonstrates satisfactory performance. Thus, `FlowMalTrans` has good translation quality and can effectively translate binaries across ISAs.

The average time to translate a basic block from one ISA to X86-64 is 10^{-4}s, and the average number of blocks in a binary is $12,320$. Thus, translating a binary takes around 1.2s. This demonstrates the good scalability of our approach.

5.4 Malware Detection Task

We train a malware detection model on X86-64 (where abundant malware samples are available), and reuse the trained model for other ISAs by translating binaries using `FlowMalTrans` [1].

Malware Detection Model. We use the LSTM model (HaddadPajouh et al., 2018) to detect malware, designed as two layers. The model details are provided in Appendix A.5. We first extract the token embeddings from `FlowMalTrans` and integrate them into the input layer of LSTM. As a result, when a binary is fed into LSTM, each input token is represented as its embedding. To further enrich the task, we also apply a CNN model, with the results presented in Appendix A.6. In the following, we focus on the results related to LSTM.

Task-Specific *Training* Datasets. We collect 2140, 1740, 1581, 1430, 962, 545, and 128 malware samples from *VirusShare.com* (VirusShare, 2020) for X86-64, i386, ARM32, MIPS32, PPC32, M68K, and ARM64, respectively. We spend significant efforts in collecting malware, especially for PPC32, M68K, and ARM64. Notably, our approach only needs malware in a high-resource ISA to train a malware detection model, significantly reducing the data collection burden for low-resource ISAs.

As we compare the results to the *Same-ISA model* trained and tested on the *same* ISA (Section 5.1), we build task-specific training datasets for each ISA. We split the malware samples into two parts: 80% are used for training and 20% for testing. Each training dataset also includes an equal number of benign samples, randomly selected from *openssl*, *binutils*, *findutils*, and *libgpg-error*.

We also evaluate the performance when testing on *all* malware samples (without comparison to the *Same-ISA model* baseline) in Appendix A.7.

Task-Specific *Testing* Datasets. The testing dataset for each ISA includes equal numbers of malware and benign samples. The benign samples are randomly selected from *zlib*, *coreutils*, and *diffutils*. Note that neither the benign nor malware samples are seen during the training of `FlowMalTrans` or the malware detection model.

Result Analysis. We first train LSTM on X86-64, and reuse it to test binaries in the other ISAs. The results in Table 3(a) show when the model trained on X86-64 is transferred to i386, ARM32, ARM64, MIPS32, PPC32 and M68K, it achieves AUC = 0.996, 0.981, 0.965, 0.971, 0.973 and 0.931, respectively. The high accuracies demonstrate the superior translation quality of `FlowMalTrans`.

Next, we analyze how `FlowMalTrans` preserves code semantics through translation. We visualize the token embeddings from different ISAs and find that tokens performing similar operations, *regardless of their ISAs*, have close embeddings. This demonstrates that `FlowMalTrans` effectively captures code semantics across ISAs. Further details are provided in Appendix A.8.

5.5 Model Comparison

We compare `FlowMalTrans` to the baselines and Same-ISA model (as described in Section 5.1).

Comparison with Baseline Methods. The first baseline is `UnsuperBinTrans`. *As it focuses solely on two ISAs (X86-64 and ARM32), we train it on the same training datasets for the additional ISA pairs.* We use it to translate binaries from other ISAs to X86-64 and test the translated binaries. The results in Table 3(a) show that `FlowMalTrans` achieves better translation quality, leading to improved malware detection performance.

The second baseline is `UniMap` (Wang et al., 2023a), which also aims to resolve the data scarcity issue in malware detection across ISAs. `UniMap` learns transferable knowledge for model reuse. It covers four ISAs, including X86-64, ARM32, MIPS32, and PPC32. To ensure a comprehensive comparison, we *train it on the same datasets to cover the three additional ISAs (i386, ARM64 and M68K)*. Table 3(a) show that `FlowMalTrans` outperforms `UniMap` across all ISA pairs.

The third baseline is `CrossIns2Vec` (Wang et al., 2024), which supports four ISAs (X86-64, ARM32,

[1] We are aware that ARM32 is also a high-resource ISA; due to its importance, our evaluation involves it. However, whether our approach can be applied to translate other ISAs to ARM32 requires further experiments.

Table 3: Malware detection results using LSTM. In Table (a), we compare the detection performance by translating malware using FlowMalTrans and UnsuperBinTrans. Then we evaluate it against UniMap, CrossIns2Vec and the IR-based model. In Table (b), we give results of the *Same-ISA Model*, which is trained and tested on the *same* ISA.

(a) FlowMalTrans vs. Four baselines.

ISA Pair (src → tgt)	FlowMalTrans (*Our Work*)	UnsuperBinTrans (*Baseline 1*)	UniMap (*Baseline 2*)	CrossIns2Vec (*Baseline 3*)	IR-based Model (*Baseline 4*)
i386 → X86-64	**0.996**	0.723	0.953	-	0.819
ARM32 → X86-64	**0.981**	0.818	0.953	0.976	0.815
ARM64 → X86-64	**0.965**	0.638	0.910	-	0.825
MIPS32 → X86-64	**0.971**	0.725	0.933	0.964	0.791
PPC32 → X86-64	**0.973**	0.722	0.919	0.967	0.639
M68K → X86-64	**0.931**	0.689	0.893	-	0.476

(b) The Same-ISA model.

Train & Test on the *same* ISA	Same-ISA Model
i386	0.998
ARM32	0.986
ARM64	0.705
MIPS32	0.974
PPC32	0.849
M68K	0.790

MIPS32, and PPC32). As it is not open-sourced, the comparison is limited to these four ISAs. The results show that FlowMalTrans achieves better performance than CrossIns2Vec.

The fourth baseline analyzes IR code. We assess whether a model trained on X86-64 IR code can be reused to test IR code in other ISAs. We use angr (angr, 2024) to lift binaries into IR. We train LSTM using the same task-specific training dataset in X86-64, and apply the model to test the same task-specific datasets in other ISAs. The AUC scores are 0.819, 0.815, 0.825, 0.791, 0.639 and 0.476 for i386, ARM32, ARM64, MIPS32, PPC32 and M68K, respectively. This indicates IR alone does not magically allow a model trained on one ISA to be reused for other ISAs. (more explanation is discussed in Appendix A.10).

Comparison with Same-ISA Model. The results of the Same-ISA model are shown in Table 3(b). When the LSTM model is trained and tested on the *same* ISA, it achieves AUC of 0.998, 0.986, 0.705, 0.952, 0.949 and 0.790 for i386, ARM32, ARM64, MIPS32, PPC32 and M68K, respectively. Comparing the results with our model in Table 3(a), we observe: (1) our model achieves performance close to Same-ISA model for i386, ARM32 and MIPS32; (2) our model significantly outperforms Same-ISA model for ARM64, PPC32 and M68K.

For i386, ARM32, and MIPS32, the results align with expectations: the Same-ISA model, trained and tested on the *same* ISA, outperforms FlowMalTrans, which is trained on X86-64 and tested on other ISAs (via translation). However, for ARM64, PPC32, and M68K, our model outperforms the Same-ISA model due to the insufficient malware samples used to train the Same-ISA model. For example, for M68K, only 80% of 545 malware samples are used for training. This highlights the importance of sufficient training data to achieve desirable performance. While including more data could enhance the model's performance, collecting malware samples for low-resource ISAs can be challenging. Our approach—translating binaries to X86-64—addresses data scarcity effectively. In an extreme case, *if only one binary in a given ISA is available, we can still detect whether it is malware using the X86-64-trained model*.

5.6 Hyperparameter and Ablation Study

We conduct hyperparameter and ablation studies.
Normalizing Flows. We 1) explore different types of NFs, including *scf* (Scaling and Coupling Flow) and *glow* (Generative Flow), 2) vary the number of sequential flows, and 3) remove the flow adapter. The results show that 1) using *scf* with 3 sequential flows yields the best performance, and 2) when the flow adapter is removed, the performance degrades, highlighting the crucial role of normalizing flows.
Normalization Rules. We conduct ablation study to evaluate the impacts of normalization rules (Section 4.1). The results show that: (1) When all rules are applied, we achieve the best performance. (2) When a subset of rules is applied, the AUC values are reduced, indicating each rule mitigates the OOV issue and positively impacts translation quality.

More details are presented in Appendix A.9.

6 Conclusion and Future Work

We proposed FlowMalTrans, a flow-adapter-based unsupervised binary code translation model. FlowMalTrans incorporates normalizing flows (NFs) to model the basic block-level representations, enhancing its code translation capability. We train FlowMalTrans to translate binaries from other ISAs to X86-64, and reuse a malware detection model trained on X86-64 to test the translated code. Our approach effectively reduces the burden of data collection and achieves better malware detection across ISAs than prior state of the arts.

Large language Models (LLMs) show extraordinary capability at coding tasks and are widely utilized (Xu et al., 2025; Ma et al., 2024; Sun et al., 2025a,b). We leave using LLM for binary code translation as future work.

Limitations

Generalizability. Our evaluation encompasses malware detection across seven distinct ISAs, demonstrating the broad applicability of our method. However, our experimental validation is constrained to Linux-based malware samples. The transferability of our binary code translation framework to alternative operating systems—including Windows, iOS, and Android environments—remains an open question that warrants dedicated empirical investigation. Additionally, while our experiments focus on translating from various ISAs to X86-64, exploring ARM32 as an alternative target architecture presents an interesting direction, given its substantial malware dataset availability.

Beyond malware detection, our translation framework shows promise for broader binary analysis applications. The underlying code transformation capabilities could be leveraged for tasks such as vulnerability discovery and code similarity analysis. The strong performance demonstrated in malware detection validates the core technical approach, establishing a foundation for future research directions. Expanding this methodology to additional security analysis tasks represents a natural progression that could unlock significant research opportunities in cross-ISA binary analysis.

Malware Packing and Obfuscation. The practice of malware packing entails hiding malicious code inside files that appear benign. Our research excludes malware packing from consideration, as it lies beyond our analytical scope. Our investigation centers on examining how well malware detection models transfer across different instruction set architectures (ISAs) through binary translation. The malware specimens employed in our detection experiments are uncompressed, enabling their examination and reverse engineering using analytical tools such as `IDA Pro` (IDA, 2023).

Additionally, when dealing with compressed malware, one approach involves utilizing sophisticated decompression utilities, including PEiD (PEiD, 2008) and OllyDbg (OllyDbg, 2000), to initially decompress the malware before analyzing the resulting content.

Code obfuscation methods present significant obstacles for malware identification systems. The malware specimens utilized in our research originated from *VirusShare.com* (VirusShare, 2020), a collection platform that gathers malware encountered in real-world environments. It is commonly understood that such malware frequently implements obfuscation strategies to circumvent detection systems. Nevertheless, we do not possess definitive information about the particular obfuscation methods employed in each malware sample, which complicates evaluating resistance against specific obfuscation approaches.

Our research primarily tackles the problem of limited data availability in resource-constrained ISAs by converting binaries from these architectures to a well-resourced ISA through `FlowMalTrans`.

Subsequent research could methodically investigate how obfuscation techniques affect detection capabilities. A significant obstacle is the lack of a reliable, high-quality dataset that correlates malware samples with their corresponding obfuscation methods. Addressing this deficiency will be a priority in our upcoming investigations.

Ethical Considerations

To train and test our translation model `FlowMalTrans`, we first collect various open-source programs and compile them for different ISAs using cross compilers. Given the wide availability of open-source programs, this requires minimal effort. Moreover, we strictly adhere to the licensing agreements and intellectual property rights associated with each program when collecting them and building the training and testing datasets.

For evaluation within the security domain, we utilized malicious software specimens obtained from the *VirusShare.com* platform (VirusShare, 2020), a well-established digital repository that serves the cybersecurity research community. We acknowledge the substantial ethical responsibilities accompanying the use of such security-sensitive materials. Our research protocols emphasized legitimate scientific inquiry, prevention of inadvertent threat proliferation, and careful attention to confidentiality and regulatory compliance.

In the interest of scientific reproducibility and community advancement, we intend to distribute our developed model architecture, implementation code, and experimental datasets through a public repository under GPL licensing terms. The compiled open-source program collections will be made fully accessible in their original form. Regarding security-critical materials, we will exclu-

sively share cryptographic hash identifiers and file references from *VirusShare.com*, enabling verification and retrieval by qualified researchers while avoiding direct transmission of potentially dangerous executable content.

Acknowledgement

This work was supported in part by the US National Science Foundation (NSF) under grant CNS-2304720, and in part by the Commonwealth Cyber Initiative. The authors would like to thank the anonymous reviewers for their invaluable comments.

References

Iftakhar Ahmad and Lannan Luo. 2023. Unsupervised binary code translation with application to code clone detection and vulnerability discovery. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Singapore. Association for Computational Linguistics.

Bastidas F Andrés and María Pérez. 2017. Transpiler-based architecture for multi-platform web applications. In *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6. IEEE.

angr. 2024. Intermediate representation. https://docs.angr.io/advanced-topics/ir.

Mikel Artetxe, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho. 2018. Unsupervised neural machine translation. In *ICLR*.

Ömer Aslan, Merve Ozkan-Okay, and Deepti Gupta. 2021. Intelligent behavior-based malware detection system on cloud computing environment. *IEEE Access*.

Sunny Behal, Amanpreet Singh Brar, and Krishan Kumar. 2010. Signature-based botnet detection and prevention. In *Proceedings of International Symposium on Computer Engineering and Technology*, pages 127–132.

Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*.

Iacer Calixto, Miguel Rios, and Wilker Aziz. 2019. Latent variable model for multi-modal translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6392–6405.

Luca Caviglione, Michał Choraś, Igino Corona, Artur Janicki, Wojciech Mazurczyk, Marek Pawlicki, and Katarzyna Wasielewska. 2020. Tight arms race: Overview of current malware threats and trends in their detection. *IEEE Access*, 9:5371–5396.

Junming Chen, Xiaoyue Ma, Lannan Luo, and Qiang Zeng. 2025. Tracking you from a thousand miles away! turning a bluetooth device into an apple {AirTag} without root privileges. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4345–4362.

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. 2018. Neural ordinary differential equations. *Advances in neural information processing systems*, 31.

Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S Bharadwaj Yadavalli, and John Yates. 1998. Fx! 32: A profile-directed binary translator. *IEEE Micro*, 18(02):56–64.

Cristina Cifuentes and Mike Van Emmerik. 2000. Uqbt: Adaptable binary translation at low cost. *Computer*, 33(3):60–66.

Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Édouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. 2020. Unsupervised cross-lingual representation learning at scale. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*.

Alexis Conneau and Guillaume Lample. 2019. Cross-lingual language model pretraining. *Advances in neural information processing systems*.

Ali Davanian and Michalis Faloutsos. 2022. Malnet: A binary-centric network-level profiling of iot malware. In *Proceedings of the 22nd ACM Internet Measurement Conference*, pages 472–487.

Jimmy Ba. Diederik P. Kingma. 2014. Adam: A method for stochastic optimization.

Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*.

Laurent Dinh, David Krueger, and Yoshua Bengio. 2014. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. 2022. Density estimation using real nvp. In *International Conference on Learning Representations*.

Dummy name. 2023. Dummy name prefixes. https://hex-rays.com/blog/igors-tip-of-the-week-34-dummy-names/.

Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. 2001. Dynamic binary translation and optimization. *IEEE Transactions on computers*, 50(6):529–548.

Bryan Eikema and Wilker Aziz. 2019. Auto-encoding variational neural machine translation. In *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*, pages 124–141.

Isaac Feldman and Rolando Coto-Solano. 2020. Neural machine translation models with back-translation for the extremely low-resource indigenous language bribri. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 3965–3976.

Mohana Gopinath and Sibi Chakkaravarthy Sethuraman. 2023. A comprehensive survey on deep learning based malware detection techniques. *Computer Science Review*.

Thamme Gowda and Jonathan May. 2020. Finding the optimal vocabulary size for neural machine translation. *arXiv preprint arXiv:2004.02334*.

Hamed HaddadPajouh, Ali Dehghantanha, Raouf Khayami, and Kim-Kwang Raymond Choo. 2018. A deep recurrent neural network based approach for internet of things malware threat hunting. *Future Gener. Comput. Syst.*, 85(C).

Jianfeng He, Xuchao Zhang, Shuo Lei, Abdulaziz Alhamadani, Fanglan Chen, Bei Xiao, and Chang-Tien Lu. 2023. Clur: Uncertainty estimation for few-shot text classification with contrastive learning. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 698–710.

Jonathan Ho, Xi Chen, Aravind Srinivas, Yan Duan, and Pieter Abbeel. 2019. Flow++: Improving flow-based generative models with variational dequantization and architecture design. In *International conference on machine learning*, pages 2722–2730. PMLR.

IDA. 2023. IDA Pro: A powerful disassembler and a versatile debugger. https://hex-rays.com/ida-pro/.

Sébastien Jean, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. 2014. On using very large target vocabulary for neural machine translation. *arXiv preprint arXiv:1412.2007*.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*.

Shuo Lei, Xuchao Zhang, Jianfeng He, Fanglan Chen, and Chang-Tien Lu. 2022. Uncertainty-aware cross-lingual transfer with pseudo partial labels. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 1987–1997.

Xiang Li, Ying Meng, Junming Chen, Lannan Luo, and Qiang Zeng. 2025. Rowhammer-based trojan injection: One bit flip is sufficient for backdooring dnns. In *USENIX Security Symposium*.

Xiaopeng Li, Qiang Zeng, Lannan Luo, and Tongbo Luo. 2020. T2pair: Secure and usable pairing for heterogeneous iot devices. In *Proceedings of the 2020 acm sigsac conference on computer and communications security*, pages 309–323.

Xuezixiang Li, Yu Qu, and Heng Yin. 2021. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*.

Wu Liu, Ping Ren, Ke Liu, and Hai-xin Duan. 2011. Behavior-based malware analysis and detection. In *2011 first international workshop on complexity and data mining*. IEEE.

Lannan Luo, Qiang Zeng, Bokai Yang, Fei Zuo, and Junzhe Wang. 2021. Westworld: Fuzzing-assisted remote dynamic symbolic execution of smart apps on iot cloud platforms. In *Proceedings of the 37th*

Annual Computer Security Applications Conference, pages 982–995.

Xiaoyue Ma, Lannan Luo, and Qiang Zeng. 2024. From one thousand pages of specification to unveiling hidden bugs: Large language model assisted fuzzing of matter {IoT} devices. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4783–4800.

Xiaoyue Ma, Qiang Zeng, Haotian Chi, and Lannan Luo. 2023. No more companion apps hacking but one dongle: Hub-based blackbox fuzzing of iot firmware. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*, pages 205–218.

Xuezhe Ma, Chunting Zhou, Xian Li, Graham Neubig, and Eduard Hovy. 2019. Flowseq: Non-autoregressive conditional sequence generation with generative flow. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4282–4292.

OllyDbg. 2000. OllyDbg. Url=https://www.ollydbg.de.

George Papamakarios, Theo Pavlakou, and Iain Murray. 2017. Masked autoregressive flow for density estimation. *Advances in neural information processing systems*, 30.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics.

PEiD. 2008. PEiD. Url=https://github.com/wolfram77web/app-peid?tab=readme-ov-file.

Matt Post. 2018. A call for clarity in reporting BLEU scores. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, Belgium, Brussels. Association for Computational Linguistics.

Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. 2008. A semantics-based approach to malware detection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*.

Cheerala Rohith and Gagandeep Kaur. 2021. A comprehensive study on malware detection and prevention techniques used by anti-virus. In *2021 2nd international conference on intelligent engineering and management (iciem)*. IEEE.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in neural information processing systems*, 33:20601–20611.

Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. 2016. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*.

V Sai Sathyanarayan, Pankaj Kohli, and Bezawada Bruhadeshwar. 2008. Signature generation and detection of malware families. In *Information Security and Privacy: 13th Australasian Conference, ACISP 2008, Wollongong, Australia, July 7-9, 2008. Proceedings 13*. Springer.

Stefano Sebastio, Eduard Baranov, Fabrizio Biondi, Olivier Decourbe, Thomas Given-Wilson, Axel Legay, Cassius Puodzius, and Jean Quilbeuf. 2020. Optimizing symbolic execution for malware behavior classification. *Computers & Security*, 93.

Hendra Setiawan, Matthias Sperber, Udhyakumar Nallasamy, and Matthias Paulik. 2020. Variational neural machine translation with normalizing flows. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7771–7777.

Mohit Sewak, Sanjay K Sahay, and Hemant Rathore. 2018. An investigation of a deep learning based malware detection system. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*.

Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. 2012. Llbt: an llvm-based static binary translator. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, pages 51–60.

Raphael Shu, Jason Lee, Hideki Nakayama, and Kyunghyun Cho. 2020. Latent-variable non-autoregressive neural machine translation with deterministic inference using a delta posterior. In *Proceedings of the aaai conference on artificial intelligence*, volume 34, pages 8846–8853.

Yu Sun, Zachary Coalson, Shiyang Cheng, Hang Liu, Sanghyun Hong, Zhao Zhang, Bo Fang, and Lishan Yang. 2025a. Demystifying the resilience of large language model inference: An end-to-end perspective.

Yu Sun, Zhu Zhu, Cherish Mulpuru, Roberto Gioiosa, Zhao Zhang, Bo Fang, and Lishan Yang. 2025b. Ft2: First-token-inspired online fault tolerance on critical layers for generative large language models.

Dhiren Tripuramallu, Swapnil Singh, Shrirang Deshmukh, Srinivas Pinisetty, Shinde Arjun Shivaji, Raja Balusamy, and Ajaganna Bandeppa. 2024. Towards a transpiler for c/c++ to safer rust. *arXiv preprint arXiv:2401.08264*.

Tung-Tso Tsai, Han-Yu Lin, Wei-Ning Huang, Sachin Kumar, Kadambari Agarwal, and Chien-Ming Chen. 2024. Anomaly detection through outsourced revocable identity-based signcryption with equality test for

sensitive data in consumer iot environments. *IEEE Transactions on Consumer Electronics*.

A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems*.

VirusShare. 2020. Virusshare: An open-source repository of malware samples. https://virusshare.com/.

Junzhe Wang, Matthew Sharp, Chuxiong Wu, Qiang Zeng, and Lannan Luo. 2023a. Can a deep learning model for one architecture be used for others? Retargeted-Architecture binary code analysis. In *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA. USENIX Association.

Junzhe Wang, Matthew Sharp, Chuxiong Wu, Qiang Zeng, and Lannan Luo. 2023b. Can a deep learning model for one architecture be used for others?{Retargeted-Architecture} binary code analysis. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7339–7356.

Junzhe Wang, Qiang Zeng, and Lannan Luo. 2024. Learning cross-architecture instruction embeddings for binary code analysis in low-resource architectures. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 1320–1332.

Justin D Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. 2021. Perfection not required? human-ai partnerships in code translation. In *Proceedings of the 26th International Conference on Intelligent User Interfaces*, pages 402–412.

Chuxiong Wu, Xiaopeng Li, Lannan Luo, and Qiang Zeng. 2022a. G2auth: secure mutual authentication for drone delivery without special user-side hardware. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 84–98.

Chuxiong Wu, Xiaopeng Li, Fei Zuo, Lannan Luo, Xiaojiang Du, Jia Di, and Qiang Zeng. 2022b. Use it-no need to shake it! accurate implicit authentication for everyday objects with smart sensing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 6(3):1–25.

Yanze Wu and Md Tanvir Arafin. 2025. Arkane: Accelerating kolmogorov-arnold networks on reconfigurable spatial architectures. *IEEE Embedded Systems Letters*.

Yucheng Xie Xie, Xiaonan Guo, Yan Wang, Jerry Q Cheng, Tianfang Zheng, Yingying Chen, Yi Wei, and Yuan Ge. 2024. mmpalm: Unlocking ubiquitous user authentication through palm recognition with mmwave signals. IEEE Conference on Communications and Network Security (CNS).

Xiaoyu Xu, Minxin Du, Qingqing Ye, and Haibo Hu. 2025. Obliviate: Robust and practical machine unlearning for large language models. *arXiv preprint arXiv:2505.04416*.

Zheyu Zhang, Jianfeng He, Avinash Kumar, and Saifur Rahman. 2024. Ai-based space occupancy estimation using environmental sensor data. In *2024 IEEE Power and Energy Society Innovative Smart Grid Technologies Conference*.

Yue Zhao, Farhan Ullah, Chien-Ming Chen, Mohammed Amoon, and Saru Kumari. 2025. Efficient malware detection using hybrid approach of transfer learning and generative adversarial examples with image representation. *Expert Systems*, 42(2):e13693.

Zhu Zhu, Yu Sun, Dhatri Parakal, Bo Fang, Steven Farrell, Gregory H Bauer, Brett Bode, Ian T Foster, Michael E Papka, William Gropp, et al. 2025. Understanding the landscape of ampere gpu memory errors. *arXiv preprint arXiv:2508.03513*.

Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2018. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706*.

A Appendix

A.1 Model Training Details

Model Pretraining. Pretraining is a key ingredient of unsupervised neural machine translation. Studies have shown that the pre-trained cross-lingual word embeddings which are used to initialize the lookup table, have a significant impact on the performance of an unsupervised machine translation model (Conneau et al., 2020; Conneau and Lample, 2019). We adopt this and pre-train both the encoders and decoders of `FlowMalTrans` to bootstrap the iterative process of our binary translation model. We employ causal language modeling (CLM) and masked language modeling (MLM) objectives to train the encoder and decoder.

(1) The CLM objective trains the model to predict a token e_t, given the previous $(t-1)$ tokens in a basic block $P(e_t|e_1, ..., e_{t-1}, \theta)$. (2) For MLM, we randomly sample 15% of the tokens within the input block and replace them with `[MASK]` 80% of the time, with a random token 10% of the time, or leave them unchanged 10% of the time.

The first and last token of an input basic block is a special token `[/s]`, which marks the start and end of a basic block. We add position embedding and architecture embedding to token embedding, and use this combined vector as the input to the bi-directional Transformer network. Position embeddings represent different positions in a basic block, while architecture embeddings specify the architecture of a basic block. Both position and architecture embeddings are trained along with the token embeddings and help dynamically adjust the token embeddings based on their locations.

Denoising Auto-Encoding (DAE). The DAE reconstructs a basic block from its noised version, as depicted in the process ① and ② in Figure 2. Given the input source block, B_{src}, we introduce random noise into it (e.g., altering the token order by making random swaps of tokens), resulting in the noised version, $B^{'}_{src}$. Then, $B^{'}_{src}$ is fed into the target encoder, whose output is analyzed by the source decoder. The training aims to optimize both the target encoder and source decoder to effectively recover B_{src}. Through this, the model can better accommodate the inherent token order divergences. Similarly, the source encoder and source decoder are optimized when the input is a target basic block, B_{tgt}. The training procedure of DAE involves only a single ISA at each time, without considering the final goal of translating across ISAs nor the latent code transformation.

Back Translation (BT). We adopt the back-translation approach (Feldman and Coto-Solano, 2020) to train our model in a translation setting, as shown in Figure 2. As an example, given an input basic block in one ISA N, we use the model in the inference mode to translate it to the other ISA M (i.e. applying the encoder of N and the decoder of M). This way, we obtain a pseudo-parallel basic block pair. We then train the model to predict the original basic block (i.e. applying the encoder of M and the decoder of N) from this synthetic translation. As training progresses and the model improves, it will produce better synthetic basic block pairs through backtranslation, which will serve to further improve the model in the subsequent iterations. In BT, the latent code transformation is performed twice and trained along with the encoders/decoders; taking BT for the source ISA as an example: first in the source-to-target direction, then in the target-to-source direction as shown in the steps of ③ and ④ in Figure 2.

A.2 Examples of Normalized Assembly Code

Table 4 shows some randomly selected examples for instruction normalization. For each sub-table, the code on the left side represents the original version, while the code on the right side shows the normalized version.

A.3 Dataset Adequacy

To train `FlowMalTrans`, we create a training dataset for each ISA: $2,789,119$ blocks for X86-64, $2,803,557$ blocks for i386, $7,413,083$ blocks for ARM64, $5,812,795$ blocks for ARM32, $4,813,685$ blocks for MIPS32, $3,964,237$ blocks for PPC32, and $5,463,395$ blocks for M68K. It should be noted that the training of `FlowMalTransis` unsupervised.

In NLP, it is widely recognized that a comprehensive dataset, which ensures that the vocabulary covers a wide range of words, is crucial for training effective code translation models. We assess the adequacy of our mono-architecture datasets. Specifically, we study the vocabulary growth as we incrementally include programs. Our findings indicate that while the vocabulary size initially increases with the inclusion of more programs, it eventually stabilizes. Take X86-64 as an example, including *openssl-1.1.1p* results in a vocabulary size of $23,029$. The size increases to $36,770$ (a 60% growth) when *binutils-2.34* is added, and then

Table 4: Comparison of original and normalized assembly code.

```
call _gpgrt_log_info
sbb al 0
mov esi 0ACh+_bss_start
lea rdi str_LogWithPid
sub rdx buffer
```

```
call <FUNC>
sbb al <VALUE>
mov esi <HEX>+<TAG>
lea rdi <STR>
sub rdx <TAG>
```

(a) X86-64

```
add esp 0Ch
call _dcgettext
lea eax [ebx-5D40h]
jmp short loc_37C3
mov eax [150h+domainname]
```

```
add esp <HEX>
call <FUNC>
lea eax [ebx-<HEX>]
jmp short <LOC>
mov eax [<HEX>+<TAG>]
```

(b) i386

```
ADD R12 R12 0x1B000
LDR PC memcpy-0x2B7C
BEQ.W loc_109B4
BL gz_uncompress
CMP R2 0
```

```
ADD R12 R12 <HEX>
LDR PC <FUNC>-<HEX>
BEQ.W <LOC>
BL <FUNC>
CMP R2 <VALUE>
```

(c) ARM32

```
LDR X0 [SP #0xC0+stream_68]
TBZ W0 #0 loc_AF38
ADRL X1 str_ErrorInitia
B _gmon_start_
MOV X19 #0
```

```
LDR X0 [SP <HEX>+<TAG>]
TBZ W0 <HEX> <LOC>
ADRL X1 <STR>
B <FUNC>
MOV X19 <HEX>
```

(d) ARM64

```
lw $fp 0x40+var_20
bal usage
beqz $v0 loc_1358
move $a2 $s3+1
sw $s0 0x40+path($sp)
```

```
lw $fp <HEX>+<VAR>
bal <FUNC>
beqz $v0 <LOC>
move $a2 $s3+<VALUE>
sw $s0 <HEX>+<TAG>($sp)
```

(e) MIPS32

```
addi r3 r4 8
cmpwi r2 8+var_12
beq _str_branch_
stw r6 r2 6+0x1C5
lwz r3 0x3H+str_56
```

```
addi r3 r4 <VALUE>
cmpwi <VALUE>+<VAR>
beq <TAG>
stw r6 r2 <VALUE>+<HEX>
lwz r3 <HEX>+<STR>
```

(f) PPC32

```
move.l #$DB3EF d0
sub.w a2 d4-stream_23
tst.b loc_5H
cmp.l #$352A #23+mem_name
divs.w #16 d2
```

```
move.l <HEX> d0
sub.w a2 d4-<TAG>
tst.b <LOC>
cmp.l <HEX> <VALUE>+<TAG>
divs.w <VALUE> d2
```

(g) M68K

increases to 39,499 (a 7.4% growth) and 39,892 (a 0.9% growth) when *findutils-4.8.0*, *libpgp-error-1.45* are included, respectively. The growth trend is similar for other ISAs. It shows that the vocabulary barely grows in the end when more programs are added. According to the vocabulary growth trend and the high performance achieved, our mono-arch datasets are adequate to cover instructions and enable effective code translation.

Note that the datasets used for training FlowMalTrans have *no overlap* with (1) the dataset used for testing the translation capability of FlowMalTrans and (2) the testing dataset used in the malware detection task. The details of these datasets are introduced in the following sections.

A.4 Model Parameters of FlowMalTrans

The encoders and decoders of FlowMalTrans are implemented using the Transformer model. Table 5 shows the details of the model parameters.

A.5 Model Parameters of Malware Detection LSTM Model

We use the LSTM model to detect malware. Table 6 shows the parameters details of the LSTM model.

A.6 Malware Detection Using a CNN model

We use a 1-dimensional Convolutional Neural Network (CNN) model to detect malware. The parameters of the CNN model are presented in Table 7.

We use the same task-specific training and testing datasets described in Section 5.4. Moreover, we compare our results against four baseline methods and the Same-ISA model. Table 8 presents the malware detection results using the CNN model.

When the CNN model trained on X86-64 is transferred to i386, ARM32, ARM64, MIPS32, PPC32 and M68K, it achieves AUC values of 0.993, 0.984, 0.973, 0.952, 0.942 and 0.937, respectively. These high accuracies highlight the superior translation quality of FlowMalTrans, outperforming both UnsuperBinTrans and the IR-based model. Compared to the Same-ISA model, we have the following observation. (1) First, our model achieves performance close to the Same-ISA model when testing malware on i386, ARM32, and MIPS32. This outcome is expected, as the Same-ISA model is trained and tested on the same ISA, while our model is trained on X86-64 and tested on other ISAs through translation. (2) Second, our model significantly outperforms the Same-ISA model on ARM64, PPC, and M68K. This is due to the limited malware samples available for training the Same-ISA model on the two ISAs, highlighting the value of our approach. By reusing a model trained on a high-resource ISA, we enable robust detection for low-resource ISAs that would otherwise face significant challenges.

Table 5: Parameter Details of FlowMalTrans.

Parameter	Value	Description
Emb. Dimension	32/64/128	Embedding layer size for tokens
Hidden Dimension	4* Emb. Dimension	Transformer FFN hidden dimension
Num. Layers	4	Number of transformer layers
Num. Heads	4	Number of attention heads per layer
Regu. Dropout	0.1	Dropout rate for regularization
Attn. Dropout	0.1	Dropout rate in attention layers
Batch Size	256	Number of sentences per batch
Max. Length	512	Maximum length of one sentence after BPE
Optimizer	Adam	Adam optimizer with sqrt decay
Clip Grad. Norm	5	Maximum gradient norm for clipping
Act. Function	ReLU	Use ReLU for activation
Pooling	Mean	Use mean pooling for sentence embeddings
Accumulate Grad.	8	Accumulate gradients over N iterations

Table 6: Parameter Details of the malware detection LSTM model.

Parameter	Value	Description
Emb. Dimension	32/64/128	Input feature dimension for sequence embedding
Num. of Layers	3	Number of stacked LSTM layers in the network
Hidden Units	16	Number of hidden units in each LSTM layer
Output Units	1	Dimension of the output layer
Batch Size	36	Number of samples processed in one batch
Optimizer	Adam	Adaptive optimization algorithm with momentum
Loss Function	BCEWithLogitsLoss	Binary cross-entropy with logits
Pooling	Max	Maximum value across temporal dimension

Table 7: Parameter Details of the malware detection CNN model.

Parameter	Value	Description
Emb. Dimension	64	Input feature dimension for sequence embedding
Conv. Layers	2	Number of convolutional layers in the CNN network
Conv. Kernel	3	Kernel size of the convolutional layers
Output Units	1	Dimension of the output layer
Batch Size	64	Number of samples processed in one batch
Optimizer	Adam	Adam optimizer with a learning rate of 0.001
Loss Function	BCEWithLogitsLoss	Binary cross-entropy with logits
Pooling	Max	Maximum pooling to reduce spatial dimensions

A.7 Malware Detection Using All Malware Samples for Testing

We train the LSTM model exclusively on X86-64, and reuse the trained model to test binaries in other ISAs, including i386, ARM32, ARM64, MIPS32, PPC32 and M68K. It is important to note the key difference between the experiment described in this Appendix and that in Section 5.4. Here, we use all malware samples from i386, ARM32, ARM64, MIPS32, PPC32 and M68K for testing. In contrast, in Section 5.4, only 20% of the malware samples from these ISAs are used for testing, as the remaining 80% are reserved for training.

Result Analysis. We first train LSTM on X86-64, and reuse the model to test binaries in the other ISAs. The results are shown in Table 9. We can see that when the model trained on X86-64 is transferred to i386, ARM32, ARM64, MIPS32, PPC32 and M68K, it achieves AUC = 0.997, 0.982, 0.963, 0.974, 0.978 and 0.940 respectively. The high accuracies demonstrate the superior translation quality of FlowMalTrans.

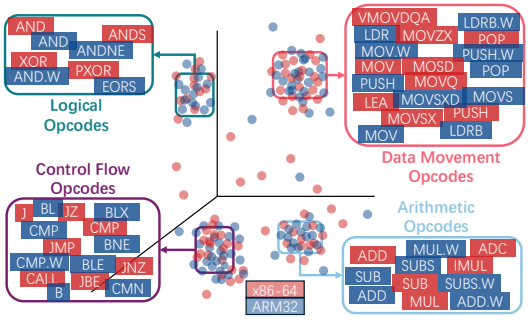

Figure 3: Visualization of opcode embeddings.

A.8 Semantic Transfer Analysis

We analyze how FlowMalTrans is able to preserve the code semantics through translation. Specifically, we visualize the embeddings of opcode tokens from different ISAs. We take the X86-64 and ARM32 pair as an example. Opcodes, which determine the operation to be performed, capture more

Table 8: Malware detection results using CNN. In Table (a), we compare the detection performance by translating malware using FlowMalTrans and UnsuperBinTrans. Then we evaluate it against UniMap, the IR-based and Same-ISA model. In Table (b), we give the results of the *Same-ISA model*, which is trained and tested on the *same* ISA.

(a) FlowMalTrans vs. Four baselines.

ISA Pair (src → tgt)	FlowMalTrans (*Our Work*)	UnsuperBinTrans (*Baseline 1*)	UniMap (*Baseline 2*)	CrossIns2Vec (*Baseline 3*)	IR-based Model (*Baseline 4*)
i386 → X86-64	**0.993**	0.745	0.953	-	0.823
ARM32 → X86-64	**0.984**	0.826	0.953	0.976	0.842
ARM64 → X86-64	**0.973**	0.641	0.910	-	0.818
MIPS32 → X86-64	**0.974**	0.762	0.933	0.964	0.774
PPC32 → X86-64	**0.972**	0.736	0.919	0.967	0.642
M68K → X86-64	**0.931**	0.696	0.893	-	0.483

(b) The Same-ISA model.

Train & Test on the *same* ISA	Same-ISA Model
i386	0.997
ARM32	0.988
ARM64	0.714
MIPS32	0.956
PPC32	0.853
M68K	0.780

Table 9: Malware detection results. We use all the malware samples in i386, ARM32, ARM64, MIPS32, PPC32 and M68K for testing. We compare the detection performance by translating malware using FlowMalTrans and UnsuperBinTrans, and evaluate it against the IR-based model.

ISA Pair (src → tgt)	FlowMalTrans (*Our Work*)	UnsuperBinTrans (*Baseline 1*)	UniMap (*Baseline 2*)	CrossIns2Vec (*Baseline 3*)	IR-based Model (*Baseline 4*)
i386 → X86-64	**0.997**	0.728	0.949	-	0.821
ARM32 → X86-64	**0.982**	0.821	0.951	0.975	0.813
ARM64 → X86-64	**0.963**	0.641	0.913	-	0.822
MIPS32 → X86-64	**0.974**	0.731	0.931	0.968	0.796
PPC32 → X86-64	**0.978**	0.732	0.916	0.973	0.641
M68K → X86-64	**0.940**	0.682	0.894	-	0.480

Table 10: Impacts of different types of normalizing flows on the malware detection task.

Type	i386	ARM32	ARM64	MIPS32	PPC32	M68K
3-scf	**0.996**	0.981	**0.965**	**0.971**	**0.973**	**0.931**
3-glow	0.994	**0.983**	0.924	0.823	0.873	0.873
5-scf	0.957	0.875	0.817	0.910	0.832	0.824
5-glow	0.919	0.891	0.843	0.804	0.821	0.760
None	0.854	0.871	0.784	0.769	0.811	0.752

semantics compared to operands; we thus focus on opcodes for this demonstration. We extract the embeddings of 138 X86-64 opcodes and 247 ARM32 opcodes from FlowMalTrans, and visualize them using t-SNE, as shown in Figure 3. Four categories of opcodes are selected for demonstration. We can see that opcodes performing similar operations, *regardless of their ISAs*, are close together. Thus, FlowMalTrans can successfully capture semantic relationships of opcodes across ISAs and preserve code semantics.

A.9 Hyperparameter and Ablation Study

Normalizing Flows. We conduct an experiment to analyze the impact of normalizing flows on translation capability based on the following aspects: (1) exploring different flow-adapter architectures, such as Glow (Diederik P. Kingma, 2014); (2) varying the number of sequential flows to build the source/target flow; and (3) removing the flow-adapter architecture in FlowMalTrans.

The results are shown in Table 10. We use two types of normalizing flows: *scf* (Scaling and Coupling Flow) and *glow* (Generative Flow), each configured with different numbers of sequential flows (e.g., 3 and 5). For instance, *5-glow* represents the scenario where five consecutive *glow* flows are utilized to construct the source/target flow. The last row, *None*, represents the scenario in which no normalizing flows are incorporated into the model FlowMalTrans.

We can observe that: (1) Using *3-scf* to construct the normalizing flows yields the best performance across all ISAs except for ARM32. However, the result for ARM32 remains satisfactory, being close to the highest value. (2) When the flow-adapter architecture is removed from FlowMalTrans, the performance degrades, highlighting the crucial role of normalizing flows in enhancing translation capability.

Normalization Rules. Each instruction in the datasets is normalized by applying the three rules (R1, R2, and R3) discussed in Section 4.1. Normalization is a vital step in our approach. In this experiment, we conduct ablation study by removing certain rules and evaluating their influence on malware detection. We consider these cases:

- (**C1**): Applying all rules to the data.
- (**C2**): Removing R1, applying R2 and R3.
- (**C3**): Removing R2, applying R1 and R3.
- (**C4**): Removing R3, applying R1 and R2.

Table 11: AUC when varying normalizing strategies.

Case #	i386	ARM32	ARM64	MIPS32	PPC32	M68K
Case 1	**0.996**	**0.981**	**0.965**	**0.971**	**0.973**	**0.931**
Case 2	0.862	0.521	0.624	0.623	0.617	0.638
Case 3	0.883	0.575	0.617	0.570	0.503	0.642
Case 4	0.827	0.533	0.613	0.422	0.451	0.546
Case 5	0.793	0.453	0.523	0.422	0.473	0.471

Table 12: AUC when varying dimension size.

ISA Pair (src → tgt)	Dimension: 32	Dimension: 64	Dimension: 128
i386→X86-64	0.995	0.996	0.994
ARM32→X86-64	0.978	0.981	0.972
ARM64→X86-64	0.857	0.965	0.860
MIPS32→X86-64	0.912	0.971	0.869
PPC32→X86-64	0.917	0.973	0.892
M68K→X86-64	0.832	0.931	0.917

- (**C5**): Not applying any rules to the data.

Table 11 shows the results. We can observe that: (1) When all normalization rules are applied (**C1**), we achieve the best performance. (2) When a subset of normalization rules is applied (**C2-4**), the AUC values are lower than in **C1**, indicating that each normalization rule mitigates the OOV issue and has an impact on translation quality, thereby influencing malware detection performance. (3) When no normalization rule is applied (**C5**), the results are the lowest.

Embedding Dimension. We evaluate the impacts of the embedding dimension. We test different dimension sizes, including 32, 64, and 128, to train `FlowMalTrans`. We then apply `FlowMalTrans` to translate binaries from the source ISA to X86-64 for malware detection. The results are shown in Table 12. We observe that when the dimension is set to 64, the AUC values are higher compared to other dimensions. Moreover, as the dimension increases, the training time also increases. We thus choose a dimension of 64, considering both the translation quality and training efficiency.

A.10 Discussion on IR-based model

Although IR can abstract away many architectural differences among ISAs, significant variations persist across IR code from different ISAs. Given two binary executables from different ISAs, compiled from the same source code, their corresponding IR representations can still appear quite different. This issue is discussed in Section 4.3.1 of the UniMap paper. Consequently, existing approaches that use IR for cross-ISA binary analysis often require additional advanced techniques, such as fuzzing or re-optimization, to bridge these differences. Similarly, in our evaluation, the IR-based baseline, which trains a model on X86 IR code and reuses it for testing on IR code from other ISAs, performs poorly due to these variations.

Decompilation, which converts binary code into readable C-like code, is inherently a lossy process with limited accuracy (e.g., complex constructs such as optimized loops, switch statements, and inlined functions are often not recovered accurately). Moreover, while IDA Pro's decompiler is among the best available, it supports only a limited set of ISAs, restricting its generalization to only those architectures. In contrast, our binary code translation approach does not suffer from these limitations. Thus, decompilation is an unsuitable solution for our needs.

A.11 Translation Demonstration

Table 13 and Table 14 shows some randomly selected examples. In each example, (1) the *source ISA* is the original basic block in the source ISA, which could be i386, ARM32, ARM64, MIPS32, PPC32 or M68K; (2) the *target ISA* is the basic block in the target ISA, X86-64, that is similar to the original basic block in the source ISA; and (3) the *translated ISA* is the X86-64 basic block translated from the original basic block in the source ISA by our model `FlowMalTrans`.

By comparing the translated X86-64 block with the target X86-64 block, we observe that `FlowMalTrans` (1) accurately predicts almost all opcodes, and (2) while a few operands are predicted incorrectly, these errors are reasonable. Note that an instruction consists of an opcode (which specifies the operation) and zero or more operands (which specifies registers, memory locations, or literal data). Thus, opcodes, which determine the operation to be performed, capture more semantic information compared to operands. On the other hand, different registers or memory locations can store data while preserving code functionality, which reduces the significance of operands.

Consider the first example of the ISA pair i386→X86-64. In the target X86-64 basic block, the second instruction is: **add** `rbp [state+<HEX>]`, whereas in the translated X86-64 basic block, the predicted instruction is: **add** `rdx [s+<HEX>]`. Here, `FlowMalTrans` successfully predicts the opcode **add**, and predicts a different register and memory cell for the operands, while preserving the functionality of the code.

A.12 Uniqueness of Our Work

We propose an entirely different approach compared to the prior works (Wang et al., 2023b, 2024),

such as UniMap (Wang et al., 2023b), which also aims to reuse models across ISAs in order to resolve the data scarcity issue in low-resource ISAs.

A key limitation of UniMap is its reliance on a linear transformation to map instruction embeddings between ISAs. This approach assumes that the embedding spaces of different ISAs exhibit structural similarity (i.e., are isomorphic). However, this assumption may not hold, particularly for ISAs with significantly different vocabulary sizes, making it infeasible to find a suitable linear transformation. In contrast, FlowMalTrans does not rely on such an assumption, making it more broadly applicable across diverse ISAs.

While UniMap learns cross-architecture instruction embeddings, our method focuses on translating binary code across ISAs. By translating code to a high-resource ISA, our approach offers several advantages. First, it allows the direct application of existing downstream models—trained on the high-resource ISA—to other ISAs through testing the translated code. In contrast, the works in (Wang et al., 2023b, 2024) require retraining the model using cross-architecture instruction embeddings. Furthermore, translating code from one ISA to another assists human analysts in understanding code from unfamiliar ISAs, supporting broader applications in code comprehension.

InnerEye (Zuo et al., 2018) applies neural machine translation (NMT) techniques for binary code similarity comparison but *does not perform binary code translation across ISAs*. It uses two encoders from NMT models, where each encoder generates an embedding for a piece of binary code of a given pairs, and measures similarity based on embedding distance. In contrast, our approach focuses on translating binary code across ISAs.

Intermediate Representation (IR) is an architecture-agnostic representation that abstracts away various architectural differences among ISAs. However, despite this abstraction, significant variations still exist across IR code from different ISAs. Consequently, even when binaries from different ISAs—compiled from the same source code—are converted into a common IR, the resulting IR code often differs significantly, as discussed in (Ahmad and Luo, 2023). We compare malware detection performance across ISAs by translating binaries using FlowMalTrans against an approach that analyzes the IR code of binaries across ISAs. The results show our model enables superior cross-ISA malware detection (see the evaluation). An interesting direction is exploring translation based on IR code, which we leave for future work.

Table 13: Examples for code translation.

i386	1	Source i386	**sub** esp <VALUE> **add** edx [s+<HEX>] **mov** ebx esi **push** len **push** edx **push** [esp+<HEX>+dictionary] **call** <FUNC> **add** esp <HEX> **jmp** <LOC>
		Translated X86-64	**sub** rsp <VALUE> **add** rdx [s+<HEX>] **mov** rbx rbp **push** len **mov** r8 [rsp+<HEX>+datalen] **mov** rcx len **call** <FUNC> **jmp** <ADDR>
		Target X86-64	**sub** rsp <VALUE> **add** rbp [state+<HEX>] **mov** eax <HEX> **push** <TAG> **mov** [state+<HEX>] rdi **mov** rdi state **call** <FUNC> **jmp** short <TAG>
	2	Source i386	**mov** edx [edi+<VALUE>] **mov** edp <VALUE> **cmp** edx <HEX> **lea** ecx [edx-<HEX>] **setnz** al **and** ecx <HEX> **lea** ecx [edx-<HEX>]
		Translated X86-64	**mov** rdx [abfd+<VALUE>] **mov** ebp <VALUE> **cmp** rdx <HEX> **lea** rcx [r1+rax+<VALUE>] **setnz** al **and** ecx <HEX> **lea** rcx [r1+rax+<VALUE>]
		Target X86-64	**mov** rdx rsp **mov** rsp <HEX> **cmp** rax rdx **lea** r8 <TAG> **setnz** a1 **and** ecx <TAG> **lea** rcx [rax+<HEX>]
	3	Source i386	**sub** esp <VALUE> **lea** eax [esi+<VALUE>] **push** eax **push** [esp+<HEX>+buf] **call** <FUNC> **mov** edi eax **mov** c [esp+<HEX>+<TAG>]
		Translated X86-64	**sub** rsp <VALUE> **lea** rax <TAG>+<HEX> **mov** cs:<TAG> rax **mov** rax [abfd+<VALUE>] **call** <FUNC> **mov** rsi rsp **mov** rdi abfd **call** <FUNC> **add** rsp <VALUE>
		Target X86-64	**sub** rsp <HEX> **lea** eax <HEX> **mov** rax rs:<HEX> **mov** [rsp+<HEX>+<VAR>] **call** <FUNC> **mov** rcx rsp **mov** rsi <VALUE> **call** <FUNC> **add** eax <HEX>
ARM32	1	Source ARM32	**XOR** R0 R2 **MOV** R1 R1 <VALUE> **SUBS** R3 R0 R3 **LDP** R6 R6 have **BNE** <TAG> **CMP** copy R7
		Translated X86-64	**xor** edi edi **mov** rcx rdx **and** esi <VALUE> **sub** <TAG> <VALUE> **mov** [rax+rdx-<VALUE>] cl **mov** ecx <VALUE>
		Target X86-64	**xor** edi edi **mov** [rax+rcx-<VALUE>] di **and** edi <TAG> **sub** init curr **mov** [s+<HEX>] rdx **mov** rsp <VALUE>
	2	Source ARM32	**LDR** R2 [R3] **MOVS** R0 <VALUE> **BNE** <LOC> **LDR** R0 [R2+<TAG>] **MOV** R1 <TAG> **BL** <FUNC>
		Translated X86-64	**mov** rsi [rsp+<HEX>+p] **movsxd** rdi eax **lea** rcx [rsp+<HEX>+id] **call** <FUNC>
		Target X86-64	**mov** rdi in **movsxd** rdi file **lea** rax [rsi+<VALUE>] **call** <FUNC>
	3	Source ARM32	**MOV** R0 strm **MOVS** R1 <VALUE> **BL** <FUNC> **ADDS** err <VALUE> **MOVS** R2 <TAG> **LDR** R1 <HEX> **BNE** <ADDR>
		Translated X86-64	**mov** edi ebx **lea** rcx <TAG> **movsxd** rax ds:<TAG> **call** <FUNC> **add** rax rcx **lea** rdx <TAG> **jmp** <ADDR>
		Target X86-64	**mov** edi ebx **lea** rsi <HEX> **movzx** rax <TAG> **call** <FUNC> **add** eax ecx **lea** rbp <LOC> **jmp** <TAG>
ARM64	1	Source ARM64	**MOV** W2 <VALUE> **MOV** W0 W2 **LDP** X2 <VALUE> [SP+<HEX>+<TAG>] **LDR** X2 [SP+<HEX>+<TAG>] **LDP** X2 X3 [SP+<HEX>+<TAG>]
		Translated X86-64	**mov** qword ptr [rax] <VALUE> **mov** rax [rbp+p] **mov** rdi rax **call** <FUNC> **test** eax eax **jnz** <ADDR>
		Target X86-64	**mov** qword ptr [rcx+<HEX>] <VALUE> **mov** [rdx+<HEX>] <VALUE> **mov** [rdx+<HEX>] rcx **call** <FUNC> **test** rax rax **jnz** <TAG>
	2	Source ARM64	**ADD** W2 W2 <VALUE> **ADD** X3 X0 X3 **SUB** W2 W2 <VALUE> **MOV** W0 <VALUE> **STRB** W0 [X1+<HEX>] **MOV** X0 X2 <VALUE>
		Translated X86-64	**mov** r1 q **mov** rsi p **movzx** ri byte ptr [p] **sub** r1 <VALUE> **add** p <HEX> **mov** rax [p] **jmp** short <LOC>
		Target X86-64	**mov** r14 rsp **mov** rsip **movzx** rax <HEX> **sub** r10 <VALUE> **add** q <HEX> **mov** r11 <VALUE> **jmp** <TAG>
	3	Source ARM64	**LDR** W5 <VALUE> **LDR** X9 <HEX> **SUB** W0 W5 W0 **ADD** W1 W0 <VALUE> **LDP** W1 <VALUE> **SUB** W0 W5 W0 **ADD** W1 W0 <VALUE> **LDP** W1 <VALUE> **LDRB** W2 <TAG>
		Translated X86-64	**mov** rdx <HEX> **mov** [rsp+<HEX>+n] rdx **movzx** edx [r1+<HEX>] **mov** esi edx **mov** r9 [r1+<HEX>] **lea** r1 [rdx+<TAG>]
		Target X86-64	**mov** rdx [rbp+mode] **mov** ecx [rbp+fd] **movzx** rax [rbp+path] **mov** esi ecx **mov** rdi rax **mov** [rbp+gz] rax **lea** r1 [rax+<VALUE>]

Table 14: Examples for code translation.

MIPS32	1	Source MIPS32	**li** $t9 <VALUE> **lw** $ra <HEX>+<VAR> ($sp) **addiu** $t9 <FUNC> **b** <FUNC> **addiu** $sp <HEX>
		Translated X86-64	**mov** r8 <VALUE> **mov** rbx rsp+<HEX>+<TAG> **lea** r10 r8+<ADDR> **call** <FUNC> add rbp <VALUE>
		Target X86-64	**mov** r10 <VALUE> **mov** rax rsp+<HEX>+<VAR> **lea** r10 r10+<TAG> **jmp** <FUNC> **add** rsp <HEX>
	2	Source MIPS32	**li** $a1 <VALUE> **addiu** $a1 <STR> **move** $a0 $s0 **la** $t9 <FUNC> **jalr** $t9 **lw** $gp <HEX>+<VAR> ($sp)
		Translated X86-64	**mov** rsi <HEX> **lea** rbp rbp+<ADDR> **mov** rdi rax **mov** r11 <FUNC> **call** <FUNC> **mov** rap rsp
		Target X86-64	**mov** rsi <VALUE> **lea** rsi rsi+<STR> **mov** rdi, rbx **mov** r10 <ADDR> **call** <FUNC> **mov** rbp rsp
	3	Source MIPS32	**li** $a2 <VALUE> **li** $a1 <VALUE> **addiu** $a1 <STR> **move** $a0 $zero **la** $t9 <FUNC> **jalr** $t9 lw $gp <HEX+<VAR> ($sp) **move** $a1 $s1 **move** $a0 $v0
		Translated X86-64	**mov** rbx <HEX> **mov** rsi <ADDR> **lea** rsi rsi+<TAG> **xor** rsi rsi **mov** r10 r9 **call** <FUNC> **mov** rbp rap **mov** rsi r14+<TAG>+<HEX> **mov** rdi rax
		Reference X86-64	**mov** rdx <VALUE> **mov** rsi <VALUE> **lea** rsi rsi+<STR> **xor** rdi rdi **mov** r10 r11 **call** <FUNC> **mov** rbp rsp+<HEX>+<VAR> **mov** rsi r14 **mov** rdi rax
PPC32	1	Source PPC32	**lis** r13 r2@ha **addi** r3 r3 at1@l **lis** r9 <VAR>+<HEX>@ha **lwz** r1 <TAG>+<LOC>@l(r9) **add** r10 r7 r8 **add** r10 r6 r10 **lwz** r3 <HEX>(r1) **bl** <FUNC>
		Translated X86-64	**mov** rbi rax **movsz** rbp [<HEX>+<VALUE>] **mov** rdx rax **lea** [rax r1+<TAG>] **mov** rdx [rsi+<LOC>] **call** <FUNC> **test** rax rax **jz** <TAG>
		Target X86-64	**mov** rbi <LOC> **mov** rbx <VALUE> **lea** [rdi+<LOC>] **mov** rdx [rbx+<VALUE>] **call** <FUNC> **test** rdx rdx **jnz** <LOC>
	2	Source PPC32	**mr** r3 r7 **li** r3 <VAR> **lis** <FUNC> **li** r6 <TAG> **mr** r5 r4 **stwu** r31 -4(r1) **b** <LOC>
		Translated X86-64	**movsx** r2 r13 **mov** rbi <VAR> **call** <FUNC> **mov** rbx <TAG> **mov** rax rdi **push** rbx **jmp** <ADDR>
		Target X86-64	**mov** r12 rbi **movsz** rsi **mov** rbx <LOC> **lea** <HEX>+<TAG> **mov** r12 r7 **mov** rbx (<VALUE>+<HEX>) **mov** r8 <HEX> **jmp** <HEX>
	3	Source PPC32	**extsh** r2 r13 **li** r9 <VAR> **bl** <FUNC> **lis** r5 <TAG>@ha **addi** r5 r5 <TAG>@l **mr** r6 r3 **stwu** r5 -4(r1) **b** <ADDR>
		Translated X86-64	**mov** rbx <LOC>+<HEX> **movsx** rbx rax **mov** eax [ecx+<HEX>] **mov** ebi [ecx+<HEX>] **mov** r5 [r2+<VALUE>] **lea** r7 [rax+<HEX>]
		Reference X86-64	**mov** rax [<VALUE>+<HEX>] **mov** eax <VAR> **mov** rbp [rsi+<VALUE>] **mov** eax ebx **mov** rsi rdx **mov** rax (rax+<VAR>) **l** rbi [r3+<VAR>]
M68K	1	Source M68K	**move.l** at2 a0 **move.l** <VALUE>+<TAG> a7 **move.l** a1 d2 **add.l** a2 d3 **add.l** a3 d2 **move.l** <HEX>(a7) a0 **jsr** <FUNC> **tst.l** d0 **bne** <LOC>
		Translated X86-64	**mov** rsi at3 **mov** rsp [<HEX>+<TAG>] **mov** rax rex **lea** [r4 r10+r3] **mov** rbx [rbp+<VALUE>] **call** <FUNC> **test** rax rax **jz** <LOC>
		Target X86-64	**mov** rsi <TAG> **mov** rbp <VALUE> **lea** [rdx+<HEX>] **mov** rdi [rax+<VALUE>] **call** <FUNC> **test** rax rax **jnz** <TAG>
	2	Source M68K	**move.l** d3 d2 **move.l** <HEX> a0 **lea** <FUNC> a1 **move.b** <TAG> d4 **ext.l** d4 **move.l** d1 d0 **move.l** a6 a7 **b** <LOC>
		Translated X86-64	**mov** r12 rbx **mov** rdi <HEX> **lea** <FUNC> **movsx** rax <TAG> **mov** rbx rdx **push** rbp **jmp** short <LOC>
		Target X86-64	**mov** r12 rbp **mov** rsp **movsx** rcx <LOC> **lea** <TAG> **mov** r2 r4 **mov** rbp (<TAG>+<HEX>) **mov** r11 <VALUE> **jmp** <ADDR>
	3	Source M68K	**move.l** <VALUE>+<VAR> d0 **move.l** d2 d1 **move.l** <VALUE>(d1) d2 **move.l** d1 a0 **move.s** d3 <VALUE> **move.l** <HEX>(a1) d3 **lea** <TAG>(a2) a1
		Translated X86-64	**mov** rax <VAR>+<HEX> **mov** rcx rbx **mov** ebx [ecx+<VALUE>] **mov** ebi ecx **mov** r9 [r1+<HEX>] **lea** r1 [rdx+<TAG>]
		Reference X86-64	**mov** rax [<VALUE>+<HEX>] **mov** eax <VAR> **mov** rbp [rsi+<VALUE>] **mov** eax ebx **mov** rsi rbx **mov** rax (rax+<TAG>) **lea** rsi [r2+<VAR>]