SLaDe: A Portable Small Language Model Decompiler for Optimized Assembly

Jordi Armengol-Estapé
School of Informatics
University of Edinburgh
Edinburgh, United Kingdom
jordi.armengol.estape@ed.ac.uk

Jackson Woodruff
School of Informatics
University of Edinburgh
Edinburgh, United Kingdom
j.c.woodruff@sms.ed.ac.uk

Chris Cummins
Meta AI Research
Menlo Park, CA, USA
cummins@fb.com

Michael F.P. O'Boyle
School of Informatics
University of Edinburgh
Edinburgh, United Kingdom
mob@inf.ed.ac.uk

2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 979-8-3503-9509-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/CGO57630.2024.10444788

***Abstract*—Decompilation is a well-studied area with numerous high-quality tools available. These are frequently used for security tasks and to port legacy code. However, they regularly generate difficult-to-read programs and require a large amount of engineering effort to support new programming languages and ISAs. Recent interest in neural approaches has produced portable tools that generate readable code. Nevertheless, to-date such techniques are usually restricted to synthetic programs without optimization, and no models have evaluated their portability. Furthermore, while the code generated may be more readable, it is usually incorrect.**

This paper presents `SLaDe`, a Small Language model Decompiler based on a sequence-to-sequence Transformer trained over real-world code and augmented with a type inference engine. We utilize a novel tokenizer, dropout-free regularization, and type inference to generate programs that are more readable and accurate than standard analytic and recent neural approaches. Unlike standard approaches, `SLaDe` can infer out-of-context types and unlike neural approaches, it generates correct code.

We evaluate `SLaDe` on over 4,000 ExeBench functions on two ISAs and at two optimization levels. `SLaDe` is up to $6\times$ more accurate than Ghidra, a state-of-the-art, industrial-strength decompiler and up to $4\times$ more accurate than the large language model ChatGPT and generates significantly more readable code than both.

***Index Terms*—decompilation, neural decompilation, Transformer, language models, type inference**

I. Introduction

Decompilation is the automatic lifting of assembly instructions to higher-level, human-readable source code [1]. Decompilation enables the porting of legacy programs to new hardware [2], and the detection of malware for security protection [3]. 50 years of decompilation research [4] has resulted in a number of commercial [5] and open-source tools [6], which represent many years of engineering effort [7].

Although well-studied, decompilers are fundamentally limited by their inability to determine the types of variables and functions declared outside the source assembly, relying on programmer assistance. In terms of readability, they rely on large bodies of pattern-matching rules [8], which generate hard-to-read code. Work to make these decompiled codes less complex [6, 9, 10] is still well short of what a programmer would write.

Neural decompilation [11] promises to overcome this producing more human-readable programs [12], but existing techniques usually rely on synthetic data and do not generate correct code. Neural decompilation techniques [7] rely on token or string-level similarity metrics, such as edit-distance [3] which are effective in natural language translation [13]. However, these metrics fail to capture the correctness of a translation. The result is generated code that looks similar to the ground-truth but is incorrect.

Recently, large-language models such as ChatGPT [14] have received considerable attention. These models are able to generate high-quality programs from text inputs, at the cost of extreme model size. However, as we show in section VII, ChatGPT performs poorly at decompilation, frequently producing incorrect and hard to read code.

In summary, decompilers fall into two categories: traditional hand-crafted decompilers that produce hard to read code and fail to tackle external type declarations, and neural techniques that produce incorrect answers. What we want is a technique with the accuracy of traditional schemes that produces code as readable as neural approaches, works on real-world code, manages external type declarations and is portable across ISAs and code optimization levels with minimal engineering effort.

We present `SLaDe`, a **S**mall **La**nguage model **De**compiler. It comprises a 200M-parameter Transformer specifically trained for decompilation at program function, rather than code fragments, scale. We develop a novel code tokenizer and train a sequence-to-sequence, assembly to C model with no dropout. Because `SLaDe` uses a neural approach, we generate readable code compared to existing approaches (Section III-A2) and have easy portability across ISAs and optimization levels. While in-principle other neural approaches are portable, all existing work targets x86 exclusively. `SLaDe` is the first neural decompiler to be applied across ISAs and optimization levels.

To decompile code, `SLaDe` applies the trained model to assembly. This produces C code, which often contains undefined types, similar to existing rule-based decompilers. We use PsycheC [15] to generate types for the resulting source code (Section VI-B). This combination of neural translation and compiler analysis is crucial to decompilation accuracy (Section VIII).

We perform a large-scale evaluation on over 4,000 executable programs from ExeBench [16] (following AnghaBench [17] methodology for obtaining compilable C functions, and

ExeBench methodology for making them executable) and compare against the neural decompiler BTC [3], the state-of-the-art industrial strength decompiler Ghidra [6] from the NSA, and the large language model ChatGPT [14]. We show that it generates code that is both more accurate and more readable than all existing schemes across ISA and optimization level. We are up to 6× more accurate than Ghidra and up to 4.2× more accurate than ChatGPT. In addition, we generate code that is 1.65× to 3.68× more edit-similar to the original source than Ghidra and 1.07× to 3.83× nearer than ChatGPT.

This paper makes the following contributions:

1) We present the first neural decompiler that works across ISAs and optimization levels.
2) We introduce type inference-augmented neural decompilation, together with novel tokenization, dropout-free training over real-world code which dramatically improves semantic accuracy compared to existing neural decompilers.
3) `SLaDe` outperforms industrial strength rule-based approaches in terms of accuracy while significantly improving readability.
4) We show in a large-scale evaluation on a dataset of 4,000 functions across two optimization levels and two ISAs that our small model significantly outperforms ChatGPT with three orders of magnitude fewer weights.

II. Motivation

This section presents an example illustrating the complexity of decompilation before sketching an outline of our approach.

A. Examples

Figure 1 shows a typical decompilation task selected from the Synth Benchmarks where Box 2 contains the original C code. If we compile this function with GCC using level -O3 optimization, we obtain the assembly in Box 4. GCC's -O3 optimization has added significant complexity, unrolling and vectorizing the loop resulting in 55 lines of assembly, multiple loops and vector instructions. By feeding this assembly code into various decompilers we receive the following outputs:

Ghidra [6], a state-of-the-art, industrial strength, patter-matching decompiler generates the code shown in Box 2; a literal translation of the input assembly into a higher-level source language. While correct, the generated code is tightly coupled to the input assembly's structure, making it verbose, hard to read, and bears little resemblance to the input source code. Ghidra makes no attempt to produce interpretable variable names.

BTC [3], the only publicly-available neural decompiler that works on non-synthetic code, aims to reduce the readability gap of traditional decompilers. BTC accepts only unoptimized assembly so we fed as input the -O0 assembly (not shown). While the output, shown in Box 5, is less verbose and more readable than Ghidra, it does not contain a function header and has incorrect behavior.

ChatGPT [14] improves over BTC in that it generates a complete function definition as shown in Box 3. It infers human-readable variable names and even adds comments. However, like Ghidra, the output is long and has complex control-flow. Surprisingly, it contains x86 intrinsic instructions obscuring the meaning of the function. Furthermore, when executed it gives incorrect results.

`SLaDe`, presented in this paper, produces output with correct control-flow structure and the same number of variables as shown in Box 6. While the names of the variables are different and the pointer argument is equivalently declared as an array, the behavior of this code is immediately obvious and is functionally equivalent to the original.

B. Our Approach

Figure 2 gives an overview of our approach. We represent programs as a sequence of tokens and use a sequence-to-sequence encoder-decoder Transformer model. We train the model with (assembly, C) sequence pairs, adjusting weights to minimize the difference between the predicted output and the correct one using the cross-entropy loss function. Unlike standard approaches, we do not use dropout regularization [18]. As per our preliminary experimentation, weight decay regularization alone (with no dropout) yielded better results.

During inference, we run the model over the source assembly, using beam search to generate a C program. We apply type inference (Section VI-B) to generate missing types required to run generated program if it is required. Provided the decompiled program is compilable and executable, we test it for equivalence using I/O examples (Section III-A1).

In this work, we tackle the task of decompilation from assembly instructions to C, not disassembly. That is, we directly feed assembly instructions to `SLaDe` rather than machine bytes. The task of high-quality disassemblers such as [19], while complementary to our work, is out of the scope of this article.

III. Decompilation as neural machine translation

We learn a model T that takes as input a low-level assembly program as input and outputs a high-level C program. Our model supports optimized an unoptimized assembly, and handles x86 and Arm assembly.

$$T : S \mapsto C \tag{1}$$

where $S = \text{x86} \cup \text{Arm}$, the set of all x86 and Arm assembly programs at optimization levels O0 and O3, and C is the set of all C-language programs.

We represent each instance of S, C as a sequence of tokens. $c = [c_1, \ldots c_n] \in C, s = [s_1, \ldots s_m] \in S$ and restrict program length $m = 1024$.

A. Equivalence

For correct translation between some assembly $s \in S$ and some source $c \in C$, we require that meaning of s and c are identical

$$M[\![s]\!] \equiv M'[\![c]\!] \tag{2}$$

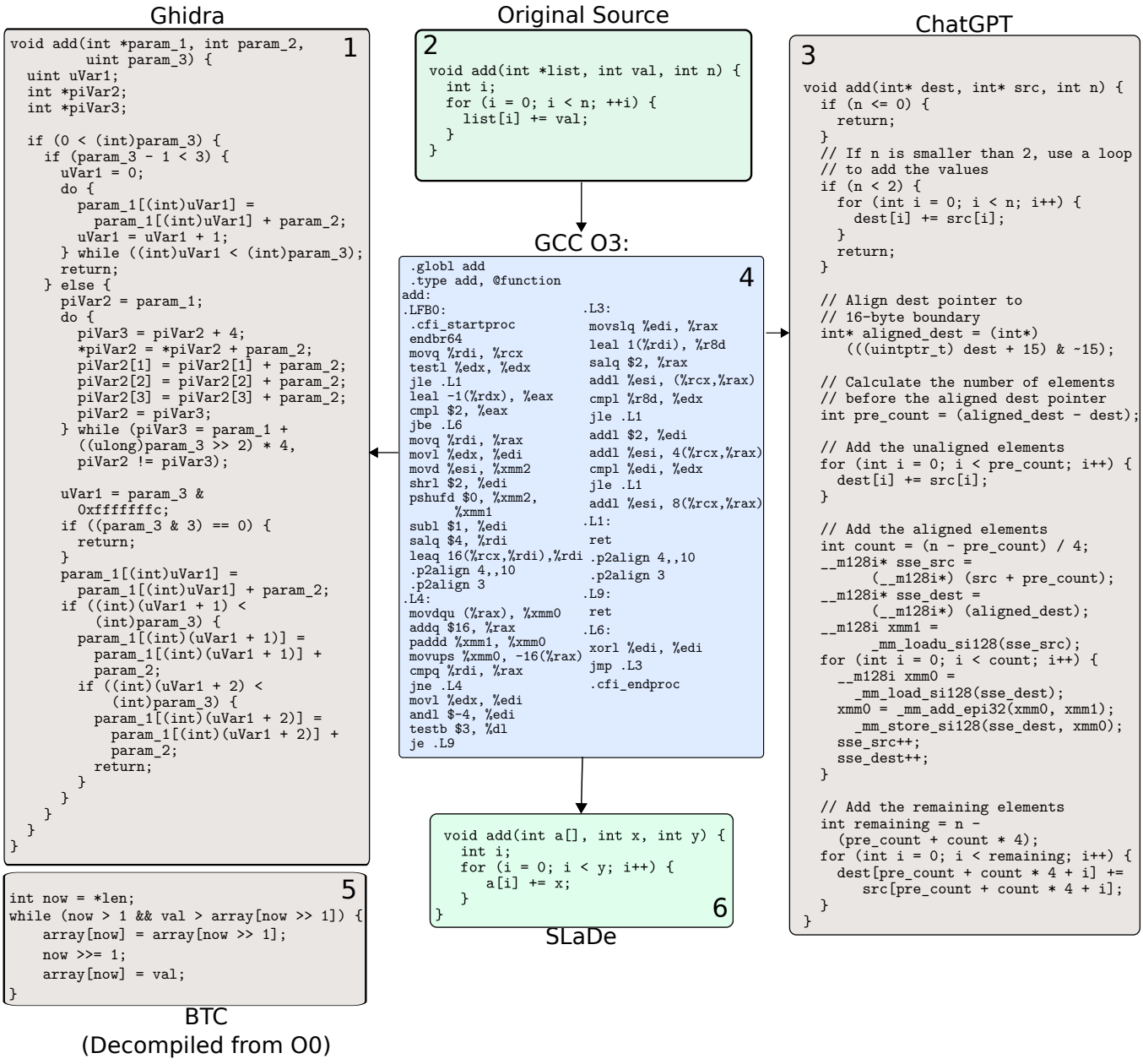

Fig. 1. Comparing decompilation techniques to the ground-truth. We compile the original code (box 2) using GCC O3 and then decompile with each technique. As BTC was trained on O0 code, we use O0 to evaluate it. We can see that Ghidra (box 1) and ChatGPT (box 3) produce very difficult to read code and in ChatGPT's case, the code is incorrect, adding two arrays together rather than adding a constant to an array. BTC (box 5) produces significantly more readable, but also incorrect, code. `SLaDe` (box 6) produces readable, correct code.

Where M, M' are functions denoting the meaning of programs in S and C respectively. We use $\equiv$ to denote equivalence of program state rather than just return values. In general equation 2 is undecideable [20] and we instead consider both more restricted and looser bounds of equivalence, Input/Output Equivalence on finite subsets.

1) Input Output (IO) Accuracy: A typical definition of semantic equivalence is to use behavioural equality. That is, for some functions $c \in C$ and $s \in S$ over some domain D: $\forall x \in D.s(x) \equiv c(x)$. This is undecidable for domains where D is infinite, which is frequent in practice. To make this notion of equivalence decidable, we instead use input/output equivalence on *finite subsets*. That is, we select some finite set $\mathcal{F} \subseteq D$ and then seek to prove that:

$$\forall x \in \mathcal{F}.s(x) \equiv c(x) \tag{3}$$

This process is decidable provided we assume that c or l terminates. In practice, non-termination is rare and we assume non-equivalence in cases on non-termination. This process used for compiler testing in [21] is successfully used in evaluating

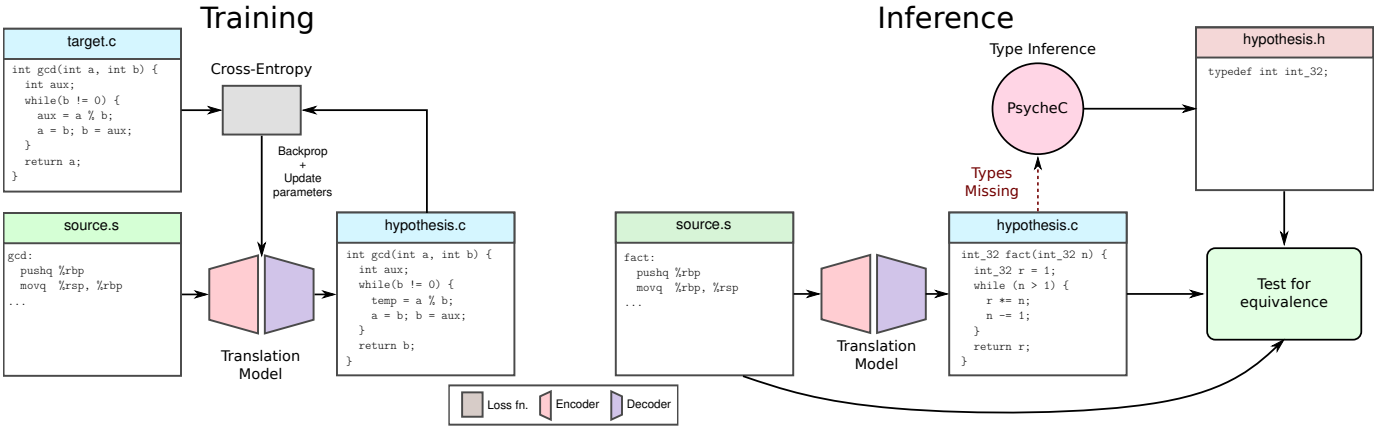

Fig. 2. We train a small Transformer to minimize the cross-entropy loss function. At inference time, we use the model to generate code. Generated code with missing typedefs is passed to PsycheC [15] to generate candidate types. We then check the inputs for correctness using input/output examples.

standard decompilers in [22]; this paper is the first to use it on neural decompilation. As the size of the finite subset increases, our confidence increases that the programs are truly equivalent also increases, but is only guaranteed in cases where D is finite and fully-explored. In many case tighter bounds on equivalence can be found via model-checking and other techniques [23].

2) Edit similarity: A key metric for decompilation tools is the readability of the code. Readbility is of course a subjective metric, but for a decompilation task, we are most interested in similarity to the original source code. We use edit distance, a standard metric in other neural approaches [3, 24] to give *edit similarity* a measure of closeness.

The edit *distance* is defined as the minimum number of *insertion*, *deletion*, and *replacement* operations needed to transform one string into another. Given some decompiled C, $c^* = \{c_1^*, \ldots, c_n^*\}$ and a ground-truth source C, $c = \{c_1, \ldots, c_n\}$, We can compute the edit distance using a dynamic-programming based algorithm defined in figure 3. We use ε to represent the empty sequence. We use *edit similarity*, is $1 - \frac{\text{Edit Distance}}{\text{Sequence Length}}$, which is normalized to sequence length (of the ground truth target) and converted so that a higher edit similarity represents better readability.

$$
\begin{aligned}
&\text{Distance}(\varepsilon, \{c_1, \ldots, c_j\}) = j \\
&\text{Distance}(\{c_1^*, \ldots, c_k^*\}, \varepsilon) = k \\
&\text{Distance}(\{c, c_2^*, \ldots, c_k^*\}, \{c, c_2, \ldots, c_j\}) = \\
&\qquad \text{First tokens equal:} \\
&\qquad \text{Distance}(\{c_2^*, \ldots, c_k^*\}, \{c_2, \ldots, c_j\}) \\
&\text{Distance}(\{c_1^*, c_2^*, \ldots, c_k^*\}, \{c_1, c_2, \ldots, c_j\}) = \\
&\qquad \text{One case for each of delete, insert and replace:} \\
&\qquad \min(\\
&\qquad\quad \text{Distance}(\{c_2^*, \ldots, c_k^*\}, \{c_1, c_2, \ldots, c_j\}) + 1, \\
&\qquad\quad \text{Distance}(\{c_1^*, c_2^*, \ldots, c_k^*\}, \{c_2, \ldots, c_j\}) + 1, \\
&\qquad\quad \text{Distance}(\{c_2^*, \ldots, c_k^*\}, \{c_2, \ldots, c_j\}) + 1 \\
&\qquad)
\end{aligned}
$$

Fig. 3. Algorithm for computing the edit-distance between two sequences. We use ε to represent the empty sequence. We use *edit similarity*, which is $1 - \text{Edit Distance}/\text{Sequence Length}$, so that a higher edit similarity represents better readability.

IV. Tokenization

Our model consumes and produces sequences of tokens from a fixed vocabulary. Since we also model identifiers, this could cause encountering unknown tokens during inference. To avoid those out-of-vocabulary tokens, we use subword tokenization [25], in which tokens are derived from the character frequencies in the train set. Because individual characters present in the train set (in this case, essentially the ASCII alphabet) are also part of the vocabulary, unseen tokens can always be built from seen subwords, even character by character if required. We base our subword tokenization on UnigramLM [26], which has been shown to either match or exceed (by up to 10 F1-score points in some downstream tasks [27]) the performance of Byte-Pair Encoding [25].

We modify the default parameters of UnigramLM to make it more amenable to code. We set a small vocabulary size of 8k due to the small number of C keywords and assembly opcodes as compared to unrestricted natural language (vocabulary sizes in natural language processing are typically >30k). We tokenize numbers digit-by-digit as in [28], e.g. $512 \rightarrow [5, 1, 2]$. This prevents inconsistencies when encoding large numbers (e.g. $512 \rightarrow \{[5, 1, 2], [5, 12], [51, 2], [512]\}$. We split all punctuation signs into different tokens (so that dots are not merged in float numbers, etc). We protect spaces by escaping with the metaspace character (unicode _) as in SentencePiece [29], but only inside double quotes for string definitions. Otherwise, spaces are normalized (replaced with a single space).

V. Training

We pose neural decompilation as a sequence-to-sequence task. Given a dataset $\mathcal{D}$ with N pairs $\{(\mathbf{s_n}, \mathbf{c_n}) | n \in \{0..N-1\}\}$, where $\mathbf{s_n}$ is a function assembly code and $\mathbf{c_n}$ is the corresponding C code that produced it, we want to train a model T parameterized by $\boldsymbol{\theta}$ with maximum likelihood estimation:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \prod_{n=0}^{N-1} P(\mathbf{c_n} | \mathbf{s_n}; \boldsymbol{\theta}) \tag{4}$$

In practice, we minimize the negative log-likelihood, which is equivalent to (multiclass) cross-entropy (CE). For each minibatch (i.e., group of $B << N$ examples in the dataset), we use stochastic gradient descent to update the parameters of the model given the gradient of the cross-entropy loss function:

$$\boldsymbol{P}(\boldsymbol{\theta}) = \mathrm{T}\left(\boldsymbol{s}_n, \boldsymbol{c}_n \mid \boldsymbol{\theta}\right) \tag{5}$$

$$\mathrm{CE}(\boldsymbol{\theta}) = -\sum_{l=1}^{L-1} \log P(\boldsymbol{\theta})\left[\boldsymbol{c_n}[l+1], l\right] \tag{6}$$

$$\boldsymbol{\theta'} = \boldsymbol{\theta} - \eta \cdot \nabla \mathrm{CE}(\boldsymbol{\theta}) \tag{7}$$

where $\boldsymbol{P}(\boldsymbol{\theta})$ are the token probabilities predicted by the model (the expected outputs $\boldsymbol{c}_n$ are also needed by the model in training due to the use of teacher forcing[1]), L is the sequence length, η is the learning rate hyperparameter, and $\boldsymbol{\theta'}$ are the updated parameters.

A. Training Dataset

AnghaBench [17] is dataset of compilable C functions scraped from public repositories. We use an expanded version with around 4M functions paired with the corresponding function-level assembly obtained from ExeBench [16]. The scraped functions contain no restrictions so that the code is representative of real-world code. We ensure that the test set functions are not present in the training set with token-level hash-based deduplication. We feed functions (as opposed to programs) to our model to avoid long sequences, which would make the learning task more challenging and more computationally expensive.

The assembly-C pairs come from GCC with different optimization levels. For each assembly-C pair, we take the assembly function without its surrounding context and ask our model to predict the corresponding C function without any surrounding context. By design, `SLaDe` does not have access to functions, typedefs or variables external to the function. These are predicted using type-inference (section VI-B). For our evaluation, we add the original context of the C program into the context we evaluate it in. We use these inferred types for compilation and execution of hypothesis decompilations.

[1]That is, in training, unlike in inference, when predicting the token l, the model is fed the ground truth up to $l-1$, not its own predictions autoregressively. This means that in training the decoder can run in parallel, and that compute is not thrown away by trying to predict sequences that are already too far off from the ground truth.

B. Sequence-to-Sequence Transformer

We use a sequence-to-sequence Transformer model [30]. Each input token from the source sequence is embedded into a real-valued vector. Then, the sequence is passed through the encoder, which contains M encoder blocks defined as follows:

$$\bar{h}_m = h_{m-1} + \mathrm{MHA}\left(\mathrm{LN}\left(h_{m-1}\right)\right) \tag{8}$$

$$h_m = \bar{h}_m + \mathrm{FFN}\left(\mathrm{LN}\left(\bar{h}_m\right)\right) \tag{9}$$

where MHA is the multi-head attention layer, LN is layer-normalization [31], and FFN is a feed-forward network. Crucially, the layers in each block sum its outputs to the outputs from the previous layers (h_{m-1}). This kind of connection is known as residual connection and it eases learning, enabling the training of deeper models [32]. Regarding the MHA, it is defined as follows:

$$MHA = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{10}$$

Where Q is the query matrix, K is the key matrix, V is the value matrix, and d_k is the dimension of the keys. In encoder's (and decoder's) self-attention, all key, value, and value vectors come from linear projections from the input sequence (from the previous layer). Note that MHA is defined with matrices, and not vectors, because the implementation is batched.

Similarly, the input tokens from the target sequence are also embedded, and passed through the decoder attention-based blocks. There are two relevant differences with respect to the encoder. The first one is that decoder blocks have an additional MHA layer (followed by an additional LN) to perform encoder-decoder attention (as opposed to self-attention). The second one is that in training, causal masking is applied so that future tokens don't leak into the input of the decoder. Finally, the output embeddings from the decoder are projected back into the vocabulary space with an additional linear layer, so that, after applying the softmax function, the output is a probability distribution for the tokens in the vocabulary.

C. Model Architecture and Training Details

We use the modifications from BART [33]. Our model has 6 encoder layers, 6 decoder layers, an embedding size of 1024, a context window of 1024 positions for both source and target sequences, and shared embeddings for the encoder, decoder and decoder output layer. We initialize the parameters from $\mathcal{N}(0, 0.02)$. Instead of vanilla stochastic gradient descent, we use the Adam optimizer [34]. Each model is trained for 72 hours on 4 Nvidia A100s. In total, we train 4 models (one for each evaluated architecture and optimization level).

We do not use dropout [18] regularization method typically used in Transformer models, following the intuition, confirmed in preliminary experiments, that it would be detrimental to our task. Instead, we regularize with weight decay.

VI. Inference

Figure 2 shows the process SLaDe uses to decompile source code. First, we use our trained model to generate source code (section VI-A). This generates a hypothesis C file. We then use type inference (Section VI-B) to infer the types that should be inserted.

A. Model Inference

SLaDe uses a trained model T that takes as input some assembly s and generates a C code hypothesis, $\hat{c}$. In the simplest case, with *greedy* decoding, we would feed s into the encoder, which would encode it in parallel. Then, autoregressively, we would keep taking the token that maximizes the probability, until predicting the special token EOS marking the end of the sequence. Since our goal is to maximize the global probability of the predicted sequence as opposed to the local probability of just the next token, we use beam search decoding with a beam size of $k = 5$. That is, at each step, we keep the top k hypotheses with the highest probability, and at the end of the decoding we select the first one passing the IO tests (if any). To speed up evaluation, we batch the examples on an NVIDIA A6000 GPU. However, the model is small enough to be run on consumer CPUs with decent latency.

B. Type Inference

The C code that our model outputs can feature missing types. For example, it may be that T generates code that relies on some type `my_int` that it has frequently seen in training, but is not part of the standard library. We use type inference (Section VI-B) to solve this problem. For any of these types we use PsycheC [15] which is a tool that infers types that respect C's type system. PsycheC takes a partial program, $\mathcal{P}$ as its input and produces a complete program $\mathcal{P}'$ that can be compiled.

It performs three key steps to do this. First, PsycheC parses partial C programs, which can contain ambiguities (e.g., `(a)*b` could be a cast or a unary operation or a binary operation depending on whether `a` is a variable or a type).

After parsing, PsycheC produces a series of constraints that must be applied to generated types. For example, if we have some type τ that is used in an assignment τ `a = b;`, and we know `b` is of type τ', we can conclude that $\tau = \tau'$. PsycheC builds a set of rules for type equivalence e.g., $\tau_1 = \tau_2 \implies \tau_1* = \tau_2*$ and uses syntax-directed generation of constraints on types within this framework. For example, $*$`E` has type $*\tau$ if `E` has type τ. To deal with the ambiguities discussed above, PsycheC uses a *lattice* model of constraint generation.

Once these constraints are generated, PsycheC solves them and produces types that make the program compile. We inject these generated dependencies into our generated decompiled code, checking that there is no conflict with the previously defined code. Once we have compilable code, we test it using IO examples as discussed in Section III-A1.

VII. Results

We evaluate SLaDe against alternative approaches on two different ISAs, at two different optimization levels on two different benchmark suites.

A. Experimental Setup

1) Benchmarks: We evaluate our approach on two benchmark suites: the small-scale 112 program synthesis benchmark used in [35], Synth, to examine in detail decompilation behavior; and a subset of ExeBench [16] to allow wider-scale evaluation and interaction with user-defined types and external function calls, a challenging task for decompilation. For all models and baselines, we discard the benchmarks in which GCC couldn't compile the original C code in our host machine used for evaluation. For a more realistic evaluation, we do not discard any benchmark based on length. None of the about 4000 test benchmark programs were seen during training as per the performed token-level hash-based deduplication. We report the normalized edit similarity with respect to the ground truth, and the percentage of functions that pass the IO tests (Section III-A). That is: how many decompiled programs when recompiled, give the same output as the original assembly for different inputs? We evaluate on two different ISAs, x86 and ARM. We consider two levels of code optimization -O0 and -O3 to evaluate the impact of code complexity.

2) Decompilers:

a) Ghidra: We invoke the headless version of the Ghidra decompiler, providing the object file of the compiled function assembly generated by GCC. We then insert the resulting C code function into the original calling program which contains context such as type declarations. Ghidra frequently makes reference to standard types that may not be declared in the original program such as `bool`. For Ghidra alone, we insert those types into the header of the program for fair evaluation.

b) ChatGPT: is a well known chatbot. with a wide range of uses, including program generation. We used a best-effort, trial-and-error approach to find an effective prompt. The following format gave the best response: `Decompile the following ARM assembly function into C code. Return the code and only the code, no explanations or introductions.`

We also experimented with *few-shot learning*, that is, including examples of the intended decompilation task in the prompt itself, but this harmed rather than helped due to the very long sequences needed to include C-assembly pairs into the prompt. We inserted the resulting function into the original calling program as for Ghidra.

c) BTC: is the only publicly available alternative neural-decompiler that tackles real-world assembly. We run it on Synth benchmark using the model provided by the authors. Elsewhere, we use the value reported in their paper (on another dataset), in the case of edit similarity, or the accuracy from the model limitations stated in the paper.

d) SLaDe: After invoking SLaDe we insert the resulting function into the original calling program as for Ghidra.

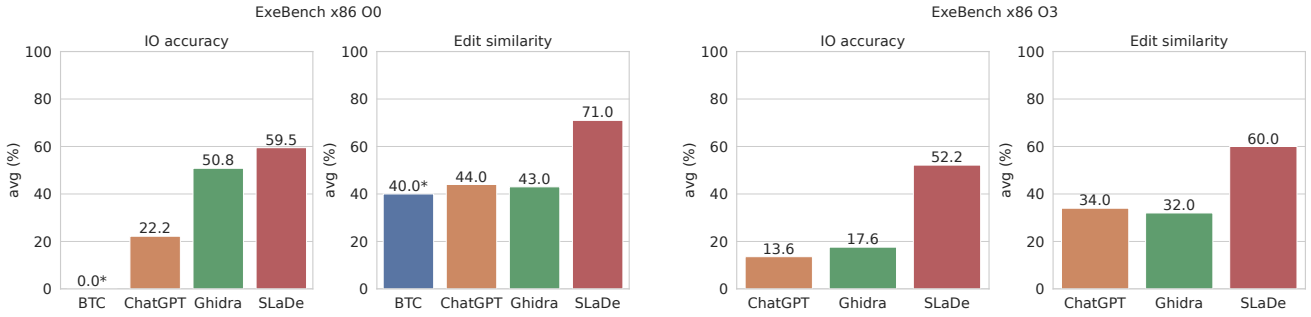

Fig. 4. ExeBench, x86: -O0 (left) -O3 (right), input-output (IO) accuracy and edit similarity. A decompiled program is IO accurate if it gives the same outputs for the same range of inputs as the original assembly. BTC's edit distance is as-repored in [3] on a different dataset. Its dataset is restricted to -O0 and does not support evaluations of correctness, so omitted. `SLaDe` out-performs existing techniques, producing 1.17x to 3.83x more correct code, and a higher edit similarity than Ghidra, ChatGPT and BTC. .

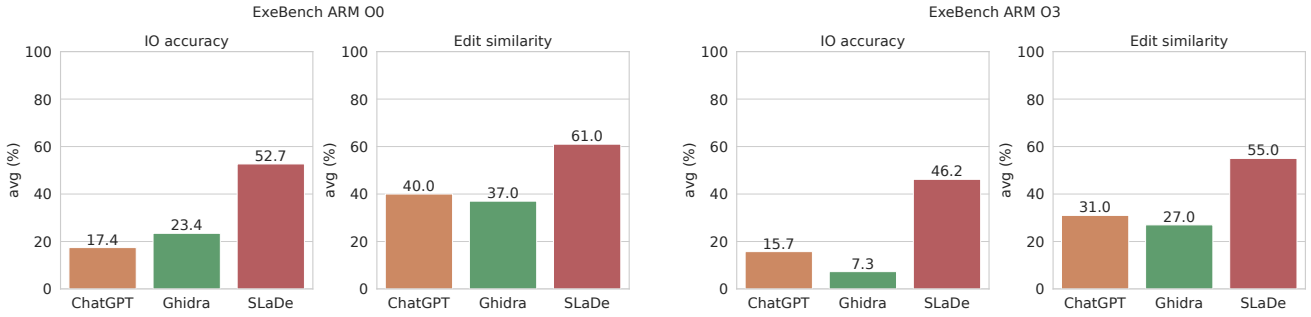

Fig. 5. ExeBench, x86: -O0 (left) -O3(right), IO accuracy and edit similarity. `SLaDe` significantly out-performs existing techniques producing 2.2x to 6.32x more accurate code and a higher edit similarity.

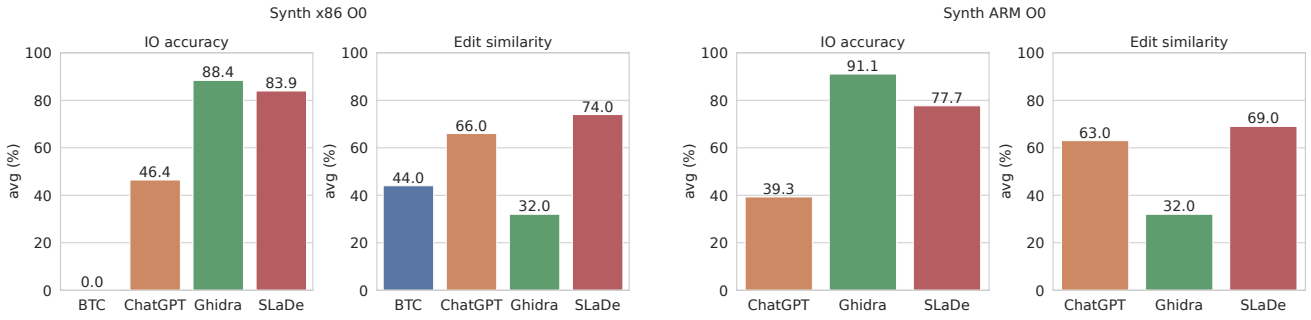

Fig. 6. Synth -O0: x86 (left) and ARM (right), IO accuracy and edit similarity. BTC edit similarity is experimentally evaluated. Ghidra has slightly higher IO accuracy than `SLaDe` on these simpler, unoptimized, benchmarks.

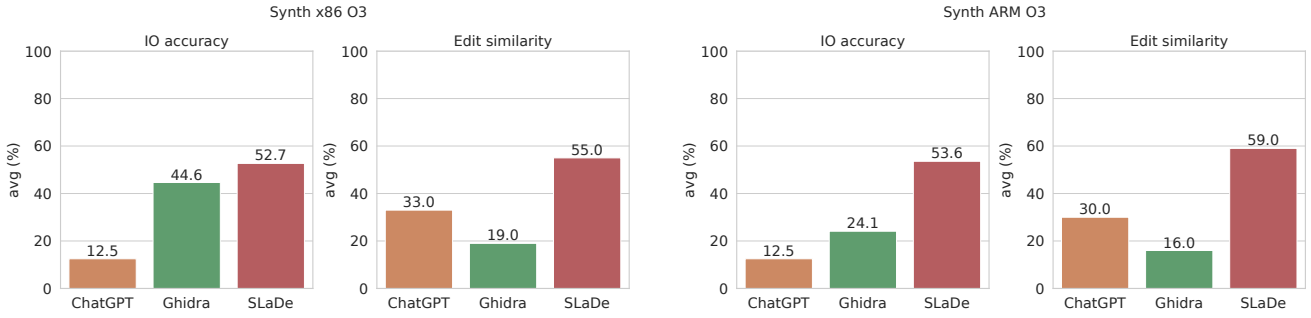

Fig. 7. Synth -O3: x86 and ARM, IO accuracy and edit similarity. `SLaDe` has a small reduction in accuracy compared to O0 while Ghidra is more negatively affected. `SLaDe` produces 1.81x to 4.28x more accurate code.

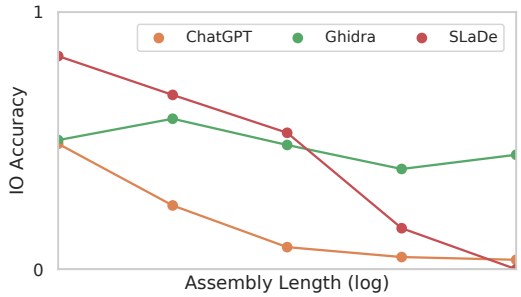

Fig. 8. IO accuracy as the program size changes.

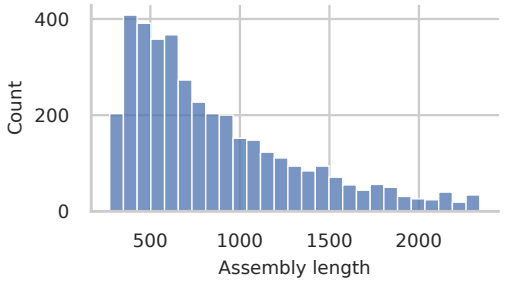

Fig. 9. Distribution of program lengths in ExeBench by character length.

SLaDe is implemented in PyTorch [36] using Fairseq [37] and Huggingface Transformers libraries [38].

B. x86 Decompilation

Figure 4 shows the performance of SLaDe relative to the alternative techniques. The two leftmost graphs show IO accuracy and edit similarity on unoptimized (-O0), x86 code from ExeBench. The rightmost two show the same results for optimized (-O3), x86 code.

a) Unoptimized -O0 decompilation: SLaDe is able to accurately decompile almost 60% of the holdout ExeBench test suite, 17% more than Ghidra and more than double ChatGPT. BTC is unable to accurately decompile any of the programs accurately which is not surprising given its limited focus. SLaDe's failures are largely due to incorrectly predicting the arguments to externally declared functions. Ghidra has the same issue as well as additional type errors due to structs.

SLaDe is able to produce highly readable code with an edit similarity of over 70%. This was a significant improvement over the other approaches: 40% to 44% similarity. The value for BTC is a reported values from [3] and is close to that experimental value of ChatGPT. In section VII-E we experimentally evaluate it on the smaller Synth bench. Surprisingly Ghidra has a similar edit similarity to ChatGPT.

b) Optimized -O3 decompilation: While SLaDe is able to achieve impressive performance on unoptimized code, in practice most shipped code is optimized and the most likely use case for decompilation. The two right-most graphs show x86, -O3 results. We do not include BTC as it targets unoptimized (-O0) code. All approaches unsurprisingly find -O3 more challenging. However the relative decline in IO accuracy for SLaDe is less than for the other schemes. SLaDe is now able to deliver almost three times the number of correctly decompiled programs as Ghidra and almost four times that of ChatGPT. Furthermore, the readability of the code generated by SLaDe is nearly twice that of ChatGPT and Ghidra.

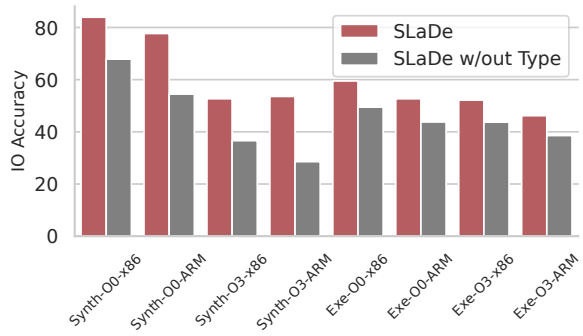

Fig. 10. SLaDe with and without type-inference (Section VI-B). On average, the type inference algorithm makes SLaDe 14% better.

C. ARM Decompilation

In principle one of the main benefits of neural decompilation is that it can be straightforwardly trained and applied to new ISAs with no re-engineering effort. Figure 5 shows performance on a new ISA, ARM, with the two leftmost graphs showing IO accuracy and edit similarity for unoptimized (-O0) code from ExeBench. The rightmost two graphs show the same results for optimized (-O3) code. Compared to x86, SLaDe has a slight degradation in accuracy and edit similarity. However, Ghidra's accuracy degrades by at least a factor of two. SLaDe is now more than 2× as accurate on -O0 and more than 6× more accurate at -O3. ARM's stack calling conventions appear to be an issue for Ghidra. ChatGPT is less affected though still 3× less accurate than SLaDe.

D. Discussion

Despite being rule-based, Ghidra has a key restriction. In cases of external type/function declarations, unlike SLaDe, Ghidra does not generate types but rather leaves them undefined. This accounts for many of the IO correctness failures (due to lack of compilability) it suffers. While a programmer could resolve these, given the challenges in understanding Ghidra's code, filling in the missing dependencies is a non-trivial task.[2] In the next section, we therefore evaluate using simpler programs where type ambiguity is not an issue.

E. Synthesis Benchmark

Here we evaluate on a set of benchmarks with simpler types. Figure 6 shows the performance of SLaDe against alternative approaches.

[2]https://github.com/NationalSecurityAgency/ghidra/issues/236

TABLE I
PEARSON'S CORRELATION COEFFICIENT BETWEEN CODE FEATURES AND IO ACCURACY ON EXEBENCH.

	x86 O3			x86 O0			ARM O0			ARM O3		
	ChatGPT	Ghidra	SLaDe	ChatGPT	Ghidra	SLaDe	ChatGPT	Ghidra	SLaDe	ChatGPT	Ghidra	SLaDe
Compiles	0.54	0.81	0.80	0.53	0.81	0.94	0.49	0.96	0.95	0.59	0.98	0.78
Edit Similarity	0.32	0.16	0.59	0.24	0.01	0.53	0.19	0.07	0.58	0.25	0.13	0.60
ASM Length	-0.10	0.03	-0.13	-0.18	-0.06	-0.30	-0.20	-0.14	-0.36	-0.13	-0.04	-0.16
C Length	-0.07	-0.01	-0.14	-0.11	-0.04	-0.21	-0.11	-0.05	-0.21	-0.09	-0.02	-0.15
Num Func Args	-0.08	-0.01	-0.12	-0.07	0.07	-0.13	-0.07	0.20	-0.09	-0.08	0.04	-0.10
Num Pointers	-0.04	0.03	-0.16	-0.03	0.14	-0.17	-0.04	0.20	-0.12	-0.06	0.08	-0.14

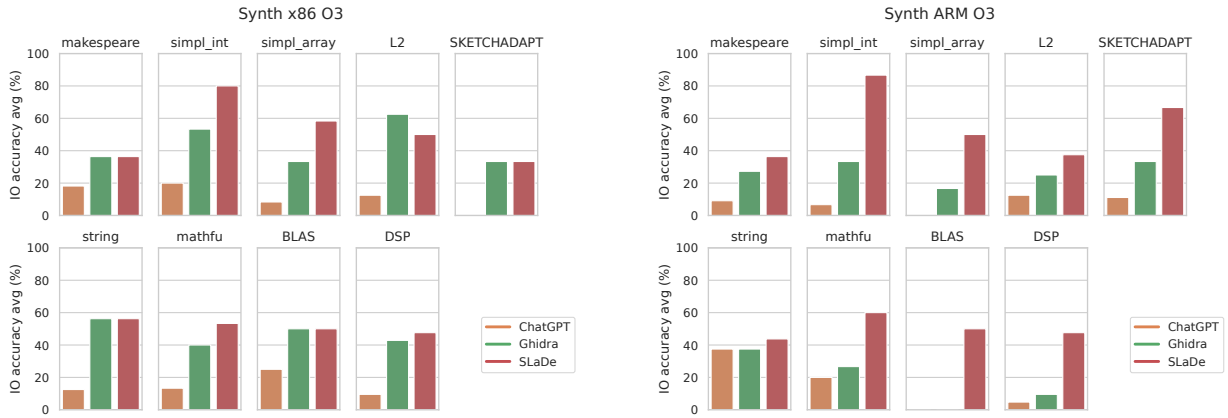

Fig. 11. Analysis of IO accuracy depending on program type and optimization level. We see that SLaDe performs significantly better on simple integer benchmarks in simpl_int, achieving more than 80% in both cases than it does on the functional programming-based benchmarks in SKETCHADAPT where it achieves 40% and 60% on x86 and Arm respectively.

a) Unoptimized x86, ARM: On unoptimized assembly, both SLaDe and Ghidra perform well with over 80% IO accuracy, showing that these benchmarks are easier to decompile. They contain only simple argument types and do not call external functions. Ghidra outperforms SLaDe by 5% on x86 and 20% on ARM but is significantly worse in the readability of the generated program. Surprisingly Ghidra has worse readability on these benchmarks relative to ExeBench. SLaDe has more than double the edit similarity of Ghidra for both ISAs. ChatGPT is significantly less accurate than both SLaDe and Ghidra. It has better readability than Ghidra but is inferior to SLaDe on both ISAs. We also evaluated BTC on this smaller set of benchmarks where it was shown to have a better edit similarity 44% than its reported result 40%.

b) Optimized x86, ARM: Figure 7 again shows that optimization has a dramatic effect on Ghidra, particularly on the ARM ISA where IO accuracy has reduced by over two-thirds. The performance of ChatGPT similarly falls. Complex types are no longer the problem for Ghidra, rather the obfuscating effect of compiler optimization has degraded its performance. In the optimized setting SLaDe has superior accuracy for both ISAs and superior readability. While the readability of Ghidra and ChatGPT almost half when optimizing, SLaDe only suffers a small degradation.

F. SLaDe failure

While SLaDe performs well, here we cover two typical failure cases. For the ground truth code:

```
void satd8x8_getDiff(unsigned int res[]) {
  memcpy(res, &mat[lastRow * 8],
         8 * sizeof(*mat));
  lastRow++;
}
```

SLaDe produces:

```
void satd8x8_getDiff( float *mat ) {
  memcpy( mat, &mat[lastRow*8], 32);
  lastRow++;
}
```

Critically, SLaDe mistypes the argument to the function, as most matrix variables it encounters are of float type. Similarly, for the ground truth code:

```
void clock_add( SClock *clk, double incr ) {
  if(clk) {
    clk->curtime += incr;
    clk->basetime += incr;
    clk->seqno++;
  }
}
```

SLaDe produces:

```
void clock_add(struct clock *__restrict ev,
               double d) {
  if(ev) {
    ev->constev += d;
    ev->constsp++;
```

```
    ev->constt--;
  }
}
```

Which is the right idea, but aside from different struct and field names, it uses the ++ and the −− operations rather than the +=incr and ++ operations, respectively. When the original code was compiled with -O0, Ghidra produced correct decompiled code. However, with -O3 GCC's optimizations obscuring the resulting assembly, Ghidra was not able to produce compilable code.

VIII. Analysis

In this section, we analyze the results and explore the limitations of traditional and language-model-based decompilation.

A. Impact of Assembly Length on Accuracy

Figure 8 shows how IO accuracy similarity varies for increasing assembler length for ExeBench, x86 -O0. All three approaches are better at decompiling shorter assembler sequences, since increased assembly complexity makes decompilation more challenging. Both ChatGPT and SLaDe have a steeper decline than Ghidra, illustrating that neural schemes are more sensitive to the distribution type of training examples than hand-crafted approaches. Figure 9 shows the distribution of assembly length in ExeBench. We see that there is indeed a bias to shorter-length assembly.

B. Impact of Type Inference

Figure 10 ablates the impact of type inference (Section VI-B). On average, using type inference improves SLaDe by 14% by increasing the number of decompiled examples that can compile. These 14% are the samples for which SLaDe generates correct code, but was missing the required typedefs. The type inference algorithm was then able to infer the correct typedefs to make the code execute correctly.

C. Features

To understand what factors affect each of the three decompilation schemes, we examined the correlation of IO accuracy to different characteristics or features of the assembler programs. The features considered are: whether the code compiles, the edit similarity, the assembler length, the ground truth C program length, the number of function arguments and the number of pointers. Table I shows the correlation coefficient for each approach across each ISA and optimization level.

Not surprisingly, whether a program compiles is important across all approaches and ISAs. A program that does not compile is unlikely to be IO accurate. However, this correlation is less significant for ChatGPT which is able to hallucinate programs that do compile but are incorrect. Edit similarity is important for the neural schemes, particularly SLaDe, but less so for Ghidra which is less concerned with readability. All other features are more weakly correlated. There is a small negative correlation with assembler and C length suggesting longer programs are harder to decompile. The number of functions and pointers is negatively correlated for ChatGPT and SLaDe but weakly positive for Ghidra. However, the magnitude is small and no conclusions can be drawn.

D. Impact of Program Type

The Synth benchmarks are broken down into groups. In Figure 11, for x86, the easiest benchmarks to decompile were the simple integer ones, Simple_int, which consists of integer types and arithmetic with trivial control-flow Sketchadapt programs were the hardest. consisting of more complex string manipulation programs. ChatGPT performed best on BLAS problems, presumably from their availability from web crawlings, but generally performed poorly across groups, failing on all Sketchadapt problems. In four groups Ghidra and SLaDe have equal accuracy. In one case, Ghidra performs better on the L2 group which consists of functional problems while SLaDe performs better on the remaining three.

On Arm, different behavior is seen with SLaDe outperforming across all categories. ChatGPT is able to decompile a Sketchapdapt problem but is now unable to manage any BLAS problems, its best-performing category on x86. Ghidra is also unable to decompile any of this group

IX. Related Work

A. Analytic decompilation

Traditional approaches based on program analysis [5, 6] represent years of hand-coded effort [39] and rely on large bodies of pattern-matching rules [8]. Research work in this direction has focused on correctness, through studies [22] and formal methods [23]. LLVM IR is a popular lifting target, to enable retargeting to new ISAs [40, 41]. More recent work lifts LLVM IR to OpenMP C to exploit parallelism while retaining meaningful names [42].

B. Neural Decompilation

Early neural decompilation work focused on generating *readable* code from short snippets, rather than semantically correct code [11]. Error-repair techniques can be used to address correctness [39] and [43]. However, these works only explore assembly generated from synthetic and restricted datasets, and only from small snippets.

Cao et al. [44] lifts binaries into intermediate representations, but is only evaluated against synthetic C programs. An alternative evolutionary approach based on genetic algorithms is explored in [45], but is only evaluated on 19 programs.

C. Binary-Source Code Matching

Gui et al. [46] propose XLIR leverage contextual embeddings of low-level representations (LLVM's intermediate representations) to find potential matches between pairs of code snippets in potentially different programming languages. While not generative (i.e., they don't generate new code snippets), XLIR is effective at tasks such as clone connection

D. Neural Code Translation

Since the advent of sequence-to-sequence models [47], neural machine translation has been applied to programming language translation [48, 49] often using data from coding websites [50], and also in unsupervised settings [51, 52, 53, 54]. Other tasks range from code style detection [55], generating accurate variable names [56], correcting syntax errors and bugs [57, 58, 59], code completion [7] and program synthesis [60] to API recommendation [61], specification synthesis [62] and full-scale code migration [51]. Regarding low-level code, in [63] they target the generation of LLVM IR from C while [64] targets the generation of x86.

X. Conclusion and Future Work

We present SLaDe, a small language neural decompiler trained on real-world code. SLaDe is the first decompiler to combine supervised training and type inference-based program analysis, allowing decompilation of code with external type and function declarations. This paper delivers the first neural decompiler portable across ISAs and code optimization levels. We conduct a large-scale evaluation on 4,000 functions against a state-of the art, industrial strength decompiler, Ghidra, and a general large language model, ChatGPT. Across two benchmark suites, two ISAs and two different types of optimized binaries, SLaDe provides superior performance in terms of both correct decompilation and readability.

Future work should increase the scope of decompilation to larger program units, potentially exploring the use of pre-training and program repair to improve accuracy. Longer-term, it would be interesting to investigate how learnable and analytic approaches could be best integrated to deliver both portable and correct decompilation. More generally, we believe the combination of neural approaches and analytic approaches such as type inference should be further explored.

Data Availability Statement

We refer to the accompanying artifact [65].

Acknowledgements

We thank Irina Rish, supported by the Canada CIFAR AI Chair Program and the Canada Excellence Research Chairs Program, and Compute Canada, for the help with part of the compute. We thank the reviewers for their insightful comments.

Appendix

Executing SLaDe's neural component can be as easy as running:

```
tokenizer.decode(model.generate(
  tokenizer.encode(ASSEMBLY)))
```

using Huggingface's Transformers Python API. Nevertheless, the type inference engine, the IO evaluation, and the baseline require additional complexity and dependencies.

The accompanying artifact [65] contains the instructions, license, model weights, tokenizers, evaluation data, and code required to evaluate SLaDe and the baselines. However, we cannot provide BTC's weights and ChatGPT API keys. We recommend the user to run the IO evaluation in a sandboxed environment.

References

[1] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Software: Practice and Experience*, vol. 25, no. 7, pp. 811–829, 1995.

[2] G. Stitt and F. Vahid, "Binary synthesis," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 12, no. 3, pp. 1–30, 2008.

[3] I. Hosseini and B. Dolan-Gavitt, "Beyond the C: Retargetable decompilation using neural machine translation," *Workshop on Binary Analysis Research*, 2022.

[4] B. C. HOUSEL III, *A study of decompiling machine languages into high-Level machine independent languages*. Purdue University, 1973.

[5] Retdec, "Retargetable decompiler," 2017. [Online]. Available: https://retdec.com/

[6] Ghidra, 2022. [Online]. Available: https://ghidra-sre.org/

[7] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav, "Towards neural decompilation," *CoRR*, vol. abs/1905.08325, 2019. [Online]. Available: http://arxiv.org/abs/1905.08325

[8] Hex-Rays, 2022. [Online]. Available: https://hex-rays.com/blog/ida-8-2-released/

[9] G. Chen, Z. Qi, S. Huang, K. Ni, Y. Zheng, W. Binder, and H. Guan, "A refined decompiler to generate c code with high readability," *Software: Practice and Experience*, vol. 43, no. 11, pp. 1337–1358, 2013.

[10] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. Internet Society, 2015.

[11] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 346–356.

[12] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models

trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[13] H. Maarif, R. Akmeliawati, Z. Htike, and T. S. Gunawan, "Complexity algorithm analysis for edit distance," in *2014 International Conference on Computer and Communication Engineering*. IEEE, 2014, pp. 135–137.

[14] OpenAI, "Chatgpt."

[15] L. T. C. Melo, R. G. Ribeiro, M. R. De Araujo, and F. M. Q. Pereira, "Inference of static semantics for incomplete C programs," *POPL*, 2018.

[16] J. Armengol-Estapé, J. Woodruff, A. Brauckmann, J. W. d. S. Magalhães, and M. F. P. O'Boyle, "Exebench: An ml-scale dataset of executable c functions," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, ser. MAPS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 50–59. [Online]. Available: https://doi.org/10.1145/3520312.3534867

[17] A. F. da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. Ferreira Guimarães, and F. M. Quinão Pereira, "Anghabench: A suite with one million compilable c benchmarks for code-size reduction," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 378–390.

[18] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[19] A. Flores-Montoya and E. Schulte, "Datalog disassembly," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1075–1092. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya

[20] G. Winskel, "The formal semantics of programming languages: An introduction," 1993.

[21] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 216–226, 2014.

[22] Z. Liu and S. Wang, "How far we have come: Testing decompilation correctness of c decompilers," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 475–487.

[23] S. Dasgupta, S. Dinesh, D. Venkatesh, V. S. Adve, and C. W. Fletcher, "Scalable validation of binary lifters," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 655–671.

[24] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav, "Towards neural decompilation," 2019.

[25] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. [Online]. Available: https://aclanthology.org/P16-1162

[26] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," 2018. [Online]. Available: https://arxiv.org/abs/1808.06226

[27] K. Bostrom and G. Durrett, "Byte pair encoding is suboptimal for language model pretraining," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 4617–4624. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.414

[28] M. Muffo, A. Cocco, and E. Bertino, "Evaluating transformer language models on arithmetic operations using number decomposition," in *Proceedings of the Thirteenth Language Resources and Evaluation Conference*. Marseille, France: European Language Resources Association, Jun. 2022, pp. 291–297. [Online]. Available: https://aclanthology.org/2022.lrec-1.30

[29] T. Kudo and J. Richardson, "SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, E. Blanco and W. Lu, Eds. Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 66–71. [Online]. Available: https://aclanthology.org/D18-2012

[30] A. Vaswani, N. shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, and L. Kaiser, "Attention is all you need," *NIPS*, 2017.

[31] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," 2016.

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[33] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *CoRR*, vol. abs/1910.13461, 2019. [Online]. Available: http://arxiv.org/abs/1910.13461

[34] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[35] B. Collie, J. Woodruff, and M. F. P. O'Boyle, "Modeling black-box components with probabilistic synthesis," in *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–14. [Online]. Available: https://doi.org/10.1145/3425898.3426952

[36] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[37] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross,

N. Ng, D. Grangier, and M. Auli, "fairseq: A fast, extensible toolkit for sequence modeling," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 48–53. [Online]. Available: https://aclanthology.org/N19-4009

[38] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://aclanthology.org/2020.emnlp-demos.6

[39] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav, "Towards neural decompilation," *CoRR*, 2019.

[40] S. B. Yadavalli and A. Smith, "Raising binaries to llvm ir with mctoll (wip paper)," in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2019, pp. 213–218.

[41] K. Anand, M. Smithson, A. Kotha, and K. Elwazeer, "Decompilation to compiler high IR in a binary rewriter," *University of Maryland*, 2010.

[42] Z. Tan, Y. Chon, M. Kruse, J. Doerfert, Z. Xu, B. Homerding, S. Campanoni, and D. I. August, "Splendid: Supporting parallel llvm-ir enhanced natural decompilation for interactive development," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 679–693.

[43] C. Fu, H. Chen, H. Liu, X. Cheng, and Y. Tian, "Coda: An end-to-end neural program decompiler," *NeurIPs*, 2019.

[44] Y. Cao, R. Liang, K. Chen, and P. Hu, "Boosting neural networks to decompile optimized binaries," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 508–518.

[45] E. Schulte, J. Ruchti, M. Noonan, D. Ciarletta, and A. Loginov, "Evolving exact decompilation," in *Workshop on Binary Analysis Research (BAR)*, 2018.

[46] Y. Gui, Y. Wan, H. Zhang, H. Huang, Y. Sui, G. Xu, Z. Shao, and H. Jin, "Cross-language binary-source code matching with intermediate representations," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2022, pp. 601–612. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SANER53432.2022.00077

[47] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, p. 3104–3112.

[48] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 2552–2562.

[49] M. Drissi, O. Watkins, A. Khant, V. Ojha, P. Sandoval, R. Segev, E. Weiner, and R. Keller, "Program language translation using a grammar-driven tree-to-tree model," 2018.

[50] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, and S. Fu, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," *CoRR*, 2021, available at https://arxiv.org/pdf/2102.04664.pdf.

[51] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," 2020.

[52] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *NeurIPs*, 2020.

[53] M. Artetxe, G. Labaka, and E. Agirre, "An effective approach to unsupervised machine translation," *CoRR*, vol. abs/1902.01313, 2019. [Online]. Available: http://arxiv.org/abs/1902.01313

[54] M. Lachaux, B. Rozière, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *CoRR*, vol. abs/2006.03511, 2020. [Online]. Available: https://arxiv.org/abs/2006.03511

[55] D. Pizzolotto and K. Inoue, "Identifying compiler and optimization level in binary code from multipler architectures," 2021.

[56] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, "Dire: A neural approach to decompiled identifier namings," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 628–639.

[57] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral, "Syntax and sensibility: Using language models to detect and correct syntax errors," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 311–322.

[58] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: Improving error reporting with language models," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 252–261.

[59] H. Hong, J. Zhang, Y. Zhang, Y. Wan, and Y. Sui, "Fix-filter-fix: Intuitively connect any models for effective bug fixing," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 3495–3504.

[60] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," *CoRR*,

2021.

[61] Y. Kang, Z. Wang, H. Zhang, J. Chen, and Y. Hanmo, "APIRecX: Cross-library API recommendation via pre-trained language model," *EMNLP*, 2021.

[62] S. Mandal, A. Chethan, V. Janfaza, S. Mahmud, T. A. Anderson, J. Turek, J. J. Tithi, and A. Muzahid, "Large language models based automatic synthesis of software specifications," *arXiv preprint arXiv:2304.09181*, 2023.

[63] Z. C. Guo and W. S. Moses, "Enabling transformers to understand low-level programs," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2022, pp. 1–9.

[64] J. Armengol-Estapé and M. O'Boyle, "Learning c to x86 translation: An experiment in neural compilation," in *Advances in Programming Languages and Neurosymbolic Systems Workshop*, 2021. [Online]. Available: https://openreview.net/forum?id=444ug_EYXet

[65] SLaDe authors, "Artifact for SLaDe: A Portable Small Language Model Decompiler for Optimized Assembler (CGO 24)," Nov. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.10205121