From Bytes to Types : Enhancing Type Prediction in Executables with Transformer-based Binary Analysis Tool SeeType

Raisul Arefin[1*], Ahmed Mostafa[2], Kevan Baker[3], Samuel Mulder[4]

[1*]Department of CSSE, Auburn University, 345 W Magnolia Ave, Auburn, 36849, Alabma, USA.
[2]Department of CSSE, Auburn University, 345 W Magnolia Ave, Auburn, 36849, Alabma, USA.
[3]Department of CSSE, Auburn University, 345 W Magnolia Ave, Auburn, 36849, Alabma, USA.
[4]Department of CSSE, Auburn University, 345 W Magnolia Ave, Auburn, 36849, Alabma, USA.

*Corresponding author(s). E-mail(s): ran0013@auburn.edu;
Contributing authors: aim0008@auburn.edu; kzb0154@auburn.edu; szm0211@auburn.edu;

Abstract

Purpose: Understanding binaries is a difficult task as the high-level abstraction used by the original programmers in source code is lost during compilation. The compiled binary code does not retain much of the structure and coherency that the programmer uses. In addition, reverse engineers frequently only have stripped binaries which omit critical information about the symbols and function names used in the source. The compilation process is lossy, making source-code variable type information recovery inherently uncertain. Our research involves regenerating the original type information using machine learning. This decreases the cognitive burden of reasoning about machine code and allows reverse engineers to focus on understanding the behavior of programs by providing additional contextual information.

Methods: In this paper, we present SeeType, a Transformer-based model capable of augmenting binary code with fine-grained source-level type information. SeeType learns to predict data types based on sequences of assembly instructions that interact with registers. SeeType is trained on a diverse dataset which we created and achieved better overall performance than Ghidra and StateFormer.

We also present new methods to preprocess the instructions so that the Natural Language Models tailored for human language can process them better.
Results: Our methods have shown significant improvement in predicting types. We have compared our tool SeeType with two State-of-the-Art (SOTA) methods, StateFormer [1] and Ghidra [2]. SeeType outperformed both of them.
Conclusion: SeeType is a progressive approach to train a Transformer-based binary analysis model. We have experimented with a set of decisions that effect the type prediction process and evaluated them to show their effectiveness. We also tried to solve some related existing limitations in type prediction which are under-explored in current literature.

Keywords: Reverse Engineering; Binary Program Understanding; Machine Learning, Type Inference

1 Introduction

Executable files are the product of design, coding, and compilation processes with each adding and removing information relative to the semantic level being addressed. The design of a program is primarily a form of communication to other humans about the semantic intent of a program and may include details and information that is helpful for other humans reading the design. The source code of the program is similarly a communication between a human programmer and the compiler about the specific implementation of that program, often discarding semantic information relative to the problem domain in favor of adding information relative to the program itself. The compiler thus transforms the human design representation into a binary file which ultimately contains only the information needed to execute the program on a specific platform, removing most of the semantic clues (e.g. type information, variable names) from the program in favor of information relative to the hardware semantics, (e.g. register selection, memory addressing). Binary code, without explicit data type annotations, poses a formidable barrier to understanding and analyzing software. Often, the source code of executables is not readily accessible, either due to proprietary constraints or other practical limitations.

Application. Binary type inference is crucial for binary program understanding. Understanding the types of data the instructions are manipulating helps to understand the semantics of the program. One of the primary applications of binary type inference is to assist [2–4] or improve [5–8] the decompilation process from binaries. Decompiled code helps reverse engineers by translating the instructions back into a more human-readable, high-level representation. Type information is also used for malware detection by identifying function signatures and data structures associated with known malware [9]. Besides, it is also crucial for binary rewriting [10], vulnerability prevention, code reuse, and integration of new code with existing binaries [10], and Control-Flow Integrity [11].

Problem Statement. Assembly instructions rely on generic registers and memory locations, which lack inherent type indications. This missing information makes it difficult to decipher the meaning of the executable's operations on a semantic level

understandable by human experts. For instance, in the case of the x86 architecture, if we are dealing with the register `EAX`, we may infer that the register has some data type that is 32-bits. Yet this would be a naive approach as there are several high-level C types such as `float`, `signed long int`, or `pointers` which share the same size [12]. However, if we look beyond that single instruction towards where the data in `EAX` came from and how it is used, we can achieve a more accurate estimation as to the data type. The primary problem we are trying to solve is source-level type information recovery from binary code. For our experiment, we are considering the prediction of C types from binaries compiled from C programs. We target the primitive (e.g. `int, float`), aggregate (e.g. `struct, array`), and `pointer` types. Our method attempts to predict any type, given sufficient examples of that type in our dataset. Our current dataset includes 52 different C types, and we measure the performance on the most common 44 of these as presented in Table 3. The input to our type prediction task is a sequence of instructions and a marker to the specific instruction we are interested in. This is a classification task where the types are the classes that the model needs to assign to a specific instruction in a program.

Prior Work. Most traditional Reverse Engineering (RE) tools use rule-based methods and type propagation to predict the types in binary programs. The RE tools keep a large collection of system calls and popular library function prototypes. Whenever a system call or a known library call is detected in the instructions, the types of the parameters are known. Afterward, this type information is propagated throughout the program using data flow analysis and more variable types can be resolved. Additionally, many tools have rules that are designed to find patterns in the code and connect those patterns with different types [2, 4, 13]. With the growing diversity of binaries, it can be overwhelming for rule-based methods to adapt [14]. Machine learning methods are able to take advantage of large quantities of data to develop pattern-matching capabilities [15, 16]. These models examine the context of the program with respect to register use and predict type information. The prediction is based on the knowledge these models learned from previously encountered examples.

Challenges and Our Solution. One of the barriers to training a Deep Neural Network (DNN) is that a substantial amount of labeled examples are necessary to make robust predictions. To address this, we gathered a large number of programs from open-source repositories (totaling 100.2k repositories, including 559k executable files). While training the models, we faced multiple problems that the previous type of predicting tools did not address well. For example, instructions have a lot of numbers, and natural language models are not good at processing them [17]. So we conducted experiments to find a better method to tune the model to process numbers. We plan to do an extensive experiment on this problem in our future work as this is not our primary problem.

Instructions are not like human language and do not follow a consecutive order. For example, two neighboring instructions can be unrelated and two distant instructions can be closest to each other in relation. If we process the instructions as they are located originally in the file, it would not be optimal. So, we used program slicing methods to formulate a method to calculate the context of an instruction and use that as the model input. This method helps us to process functions that have a lot of

instructions and some instructions need to be rejected from the model input. We also devised a new approach to pretrain our natural language model to adapt to machine language better.

Contributions. The contributions of our work can be summarized as follows:

(1) We developed a novel way of using source code type information to assist in labeling type information of machine instructions.

(2) We created a diverse and large corpus of compiled binaries and labeled them with type information to train a DNN model.

(3) We designed and evaluated a new tokenization technique to handle numerical values more effectively.

(4) Our model outperformed two SOTA tools, StateFormer [1] and Ghidra [2].

(5) We devised and evaluated a technique to analyze functions with an instruction count greater than the capability of the DNN model.

(6) We designed and evaluated a new pre-training technique that increases the performance of the DNN model.

We plan to release the code and datasets used for this work upon acceptance for publication.

2 Background

2.1 The Complexity of Executables

One of the biggest challenges in reverse engineering is trying to understand the semantics of a program when most of the abstractions created by the original programmer are lost in the compilation process. The most basic starting points for analyses of an executable are code discovery, Control Flow Graph (CFG) construction, and Data Flow Graph (DFG) construction. Each of these rely on and have interdependence with the recovery of a correct disassembly. There has been significant research on this task [18, 19], but the underlying problem of generating a correct and complete disassembly is undecidable [20] and modern tools rely on a variety of heuristics. Our goal is to recover the abstract source-level types from executables. The process will have some inherent noise, which we ignore. For example, we assume that the disassembly and DFG are correct. This is the case for our ground truth set, because of the generation process used, although it may not be the case for binary executables in general.

2.2 Classification with BERT

Our experiment involves analyzing a set of sequential instructions and classifying a target instruction into a source-level type. BERT (Bidirectional Encoder Representations from Transformers) [21] is a popular Transformer [22] model for text classification. The following steps describe how a Transformer classifies a sequence.

2.2.1 Input Encoding

The input sequence undergoes tokenization, breaking it down into smaller units like words or subwords, which in the case of machine instructions are the elementary parts of the instructions such as `mov, add, +, [` etc. Each token is subsequently

converted into an embedding vector, effectively capturing the semantic meaning of the token. These embeddings encapsulate the contextual information of each word within the text. An advantage of using encoding instead of actual tokens is that tokens with similar meanings should have similar embeddings which helps the model to generalize. We have trained our custom tokenizer so that the embeddings generated for instruction tokens work more effectively than the off-the-shelf BERT tokenizer trained with natural language data.

2.2.2 Positional Encoding

Along with token embeddings, BERT uses positional embeddings and segment embeddings for each token, which incorporate regional information into the input. Positional embeddings play a crucial role in Transformer-based architectures for preserving token order, as their absence would result in a bag-of-words representation [23]. This step is crucial as Transformers do not inherently understand the sequential order of tokens.

2.2.3 Self-attention

The attention mechanism, first introduced by Vaswani et.al. [22], helps Transformers focus on relevant parts of the input when making predictions. It assigns different weights to different parts of the input, indicating their importance. This allows the model to "pay attention" to the most relevant information while ignoring irrelevant details. This is vital for our task as some parts of the program are more important than others when predicting the type of information of a particular instruction.

BERT uses a self-attention mechanism. In simple terms, when BERT is analyzing a sentence, self-attention enables the model processing a word in the sentence to look at other words to know which words contribute to the current word. So, the model evaluates the sentence by looking at all the tokens to figure out how to represent each token.

2.2.4 Transformer Encoder

While the original transformer architecture has two main components, an encoder and a decoder, BERT only uses the encoder. The encoder has several repeated layers of self-attention and feedforward neural networks. The augmented encodings from the tokens are passed through the encoder layer. The output vector from the encoder then can be used for various downstream tasks such as classifying types.

2.2.5 Classification and Learning

The final output from the encoder obtained after processing the input sequence is commonly pooled or concatenated to create a fixed-size representation for the entire input. This aggregated representation is then fed into a classification layer, such as a softmax [24] layer, to predict the desired class label. The predicted labels are then compared with the ground truth labels and loss is calculated, representing how good or bad the prediction was. The loss is then backpropagated to update the model parameters.

3 Related Works

3.1 Conventional Approaches

Researchers and practitioners have developed two primary approaches to tackle the challenge of type inference from executables: static [25–27] and dynamic analysis [28–30]. Static analysis involves examining the binary's structure, instructions, and control flow without executing it. Various static analysis techniques, such as pattern recognition, hand-written heuristics, and symbolic execution [31–36], have been employed to deduce data types based on the assembly code's structure and characteristics. One of the limitations of this method is its dependence on the correctness of the disassembly. As static analysis techniques continue to advance, driven by innovations in machine learning, formal methods [37], and code analysis [38], their popularity continues to rise.

On the other hand, dynamic analysis involves executing the binary and observing its behavior during runtime. By monitoring memory accesses, register manipulations, and system calls, dynamic analysis provides insights into how data is manipulated, offering valuable context for type inference. One of the issues of this approach is that it only considers code that is executed during the analysis, and even combining multiple execution results coverage will likely be incomplete [39–41].

Lin et. al. developed Rewards [42], which resolves type information by looking for type-revealing executions, such as system calls. That information is then propagated throughout the program to resolve even more types. Slowinska et. al. developed another tool, Howard [40], which leverages the fact that memory access patterns can effectively be linked to data structure layouts. The tool identifies a structure by examining how the structure is accessed, e.g. an array is accessed how an array is accessed. Hallar et. al. developed MemPick [43], which detects data structures based upon the observation that the shape of a data structure reveals information about its type. These are mostly rule-based approaches. Laika [44], by Cozzie et. al., uses machine learning techniques to detect the data structure of a program given an image of the memory while the program is loaded.

One of the biggest weaknesses of dynamic analysis is its low coverage of code. Any code that is not executed is not going to be discovered by a dynamic analysis tool. Many tools perform executions multiple times to increase the coverage but for that good quality input is necessary. Besides, programs can hide their true intension if they detect they are bening executed in a simulated environment which will worsen the situation. Besides, compared to static analysis, there are significantly more resources needed for dynamic analysis. This is the reason most of the latest type inference tools use static analysis.

OSPREY [13] presented by Zhang et al. is a probabilistic variable and data structure recovery technique that is closely related to SeeType. They tried to address the uncertainty of variable information recovery from binaries using random variables and probabilistic constraints. They defined different hints that can help predict the variable information, such as data-flow hints that takes data-dependency into consideration. They also defined hints for memory access patterns that consider the fact that fields of the same data structures would be accessed in a unified manner. Another kind of

hint they came up with is points-to hint which relates two variables by the pointers they are pointed to. Afterwards, they construct probabilistic constraints where the predicates describe the structural and type properties of memory chunks, which are denoted by random variables. Then they solve these probabilistic constraints to predict the variable information. However, we did not consider OSPREY as a baseline as it is not open-sourced. However, as this is dependent on patterns, a lot of manual work is needed to keep it up to date with the ever-changing compilers and architectures. Machine learning methods mitigate this problem by learning the patterns automatically given enough samples.

Additional efforts have attempted to combine both static and dynamic analysis, such as TIE [12] and TypeSqeezer [45]. However, modern approaches have mostly relied on static methods applied to binary executables rather than dynamic methods. Some popular general-purpose reverse engineering tools that have type recovery features are Ghidra [2], Ida Pro [4], Angr [31], Binary Ninja [36], and Radre2 [35]. These tools mostly rely on the type information propagation method where they propagate the known types from system calls and known library function calls. This method is often very precise, but its coverage is low as not all variable types can be resolved through this process.

3.2 Machine Learning Approaches

The use of machine learning techniques has shown promising results in the automatic recovery of data types. This application of machine learning involves training models on datasets comprised of binary code samples alongside their corresponding type annotations. The models learn relationships between the binary code's features, such as opcode sequences, control flow patterns, memory access patterns, and the associated data types. Once trained, these models can effectively predict data types in previously unseen binary code.

Academic research has recently focused on recovering type information from C binaries through the application of machine learning technologies. Notable contributions have been made by early studies such as EKLAVYA [46], DEBIN [5], TypeMiner [47], BITY [48], SnowWhite [49], and the work of Escalada et al. [8]. These projects predominantly rely on relatively simple classification or probabilistic machine learning algorithms. Additionally, the volume of data employed in their experiments is relatively limited, which may be insufficient for training more sophisticated models, such as large-scale language models. Furthermore, CATI [50] used statistical analyses and a multi-stage classifier consisting of a set of Convolutional Neural Networks (CNN) for predicting types.

One of the earliest uses of sequence-to-sequence networks to predict type information from binaries was EKLAVYA [46]. Another closely related project is StateFormer [1]. StateFormer uses pre-training on Generative State Modeling (GSM), where they train the model to predict execution characteristics of the binary. Then they fine-tune the pre-trained model to predict the type information. TypeMiner [47] does similar pre-training on dependence analysis and trace execution to provide the model with more contextual information. Additionally, SnowWhite [49] uses bidirectional LSTM [51] with global attention as the machine learning model to predict types.

It is worth mentioning that while DEBIN, for instance, does not directly provide type information, it predicts DWARF [1] information, which is rich in type-related data. SnowWhite [49] also takes a similar approach. SnowWhite works on WebAssemblies which allow execution of code written in languages other than JavaScript on the browser at a near-native speed [52]. Programs written in various languages such as C/CPP, C#, GO, or Rust can be compiled into WebAssemblies. So, instead of predicting different types for different languages, they predict types in a way presented in DWARF.

Furthermore, research has been conducted on enhancing the decompiled code generated from binary analysis tools using Transformer models. For instance, VARBERT [53] improves the decompiled source code variable name generated by reverse engineering tools. Another closely related work is DIRTY (DecompIled variable ReTYper) [7], which improves on the type information of the decompiled source code. One major difference between DIRTY and SeeType is that DIRTY uses decompilation as the input. One limitation of this work is the accuracy of the tool will be dependent on the accuracy of the decompilers, as the decompilers try to recover the type information themselves. If there is any error in the decompiled code, there is no way for DIRTY to fix that.

Besides, the current LLM methods did not present any experiments to show how they selected their tokenization method, or how they handled longer functions which exceeds the input limit of the models. Tokenization is one of the primary task every machine learning methods using LLMs need to perform. How you perform tokenization can effect the performance of the LLMs significantly. Longer functions are common and must be handled with care. Our experiments shed light on these problems and we conducted experiments to show how our selection of methodology is better than the current tools. Cati [50], BITY [48], and Stride [6] use some notion of context that can handle longer functions. But those methods are trivial compared to our method which takes advantage of a well-established program slicing method.

A further challenge is that deep learning models often struggle in explaining their decisions and behaviors [54]. Hopefully, our experiment will shed light on how a Natural Language Processing (NLP) model designed for natural human languages can learn the structure of binary programs.

4 SeeType

We treat our type prediction problem as a classification problem and use BERT, a Transformer machine learning model, as a tool to approach the problem. We first preprocess the disassembled instructions before feeding them to BERT. In the case of large functions with an instruction count higher than the maximum limit of BERT, we find the partial part of the function relevant to the target instruction using data dependency analysis and program slicing. We explain the program slicing process in detail in Section 6.1. We then train a tokenizer suitable for tokenizing machine instructions. Tokenizing instructions differs from tokenizing natural languages and we discuss our approach in Section 8. Next, we pre-train the BERT model with Masked

[1] www.dwarfstd.org

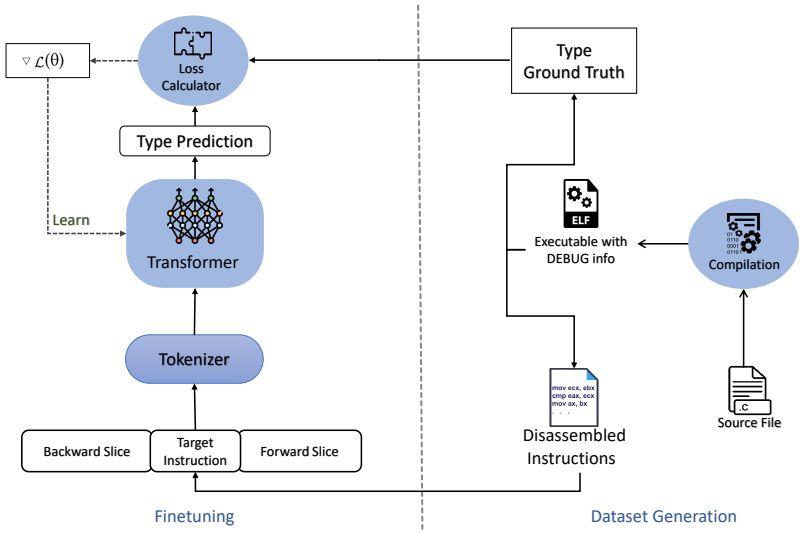


Fig. 1: System design of SeeType and Dataset pre-processing.

Language Modeling and Data Dependency Modeling to make the BERT model more suitable for machine instructions, as described in 7. After pre-training, we transfer the learned knowledge during pre-training for our desired task by fine-tuning the BERT model on the type prediction task, discussed in 9. The fine-tuning process is shown in Figure 1. The right half of the figure shows how an executable is compiled from a source file with debug information. The executable is then disassembled which is used for generating the input of the model. At the same time, Ground truth is also generated with the help of debugging information which is used to train and validate the model. The left side of the figure shows how SeeType works. Initially, the input sequence is produced from the disassembly. Only the relevant instructions to the target instructions are included in the input in case the original function disassembly is larger than the maximum length allowed for the model. Then the input sequence is tokenized and fed to the transformer model. Analyzing the input, the model makes a prediction about the type which is then compared with the ground truth type. During training, the model parameters are updated based on the correctness of the prediction. After the training is done, the correctness of the model is evaluated on a test dataset.

5 Challanges Of Building A Type Data Set

Training a large language Transformer model requires much more labeled data than other traditional approaches. To generate this data, we created a fully automatic process that compiled a multitude of programs and labeled the disassembled instructions of the programs with their respective types. The challenges of making this kind of dataset are discussed below.

5.1 Magnitude and Diversity

A large dataset with diverse examples mirrors the various scenarios the model may face in real-world applications. This facilitates improved generalization, enhancing the model's robustness and capacity to handle novel and unseen data. The dataset needs to cover a wide range of program structures, functions, logic, implementation styles, and compiler optimizations to enable the model to learn robust representations and patterns.

We searched GitHub for suitable open-source code repositories that are mostly written in the C programming language. We found 100.2k repositories and cloned them. We then compiled all of the C files to build their respective 559k executables. During the compilation process, we turned on the debug information generation mode to create the ground truth set.

Figure 2 shows the frequency of ground truth samples recovered for types in our dataset and Figure 1 shows the dataset generation process.

5.2 Challenges With Variations of Optimization Level

The selection of an optimization level, spanning from minimal optimization (e.g. 'O0') to aggressive optimization (e.g. 'O3'), significantly shapes the characteristics of the resultant executables. At lower optimization levels, the compiler maintains the original structure of the source code as much as possible, resulting in larger, slower executables. Higher optimization levels employ advanced optimization techniques, such as loop unrolling, function inlining, and constant propagation, yielding more compact and faster code, and critically, using different instruction idioms to produce the final assembly. Recognizing the impact of these differences is crucial for any model analyzing binary programs. In our dataset creation process, each program was compiled with several optimization levels from the same source file.

We could collect fewer samples from higher optimization-level binaries. This is because it gets harder to label types for highly optimized binaries, even with DWARF information, as instructions no longer maintain as close a relationship to the original source language.

5.3 Comprehensive Type Coverage

The C programming language supports a wide range of data types, ranging from commonly used ones like `int` and `float` to more specialized types such as `long long int`, `long double`, `pointer` and `structure`. Despite their varying degrees of prevalence, each type holds significance in the semantics and other properties of a program. We have generated and trained our model on 41, 39, 50, and 39 fine-grained types for O0, O1, O2, and O3 respectively. We differentiate array types, which have not been explored by previous research [1, 47]. We have ignored some types such as `Union` and `Enum`. Enums, which are symbolic names for integer values, are replaced by their corresponding integer values during compilation. Unions allow different data types to share the same memory location, and may not leave a recognizable footprint in the

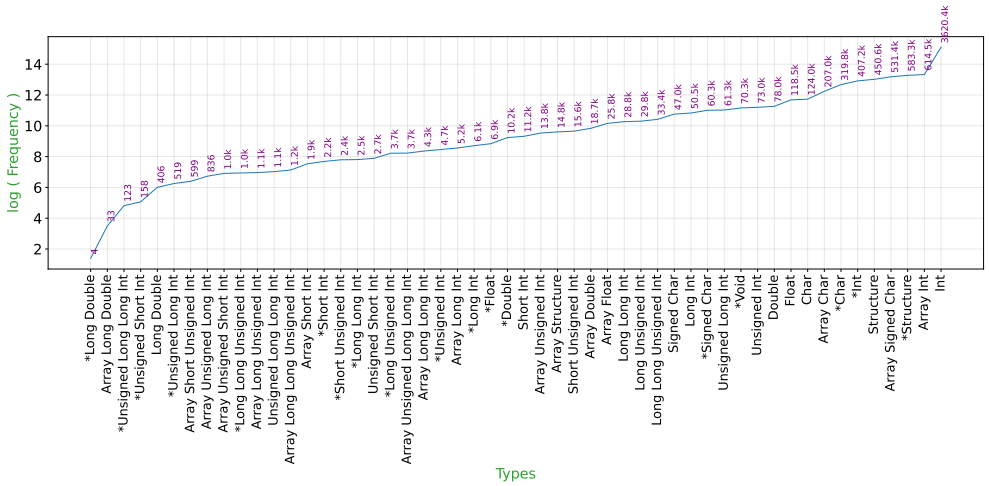


Fig. 2: Frequencies of ground truth samples of different C types extracted from compiled binaries. The numbers on the curve represent the average number of samples per optimization level of that particular C type in our dataset.

compiled instructions, as the compiler resolves their implementation during the compilation process. Both of these types act more as hints to the compiler, rather than fundamental types that are represented in the compiled binary.

The ground truth generation process also yielded different numbers of samples from different binaries from the same source but different optimization levels. Some type samples were simply absent in data generated for one optimization level but were present in another. So, as we are saying we are classifying more than 40 classes, the set of classes varies for different optimization-level experiments. In Figure 2 the Frequency of the samples per type represents the average frequency of the samples of 52 types. SeeType trained on fewer types because some types had insufficient samples for training and testing.

5.4 Challenges With Heterogeneity in Binaries

The binary output from the compilation of a specific C program can vary widely based on factors such as different compilers, compilation parameters, and architectures. Each compiler uses its custom optimization process and idioms to produce output assembly, producing very different binaries. Compiling for different instruction set architectures radically changes the binary code, with different underlying types, registers, and operations. For this experiment, we did not explore these differences, but as our dataset generation process is automatic and we have all the source code, we can generate binaries with all these variations for further experiments.

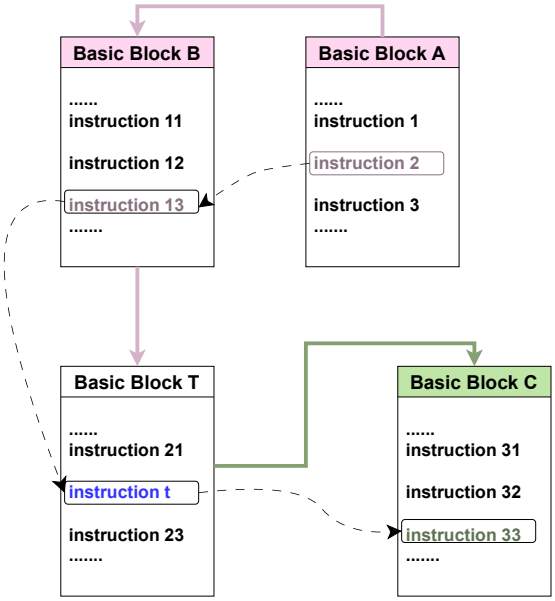
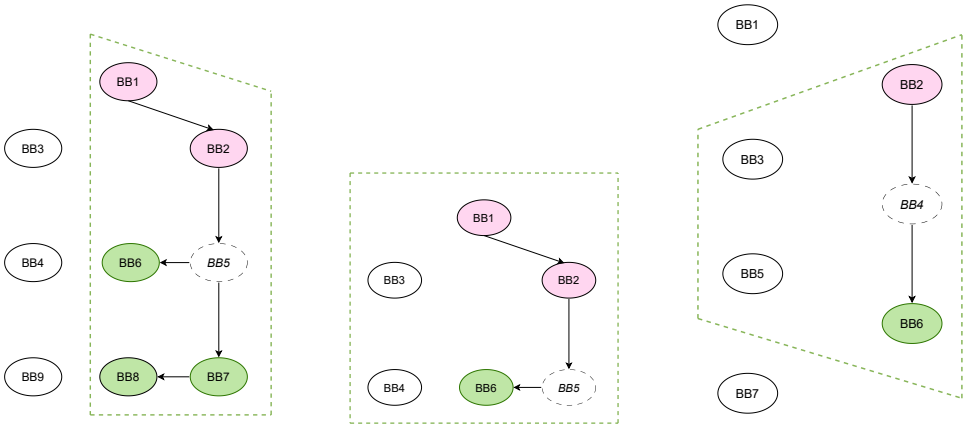


Fig. 3: Basic Block T is dependent on Basic Block B because target instruction t in Basic Block T is dependent on instruction 13 from Basic Block B. Additionally, Basic Block A is dependent on Basic Block B because of the dependency relation between instruction 13 and instruction 2 from Basic Block B and A respectively. Similarly, Basic Block C is dependent on Basic Block T because instruction 33 from Basic Block C depends on the target instruction t from Basic Block T.

6 Challenges of Longer Functions

There are certain challenges Transformer models face when analyzing binary programs with large functions due to their inherent complexity. Longer functions mean more instructions and more tokens the Transformer has to process. In addition, the control flow and data flow complexity often scale with function length. The full-attention mechanism in Transformers like BERT allows them to consider all positions in a sequence simultaneously, but this becomes computationally intensive for long sequences, leading to increased memory requirements and processing time [55]. Transformers also often struggle to capture long-range dependencies [56]. There has been significant research on how to handle longer sequences in NLP models such as GPT-2 [57], and Pegasus [58].

Type recovering tools such as BITY [48] and CATI [50] tried to utilize the context of the target instruction, instead of the whole function. BITY analyzed the dependency relation among instructions and selected a few instructions directly related to the target instruction as the input for their tool. However, they encode the instructions as a vector where the order of the instructions is lost. CATI used machine



(a) (b) (c)

Fig. 4: Three examples of selecting program context using program slicing. The nodes in the graph represent Basic Blocks (BB). The enclosed areas by the dashed lines represent the final model input. The green nodes are in forward slices and the pink nodes are in backward slices. In Figure (a), the target instruction is in BB5 and the context contains the backward and forward nodes. As the context size equals the model input length, the context is set to be the final input. In Figure (b), the function is small so the whole function is selected as the model input. In Figure (c), BB4 contains the target instruction. BB2 and BB6 are the only two nodes in the backward and forward slices respectively. However, as there is space left in the input, the BB3 and BB5 are included in the input because of their proximity to the target instruction.

learning techniques for type prediction. They also tried to capture the context of the target instruction. They only considered the backward and forward 10 instructions to the target instruction as the context without considering the dependency relationship among instructions. Both of those efforts were for capturing the context of the target instruction, rather than an attempt to analyze large functions for type recovery. Besides Stride [6] uses n-grams before and after the target instruction for capturing the context. Tiara [59] uses only the forward slice for context.

6.1 Our Solution Using Program-Slicing

To avoid this limitation with large functions, previous related research that processes machine instructions with NLP models, such as StateFormer [1] discarded the longer functions that generated more instructions than the model can handle. We think this is an interesting area that deserves more attention as larger functions are very common. Despite BERT having a smaller capacity for sequence length handling, we decided to use it due to its performance and prevalence in the literature. This is acceptable because there will always be functions larger than the maximum limit of the largest model. Rather than trying to handle entire large functions, we decided to

feed the Transformer only the portion of the function that is relevant for a particular instruction, the context. To do this, we used the idea of program-slicing [60]. This allowed us to handle both long sequences and distant dependencies. Program slicing is a disintegration process that discards parts of the program that are irrelevant to a particular computation process based on a criterion known as the slicing criterion. In our case, the criterion is the target instruction that we are trying to predict the type of. For longer functions, we try to discard the instructions irrelevant to the target instruction we are analyzing. Ideally, only the related instructions are included in the final input sequence, allowing us to handle functions of all sizes.

Program Slicing [61, 62]. Previous research such as [63, 64] used binary program slicing for analyzing binary programs. For our method, we are interested in both forward slicing and backward slicing. The backward slice contains all the instructions that affect the target instructions in any way. The forward slice includes all the instructions that are affected by the target instruction. Ideally, the final context for our problem should include all the instructions on which the target instruction depends on and all the instructions that depend on the target instruction.

Context Generation. For producing the context, we utilized the directed dataflow graph of a function. The nodes in the dataflow graph are instructions from the function and the edges are the dependency relation between the nodes. The nodes connected to the target instruction constitute the primary context, which includes both forward and backward slices.

We wanted to preserve the basic block structures of the code in our model input. This means if an instruction is chosen to be in the context, we include the whole basic block containing that instruction. The reason behind this is there might be instructions in a basic block that do not have any dependency relation with the target instruction but might contain clues for the pattern-recognizing machine learning model. According to this scheme, the context should have all the instructions that have a dependency relation with the target instruction along with the complete basic blocks they are located. To achieve that, we generated another dependency graph where the basic blocks are the nodes. The nodes are connected with directed edges which are produced from the original dependency graph. We presented this procedure in Figure 3 with an example. The connected basic blocks in the dependency graph are the final context of the target instruction.

Finalizing Model Input. The maximum model input length is fixed. But, the length of the context can vary and we want to put as many instructions as possible in the input. There can be four cases. **Case 1.** The context can be equal to our model input. If we have that perfect size, we select all the instructions from those basic blocks in our final model input. See Figure 4 (a) for an example. **Case 2.** If all the instructions of the function fit our model input, then we do not need to do any analysis and the whole function can be used as the context, like in Figure 4 (b). **Case 3.** In case the context is smaller than the input limit, we add surplus unrelated basic blocks to the context given that the context length stays less or equal to the model input. To select unrelated basic blocks to include, we find the nearest basic blocks to the target instruction. An example of this can be found in Figure 4 (c). **Case 4.** In rare cases where the context size is greater than the maximum input limit, we follow

a simple approach of selecting basic blocks from the context that are nearest to the target instruction.

7 Challenges of Using Models Specialised for Natural Languages with Assembly Language

LLM models are designed to work with natural languages such as English, so we devised our own pre-training method to make the model more optimized for assembly language rather than natural languages. Self-supervised pre-training has proven extremely valuable in preparing NLP Transformer models for a variety of tasks [65]. Using the inherent bidirectional context modeling in a Transformer architecture and providing the model an initial phase of unsupervised learning on a large corpus of unlabeled data allows the model to learn contextual embeddings. These embeddings encapsulate latent semantic structure, syntactic relationships, and nuanced patterns. When combined with fine-tuning task-specific labeled data, the model exhibits greatly improved performance.

Previous research on StateFormer [1] and PalmTree [66] and work by Ahn et. al. [67] have leveraged self-supervised training with binary instructions. For our task, we pre-train our model with two tasks, Masked Language Modeling (MLM) and Data Dependency Relation Modeling.

7.1 Pre-training SeeType with Masked Language Modeling

MLM, depicted in Figure 5, is frequently used for pre-training Transformers, where a subset of tokens in a given sentence is randomly masked, and the model is trained to predict the masked tokens based on the surrounding context. To predict the masked tokens, the model has to learn the syntax and semantic relationship between tokens within a relatively shorter range. In the example presented, we can see that one of the tokens from the instruction is masked. The model needs to predict the correct token which is `ptr`.

7.2 Pre-training SeeType with Data Dependency Modeling

The input of the model during training on data dependency is a sequence of instructions from a function with two particular instructions marked. The model predicts if there is any data dependency, direct or indirect, between that pair of instructions. Correctly predicting the data dependency of a pair of instructions within a function requires a deep understanding of the long-range dependencies and relationships among the instructions. Figure 6 shows a positive and negative example of data dependency modeling training data. In the positive example on the left, we can see that the instruction on `0x1201` uses the value of register `RAX` which was loaded by the instruction at `0x11fa`. We also find that the value loaded into the register `RAX` depends on the previous instruction at `0x11f6`. Thus the instruction at `0x1201` depends on `0x11f6`. To understand this kind of relationship the model needs to understand the operation of instructions.

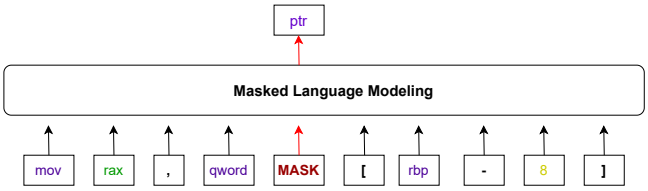


Fig. 5: Masked Language Modeling

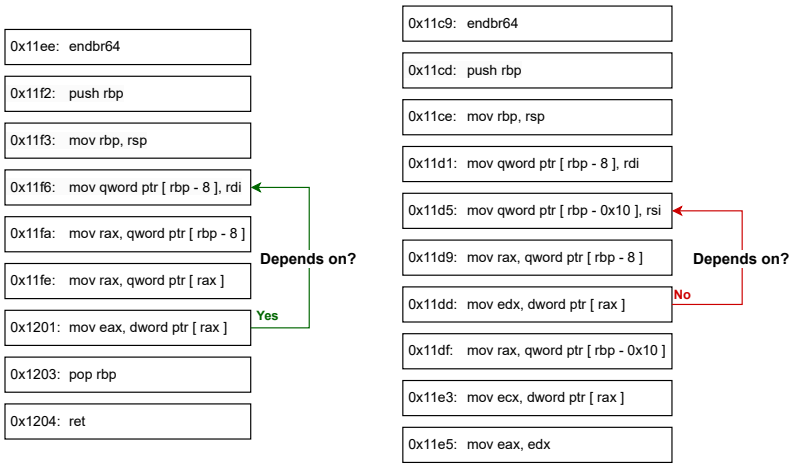


Fig. 6: Examples of data dependency samples. The arrows indicate the data dependency relation between a pair of instructions.

8 Challenges of Tokenizing the Instructions

Recent ML-based systems such as TypeMiner [47] eliminate addresses and other numerical values from the instructions before tokenization. Transformers tailored for NLP tasks are not the best models for handling numerical values [17]. StateFormer [1] tries to mitigate the limitation by using Neural Arithmetic Units (NAU) [68].

For our application, however, the addresses and other numerical values are valuable because they often carry information about the control flow and the storage location of source-level variables [69]. They indicate relationships between instructions and are necessary for reconstructing the control flow of the program. Therefore, we worked to find a system for handling addresses and other numerical values. However, the range of possible numerical values which includes addresses, offsets, and data is vast.

Our intuition was that, with a large enough dataset, the model may learn some relationships between instructions based on addressing. We set the vocabulary which is the maximum unique token count to 3000. This is much lower than the maximum allowed for a large language model. For instance, the BERT-Base-Uncased model has a vocabulary size of 30522 unique tokens [70]. We made this decision because the small numbers and file offsets are frequently repeated in the dataset. For example,

small numbers from 0 to 100 are seen numerous times as those numbers are often used as offsets. Offsets are very important for type prediction because they can be used as clues for the size of variables which are indicative of their types. For instance, if a variable takes 2 bytes we can say that variable can't be a `long`. Similarly, if a variable takes only 1 byte, that is a hint that that variable might be a `char`. The size of a variable can be recovered from the offsets used to access the variable. Then the other kind of very frequent numbers are the addresses. Addresses are important for retaining the control flow of the program. Even though these addresses might seem very random, they are not. Usually, programs are addressed with a virtual address, and the range of addresses the programs can span is not that vast. For example, if the base virtual address for all programs is set to 10000, addresses close to that will be seen repeated among all the programs. So the addresses close to that base address will be more frequent compared to some random integer like 3000000. The vocabulary is generated in such a way that it will retain the most frequently seen addresses and will discard the rarely seen numerical values. So, setting the vocabulary size low would force the tokenizer to keep all the usual tokens from the assembly language and the most frequently seen numerals like offsets and addresses and discard any outliers.

The problem of machine instruction tokenization for LLMs is often not addressed well in the current literature. Even though analyzing machine instructions using neural models [1, 5, 46, 66] is becoming more popular, we believe the tokenization problem should receive more attention.

9 Challenges of Fine-tuning for Type Prediction

We transfer the knowledge learned during pre-training to the final model. Pre-training gives a warm-start to the fine-tuning process and helps improve accuracy in the long run [71]. We faced two major challenges during fine-tuning. As shown in Figure 2, type data is unbalanced, which causes the model to become biased. The model might see some dominant types excessively and predict everything it sees as those types giving a higher accuracy score overall at the expense of consistently misidentifying rare types.

Undersampling is a technique employed in the training of models on unbalanced datasets. Undersampling involves randomly reducing the frequency of the dominant class to balance the class distribution [72]. This helps the model to pay proportional attention to each class during training, preventing it from being dominated by the more prevalent class [73]. Undersampling further helps to improve the generalization of the model across all classes, ensuring that it learns meaningful patterns from both frequent and rare classes. We fixed a maximum limit of 20k samples a single type can have in the training data. Extra samples were rejected randomly so that we have all the variations present in our training set.

Another limitation of our dataset is that it is not complete. We used a combination of different tools for labeling the types of as many instructions as possible, but there is still a large portion of instructions without labels. This limitation is also present in all previous datasets presented by other groups because there is simply no way to get ground truth for all instructions in a binary. In many cases, this is because the register usage has no type equivalent in the source language. This is often the case

for compiler generated idioms around managing the stack or address calculations. In other cases, it may be a limitation of the tools being used. For this experiment, we are electing to omit predicting the unlabeled instructions. All the scores presented are for predicting the types of only those instructions for which we have the ground truth.

The model receives a series of instructions as input. An efficient method would be to predict all the types of instructions in a single pass as StateFormer [1] did. While this method is fast, it has some serious drawbacks. One is that we do not have labels for all instructions. If the model predicts all the types for each instruction given to it, what about its prediction for the instructions that do not have ground truth? StateFormer tackles this issue by ignoring the prediction for those instructions in calculating evaluation metrics. During the training process, StateFormer will keep predicting the types for those unlabeled instructions and it would have no way to calculate loss for those instructions and learn from mistakes.

Another major problem with this approach is that there is no way to handle the imbalance in the dataset. For example, given the instruction set of a function as input for the model, and the model trying to predict types for all the instructions, there is no way to ignore some instructions for downsampling. Using the concept of program slicing would also not be possible. Due to these limitations, we have taken the approach of predicting types for one instruction at a time. This process is slower but can handle the above mentioned problems better, giving a more generalized and better performance.

10 Implementation Details

10.1 Ground Truth Generation

We used a wide variety of tools for the experiment. For dataset generation, we used IDA Pro for disassembly, Ghidra, and pyelftools [74] for ground truth extraction, and Miasm [75] for calculating forward and backward slices. For generating the data dependency graph of a program we used Miasm. We used a 32-core Intel i9-13900k processor with 64 GB of RAM for processing. The repositories we downloaded took around 500 GBs of disk space. The compiled executables with all the optimization level variations and analysis temp files created by IDA Pro took another 1 TB of space. Processing those files with Ghidra using our 32-core machine took around 70 hours and for IDA Pro around 20 hours.

10.1.1 Ground Truth Labeling Method

This step includes aligning the disassembled instructions with the source information with the help of DWARF information and tagging the content of the registers in the instructions with their respective type information. We used Ghidra's type extraction feature which reads the debug information and correlates types for instructions. However, Ghidra cannot extract all labels as this is a complex task even with the DWARF information. Since we have access to the original C source, which Ghidra does not use, we wanted to develop a custom tool that uses both DWARF and source information to label type information in instructions.

The resulting process for labeling types per instruction was fairly complex. The first step was to read the binary and disassemble it. Any modern reverse-engineering tool such as Ghidra or IDA Pro can be used for disassembly. Then we used pyelftools to parse the DWARF information embedded in the binaries during compilation.

```
1  0x1149: endbr64                                  # {
2  0x114d: push rbp
3  0x114e: mov rbp, rsp
4  0x1151: sub rsp, 0x10
5  0x1155: mov dword ptr [rbp - 8], 1                #  unsigned |i| = 1;
6  TYPE:>  [ unsigned int ]
7
8  0x115c: mov dword ptr [rbp - 4], 0xfffffffc      #  int j = -4;
9  TYPE:>  [ int ]
10
11 0x1163: mov edx, dword ptr [rbp - 4]             #  printf("%d", i + j); //prints -3
12 TYPE:>  [ int ]
13
14 0x1166: mov eax, dword ptr [rbp - 8]
15 TYPE:>  [ unsigned int ]
16
17 0x1169: add eax, edx
18 0x116b: mov esi, eax
19 0x116d: lea rax, [rip + 0xe90]
20 0x1174: mov rdi, rax
21 0x1177: mov eax, 0
22 0x117c: call 0x1050
23 0x1181: mov eax, 0                               #  return |0|;
24 0x1186: leave                                    # }
25 0x1187: ret
26
```

Fig. 7: Illustration of instruction and source code mapping using DWARF line information and how we assigned C types to instructions from their source code information.

10.1.2 Aligning Source Code and Machine Code with Line Information

DWARF line information provides essential details about the correlation between machine code instructions and the original source code. Embedded within binary executables compiled with required parameters, this information includes mappings between memory addresses and source code lines, file names, and function names. Using this, we were able to align machine instructions with the corresponding lines of source code. One limitation of this method is that line information is only found for some instructions, as shown in Figure 7. To fill in more detail, we used the heuristic that all the instructions between two instructions with line info are related to the same source as the prior instruction. For example, in Figure 7, the instruction in line 1 is mapped with the beginning curly brace, which implies the start of a function. The next three instructions have no mapping information, but we include them with the start of the function. In fact, these instructions are setting up the function stack

frame, which is not directly correlated with anything in the source code other than the beginning of a function.

Debuggers use DWARF information to dynamically map memory locations to variable names and their corresponding types during runtime. By associating memory addresses with specific variables and their data types, debuggers can provide real-time insights into variable values, enabling developers to inspect and monitor program state during execution. We also use that information to correlate source-level variables with the specific instructions. When we see an instruction accessing a location where a source-level variable is saved, we can associate the instruction with the variable. We then tagged the instruction with the appropriate source-level type.

10.2 Pre-training and Fine-tuning

For implementing SeeType with BERT, we used Huggingface's Transformer library [76]. For training and testing the model during different steps, we used an NVIDIA 4090 GPU with 24 GB of G6X memory which took around 30 hours.

10.3 Hyperparameters

During training the batch size was set to 32. We initially found that the validation score was far below the training score and we decided to implement a decaying learning rate which gave us a better validation score compared to the training score. The optimizer we used was AdamOptimizer [77]. We split the dataset 80%-20% for training and validation. Cross Entropy loss was used for the fine-tuning task.

10.4 Evaluation Metrices

We selected our evaluation metrics to measure our model's performance and compare it with the baseline method used by StateFormer and Ghidra. Key measures including F1-Score, Recall, and Precision are crucial for classification tasks. F1-Score, a balance of recall and precision, addresses trade-offs between false positives and negatives. Recall gauges the model's ability to identify instances of a class and is crucial when missing positives have significant consequences. Precision emphasizes the accuracy of positive predictions. Together, these metrics provide a nuanced understanding, guiding decisions for optimizing classification accuracy.

11 Results

In this section, we aim to answer the following research questions.

- **RQ1:** How accurate is our method compared to the State-of-the-Art methods?
- **RQ2:** Does pre-training SeeType improve performance?
- **RQ3:** How does our approach to handling long functions perform?
- **RQ4:** How does our tokenization method perform compared to other methods?

Tool	Compiler Optimization	Accuracy	Precision	Recall	F1
StateFormer	O0	93.9	87.8	87.6	87.7
Ghidra	O0	87	93	87	89
SeeType	O0	**96.4**	**96.4**	**96.4**	**96.4**
StateFormer	O1	**97**	79.1	85.1	82.0
Ghidra	O1	84	91	84	86
SeeType	O1	96	**95.5**	**95.5**	**95.5**
StateFormer	O2	**97**	81.8	84.1	82.8
Ghidra	O2	83	90.2	83	85
SeeType	O2	94	**93.5**	**93.5**	**93.5**
StateFormer	O3	**97**	81.6	83.4	82.3
Ghidra	O3	82.3	90.0	82.3	84.2
SeeType	O3	93.6	**93.6**	**93.6**	**93.5**

Table 1: Accuracy, precision, recall, and F1-scores obtained after fine-tuning SeeType and StateFormer with various compiler-optimized binaries. We also present the performance of Ghidra on the same dataset.

11.1 RQ1: Correctness of Prediction

Creation of Baselines. We trained StateFormer with our dataset to use as a baseline. Initially, we pre-trained StateFormer with our dataset and then finetuned it with the respective optimization level we are going to compare it with. So, all the evaluation scores presented for StateFormer in this paper are based on training and testing on the same dataset SeeType was trained and tested on. For pre-training and fine-tuning StateFormer, we used the code and other materials provided by StateFormer through github [78]. As for evaluating Ghidra on our dataset, we made Ghidra predict the type information without the debug information. To do that, we stripped each binary and generated the decompilation. We obtained the type prediction of Ghidra from the decompilation it generated.

Comparison with StateFormer. Table 1, Table 3, and Figure 8 present the performance of SeeType and StateFormer on datasets created with binaries compiled with various levels of optimization passes. Our methods outperform the baseline method in the measure of precision, recall, and F1 measures. While StateFormer has higher accuracy, the F1-score represents a better picture of the performance of classification models over an imbalanced dataset.

SeeType has outperformed the baseline due to the improved pre-training and training methods. Notably, as we can see in Table 3 and Figure 8, SeeType performs well for the majority of the types, irrespective of their sample frequency, as stated in Figure 2. For example for the type `*long int`, which has a comparatively lower frequency in the dataset, our model is able to outperform the baseline on Table 3 and Figure 8. The F1-score of the baseline method on different types depends on the frequency of

Method	Accuracy	Precision	Recall	F1
SeeType (No Pre-training)	95.2	95.2	95.2	95.2
SeeType	**96.4**	**96.4**	**96.4**	**96.4**

Table 2: Accuracy, precision, recall, and F1-scores obtained for O0 binaries during fine-tuning without and with pre-training.

that particular type in the training dataset. This shows that SeeType handles data imbalance better by using undersampling during training.

Comparison with Ghidra. Ghidra is one of the most popular reverse engineering tools. We can see that Ghidra performed better than StateFormer in all the metrics other than accuracy, like SeeType. SeeType performed better than Ghidra in all the metrics in Table 1. In the more fine-grained comparison in Figure 8, we can see that Ghidra is performing better than SeeType only for the `int` and `char` classes. For, all the other classes, SeeType is performing better than Ghidra. This shows the robustness of SeeType over all the different types.

11.2 RQ2: Impact of Pre-training

We experimented with omitting pre-training. Instead of initiating fine-tuning from the pre-trained model, we started from a vanilla BERT base model. The experiment showed that the model that started fine-tuning from the pre-trained model outperformed its counterpart by 1.4% in the overall F1-score as reported in Table 2.

11.3 RQ3: Performance of our technique on Handling Longer Functions

For evaluating our approach to handling larger functions, we created a different training set, as the original dataset has functions both small and large. A naive approach for handling larger functions would be to take as many nearest instructions as possible, akin to a sliding window where the target instruction sits in the middle. Here we are defining proximity as the file offset of the instructions. This approach has some merit as the maximum number of instructions we were taking as input was 48, and there is a good chance that those 48 instructions contain enough context for the model to make a good inference. To evaluate our approach, we created another dataset that consists of functions only with more than 48 instructions. Then we discarded any sample with less than 10 instructions in their Symmetric difference of the instructions sets obtained by both methods. We found that our method resulted in a better F1-score compared to the naive approach as reported in Table 4. This experiment shows us that there are multiple ways to handle larger functions for feeding into NLP models and this might be interesting future work.

Optimization	Array Long Long Unsigned Int	Long Int	Array Long Int	Unsigned Long Long Int	Array Structure	Double	*Void	*Char	Signed Char	Array Char	*Signed Char	Long Unsigned Int	Long Double	*Int	Long Long Unsigned Int	*Float	Array Long Unsigned Int	*Long Unsigned Int	Structure	Array Signed Char	Array Short Unsigned Int	*Short Int
O0	97	93	97	-	96	99	-	94	98	95	100	96	80	96	98	98	96	98	92	99	96	99
O1	-	94	96	91	-	98	99	96	97	98	93	-	98	96	-	87	-	-	88	90	-	-
O2	-	90	96	87	-	98	99	92	96	97	89	-	83	93	-	80	-	-	83	87	-	98
O3	40	90	95	84	88	98	99	90	91	82	92	85	90	94	88	82	81	0	82	81	87	93

Optimization	*Long Int	*Unsigned Int	Unsigned Short Int	Array Unsigned Int	*Double	Array Float	Array Unsigned Long Int	Short Unsigned Int	Array Unsigned Long Long Int	*Structure	Array Short Int	*Long Long Int	Array Long Long Int	Unsigned Int	Array Int	Long Long Int	Short Int	Int	Unsigned Long Int	Char	Array Double	Float
O0	99	97	-	95	99	97	-	99	-	92	95	96	95	96	95	95	99	94	-	99	97	99
O1	88	96	99	98	91	98	98	-	100	95	99	90	96	96	93	93	97	91	97	98	99	98
O2	81	94	93	96	85	97	92	-	97	93	95	97	94	95	88	90	95	87	97	96	98	97
O3	72	86	92	96	89	97	89	94	98	94	96	94	93	94	90	90	96	89	96	91	98	97

Table 3: Percent F1-scores obtained for various C types of all the optimization levels. Not all classes and their scores are shown due to space constraints.

Method	Accuracy	Precision	Recall	F1
Naive	85.4	86.0	85.4	85.4
Slicing	**87.5**	**87.6**	**88**	**87.4**

Table 4: The accuracy, precision, recall, and F1-scores obtained during the comparison of SeeType program slicing with a naive approach to handling longer functions.

Method	Accuracy	Precision	Recall	F1
No Numericals	91.6	91.6	91.7	91.6
SeeType	**93.6**	**93.6**	**93.6**	**93.5**

Table 5: Comparison of SeeType tokenization with baseline on O3 binaries.

11.4 RQ4: Performance of SeeType's tokenization method

We conducted experiments to evaluate the effectiveness of our tokenization scheme discussed in 8. We compared our method's performance with a naive baseline method. In the baseline method, we replaced all the numerical values with a single `<number>` token. Thus the numerical values in the instruction sequence were converted to the token `<number>`. Previous tools such as TypeMiner [47] and PalmTree [66] used a similar scheme for handling numerical values. In our approach, we kept the numerical values unchanged and treated them as general tokens. But, we set the maximum token count of the tokenizer to 3000. This means the tokenizer should train itself to recognize the most frequent 3000 tokens from all the instructions present in the dataset. The tokenizer should learn all the non-numerical tokens and the most frequent numerical values. The comparison of the performance of these two methods is presented in Table 5.

12 Limitations

The scores presented in Table 1 should be observed with an understanding of the limitations of the dataset. In a real-world scenario, this kind of performance may not be reflected. As previously discussed, the models are only as good as the samples they are trained on and the dataset was not complete. There is still work needed to improve the ground truth generation process.

Although our approach of training on a single instruction per sample takes more time to train per instruction, the total time required for the model to be trained is relatively low. This is because during down-sampling we are discarding a large number of samples. For example, 95% of the total samples are discarded during the downsampling of the O0 dataset. Therefore, SeeType is being trained on only 5% of the total dataset on which StateFormer needs to be trained. Training StateFormer is 5 times faster than SeeType's 30 hours on average.

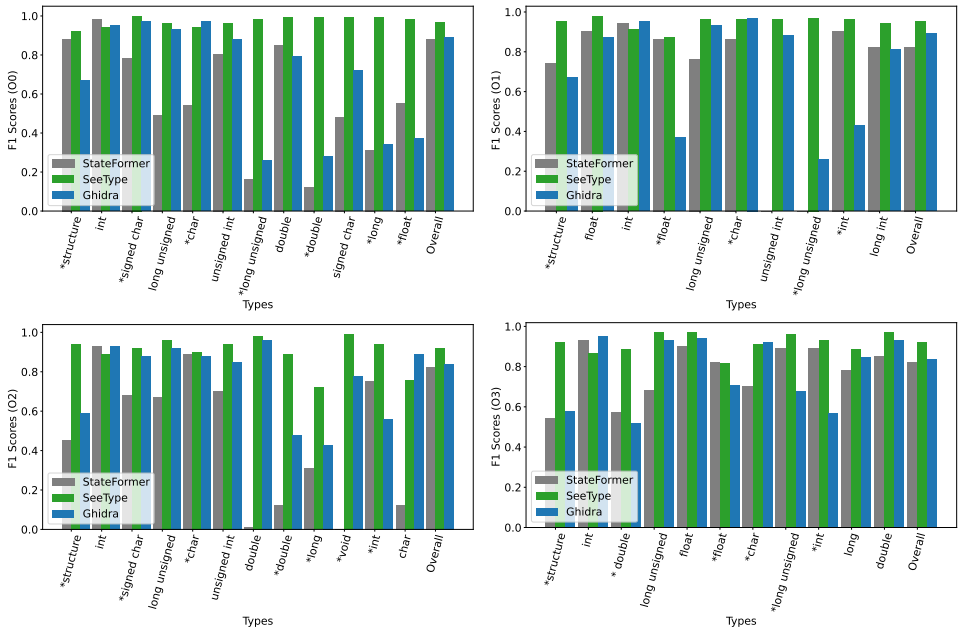


Fig. 8: Comparison of F1-scores between SeeType, Ghidra, and StateFormer for different C types.

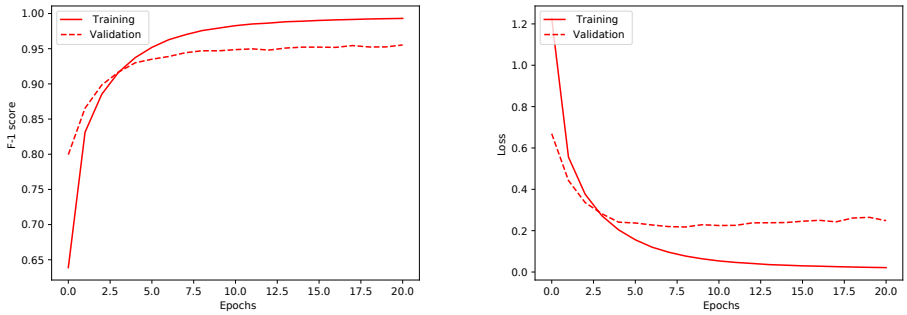


Fig. 9: F1-score and Training loss curves obtained during fine-tuning SeeType with Optimization level 1 dataset.

Figure 9 shows the loss and F1 curves during training and validation of SeeType during fine-tuning with the optimization level O1 dataset. We can see that the model gains its peak performance quite early and hence is suitable for fine-tuning with a massive dataset.

13 Future Work

In this research, we experimented with inferring the C source-level types from binaries. However, there are other higher abstraction data structures, such as linked lists and stacks, that potentially provide even more semantic context to the reverse engineering process. As we have a large number of executables with their corresponding source information, we intend to generate ground truth information for those abstractions and train a model to predict them as well.

14 Conclusion

We have presented a progressive approach to train a Transformer-based binary analysis model SeeType. Our method provides a warm-start to the BERT classifier model by pre-training with two self-supervising tasks, Masked Language Modeling, and Data Dependency prediction. Then, our model transfers the learned knowledge to the data type prediction task. Additionally, we have developed an automated method to create a large-scale collection of binary programs along with their corresponding source code to facilitate the pre-training and fine-tuning tasks. We also developed a method to handle large programs using backward and forward slicing. Through training on the dataset we created, our method SeeType showed improvement over the baseline method. We covered a wide variety of granular C types and our model outperformed the baseline classifying the majority of the types. As we have an automated method to create a large set of binaries, we plan on investigating the broader implications of using data-driven binary analysis methods.

15 Statements and Declarations

15.1 Availability of data and materials

Our dataset, including the binaries and the final processed input files for training the model, is over 1TB. Due to its large scale, it is not feasible to host or share the entire dataset openly as it would require substantial resources. But, we are open to sharing the dataset upon request. The dataset can be reproduced using the method described in the paper. We plan to release our other materials, such as code and model weights upon publication.

15.2 Competing interests

The authors declare that they have no competing interests.

15.3 Funding

Not applicable.

15.4 Acknowledgements

Not applicable.

15.5 Author contribution

Raisul Arefin: The first author contributed to the project by conceptualizing the study, creating the dataset, conducting the experiments, and drafting the manuscript.

Ahmed Mostafa: The second author contributed by writing a script to generate the dataset and editing the manuscript.

Kevan Baker: The third author contributed by editing the draft significantly.

Samuel Mulder: The fourth author, as the Principal Investigator (PI), provided guidance throughout the research process. He significantly contributed to conceptualizing the study and planning and analyzing the experiments. He also significantly contributed to drafting and revising the manuscript.

15.6 Research Involving Human and /or Animals

Not applicable.

15.7 Informed Consent

Not applicable.

References

[1] Pei, K., Guan, J., Broughton, M., Chen, Z., Yao, S., Williams-King, D., Ummadisetty, V., Yang, J., Ray, B., Jana, S.: Stateformer: Fine-grained type recovery from binaries using generative state modeling. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2021, pp. 690–702. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3468264.3468607 . https://doi.org/10.1145/3468264.3468607

[2] Eagle, C., Nance, K.: The Ghidra Book: The Definitive Guide, (2020). https://nostarch.com/GhidraBook

[3] Retdec, A.: Retdec Github Repository. GitHub. https://github.com/avast/retdec

[4] Eagle, C.: The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler. No Starch Press, USA (2011)

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

[6] Green, H., Schwartz, E.J., Goues, C.L., Vasilescu, B.: Stride: Simple type recognition in decompiled executables. arXiv preprint arXiv:2407.02733 (2024)

[7] Chen, Q., Lacomis, J., Schwartz, E.J., Goues, C.L., Neubig, G., Vasilescu, B.: Augmenting Decompiler Output with Learned Variable Names and Types (2021)

[8] Escalada, J., Scully, T., Ortin, F.: Improving type information inferred by decompilers with supervised machine learning. ArXiv **abs/2101.08116** (2021)

[9] Cozzie, A., Stratton, F., Xue, H., King, S.T.: Digging for data structures. In: OSDI, vol. 8, pp. 255–266 (2008)

[10] Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. **13**(1) (2009) https://doi.org/10.1145/1609956.1609960

[11] Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing Forward-Edge Control-Flow integrity in GCC & LLVM. In: 23rd USENIX Security Symposium (USENIX Security 14), pp. 941–955. USENIX Association, San Diego, CA (2014). https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice

[12] Lee, J., Avgerinos, T., Brumley, D.: Tie: Principled reverse engineering of types in binary programs (2011)

[13] Zhang, Z., Ye, Y., You, W., Tao, G., Lee, W.-c., Kwon, Y., Aafer, Y., Zhang, X.: Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 813–832 (2021). https://doi.org/10.1109/SP40001.2021.00051

[14] Caballero, J., Lin, Z.: Type inference on executables. ACM Comput. Surv. **48**(4) (2016) https://doi.org/10.1145/2896499

[15] Bishop, C.M.: Pattern recognition and machine learning. Springer google schola **2**, 645–678 (2006)

[16] Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data. SIGMOD '93, pp. 207–216. Association for Computing Machinery, New York, NY, USA (1993). https://doi.org/10.1145/170035.170072 . https://doi.org/10.1145/170035.170072

[17] Thawani, A., Pujara, J., Ilievski, F., Szekely, P.: Representing numbers in NLP: a survey and a vision. In: Toutanova, K., Rumshisky, A., Zettlemoyer, L., Hakkani-Tur, D., Beltagy, I., Bethard, S., Cotterell, R., Chakraborty, T., Zhou, Y. (eds.) Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 644–656. Association for Computational Linguistics, Online (2021). https://doi.org/10.18653/v1/2021.naacl-main.53 . https://aclanthology.org/2021.naacl-main.53

[18] Wartell, R., Zhou, Y., Hamlen, K.W., Kantarcioglu, M., Thuraisingham, B.: Differentiating code from data in x86 binaries. In: Joint European Conference on

Machine Learning and Knowledge Discovery in Databases, pp. 522–536 (2011). Springer

[19] Meng, X., Miller, B.P.: Binary code is not easy. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ISSTA 2016, pp. 24–35. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2931037.2931047 . https://doi.org/10.1145/2931037.2931047

[20] Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society **74**, 358–366 (1953)

[21] Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: BERT: Pre-training of deep bidirectional transformers for language understanding. In: Burstein, J., Doran, C., Solorio, T. (eds.) Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pp. 4171–4186. Association for Computational Linguistics, Minneapolis, Minnesota (2019). https://doi.org/10.18653/v1/N19-1423 . https://aclanthology.org/N19-1423

[22] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. NIPS'17, pp. 6000–6010. Curran Associates Inc., Red Hook, NY, USA (2017)

[23] Wang, B., Shang, L., Lioma, C., Jiang, X., Yang, H., Liu, Q., Simonsen, J.G.: On position embeddings in bert. In: International Conference on Learning Representations (2021). https://openreview.net/forum?id=onxoVA9FxMw

[24] Liu, W., Wen, Y., Yu, Z., Yang, M.: Large-margin softmax loss for convolutional neural networks. arXiv preprint arXiv:1612.02295 (2016)

[25] Mycroft, A.: Type-based decompilation (or program reconstruction via type reconstruction). In: European Symposium on Programming, pp. 208–223 (1999). Springer

[26] Zhang, C., Song, C., Chen, K.Z., Chen, Z., Song, D.: Vtint: Defending virtual function tables' integrity. In: Symposium on Network and Distributed System Security (NDSS), vol. 160, pp. 173–176 (2015)

[27] Emmerik, M., Waddington, T.: Using a decompiler for real-world source recovery. In: 11th Working Conference on Reverse Engineering, pp. 27–36 (2004). IEEE

[28] Zhang, M., Prakash, A., Li, X., Liang, Z., Yin, H.: Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis (2012)

[29] Srinivasan, V., Reps, T.: Recovery of class hierarchies and composition relationships from machine code. In: International Conference on Compiler Construction,

pp. 61–84 (2014). Springer

[30] Zeng, J., Fu, Y., Miller, K.A., Lin, Z., Zhang, X., Xu, D.: Obfuscation resilient binary code reuse through trace-oriented programming. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 487–498 (2013)

[31] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., *et al.*: Sok:(state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 138–157 (2016). IEEE

[32] Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: Bap: A binary analysis platform. In: Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23, pp. 463–469 (2011). Springer

[33] Jin, W., Cohen, C., Gennari, J., Hines, C., Chaki, S., Gurfinkel, A., Havrilla, J., Narasimhan, P.: Recovering c++ objects from binaries using inter-procedural data-flow analysis. In: Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014. PPREW'14. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2556464.2556465 . https://doi.org/10.1145/2556464.2556465

[34] Flores-Montoya, A., Schulte, E.: Datalog disassembly. In: 29th USENIX Security Symposium, pp. 1075–1092 (2020). https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya

[35] radare2. GitHub. https://github.com/radareorg/radare2

[36] 35, V.: Binary Ninja. GitHub. https://github.com/Vector35/binaryninja-api

[37] Djoudi, A., Bardin, S.: Binsec: Binary code analysis with low-level regions. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 212–217 (2015). Springer

[38] Harris, L.C., Miller, B.P.: Practical analysis of stripped binary code. SIGARCH Comput. Archit. News **33**(5), 63–68 (2005) https://doi.org/10.1145/1127577.1127590

[39] Caballero, J., Johnson, N.M., McCamant, S., Song, D.: Binary code extraction and interface identification for security applications. In: NDSS, vol. 10, pp. 391–408 (2010)

[40] Slowinska, A., Stancescu, T., Bos, H.: Howard: A dynamic excavator for reverse engineering data structures. In: Network and Distributed System Security Symposium (2011). https://api.semanticscholar.org/CorpusID:43281

[41] Caballero, J., Grieco, G., Marron, M., Lin, Z., Urbina, D.I.: Artiste: Automatic generation of hybrid data structure signatures from binary code executions. (2012). https://api.semanticscholar.org/CorpusID:54749001

[42] Lin, Z., Zhang, X., Xu, D.: Automatic reverse engineering of data structures from binary execution. In: Proceedings of the 11th Annual Information Security Symposium. CERIAS '10. CERIAS - Purdue University, West Lafayette, IN (2010)

[43] Haller, I., Slowinska, A., Bos, H.: Mempick: High-level data structure detection in c/c++ binaries. In: 2013 20th Working Conference on Reverse Engineering (WCRE), pp. 32–41 (2013). https://doi.org/10.1109/WCRE.2013.6671278

[44] Cozzie, A., Stratton, F., Xue, H., King, S.T.: Digging for data structures. In: 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08). USENIX Association, San Diego, CA (2008). https://www.usenix.org/conference/osdi-08/digging-data-structures

[45] Lin, Z., Li, J., Li, B., Ma, H., Gao, D., Ma, J.: Typesqueezer: When static recovery of function signatures for binary executables meets dynamic analysis. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. CCS '23, pp. 2725–2739. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3576915.3623214 . https://doi.org/10.1145/3576915.3623214

[46] Chua, Z.L., Shen, S., Saxena, P., Liang, Z.: Neural nets can learn function type signatures from binaries. In: Proceedings of the 26th USENIX Conference on Security Symposium. SEC'17, pp. 99–116. USENIX Association, USA (2017)

[47] Maier, A., Gascon, H., Wressnegger, C., Rieck, K.: Typeminer: Recovering types in binary programs using machine learning. In: Perdisci, R., Maurice, C., Giacinto, G., Almgren, M. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 288–308. Springer, Cham (2019)

[48] Xu, Z., Wen, C., Qin, S.: Type learning for binaries and its applications. IEEE Transactions on Reliability **68**(3), 893–912 (2019) https://doi.org/10.1109/TR.2018.2884143

[49] Lehmann, D., Pradel, M.: Finding the dwarf: Recovering precise types from webassembly binaries. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2022, pp. 410–425. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523449 . https://doi.org/10.1145/3519939.3523449

[50] Chen, L., He, Z., Mao, B.: Cati: Context-assisted type inference from stripped

binaries. In: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 88–98 (2020). https://doi.org/10.1109/DSN48063.2020.00028

[51] Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. CoRR **abs/1409.0473** (2014)

[52] Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the web up to speed with webassembly. SIGPLAN Not. **52**(6), 185–200 (2017) https://doi.org/10.1145/3140587.3062363

[53] Banerjee, P., Pal, K.K., Wang, F., Baral, C.: Variable name recovery in decompiled binary code using constrained masked language modeling. CoRR **abs/2103.12801** (2021) 2103.12801

[54] Arefin, R., Samad, M.D., Akyelken, F.A., Davanian, A.: Non-transfer deep learning of optical coherence tomography for post-hoc explanation of macular disease classification. In: 2021 IEEE 9th International Conference on Healthcare Informatics (ICHI), pp. 48–52 (2021). IEEE

[55] Zaheer, M., Guruganesh, G., Dubey, K.A., Ainslie, J., Alberti, C., Ontanon, S., Pham, P., Ravula, A., Wang, Q., Yang, L., *et al.*: Big bird: Transformers for longer sequences. Advances in neural information processing systems **33**, 17283–17297 (2020)

[56] Tay, Y., Dehghani, M., Abnar, S., Shen, Y., Bahri, D., Pham, P., Rao, J., Yang, L., Ruder, S., Metzler, D.: Long range arena : A benchmark for efficient transformers. In: International Conference on Learning Representations (2021). https://openreview.net/forum?id=qVyeW-grC2k

[57] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I.: Language models are unsupervised multitask learners. (2019). https://api.semanticscholar.org/CorpusID:160025533

[58] Zhang, J., Zhao, Y., Saleh, M., Liu, P.: Pegasus: Pre-training with extracted gap-sentences for abstractive summarization. In: International Conference on Machine Learning, pp. 11328–11339 (2020). PMLR

[59] Wang, X., Xu, X., Li, Q., Yuan, M., Xue, J.: Recovering container class types in c++ binaries. In: 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 131–143 (2022). IEEE

[60] Weiser, M.: Program slicing. IEEE Transactions on software engineering (4), 352–357 (1984)

[61] Cifuentes, C., Fraboulet, A.: Intraprocedural static slicing of binary executables.

In: 1997 Proceedings International Conference on Software Maintenance, pp. 188–195 (1997). https://doi.org/10.1109/ICSM.1997.624245

[62] Kiss, A., Jasz, J., Lehotai, G., Gyimothy, T.: Interprocedural static slicing of binary executables. In: Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation, pp. 118–127 (2003). https://doi.org/10.1109/SCAM.2003.1238038

[63] Tian, J., Xing, W., Li, Z.: Bvdetector: A program slice-based binary code vulnerability intelligent detection system. Information and Software Technology **123**, 106289 (2020) https://doi.org/10.1016/j.infsof.2020.106289

[64] Xue, H., Venkataramani, G., Lan, T.: Clone-slicer: Detecting domain specific binary code clones through program slicing. In: Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation, pp. 27–33 (2018)

[65] Turc, I., Chang, M.-W., Lee, K., Toutanova, K.: Well-Read Students Learn Better: On the Importance of Pre-training Compact Models (2020). https://openreview.net/forum?id=BJg7x1HFvB

[66] Li, X., Qu, Y., Yin, H.: Palmtree: Learning an assembly language model for instruction embedding. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 3236–3251 (2021)

[67] Ahn, S., Ahn, S., Koo, H., Paek, Y.: Practical binary code similarity detection with bert-based transferable similarity learning. In: Proceedings of the 38th Annual Computer Security Applications Conference, pp. 361–374 (2022)

[68] Madsen, A., Johansen, A.R.: Neural arithmetic units. CoRR **abs/2001.05016** (2020) 2001.05016

[69] Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: Duesterwald, E. (ed.) Compiler Construction, pp. 5–23. Springer, Berlin, Heidelberg (2004)

[70] Nayak, A., Timmapathini, H., Ponnalagu, K., Venkoparao, V.G.: Domain adaptation challenges of bert in tokenization and sub-word representations of out-of-vocabulary words. In: Proceedings of the First Workshop on Insights from Negative Results in NLP, pp. 1–5 (2020)

[71] Siddique, A.B., Oymak, S., Hristidis, V.: Unsupervised paraphrasing via deep reinforcement learning. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. KDD '20, pp. 1800–1809. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3394486.3403231 . https://doi.org/10.1145/3394486.3403231

[72] Mohammed, R., Rawashdeh, J., Abdullah, M.: Machine learning with oversampling and undersampling techniques: Overview study and experimental results. In: 2020 11th International Conference on Information and Communication Systems (ICICS), pp. 243–248 (2020). https://doi.org/10.1109/ICICS49469.2020.239556

[73] Liu, X.-Y., Wu, J., Zhou, Z.-H.: Exploratory undersampling for class-imbalance learning. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) **39**(2), 539–550 (2009) https://doi.org/10.1109/TSMCB.2008.2007853

[74] Bendersky, E.: pyelftools. GitHub. https://github.com/eliben/pyelftools

[75] Security, C.I.: miasm. GitHub. https://github.com/cea-sec/miasm

[76] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., Rush, A.: Transformers: State-of-the-art natural language processing. In: Liu, Q., Schlangen, D. (eds.) Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pp. 38–45. Association for Computational Linguistics, Online (2020). https://doi.org/10.18653/v1/2020.emnlp-demos.6 . https://aclanthology.org/2020.emnlp-demos.6

[77] Zhang, Z.: Improved adam optimizer for deep neural networks. In: 2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS), pp. 1–2 (2018). Ieee

[78] stateformer github. GitHub. https://github.com/CUMLSec/stateformer