Transformers can do it: Recovering Types from Executables using Transformer based LLMs

Ruturaj Vaidya[1] and Prasad A. Kulkarni[2]
[1]*AMD Inc. and EECS/I2S, University of Kansas, Lawrence, KS, USA*
[2]*EECS/I2S, University of Kansas, Lawrence, KS, USA*
ruturaj.vaidya@amd.com, prasadk@ku.edu

Keywords: Binary type recovery, LLMs, Machine Learning

Abstract: Accurate *type recovery* from stripped binaries is important. Reconstructed types help improve the capability of binary analysis, which can aid reverse engineers to gain a better understanding of source semantics and syntax, and establish a foundational contrivance to support many security applications such as control-flow integrity (CFI), binary similarity, malware analysis, software forensics, vulnerability assessment, etc. In this work, we propose a novel architecture agnostic type recovery technique called *YĀLI* ("**Y**et **A**nother **L**anguage model for type **I**nference") to predict function parameter and callsite argument type information from stripped and optimized binaries. Our approach is a two stage process - firstly, in the static analysis and data collection phase we leverage the Ghidra binary analysis tool, to lift low-level binary executables to Ghidra's intermediate representation called *P-Code*, and recover *P-Code* slices for both function parameters and callsite arguments. Secondly, in the training stage, we utilize a light-weight *BERT* based *Transformer* model called *DistilBERT* to capture *P-Code* semantics and understand data-flow patterns to accurately perform the task of type classification. To assess our technique, we use a corpora of around 33k binaries compiled on different architectures, using various compilers and optimization levels. YĀLI achieves on average around 94% and 92% accuracy for function parameter and callsite argument recovery tasks respectively, significantly surpassing conventional type recovery techniques.

1 INTRODUCTION

Binary analysis is an important area of research that aims to recognize the structural properties of the compiled binary, in order to understand its functional behavior without having access to the source code. During compilation, when high-level code gets converted into its binary counterpart, useful high-level features such as function and variable names, types, control-flow constructs, function boundaries, etc. get lost, making binary analysis difficult. Additionally, debugging symbols are usually stripped from production-level binaries to reduce the size, improve performance and to maintain the security and confidentiality of the product. Debug information assists binary analysis frameworks in program analysis and thus the absence of such high-level symbol information hinders their analysis capabilities.

An important artifact that is missing in stripped binaries is source-level type indication. Program variables are represented either as registers or as memory locations at low-level after compilation, without any indication of explicit source-level types. Function and callsite type knowledge is important to accurately construct the program call-graph and apply techniques such as CFI (control-flow integrity) (Muntean et al., 2018; Lin and Gao, 2021), vulnerability detection (Mantovani et al., 2022; Han et al., 2023), binary rewriting/ hardening (Lu and Hu, 2019; Williams-King et al., 2020), decompilation (Brumley et al., 2013; Burk et al., 2022), etc. However, type recovery from binary programs is challenging and inherently speculative as type recuperation is affected due to a loss of source-level semantics during compilation.

The type recovery problem has been traditionally tackled using rule-based as well as machine learning based techniques. Manually defined rule based techniques (Lin et al., 2010; Lee et al., 2011; Slowinska et al., 2011; Caballero et al., 2012; Katz et al., 2016; Noonan et al., 2016; Zhang et al., 2021) tend to be tedious and fragile (Caballero and Lin, 2016). Moreover, architecture and compiler specific knowledge is required to support and augment such techniques with domain information (Bao et al., 2014), making them harder to maintain. Such binary-level algorithms often use abstract interpretation and data-flow analysis to recognize variables and types, and their static analysis often depends on Value Set Analysis (VSA) (Balakrishnan and Reps, 2004). VSA is known to produce inaccurate results, due to conservative analysis, over-approximation and inaccuracy in pointer arithmetic. Many existing reverse engineering frameworks such as Ghidra (National Security Agency ghidra, 2023) and IDA pro (hexrays, 2023), offer sophisticated algorithms to recover type information using traditional rule based techniques.

In this work, we focus on recovering function and callsite type information from binary executables using transformer based Large Language Models (LLMs). This problem has a variety of applications in control-flow recovery and binary hardening (Abadi et al., 2005; van der Veen et al., 2016; Burow et al., 2017; Muntean et al., 2018), data dependency analysis (Saxena et al., 2008), etc. Ours is a novel technique that leverages Ghidra's *P-Code slices* to teach our "DistilBERT" (Sanh et al., 2019) based transformer models to learn the inherent characteristics of Ghidra's *P-Code* intermediate representation in the first stage, and fine tune the model for type classification task in the second stage to recover advance variable types.

Ghidra converts the generated disassembly into an intermediate representation called *P-Code* (Naus et al., 2023) (before transforming it into `C`-like pseudo-code representation during the process of decompilation). The *P-Code* is an intermediate representation used by Ghidra. It consists of two levels of abstractions - *low-level* and *high-level* P-Code. The low-level P-Code is created by mapping assembly instructions to P-Codes in a one-to-many fashion. The low-level P-Code representation is closer to the underlying processor architecture. The high-level P-Code is a result of many transformations on low-level P-Code during the decompilation process. The high P-Code obscures away the underlying architecture and consequently it is more suitable for architecture agnostic applications. We refer to high-level P-Code as *P-Code* in the rest of this paper.

We use Transformer based Large Language Models (LLMs) to learn P-Code attributes and variable data-flow patterns, and later use it for our type classification task. BERT-like (Devlin et al., 2018) Large Language Models have adeptly outperformed RNNs, CNNs and LSTMs in many domains by leveraging transformer architecture and incorporating large-scale pre-training to learn comprehensive context and language intricacies. Additionally, their impressive transfer learning capabilities and ability to support diverse Natural Language Processing (NLP) tasks have made them the preferred choice over earlier neural network (NN) architectures.

For this work, we use the DistilBERT model to fit in our resource constraint experimental setup. DistilBERT (Sanh et al., 2019) is a transformer based Large Language Model introduced by Hugging Face[1]. DistilBERT is a smaller version of Bert (Devlin et al., 2018) and has 6 transformer layers (compared to 12 in Bert) and fewer parameters (66 Million) compared to the original Bert model (110 Million). The DistilBERT model is trained using the process of language distillation and hence it is computationally more efficient than the Bert model, while maintaining competitive performance.

We propose an architecture agnostic technique called *YĀLI*[2] ("**Y**et **A**nother **L**anguage model for type **I**nference") to recuperate type indications in stripped and optimized binaries using natural language processing techniques employed on Ghidra's P-Code. We leverage pre-trained large language model "DistilBERT" and fine-tune it for type classification task on raw P-Code instructions of recovered parameter and argument slices. We focus primarily on two important type inference tasks. In the first task, we focus on the type recovery of function parameters and the second task centers around finding the correct argument types at both direct and indirect callsites.

The primary motivation behind using the Ghidra framework is to leverage its analysis capabilities of static rule-based approach to find variable data-flow and augment the analysis to recover types using deep learning techniques, consequently benefiting from the best of both worlds. Additionally, Ghidra's P-Code representation and P-Code based architecture-agnostic algorithms allow us to significantly reduce the programming effort across multiple architectures and enhance the portability of our implementation.

We compare the accuracy of our technique with four state-of-the-art type recovery techniques having a combination of heuristic and deep learning based frameworks. Our model achieves around 94% for function parameter and around 92% for callsite argument type recovery, averaging over four different architectures.

Our primary contributions of this work are:

1. We create a novel technique that leverages static binary analysis and large language models to recover types from optimized and stripped binaries.

2. We develop a design that uses Ghidra and P-Code to enable our technique to support multiple archi-

[1]https://huggingface.co/

[2]Named after a legendary creature from South Asia

tectures and compilers.

3. We thoroughly evaluate our technique by first training our model on a large corpora of binaries, and comparing the results with existing state-of-the-art. We show that, our technique significantly improves on existing techniques.
4. We plan to open source our framework and trained models to support academic research.

2 BACKGROUND

In this section we explain the primary concepts and techniques that we use to develop our novel approach.

2.1 Transformer Architecture

Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) models were introduced to tackle the problem of vanishing gradient descent that was limiting RNNs to capture the long-term information. Consequently, LSTMs quickly became popular in the area of natural language processing (NLP) and other sequential tasks. LSTMs incorporate cell state and gate layers to carefully 'forget' and 'remember' information context over time. However, LSTMs demand sequential processing of input sequences, thus significantly hindering the GPU parallelization support. In addition to that, LSTMs struggle with maintaining long term dependencies, as the gradients of the loss function could become extremely small in longer sequences, inadvertently due to repeated multiplications during back-propagation.

In more recent years, Transformer architecture (Vaswani et al., 2017) initiated a groundbreaking revolution in the field of NLP. Primarily, the introduction of 'Self Attention' mechanism helps understanding the context better and provides the model an ability to remember the long-term dependencies. Additionally, positional encoding allows the model to keep sequential context and simultaneously provides the ability to support parallel computations. Transformers are comprised of multiple encoder-decoders with numerous multi-headed attention and feed forward NN layers. Transformer-based architectures are shown to be scalable and exhibit outstanding performance compared to the older NNs (Brown et al., 2020). Accordingly, transformers (especially the encoder layers) are well-suited for our sentence-based data-type classification task.

2.2 Knowledge Distillation

Knowledge distillation (Hinton et al., 2015) is a technique of transferring the knowledge of an intricate larger (teacher) model to a less intricate smaller (student) model. The fundamental goal is to adapt the knowledge and generalizability of the teacher model into a compact and computationally less intensive student model that can be deployed in resource constrained environment. DistilBERT (Sanh et al., 2019) is one of many models that implement distilled version of the original BERT model. The loss function of DistilBERT is a combination of three loss functions - Knowledge Distillation loss (L_{KD}), Masked Language Modelling loss (L_{MLM}) and Cosine embedding loss (L_{Cosine}), with γ, α and β being hyperparameters.

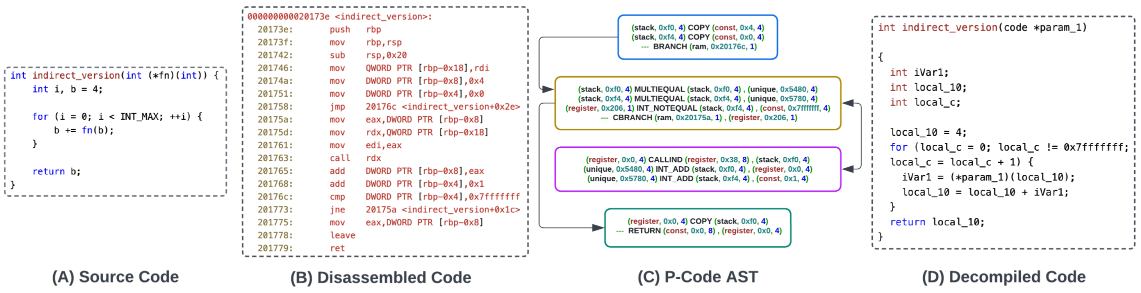
$$L_{Total} = \gamma . L_{KD} + \alpha . L_{MLM} + \beta . L_{Cosine}$$

DistilBERT attains about 97% of BERT model's accuracy despite being 40% smaller. It aims to incur minimum loss during the knowledge transfer process, while achieving comparable accuracy as the teacher model. Considering the availability of limited resources, in this work we use pre-trained DistilBERT model and fine-tune it for the downstream task of type classification.

2.3 Ghidra's P-Code Intermediate Representation

Ghidra (National Security Agency ghidra, 2023) is a Software Reverse Engineering (SRE) framework developed by the National Security Agency (NSA) and made open source in recent years. Ghidra constitutes two important components called disassembler and decompiler. The former converts low-level machine code to assembly and the later transforms it into a pseudo-code or `C`-like high-level representation. Ghidra elevates the reversed assembly to an intermediate level representation called P-Code, before transforming into a use-friendly decompiled code. Knowledge of Source-level type information, such as function parameters and callsite arguments, is essential for accurately constructing a high-level representation of program binary. Ghidra implements its heuristic-based algorithms on its P-Code intermediate representation for the task of type recovery.

Figure 1 demonstrates a running example that illustrates a high-level overview of Ghidra's decompilation process. Firstly, Figure 1(a) shows the high-level 'C' source code (function `indirect_version`) used in this example. The corresponding binary code is input to Ghidra. Figure 1(b) shows the corresponding disassembly that is generated by the Ghidra disassembler. Then, Ghidra transforms the disassembly



```
int indirect_version(int (*fn)(int)) {
    int i, b = 4;

    for (i = 0; i < INT_MAX; ++i) {
        b += fn(b);
    }

    return b;
}
```

(A) Source Code

```
000000000020173e <indirect_version>:
  20173e:       push   rbp
  20173f:       mov    rbp,rsp
  201742:       sub    rsp,0x20
  201746:       mov    QWORD PTR [rbp-0x18],rdi
  20174a:       mov    DWORD PTR [rbp-0x8],0x4
  201751:       mov    DWORD PTR [rbp-0x4],0x0
  201758:       jmp    20176c <indirect_version+0x2e>
  20175a:       mov    eax,DWORD PTR [rbp-0x8]
  20175d:       mov    rdx,QWORD PTR [rbp-0x18]
  201761:       mov    edi,eax
  201763:       call   rdx
  201765:       add    DWORD PTR [rbp-0x8],eax
  201768:       add    DWORD PTR [rbp-0x4],0x1
  20176c:       cmp    DWORD PTR [rbp-0x4],0x7fffffff
  201773:       jne    20175a <indirect_version+0x1c>
  201775:       mov    eax,DWORD PTR [rbp-0x8]
  201778:       leave
  201779:       ret
```

(B) Disassembled Code

```
(stack, 0xf0, 4) COPY (const, 0x4, 4)
(stack, 0xf4, 4) COPY (const, 0x0, 4)
--- BRANCH (ram, 0x20176c, 1)

(stack, 0xf0, 4) MULTIEQUAL (stack, 0xf0, 4) , (unique, 0x5480, 4)
(stack, 0xf4, 4) MULTIEQUAL (stack, 0xf4, 4) , (unique, 0x5780, 4)
(register, 0x206, 1) INT_NOTEQUAL (stack, 0xf4, 4) , (const, 0x7fffffff, 4)
--- CBRANCH (ram, 0x20175a, 1) , (register, 0x206, 1)

(register, 0x0, 4) CALLIND (register, 0x38, 8) , (stack, 0xf0, 4)
(unique, 0x5480, 4) INT_ADD (stack, 0xf0, 4) , (register, 0x0, 4)
(unique, 0x5780, 4) INT_ADD (stack, 0xf4, 4) , (const, 0x1, 4)

(register, 0x0, 4) COPY (stack, 0xf0, 4)
--- RETURN (const, 0x0, 8) , (register, 0x0, 4)
```

(C) P-Code AST

```
int indirect_version(code *param_1)

{
  int iVar1;
  int local_10;
  int local_c;

  local_10 = 4;
  for (local_c = 0; local_c != 0x7fffffff;
  local_c = local_c + 1) {
    iVar1 = (*param_1)(local_10);
    local_10 = local_10 + iVar1;
  }
  return local_10;
}
```

(D) Decompiled Code

Figure 1: High-level Overview of Ghidra's Decompilation Process

into P-Code representation, shown in the format of abstract syntax tree (AST) in Figure 1(c). Finally, the corresponding high-level decompiled code generated by utilizing the P-Code produced in the previous stage, after performing multiple analysis passes is shown in Figure 1(d). P-Code is architecture agnostic and beneficial in discovering the program attributes using high-level binary analysis. Consequently, we use the P-Code format for the type recovery task in this work. At the same time, we believe that our approach is extensible and can be implemented on other intermediate formats such as IDA Pro's *Microcode*.

3 RELATED WORK

In this section we present and contrast our work with other binary-level type recovery approaches.

3.1 Rule-based Type Recovery

Rule-based type recovery techniques are those that employ rules and heuristics manually crafted by expert humans to identify and assign high-level language types to data objects in a binary program. Researchers have designed many rule-based type recovery techniques (Lin et al., 2010; Lee et al., 2011; Slowinska et al., 2011; Caballero et al., 2012; Noonan et al., 2016; Zhang et al., 2021). While some of these techniques use a dynamic approach (Lin et al., 2010; Slowinska et al., 2011; Caballero et al., 2012), others use static heuristics-based methodologies to recover types (Lee et al., 2011; Noonan et al., 2016; Zhang et al., 2021). Dynamic analysis techniques usually use run-time taint tracking to determine the flow of data through the program, which in turn is used to deduce variable types. Such techniques generally suffer from coverage issues, time constraints and infeasibility of execution for programs that require specialized hardware and operating conditions.

The Retypd (Noonan et al., 2016) type system uses constraint-based solving with static type inference that supports polymorphism, sub-typing and recursive types. OSPREY (Zhang et al., 2021) is a recently published technique that uses probabilistic approach that surpasses state-of-the-art reverse engineering frameworks such as Ghidra, IDA Pro, etc in terms of variable and structure type accuracy. However, OSPREY doesn't recover register allocated variables and only recovers variables allocated on the stack and heap. Many frameworks that are popular in the reverse engineering community, such as IDA Pro (hexrays, 2023) and Ghidra (National Security Agency ghidra, 2023), employ static rule-based techniques to enhance decompilation, however are very limited in their capabilities. Static analysis techniques do not rely on run-time program behaviour but suffer from incompleteness due to lack of execution context.

Rule-based techniques exclusively rely on program analysis, are generally limited to small set of syntactic types, use handwritten rules which tend to be fragile and tedious to develop, and are difficult to extend across different architectures.

3.2 Type Recovery Using ML

Automated machine learning based approaches have also been employed to resolve the type recovery problem in binary software and alleviate some of the issues with the manual rule-based techniques. EKLAVYA (Chua et al., 2017) is one such technique that focuses on recovering the count and types of function parameters and callsite arguments. In EKLAVYA, the task of type inference for each argument is considered as a distinct classification task. In the first stage, they leverage *word2vec* to generate instruction embeddings, and then in the second stage these embeddings are provided to the downstream multi-layer Gated Recurrent Unit (GRU) network model for the type inference task. DEBIN (He et al., 2018) uses probabilistic machine learning models to recover name and type pairs and variable locations to essentially reconstruct missing debug information from stripped binaries. However, EKLAVYA supports a very limited set (seven) of argument types and DEBIN does not support floating point types.

TypeMiner (Maier et al., 2019) is a simple machine learning based technique that employs multi-stage classification on object data-flow traces to recuperate preliminary types. Their technique is evaluated on a fairly small-scale dataset. STATEFORMER (Pei et al., 2021) leverages micro-execution traces and employs transformers to acquire the instruction semantics and bypasses the challenge of feature selection, in turn producing pre-trained models. Subsequently, these pre-trained models are then leveraged for conducting type prediction. None of these techniques make use of decompiler output to revamp their results. DIRTY (Chen et al., 2022) on the other hand concentrates on recovering type and name information by using decompiler output, significantly invigorating the final outcome. DIRTY trains transformer-based encoder-decoder models using an extensive collection of binaries using their corresponding ground truth symbol information. Subsequently, this trained model is employed to generate variable names and types for binary programs decompiled using IDA Pro. DIRTY is also capable of capturing 48,888 syntactic and non-syntactic types.

In this work we are the first to explore type recovery using transformer-based classification of intermediate instruction slices that capture function and callsite argument data-flow. Although YĀLI does not recover as many types as DIRTY, we believe that YĀLI's type system is extensive and practical enough to identify sufficient range of types.

Rather than doing pure type recovery, researchers have also developed related techniques that exclusively focus on improving the *Binary Code Embedding (BCE)* task using numerical features learned from binary code (Li et al., 2021; Zhu et al., 2023; Wang et al., 2022). Such work can result in improving the accuracy of downstream binary analysis tasks, including type recovery. Likewise, there are related techniques, such as (Nitin et al., 2021; **?**; **?**), that focus on recovering variable names to improve readability of the decompiled code, while others (Wong et al., 2023) that focus on improving its overall explainability. However, ours is orthogonal to these works as we exclusively focus on recovering types of function parameters and callsite arguments.

4 DESIGN

The problem of recovering function and callsite signatures from a binary requires identifying the number of parameters/ arguments and their data types. In this work, we focus on the later part, i.e. identification of function and callsite parameter/ argument types. Function and callsite signatures are monumental in constructing accurate function call-graphs, improving data-flow analysis, and producing readable and understandable decompilation output, which in turn can help the evaluator in accurate program comprehension, symbolic analysis, malware and vulnerability detection, control-flow integrity (CFI), etc.

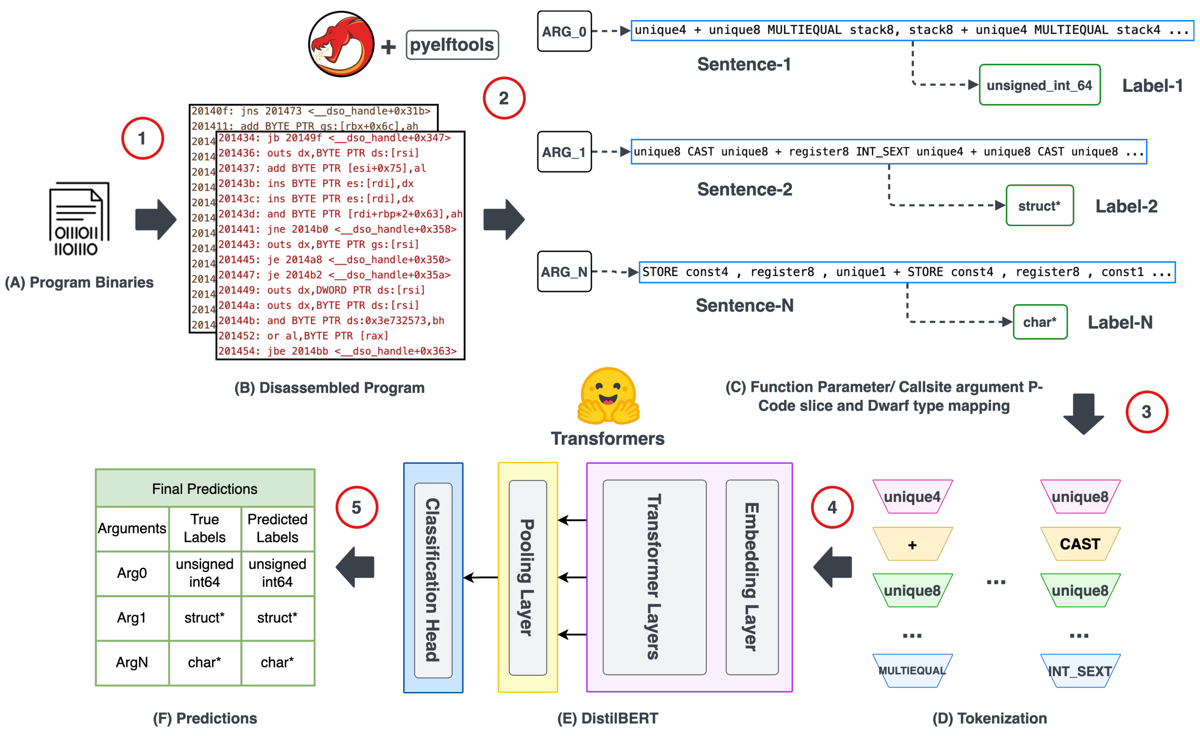
Traditionally, function-callsite arguments and their type recovery is achieved by identifying various rules and conventions that manifest function and callsite signatures in the assembly. First, the binary gets disassembled, to capture an understanding of the high-level assembly. Second, function calling conventions and parameter/argument passing patterns are analyzed. Expert knowledge of compilers and machine architecture is required to perform such analysis, as such patterns differ according to different architectures, compilers, optimization levels, etc. For example, in the x86-64 architecture, the *GCC* compiler typically uses the register `rbp` to track the frame pointer in unoptimized binaries, while in highly optimized binaries it tends to use the register `rsp` to manage the stack frame. Rule-based argument recovery techniques are difficult to implement as they are error prone and are harder to scale.

To address the problem of function parameter and callsite argument type recovery, we introduce *YĀLI* (**Y**et **A**nother **L**anguage model for type **I**nference). YĀLI treats the problem of argument type recovery as a natural language processing (NLP) based text classification problem. The Large language models (LLMs) have shown to be better for text classification task due to better contextual understanding, scalability and faster inference, and thus are well suitable for the task like this. YĀLI employs DistilBERT transformer model along with an additional classification head as the uppermost layer to implement type classification. Ghidra P-Code slices are used as input to the model and labels are the types recovered from DWARF symbol information. We treat the task of type classification as two sub-tasks:

1. **Task-1:** Function parameter type recovery.
2. **Task-2:** Argument type recovery at callsites.

4.1 Technical Overview

Figure 2 gives the high-level overview of the architecture of YĀLI. The execution of YĀLI's type classification routine broadly consists of following key phases. In the first phase, our framework employs binary analysis techniques to the extract P-Code slices



```
20140f: jns 201473 <__dso_handle+0x31b>
201411: add BYTE PTR gs:[rbx+0x6c],ah
```

```
201434: jb 20149f <__dso_handle+0x347>
201436: outs dx,BYTE PTR ds:[rsi]
201437: add BYTE PTR [esi+0x75],al
20143b: ins BYTE PTR es:[rdi],dx
20143c: ins BYTE PTR es:[rdi],dx
20143d: and BYTE PTR [rdi+rbp*2+0x63],ah
201441: jne 2014b0 <__dso_handle+0x358>
201443: outs dx,BYTE PTR gs:[rsi]
201445: je 2014a8 <__dso_handle+0x350>
201447: je 2014b2 <__dso_handle+0x35a>
201449: outs dx,DWORD PTR ds:[rsi]
20144a: outs dx,BYTE PTR ds:[rsi]
20144b: and BYTE PTR ds:0x3e732573,bh
201452: or al,BYTE PTR [rax]
201454: jbe 2014bb <__dso_handle+0x363>
```

```
unique4 + unique8 MULTIEQUAL stack8, stack8 + unique4 MULTIEQUAL stack4 ...
unique8 CAST unique8 + register8 INT_SEXT unique4 + unique8 CAST unique8 ...
STORE const4 , register8 , unique1 + STORE const4 , register8 , const1 ...
```

Final Predictions		
Arguments	True Labels	Predicted Labels
Arg0	unsigned int64	unsigned int64
Arg1	struct*	struct*
ArgN	char*	char*

Figure 2: Overview of YĀLI's Architecture. It takes program binaries as input, disassembles them, extracts argument P-Code slices and related metadata and performs preprocessing on extracted slices. Then, it uses Transformer model to learn the P-Code slice intricacies and finally performs type classification for function and callsite parameter/argument recovery tasks.

and corresponding source-level data types per argument. In the second phase, this recovered information serves as the training input to the BERT-based Large Language Model (LLM). The predictive capabilities of LLM capture the intricate patterns from the P-Code sentences. This fine-tuned model is then harnessed for text classification task to effectively enhance the predicted information that can then be used to augment the decompilation result in the final phase. The seamless integration of program analysis information recovered using SRE frameworks and machine learning based methodology for type classification, sets out as the foundation of our framework. Now, we explain the fundamental stages involved in running YĀLI's training routine for the type classification task.

4.1.1 Disassembling the binaries.

Initiating the workflow, we employ Ghidra binary analyzer to disassemble input binaries as shown in Figures 2 (A-B). We disable any effect of DWARF symbols on Ghidra analysis, so that the Ghidra output remain uninfluenced by such symbols. At the same time, DWARF debugging symbol information from the binaries is analyzed using a popular *ELF* and *DWARF* analysis tool *pyelftools*. The integration of *pyelftools* significantly facilitates the parsing of *Type-DIEs* from the *DWARF* debugging symbols, that in turn allows us to collect index and original source-level type information for each parameter of the function. Subsequently, this acquired information is utilized to create mappings to the types recovered using Ghidra to establish the ground truth, as elaborated in the following section.

4.1.2 Sentence collection.

Next, our framework *decompiles* the the generated assembly on a per-function basis using Ghidra's decompiler API. At this stage, two sets of essential program information are extracted. The first set encompasses the address-taken functions in binaries, their parameters, parameter indexes along with the forward slice of P-Code instructions. The mapping between the function parameters recovered using Ghidra and their corresponding source-level types is then achieved using the parameter types recovered using *DWARF* in the previous stage. Correct mapping is facilitated by the utilization of parameter indexes. The forward slice allows us to collect the P-Code instructions that are influenced by a particular parameter.

The second set of program information comprised of callsite arguments, argument indexes and an additional combination of forward and backward P-Code instruction slice per argument is also recovered using Ghidra. We employ a combination of backward and forward slices to expand the context. Note that we do not consider the *indirect* program callsites at this stage. Only the *direct* callsite arguments are used to correlate with the corresponding target function. In this way we can establish the ground truth types for the callsite signature and align them with Ghidra callsite arguments. P-Code is architecture agnostic, and thus it allows us to reproduce all the steps without architectural constraints. This streamlines the integration of YĀLI by making it adaptable across various architectures.

Figure 2 (C) shows the collected sentences and mapped labels. The sentences are sanitized to remove addresses and literal values. For example, P-Code instruction "`(register, 0x30, 8) PIECE (register, 0x34, 4)`" is cleaned so that the sentence becomes "`register8 PIECE register4`". Note that the register and memory sizes are preserved to insure that the model can do precise predictions about type sizes. Let $\mathbb{F}_w$ be the forward slice and and $\mathbb{B}_w$ be the backward slice of an argument, and let *Pi* be the i^{th} P-Code instruction in the slice. The forward and backward slices with "N" P-Code instructions are depicted as follows:

$$\mathbb{F}_w(Forward\ Slice) := \{P_1, P_2, \ldots P_N\}$$
$$\mathbb{B}_w(Backward\ Slice) := \{P_1, P_2, \ldots P_N\}$$

The sentence utilized for the function parameter prediction consists of only forward slice ($\mathbb{S}_p = \mathbb{F}_w$), while the sentence utilized for the callsite argument prediction consists of a combination of forward and the backward P-Code slices ($\mathbb{S}_a = \mathbb{F}_w \cup \mathbb{B}_w$). Consequently, we train two separate models - one for function parameter type detection and the other for callsite argument type detection.

4.1.3 Tokenization

Tokenization serves as an important preprocessing step for Natural Language Processing (NLP) tasks that helps extracts features for model training. It involves breaking down the text into smaller units called *tokens*. We use the pretrained "*DistilBertTokenizerFast*" tokenizer for *"distilbert-base-uncased"* that uses *WordPiece* subword tokenization algorithm to achieve high training efficiency. For instance, given the sentence like "`ram8 MULTIEQUAL ram8`", the tokenizer converts it into subword tokens "ram", "##8", "multi", "##e", "##qual", "ram" and "##8". The sentences are tokenized (Figure 2 (D)) and are then fed to the model.

4.1.4 Training the DistilBERT Model

The "input ids", i.e. numerical representation of tokenized sentences ($\mathbb{S}_a$/$\mathbb{S}_p$) along with the associated attention mask are then provided to the DistilBERT model for representation learning (Figure 2 (E)). For training, we use the "*DistilBertForSequenceClassification*" model architecture adapted by incorporating "distilbert-base-uncased" from Hugging Face Transformers library. The model is specifically designed for sequence classification task and comprised of the base DistilBERT model along with a classification layer on top that maps the output of the DistilBERT model to the output labels (types). Apart from input sentences, *true type labels* are also passed to the model for the sentence classification task.

4.1.5 Prediction generation

The final stage of our framework involves generating predictions from the models trained in the previous stage to collect type characteristics. Figure 2 (F) shows "True labels" and "Predicted Labels" for each argument. The predicted types are gathered and mapped to the corresponding function parameters or callsite arguments. At this stage the evaluator can compare the type recovery results over different architectures. The results can then be re-introduced in Ghidra's analysis routine to improve the types recovered by Ghidra. Note that the recovered types can be easily integrated in other SRE tools such as IDA Pro.

5 TRAINING/TESTING SETUP

In this section we explain the specifications and process for training and testing our model to predict function and callsite parameter/argument types.

Table 1: Benchmark Statistics show total number of parameters/arguments used during training of our primary models

	x64	**x86**	**MIPS**	**ARM**
#Total Binaries	8460	8460	8460	8460
#Function Pars	922,720	800,640	996,640	1,126,144
#Callsite Args	1,862,016	1,482,912	2,006,016	2,149,216

5.1 Dataset

We use a part of BinKit dataset introduced by (Kim et al., 2022). The dataset consists of 51 projects from GNU software such as binutils, coreutils, gzip, etc. The binaries from the dataset contain *DWARF* debug symbols information, which is essential in establishing the ground truth. We collect 33840 binaries in total (8460 binaries per architecture) that are compiled using different version of *GCC* and *LLVM* compilers for a combination four different architectures (x86-64, x86-32, ARM-64 and MIPS-64), and are compiled using four optimization levels (**O0-3**). Table 1 shows the number of function parameters and callsite arguments leveraged from our primary benchmark set for training our models. As there is a lack of standard dataset to test industry standard binary analysis techniques, to compare with previous techniques, we either use their own dataset or we train and run their models on our dataset for the evaluation.

5.2 Model Hyperparameters

We incorporate *distilbert-base-uncased*, the most popular version of DistilBERT transformer architecture to train our model. It uses a 6-layer transformer encoder with the default sequence length of 512 tokens. The number of attention heads are set to 12 and the hidden size is set to 768. We use *GELU* activation function according to the standard *DistilBertConfig*. The model is trained using 32 batch size and learning rate of 2×10^{-5}, and trained by employing mixed precision training. All the models are trained for 20 epochs. *AdamW* optimizer is used with the default weight decay of 0.01. Dropout of 0.2 is used for sequence classification head.

5.3 Experimental Setup

Our experiments are performed on 4GHz i7-6850K Linux servers with 64 GB memory and two NVIDIA TITAN Xp GPUs and took around 91 hours and 188 hours to run on average for Task-1 and Task-2 respectively for 20 epochs.

6 EVALUATION

We explain our evaluation methodology, results and observations in this section. We answer following research questions (**RQs**) during our evaluation.

1. **RQ1.** How does our technique perform in terms of Accuracy, Precision and Recall for Task-1 and Task-2.
2. **RQ2.** How does the technique perform with regard to the recovery of each type.
3. **RQ3.** How does the technique perform compared to state-of-the-art rule based techniques.
4. **RQ4.** How does the technique perform compared to the state-of-the-art deep learning techniques.

6.1 RQ1. Overall Accuracy of YĀLI

We assess the accuracy of our technique with 80-10-10 train, validation and test split on the complete dataset for each selected architecture. Given the set of classes (types) $C = \{c_1, c_2, c_3, ..., c_i\}$, we calculate the Precision, Recall and F1 scores to measure the performance of YĀLI for each type as shown below:

$$Precision_{c_i} = \frac{TP_{c_i}}{TP_{c_i}+FP_{c_i}}, Recall_{c_i} = \frac{TP_{c_i}}{TP_{c_i}+FN_{c_i}}, F1\_Score_{c_i} = \frac{2*(Precision_{c_i}*Recall_{c_i})}{Precision_{c_i}+Recall_{c_i}}$$

Where TP_{c_i}, FP_{c_i} and FN_{c_i} are ”True Positives”, ”False Positives” and ”False Negatives” respectively for class c_i. Table 2 shows the weighted average of Precision, Recall and F1 score for each type assessed over four different computer architectures and for four optimization levels for Task-1 and Task-2. Task-1 depicts function parameter type recovery and Task-2 depicts argument type recovery at callsite, as described previously.

The overall F1 scores noted for architectures x86-64, x86, MIPS and ARM are around 97%, 93%, 94% and 95% respectively for Task-1 and 93%, 90%, 92% and 94% respectively for Task-2. Note that the number of sentences (arguments and parameters) recovered (indicated by the *Support* rows in Table 2) are higher at low optimization levels. In some cases, the difference in the number of recovered sentences is twice as high for optimization `O0` than at optimization `O3`. These discrepancies may arise due to compiler optimizations such as function inlining, dead code elimination, etc. However, this difference has a minimal impact on the overall F1 scores. In general, the F1 scores are only slightly higher for binaries with low optimization levels. Even this trend is not always followed, as seen with the ARM model for Task-2. Thus, we find that our models achieve high precision and effectiveness for binary-level type detection across various architectures and optimization levels.

6.2 RQ2. On the Accuracy of YĀLI for the Recovery of Each Type

We employ YĀLI’s type system, as outlined in Figure 5, to implement YĀLI’s prediction scheme. We consider preliminary types such as boolean, character, integer and floating point types; and complex types such as arrays, structs, enums and unions. In addition to that we also consider multi-level pointer types. Thus, we design our models to employ more fine-grained types than most of the previous machine learning based type recovery techniques. For example, DEBIN (He et al., 2018) does not consider floating point types, and signedness of integral types is excluded by EKLAVYA (Chua et al., 2017), while TypeMiner (Maier et al., 2019) does not consider precise

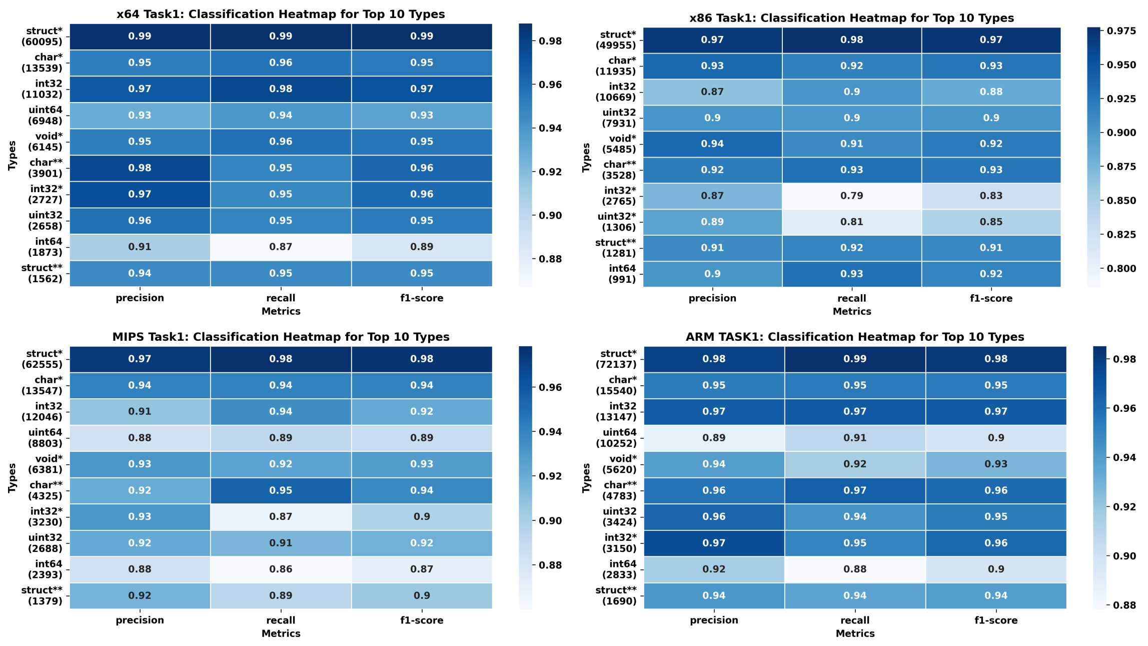


Figure 3: Heatmap for Metrics (Precision, Recall and F1 Score) collected over Top 10 types for Task-1 considering all 4 architectures - x86-64, x86, MIPS and ARM

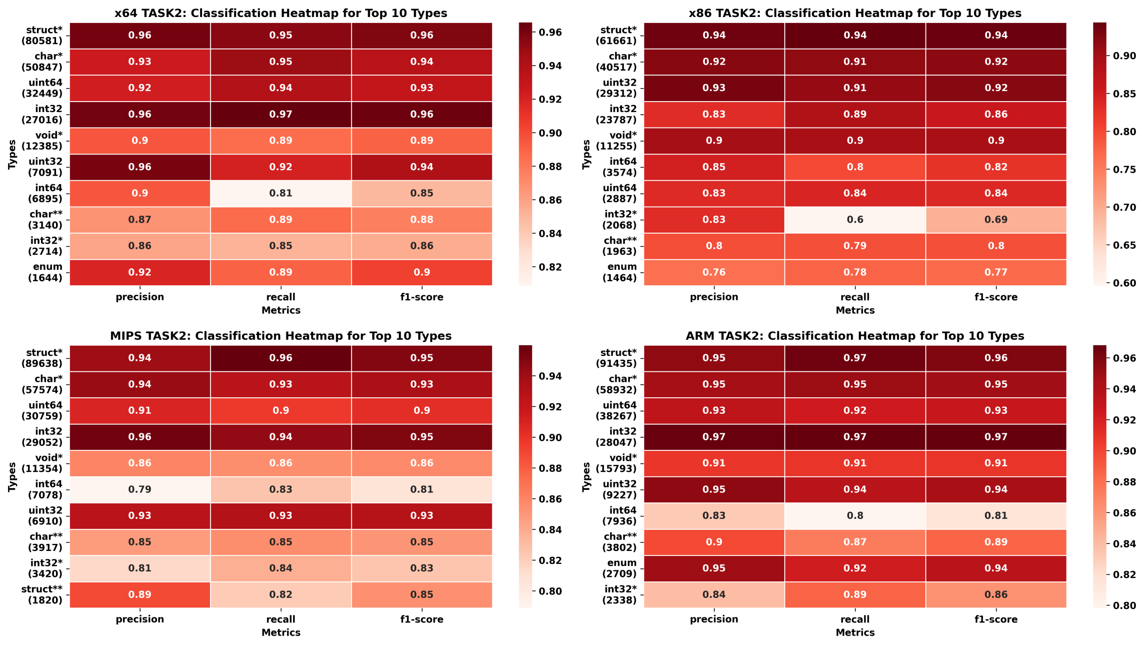


Figure 4: Heatmap for Metrics (Precision, Recall and F1 Score) collected over Top 10 types for Task-2 considering all 4 architectures - x86-64, x86, MIPS and ARM

Table 2: Evaluation results display the weighted Precision, Recall and F1 Score considering each type and computed over x86-64, x86, MIPS and ARM architectures for Task-1 and Task-2

		Architectures															
		x64				**x86**				**MIPS**				**ARM**			
Metrics	**Task**	**O0**	**O1**	**O2**	**O3**	**O0**	**O1**	**O2**	**O3**	**O0**	**O1**	**O2**	**O3**	**O0**	**O1**	**O2**	**O3**
Precision	**Task-1**	96.89	97.14	97.00	96.17	94.50	93.21	92.48	92.29	95.26	93.55	92.87	92.53	95.82	95.78	95.58	94.33
	Task-2	94.08	93.37	93.05	93.46	91.37	90.08	89.28	89.93	93.40	91.30	91.75	92.72	93.77	93.56	94.08	94.49
Recall	**Task-1**	96.88	97.12	96.97	96.17	94.49	93.23	92.49	92.32	95.29	93.67	93.06	92.60	95.88	95.84	95.71	94.59
	Task-2	94.08	93.37	93.06	93.43	91.34	90.07	89.24	89.84	93.38	91.26	91.76	92.71	93.81	93.59	94.10	94.51
F1 Score	**Task-1**	96.87	97.12	96.98	96.16	94.46	93.19	92.45	92.27	95.25	93.58	92.93	92.53	95.84	95.79	95.62	94.43
	Task-2	94.06	93.34	93.01	93.43	91.29	90.04	89.21	89.83	93.36	91.25	91.73	92.69	93.77	93.56	94.07	94.49
Support	**Task-1**	37898	35216	21213	21017	41955	24387	17065	16676	38665	33572	26411	25933	49344	37939	27398	26089
	Task-2	79865	64075	43984	44832	72111	42950	34308	35996	89169	62479	48591	50517	101169	65878	50466	51143

```
<size> ::= 8 | 16 | 32 | 64 | 128

<sign> ::= signed | unsigned

<base> ::= void | char | bool | <sign>int<size> |
           <sign>float<size>

<comp> ::= Array<<base>> | Array<<ptr>> | struct |
           enum | union

<ptr>  ::= <base>* | <comp>* | <ptr>*

<type> ::= <base> | <Comp> | <ptr>
```

Figure 5: YĀLI Type System

array types such as "`array<char*>*`". Our technique also recovers multi-level pointers unlike most previous techniques (Chua et al., 2017; He et al., 2018; Pei et al., 2021). However, we do not distinguish between const and non-const types, and unlike DIRTY do not recover structure member types.

In order to understand how YĀLI behaves, it is important to dissect the individual type recovery results. Figures 3 and 4 depict the Precision, Recall and F1 score of each type over our four tested architectures. The technique shows high accuracy across all the architectures for both the tasks and across most types. The type "`struct*`" is the most frequently observed type across all architectures, occurring more than four times as frequently as the second most common type "`char*`" for Task-1. Additionally, "`struct*`" is the most common type for Task-2 as well. All the heatmaps reveal darker colors for the types with higher support, which indicates superior Precision, Recall and F1 scores for the most common types. This in turn shows that increased support leads to higher F1 scores. The varying support numbers for different types result from a random sampling of dataset used during the training. The F1 scores for the types in Task-2 across all architectures are in general slightly lower than the F1 scores in Task-1.

6.3 RQ3. Comparison with Prior Rule Based Techniques

In this section, we compare YĀLI with rule-based type recovery techniques. Although there are multiple prominent previously published type recovery techniques such as TIE (Lee et al., 2011), Howard (Slowinska et al., 2011), ARTISTE (Caballero et al., 2012), OSPREY (Zhang et al., 2021) etc. none of these techniques are available in open source. Authors of *OSPREY* (Zhang et al., 2021) generously provided us their dataset consisting of binaries from coreutils package with `O0` optimization level (100 binaries in total). OSPREY uses a probabilistic constraint solving technique for type recovery. However, OSPREY only considers stack and heap allocated variables, and cannot recover register allocated variables. Therefore, OSPREY cannot be used to recover function parameters and callsite arguments that are typically passed in registers. We also reviewed SRE frameworks such as IDA Pro (hexrays, 2023) and Ghidra (National Security Agency ghidra, 2023) as they perform type recovery to facilitate decompilation. However, we do not have access to multi-architecture support for IDA Pro's Hex-Rays decompiler. Ultimately, we compare the accuracy of our technique with two open-source rule-based type detection mechanisms, the Ghidra Decompiler and *Retypd* (Noonan et al., 2016).

We use the open source version of Retypd [3] and use their Ghidra plugin *GhidraRetypd* [4] by converting it into headless mode to integrate recovered types back into Ghidra analysis. None of the previous techniques analyzes Ghidra in terms of function and callsite argument type recovery. We found that the execution and inference time of Retypd is excessively long, that is why for brevity, we test accuracy of Retypd and Ghidra on "coreutils" binaries (total 3748) for our x86-64 dataset. We take out 32 binaries on which Retypd failed to work. To assess and compare the performance of YĀLI, we train a new model (de-

[3]https://github.com/GrammaTech/retypd

[4]https://github.com/GrammaTech/retypd-ghidra-plugin

noted as YĀLI$_\gamma$) by removing all "coreutils" binaries from our original dataset. This also allows us to systematically assess the effectiveness and competence of our model on a dataset completely orthogonal from the train set.

Table 3: Accuracy of YĀLI$_\gamma$, Ghidra and Retypd is shown for Task-1 for and x86-64 binaries with four optimization levels

	O0	**O1**	**O2**	**O3**
YĀLI$_\gamma$ %	73.00%	74.74%	77.92%	80.64%
Retypd %	28.52%	27.57%	26.01%	28.24%
Ghidra %	31.14%	38.17%	36.96%	41.92%

Table 4: Accuracy of YĀLI$_\gamma$, Ghidra and Retypd is shown for Task-2 for and x86-64 binaries with four optimization levels

	O0	**O1**	**O2**	**O3**
YĀLI$_\gamma$ %	69.47%	71.86%	73.12%	81.10%
Retypd %	33.92%	33.72%	27.02%	41.67%
Ghidra %	24.42%	25.33%	23.55%	24.47%

To accurately compute the accuracy, we convert Ghidra and Retypd types in post-hoc manner to make them compatible with the source types. For e.g. "`struct_100*`" is replace with "`struct*`". And complex types such as "`array[10]`" are converted to their simplistic form such as "`array`". Tables 3 and 4 shows the accuracy of YĀLI$_\gamma$, over four optimization levels for Task-1 and Task-2. Overall, YĀLI$_\gamma$ achieves 75.48% accuracy for Task-1 and 73.58% accuracy for Task-2. Note again that the YĀLI$_\gamma$ model achieves lower accuracy than the YĀLI model seen earlier in Table 2 because we evaluate YĀLI$_\gamma$ on an orthogonal benchmark set that was not at all represented in the training dataset. Thus, we find that our LLM based model achieves high effectiveness even in considerably adverse situations.

Retypd achieves 27.74% accuracy for the Task-1 and Ghidra shows 35.82% accuracy for the same task, beating Retypd by around 8%. On the contrary, for Task-2, Retypd and Ghidra's overall accuracy scores are 35.81% and 24.50% respectively. Thus, YĀLI$_\gamma$ achieves significantly higher overall accuracy than the other two techniques. We also found that Retypd beats Ghidra for "`Struct`" type recovery. Specifically, Retypd achieves 22% for Task-1 and 24% Task-2, while Ghidra achives 18% and 12% for Task-1 and Task-2 respectively. YĀLI$_\gamma$ attains improved accuracy for "`Struct`" type recovery as well, reaching 87% for Task-1 and 72% for task-2.

In terms of inference speed, we observe Retypd plugin takes 2 minutes 52 seconds on one of the "coreutils" binaries (*md5sum*), while Ghidra takes around 27 seconds to compute results on the same binary. YĀLI's primary model takes around 11 seconds to finish inference on complete "coreutils" dataset demonstrating greater speed scalability. We notice that Retypd sometimes takes unusually long time for certain binaries in our dataset, occasionally taking several hours to analyze.

6.4 RQ4. Comparison with Deep Learning Techniques

We now compare YĀLI with two state-of-the-art deep learning techniques - EKLAVYA and DIRTY.

6.4.1 Comparison with EKLAVYA

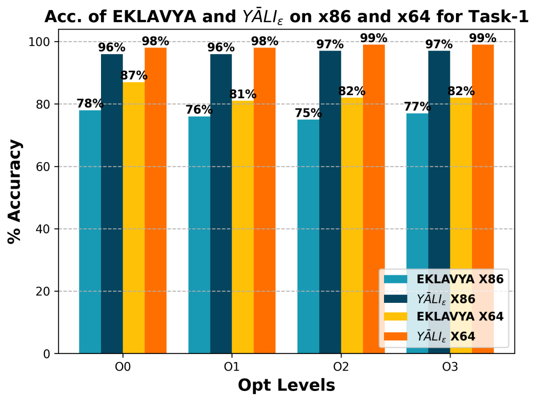
To compare with EKLAVYA we leverage EKLAVYA's publically available dataset. We train our model directly on their dataset. The dataset consists of binaries from 8 projects - binutils, coreutils, findutils, sg3utils, utillinux, inetutils, diffutils, and usbutils compiled for x86 and x86-64 architectures. The performance evaluation on the EKLAVYA benchmark set shows the reliability of our model on the relatively smaller training dataset. We refer to this model YĀLI$_\varepsilon$. To evaluate the performance of YĀLI$_\varepsilon$ against EKLAVYA, we consider seven type categories identified in EKLAVYA. The EKLAVYA technique considers each function argument's type inference as a distinct classification task. They report accuracy numbers, rather than F1 scores. Consequently, to gauge the accuracy of YĀLI$_\varepsilon$, we divide the total number of accurately predicted types by the total number of tokens. EKLAVYA considers seven different `C`-style types from the following type lattice.

$$\tau := \{int|char|float|void*|enum|union|struct\}$$

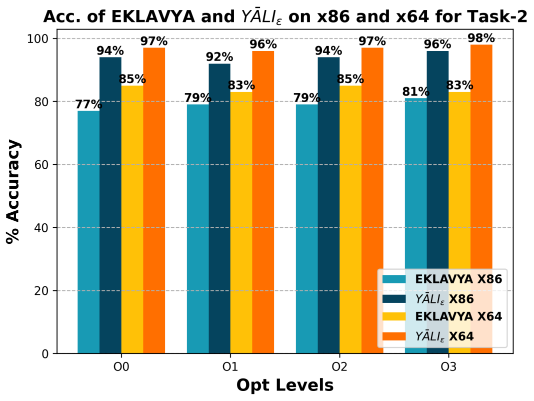
EKLAVYA focuses on acquiring function types based on callee and caller instructions i.e. at the callsite. As EKLAVYA's pre-trained model is not released, we use their datasets for training our model and for equivalent comparison. The accuracy is reported for first three arguments in EKLAVYA paper due to prevalence and better support, and we do the same for fair comparison. We train our models for 10 epochs and display the results in Figure 6. Overall our technique has achieved 20% and 15% improvement over EKLAVYA for x86 and x86-64 binaries respectively for Task-1. And we observed improvement of 15% for x86 binaries and 13% for x86-64 when compared YĀLI$_\varepsilon$ to EKLAVYA for Task-2.

6.4.2 Comparison with DIRTY

DIRTY is a deep learning technique based on transformer based NN model that enhances the quality of



(a)



(b)

Figure 6: Accuracy of EKLAVYA and YĀLI_ϵ is shown for Task-1 (subfigure (a)) and Task-2 (subfigure (b)) for x86 and x86-64 binaries over four optimization levels

decompiler output by accurately predicting variable names and types. The primary DIRTY model presented in the paper is focused on recovering variable type-name pairs, resulting in reduced type associated data. Accordingly, we use DIRTY_{Light} model customized for type recovery task. Original DIRTY paper aims to support x86-64 architecture, thus we directly train DIRTY_{Light} on our x86-64 dataset (total 8460 binaries) and collect the results. We refer to this model as DIRTY in the rest of the paper. DIRTY considers all types rather than function or callsite arguments. Therefore, we only collect function parameters statistics from generated test results from the DIRTY_{Light} to calculate accuracy for comparison, and compare the data for Task-1.

DIRTY can predict up to 48,888 syntactic and non-syntactic types. Consequently, we adapt DIRTY types and convert them into YĀLI types in post-hoc manner to make them compatible to YĀLI types that are synonymous to the DWARF Debug information. It is important to achieve this conversion with attention and accuracy, otherwise the results would suffer. Firstly, we declass the composite types such as "`union __WAIT_STATUS {wait* __uptr;int* __iptr;}`" to its primordial form "`union`" as YĀLI doesn't detect union or structure member hierarchy. DIRTY contemplates the types recovered from IDA pro's Hex-Rays decompiler for ground truth comparison, rather than the actual syntactic types intended by the programmer. The types are imported using IDA pro to establish the baseline. Thus, it may not accurately identify the user-defined types. For e.g. types such as `_BYTE` are defined by IDA and do not represent actual types declared in the source code. Secondly, we also convert types such `gl_list_impl*` to `struct*` as it obviously signifies structure type in the program. Lastly, we observe numerous parameters marked with "*disappear*" label - "*disappear*" are the types that signify lost types during the process of decompilation. To confirm this tendency, we also trained another DIRTY model by replacing IDA Pro by Ghidra decompiler, yet we still continued to notice the missing types. We decided to remove such types when we calculate and compare DIRTY's overall accuracy, which comes with at a cost of lesser number of samples.

Table 5: Accuracy of DIRTY and YĀLI is shown for Task-1 for x86-64 binaries over four optimization levels

	O0	O1	O2	O3
DIRTY %	83.36%	58.40%	61.61%	64.53%
YĀLI %	96.87%	97.12%	96.97%	96.17%

Table 5 compares YĀLI with DIRTY using x86-64 binaries compiled over four optimization levels and presents the percentage accuracy for Task-1. The overall accuracy of DIRTY's type recovery technique for function parameter types is 83.36% (highest) for opt. `O0` and 58.40% (lowest) for opt. `O1`. YĀLI outperforms DIRTY in all optimization levels having 96.17% accuracy for opt. `O2` (lowest) to 97.12% for opt. `O1` and shows optimal performance. We would like to note that the effectiveness of DIRTY model depends on the dataset utilized during its training. The dataset we use to train DIRTY's model is smaller - 8460 binaries compared to 4,346,134 compiled binaries in actual the dataset used in original work. And, therefore it may affect the overall accuracy, since the model is exposed to fewer samples during training. However, this showcases a compelling evidence that our model shows strong performance, despite having trained on a comparatively smaller dataset.

7 CONCLUSION

Function and callsite parameter/argument type recovery is important as it lays a foundation for numerous binary analysis and security tasks. We develop a novel framework, called YĀLI, that approaches this binary type recovery problem as a NLP problem, and helps achieve this task using a transformer based LLM. We conduct a thorough evaluation that shows that our technique is portable across multiple processor architectures, and significantly outperforms many existing state-of-the-art mechanisms for binary-level type recovery. We anticipate that our technique will provide a novel perspective on understanding and addressing the task of type recovery, and assist researchers that rely on accurate type recovery for their analysis and security tasks.

REFERENCES

Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. (2005). Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, New York, NY, USA. Association for Computing Machinery.

Balakrishnan, G. and Reps, T. (2004). Analyzing memory accesses in x86 executables. In *International conference on compiler construction*, pages 5–23. Springer.

Bao, T., Burket, J., Woo, M., Turner, R., and Brumley, D. (2014). {BYTEWEIGHT}: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 845–860.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Brumley, D., Lee, J., Schwartz, E. J., and Woo, M. (2013). Native x86 decompilation using Semantics-Preserving structural analysis and iterative Control-Flow structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 353–368, Washington, D.C. USENIX Association.

Burk, K., Pagani, F., Kruegel, C., and Vigna, G. (2022). Decomperson: How humans decompile and what we can learn from it. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2765–2782, Boston, MA. USENIX Association.

Burow, N., Carr, S. A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., and Payer, M. (2017). Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33.

Caballero, J., Grieco, G., Marron, M., Lin, Z., and Urbina, D. (2012). ARTISTE: Automatic generation of hybrid data structure signatures from binary code executions.

Caballero, J. and Lin, Z. (2016). Type inference on executables. *ACM Computing Surveys (CSUR)*, 48(4):1–35.

Chen, Q., Lacomis, J., Schwartz, E. J., Goues, C. L., Neubig, G., and Vasilescu, B. (2022). Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4327–4343, Boston, MA. USENIX Association.

Chua, Z. L., Shen, S., Saxena, P., and Liang, Z. (2017). Neural nets can learn function type signatures from binaries. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 99–116, Vancouver, BC. USENIX Association.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805.*

Han, H., Kyea, J., Jin, Y., Kang, J., Pak, B., and Yun, I. (2023). Queryx: Symbolic query on decompiled code for finding bugs in cots binaries. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3279–312795.

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

hexrays (2023). https://hex-rays.com/ida-pro/. In *Interactive Disassembler (IDA).*

Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531.*

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Katz, O., El-Yaniv, R., and Yahav, E. (2016). Estimating types in binaries using predictive modeling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 313–326.

Kim, D., Kim, E., Cha, S. K., Son, S., and Kim, Y. (2022). Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Trans. Softw. Eng.*, 49(4):1661–1682.

Lee, J., Avgerinos, T., and Brumley, D. (2011). TIE: principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6-9 February 2011.*

Li, X., Yu, Q., and Yin, H. (2021). Palmtree: Learning an assembly language model for instruction embedding.

Lin, Y. and Gao, D. (2021). When function signature recovery meets compiler optimization. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 36–52. IEEE.

Lin, Z., Zhang, X., and Xu, D. (2010). Automatic reverse engineering of data structures from binary execution. West Lafayette, IN. CERIAS - Purdue University.

Lu, K. and Hu, H. (2019). Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1867–1881, New York, NY, USA. Association for Computing Machinery.

Maier, A., Gascon, H., Wressnegger, C., and Rieck, K. (2019). Typeminer: Recovering types in binary programs using machine learning. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, pages 288–308. Springer.

Mantovani, A., Compagna, L., Shoshitaishvili, Y., and Balzarotti, D. (2022). The convergence of source code and binary vulnerability discovery – a case study. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, page 602–615, New York, NY, USA. Association for Computing Machinery.

Muntean, P., Fischer, M., Tan, G., Lin, Z., Grossklags, J., and Eckert, C. (2018). τcfi: Type-assisted control flow integrity for x86-64 binaries. In *Research in Attacks, Intrusions, and Defenses*, pages 423–444, Cham. Springer International Publishing.

National Security Agency ghidra, N. (2023). Ghidra. https://ghidra-sre.org/.

Naus, N., Verbeek, F., Walker, D., and Ravindran, B. (2023). A formal semantics for p-code. In *Verified Software. Theories, Tools and Experiments.: 14th International Conference, VSTTE 2022, Trento, Italy, October 17–18, 2022, Revised Selected Papers*, page 111–128, Berlin, Heidelberg. Springer-Verlag.

Nitin, V., Saieva, A., Ray, B., and Kaiser, G. E. (2021). Direct : A transformer-based model for decompiled identifier renaming. In *NLP4PROG*.

Noonan, M., Loginov, A., and Cok, D. (2016). Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 27–41.

Pei, K., Guan, J., Broughton, M., Chen, Z., Yao, S., Williams-King, D., Ummadisetty, V., Yang, J., Ray, B., and Jana, S. (2021). 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*, pages 690–702.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2019). Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*.

Saxena, P., Sekar, R., and Puranik, V. (2008). Efficient fine-grained binary instrumentation with applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 74–83.

Slowinska, A., Stancescu, T., and Bos, H. (2011). Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society.

van der Veen, V., Göktas, E., Contag, M., Pawoloski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E., and Giuffrida, C. (2016). A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 934–953.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Wang, H., Qu, W., Katz, G., Zhu, W., Gao, Z., Qiu, H., Zhuge, J., and Zhang, C. (2022). Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–13.

Williams-King, D., Kobayashi, H., Williams-King, K., Patterson, G., Spano, F., Wu, Y. J., Yang, J., and Kemerlis, V. P. (2020). Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 133–147.

Wong, W. K., Wang, H., Li, Z., Liu, Z., Wang, S., Tang, Q., Nie, S., and Wu, S. (2023). Refining decompiled c code with large language models. *arXiv preprint arXiv:2310.06530*.

Zhang, Z., Ye, Y., You, W., Tao, G., Lee, W.-c., Kwon, Y., Aafer, Y., and Zhang, X. (2021). Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 813–832. IEEE.

Zhu, W., Wang, H., Zhou, Y., Wang, J., Sha, Z., Gao, Z., and Zhang, C. (2023). ktrans: Knowledge-aware transformer for binary code embedding. *arXiv preprint arXiv:2308.12659*.