ABSTRACT

Binary type inference is a critical reverse engineering task supporting many security applications, including vulnerability analysis, binary hardening, forensics, and decompilation. It is a difficult task because source-level type information is often stripped during compilation, leaving only binaries with untyped memory and register accesses. Existing approaches rely on hand-coded type inference rules defined by domain experts, which are brittle and require nontrivial effort to maintain and update. Even though machine learning approaches have shown promise at automatically learning the inference rules, their accuracy is still low, especially for optimized binaries.

We present STATEFORMER, a new neural architecture that is adept at accurate and robust type inference. STATEFORMER follows a two-step transfer learning paradigm. In the pretraining step, the model is trained with Generative State Modeling (GSM), a novel task that we design to teach the model to statically approximate execution effects of assembly instructions in both forward and backward directions. In the finetuning step, the pretrained model learns to use its knowledge of operational semantics to infer types.

We evaluate STATEFORMER’s performance on a corpus of 33 popular open-source software projects containing over 1.67 billion variables of different types. The programs are compiled with GCC and LLVM over 4 optimization levels and 3 obfuscation passes based on LLVM. Our model significantly outperforms state-of-the-art ML-based tools by 14.6% in recovering types for both O0 and LLVM over 4 optimization levels. Our model significantly outperforms state-of-the-art ML-based tools by 14.6% in recovering types for both O0 and LLVM over 4 optimization levels.

STATEFORMER improves type inference accuracy by 33%.

1 INTRODUCTION

Recovering source-level data types from binaries is very useful for many security-critical software engineering tasks, such as vulnerability analysis [18, 37, 45], binary hardening [30, 53, 57, 74, 96, 99, 101], memory introspection [43, 89], and decompilation [4, 26]. Type inference in binaries involves reconstructing source-level constructs, such as local function variables and data types, from untyped byte-addressed memory and registers. This process is challenging because the reconstruction is based on incomplete information – most source-level information is stripped during compilation for optimization and to deter reverse engineering.

Traditional approaches to type inference rely extensively on hand-coded rules defined by domain experts. These rules facilitate (1) recognizing types directly from specified patterns (e.g., consecutive printable characters for detecting strings); and (2) propagating types from known type sinks (e.g., known string manipulation functions) to registers and memory regions storing the source-level variables. Unfortunately, these rules are brittle [16, 66] and require continuous
effort to adapt to new instruction sequences introduced by compiler and architecture evolution [10].

As a result, recent years have witnessed a growing interest in data-driven approaches leveraging Machine Learning (ML) for binary type inference [20, 40, 59]. These approaches mitigate the reliance on hand-coded heuristics by learning from a rich training set of diverse binaries. Moreover, their learned representations have been shown to generalize across various compilers, operating systems, and architectures and are highly efficient to compute (i.e., the underlying learning algorithms are amenable to GPU parallelization).

While promising, existing ML-based approaches still cannot recover data types with high accuracy or robustness, especially in the presence of compiler optimizations [20, 40]. The types are essentially abstractions describing *how a data object is expected to be manipulated and used during execution*. Therefore, the inherent challenge faced by all ML-based approaches is to understand the effects of the runtime execution of instructions in the target binary [55, 83], i.e., the operational semantics of code blocks [64]. For example, on x64, the runtime effect of iterative increments of the `rcx` register by 1 together with the instruction `mov rax, [rdx+rcx*4]` is indicative of traversing an `int` array.

Unfortunately, existing ML-based approaches are agnostic to the execution effects of code as they learn the direct mapping between static code tokens and corresponding types in an end-to-end fashion. A model trained this way often learns spurious correlations [7], taking shortcuts to leverage simple yet brittle patterns for inferring types. For example, Chua et al. [20] showed that their model, EKLAVYA, mispredicts the type of the argument to the function `ck_fopen` from Diffutils to be integer instead of pointer. A completely unrelated instruction within `ck_fopen` (namely `callq Oxf3fc`) contributes the most to the misprediction. Without understanding how an integer is accessed and manipulated during execution and the effects of `callq`, EKLAVYA establishes a spurious correlation that the internal call instruction implies an `int` argument to `ck_fopen`.

In this paper, we present STATEFORMER, a new neural architecture that explicitly learns the operational semantics of assembly for type inference. Specifically, we design a novel pretraining task to teach the STATEFORMER model the operational semantics of both *data and control* flow behavior of diverse code blocks, and then finetune the pretrained model for type inference.

**Learning operational semantics.** A human reverse engineer often makes sense of a target binary by following its assembly instructions through *mental simulation* of their execution. While the reverse engineer might not accurately resolve all invoked branches by following control flow or compute the precise values of all states by following data flow during simulation, she can still get a rough idea of what the code does by approximately following the operational semantics of code blocks. Our key insight is to teach the STATEFORMER model, via a novel pretraining task, the approximate operational semantics of assembly by forcing the model to predict how different sequences of instructions *transform the underlying program states*. Specifically, the pretraining task asks the model to predict the changed values of registers and memory after executing each instruction, which captures the operational semantics of assembly code [29, 64, 70]. This gives the model an understanding of the execution effects of code, which helps the model to infer the types of low-level registers and memory regions based on the instructions used to manipulate them without executing any parts of the code during inference.

**Generative State Modeling.** We design a novel pretraining task, Generative State Modeling (GSM), where we train a neural network to reconstruct the complete set of its execution states while taking the assembly code and a very small subset of its execution states (e.g., register values at specific program points) as input. For example, given the instruction sequence: `inc ecx; add ecx, 3; xor ecx, ecx; mov ebx, ecx;` and its corresponding execution states `ecx=0; ecx=1; ecx=4; ecx=0; (ebx=0, ecx=0)`, we feed the model with all the instructions and only the execution state after the second instruction, i.e., `ecx=4`. Our training process forces the model to compute all the preceding and succeeding states: `ecx=0; ecx=1;` and `ecx=0; (ebx=0, ecx=0)`. Therefore, to achieve low loss on the GSM task, the model needs to understand the operational semantics of `inc, add, xor, and mov`.

GSM dynamically selects random subsets of states as inputs across different training samples and iterations. Moreover, GSM is fully self-supervised [24], implying that we can collect data from an unrestricted number of binaries found in the wild. As a result, GSM creates diverse prediction tasks that compel the model to approximately reason about the effects of both *data and control* instructions, in both forward and backward directions – a critical capability for type recognition and propagation [55, 83]. During pretraining with GSM, STATEFORMER encodes such a reasoning capability as part of its network parameters, known as embeddings. Such embeddings can then be *finetuned* for type inference as a finetuning task with a few binaries with labeled types.

Consider again the example of inferring the traversal of an `int` based on iterative increments of the `rcx` register by 1 together with the instruction `mov rax, [rdx+rcx*4]`. The output of pretrained STATEFORMER will be a sequence of embeddings encoding the effects of `inc, mov` on other registers and memory locations. Therefore, instead of training on raw code sequences from scratch, the finetuning process can easily exploit the learned execution effects of code compressed in these embeddings to predict that `rdx` contains the base address of an `int` array.

**STATEFORMER neural architecture.** To efficiently pretrain with GSM, we develop a novel neural architecture specifically designed for learning operational semantics of assembly instructions. First, as the model takes as input both the program code and program states, we develop a *multi-modal encoding module* that can be trained on heterogeneous inputs in different formats.

Second, we construct two explicit objective functions to jointly optimize the model to understand the operational semantics of both *data flow* and *control flow*. Specifically, to help the model learn about control-flow, which requires learning operational semantics of comparison instructions (e.g., `cmp`), we annotate the non-executed paths with dummy program states to incorporate predicting non-executed paths as a part of the STATEFORMER’s pretraining task. To help the model to better understand the operational semantics of data flow, which often involves assignment (e.g., `mov`) and arithmetic instructions (e.g., `add`) on numerical values, we explicitly model the numerical values in both decimal and hexadecimal formats with a trainable *numerical representation module* based on the neural arithmetic unit (NAU) [58].
Finally, as the composite execution effects of a piece of code can result from the interactions between faraway instructions, we leverage self-attention layers from Transformer [93], which is amenable to learning long-range dependencies without manually constructing the dependencies (e.g., graph neural net [63, 97]). We show in Section 5.5 that such a design indeed achieves high testing accuracy in GSM for unseen program state traces.

**Result summary.** We evaluate STATEFORMER on a corpus of 33 popular open-source software projects with 1.67 billion source variables of different types. The programs are compiled for 4 instruction set architectures (x86, x64, ARM, and MIPS), by 2 compilers (GCC and LLVM), and with 4 optimization levels (O0-O3) and 3 obfuscations. By training with GSM, our model outperforms the state-of-the-art ML-based tools by up to 14.6% in recovering types for both function arguments and variables. Our extensive ablation studies show that STATEFORMER trained with GSM substantially boosts the type inference accuracy by 33%. We make the following contributions.

- We propose a new pretraining task, Generative State Modeling (GSM), to explicitly learn the operational semantics of assembly code for accurate and robust type inference.
- We develop a novel neural architecture, STATEFORMER, with specially designed sub-modules to learn the operational semantics of both data flow and control flow instructions.
- We evaluate STATEFORMER on an extensive collection of 33 open-source software projects across different architectures, compilers, optimizations, and obfuscations. After training with GSM, STATEFORMER outperforms the state-of-the-art learning-based tools by 14.6%. Our ablation studies unveil that training with STATEFORMER boosts the type inference accuracy by 33%. We release the code and datasets of STATEFORMER at https://github.com/CUMLSec/stateformer.

## 2 OVERVIEW

The high-level workflow of STATEFORMER follows the general transfer learning paradigm. As shown in Figure 1, we first pretrain STATEFORMER with GSM by training it to reconstruct the masked states (grayed-out) in the trace of program states of various assembly instructions (Section 2.3). We train STATEFORMER to reconstruct both the data and control states (Section 2.4). After pretraining STATEFORMER with GSM, we transfer its learned knowledge by finetuning on the type inference task (defined in Section 3.4).

### 2.1 Problem Definition

We consider the problem of mapping untyped low-level registers or memory regions (specified by memory offsets) to the corresponding source-level types. The source-level types are associated with function arguments, local, static, and global variables. The granularity of recovered source-level types varies widely across existing works [16], ranging from primitive (e.g., int, float) and aggregate (e.g., struct, array) types to classes in object-oriented programs and recursive types such as trees and lists.

We focus on the standard C primitive, aggregate, and pointer types. Our supported types are more fine-grained than prior works [20, 40, 59], which only support a strict subset of ours (see Section 3.4 for the complete list of types we support). Predicting fine-grained types, while very helpful to the human reverse engineer to better understand the target binary, is a challenging learning task that must distinguish between subtly different access patterns of different types [83].

We formulate type inference as a classification task. Specifically, given a sequence of assembly instructions, STATEFORMER predicts the type labels for each operand in the instructions. Note that STATEFORMER performs the type prediction in one shot (see Section 3 for design specifics), as opposed to the traditional type propagation approaches that infer the types one-by-one in a sequence of instructions. As we show in Section 5.3, this design brings significant performance gains during inference.

### 2.2 Understanding Operational Semantics Helps Type Inference

While reverse engineering types from binaries, human analysts need to understand what a target function computes without actually executing the binary. Often the analyst follows the assembly instructions by simulating the execution in their mind. Without knowing the exact initial program state during the function call, the analyst cannot accurately resolve the taken branches or compute the precise value of all states during the simulation. Still, they can get a rough idea of what the code does. This loose approximation of the operational semantics of the code allows the understanding of its runtime behavior, providing strong hints about the underlying data types [83].

For instance, given a pointer a, observing a dereference like *(a+4) in the execution behavior might imply a 4-byte read of the object at a, indicating an int or a pointer type on 32-bit systems. Similarly, contiguous dereferencing of sequential addresses like *(a), *(a+1), suggests that a is likely an array of chars. Examining precise runtime behaviors of a binary over many inputs with high-coverage dynamic analysis is prohibitively expensive [51, 83, 87]. Therefore, in this paper, we use ML models to learn approximate operational semantics of binaries in a data-driven manner and use this knowledge to statically infer types.

### 2.3 Learning Operational Semantics with GSM

Our key motivation for developing GSM is to teach an ML model to approximate operational semantics of code, i.e., its execution effects, which we then exploit for type inference. Teaching an ML model the code execution effects is challenging due to many possible combinations of instructions that introduce complex data and control flow dependencies. Therefore, it is not practical to manually engineer input features or target labels to represent the execution effects and train the model to understand them. To this end, GSM explores a self-supervised approach that exploits a large number of traces that can be cheaply generated from many code blocks using under-constrained dynamic execution, e.g., micro-execution (detailed in Section 3.1), to automate learning diverse instructions’ execution effect with a carefully-designed training task.

**Predicting masked states.** The training task performed in GSM requires a neural network to reconstruct the whole micro-execution traces (i.e., all recorded program states) in the training data based on the corresponding code blocks. To learn on a huge number of traces, we exploit stochasticity to efficiently train a network for GSM. Specifically, for each training sample in each epoch, we randomly mask (i.e., remove) some states in the traces. Such randomness
ensures that the model cannot consistently achieve low loss by taking shortcuts that only work well for a few states, traces, or code blocks. While deciding which states to mask, GSM does not follow the sequential execution order of states as recorded in the traces. This design choice ensures that the model learns to reason about both forward and backward execution effects of a diverse set of code blocks. Understanding these forward and backward dependencies is known to be crucial for accurate type inference [55, 83].

Difference with masked language models. While our stochastic masking setup is inspired by the Masked Language Modeling (MLM) used in learning natural language semantics [24], the key difference is that natural languages are not stateful, i.e., they have no notion similar to program execution. Therefore, the model trained by MLM only uses unmasked words in the neighboring context to predict the masked words, exploiting the common local word phrase patterns. While in GSM, the unmasked states alone provide little observable patterns due to high masking rate – the model has to also look at the corresponding instructions, understand their execution effects on the unmasked states, in order to correctly predict the masked states.

2.4 STATEFORMER Architecture

**Learning instruction-state dependencies.** Achieving low loss on GSM, by design, requires a neural network to understand the long-range dependencies between instructions and unmasked program states. However, standard fully-connected or recurrent networks are inefficient at learning long-range dependencies between different parts of the network inputs [61, 93, 98].

To avoid these issues, we develop a hierarchical input embedding module to learn the interactions between program states and instructions. Specifically, we design two input sub-networks for learning two embeddings of the binary code and traces – one for the registers and instruction opcodes and another for the concrete data values. We combine these representations by aggregating the embeddings with a vector addition operation and feeding them into a sequence of self-attention layers that facilitate capturing long-range dependencies [93] (Section 3.2). Finally, we use two output sub-networks to decode the output of self-attention layers for two different objectives: (1) regression for predicting the program data state and (2) classification for predicting the program control state (Section 3.3).

**Learning representations for numerical values.** Typical embeddings for numerical tokens (i.e., register values) – just like how any discrete token is embedded – are known to fail to extrapolate to unseen values even on the outputs of simple arithmetic operations like addition [88]. As understanding data and control flow often requires reasoning the execution effect of arithmetic instructions (e.g., add rax, rbx), we use Neural Arithmetic Units (NAUs) [58] as part of the subnetwork for data value embeddings. Note that our NAU layers, unlike the original NAU model that directly takes numerical values as input, learn to represent the numerical values (both decimal and hexadecimal formats) as embeddings. We have done a thorough study and refer interested readers to our supplementary material.

3 METHODOLOGY

We now provide the details of our methodology, including how we collect runtime states of binary programs, the architecture of STATEFORMER, and how we distill the learned knowledge in STATEFORMER from training GSM for type inference.

3.1 Collecting Program States

To train STATEFORMER with GSM, we need to obtain runtime execution traces of binary programs. Ideally, we want to collect diverse traces with different instructions and control flow to learn miscellaneous operational semantics for type inference in various scenarios. However, the typical dynamic analysis approach is often limited by path coverage, resulting in potentially restricted sets of covered instructions. Therefore, we adopt micro-execution [33] to support tracing arbitrary parts of a binary program without having to find concrete program inputs that maximize coverage.

Without executing the program from its entry point, our execution engine needs to initialize intermediate program states (i.e., registers and memory content) with randomized values, which can be under-constrained (i.e., infeasible when executing the program normally). In addition, we focus on program states that are explicitly manipulated in instructions (e.g., we only log the value of eax in...
We construct 5 sequences for $S$ (Section 3.2). To construct $\mu = x$ values of all registers, memory addresses, and hardcoded offsets that appear in the instruction, dubbed $\mu$Trace. We aggregate (tor) with the same dimensions, such that they can be easily aggregated into a sequence, (2) DataState, (3) instruction position and the opcode/operand position within each position, (4) opcode/operand position sequence, and (5) architecture sequence. Each token of the 5 sequences are aligned and embedded into an embedding (a learned low-dimensional vector) with the same dimensions, such that they can be easily aggregated (i.e., summation) as a single sequence of $n$ embeddings: $x = \{x_1, ..., x_n\}$. Figure 3 illustrates an example input of STATEFORMER when training on GSM.

**Encoding assembly code.** The assembly code sequence with length $n: c = \{add, ebp, ...\}$ is constructed by tokenizing the assembly instructions from disassembled binaries. Besides treating both opcodes and operands as tokens, we keep punctuations as they provide crucial contextual hints, e.g., the comma delimits the source and destination, and brackets indicate a dereference of a memory address.

Assembly code can have concrete numerical values hardcoded in instructions, which leads to a prohibitively large vocabulary size (e.g., $2^{32}$ possible values in x86), making it challenging to embed all tokens in $c$. Therefore, we place the concrete value into the $\mu$DataState sequence and replace all numerical values (in both hexadecimal and decimal forms) with a special token $\text{hex}$. This reduces the vocabulary size of $c$ across all instruction set architectures to only 648. We describe how we encode the numerical values in the following.

**Encoding $\mu$DataState.** We normalize $\mu$DataState sequence $v$ as a two-dimensional array $v = \mathcal{V}^{\text{hex}}$, where $\mathcal{V} = \{0x00, ..., 0xff\} \cup \{\text{\$}\}$ (the union of 256 bytes and a dummy token $\text{\$}$). Each $\mu$DataState $v_i$ can thus be viewed as a sequence of 8-byte tokens $\mathcal{V}^8$, where we transform all the numerical values into an 8-byte hexadecimal representation. For example, Figure 3 shows that a $\mu$DataState $0x6$ is padded to (00,00,00,00,00,00,00,06). As each $v_i$ is aligned with each token $c_i$ in code sequence, we put 8 $\text{\$}$s for those $c_i$ that do not have dynamic values (e.g., opcode).

Such a setting reduces the vocabulary size used to encode all possible numerical values from $2^{64}$ (assume 64-bit architectures) to only 257. Moreover, representing a numerical value with fixed dimensions makes it easy to stack a single learnable module (see Section 3.3) to compute inter-dependencies between digits, learning useful hierarchical knowledge (i.e., an address 0x104c might be decomposed as a section base at 0x1000 with the offset 0x4c).

**Encoding spatial information and syntactic hint.** As we flatten and concatenate all the assembly instructions as a plain code token sequence, the instruction boundaries and the relative location of tokens within each instruction become ambiguous. To this end, we introduce two positional encodings [93], namely the instruction positional encoding and opcode/operand positional encoding. The resulting instruction position sequence $p = \mathcal{Z}^p$ and opcode/operand position sequence $o = \mathcal{Z}^o$ annotate each token in $c$ with their instruction position and the opcode/operand position within each position, respectively. Figure 3 shows the example of $p$ and $o$.

When training with GSM, we mix the training samples from different instruction set architectures, which introduce disparate syntax in their assembly code. We thus append the architecture sequence $a$ to indicate the architecture, which assists the model to transfer the learned instruction semantics useful on one architecture to another (e.g., push eax in x86 has the similar semantics to addi $sp,$sp,−4; sw $t0,(sp)$ in MIPS) [49].

STATEFORMER output. STATEFORMER have different outputs depending on the training tasks. When it is in the pretraining stage with GSM, its output consists of complete $\mu$State trace including both $\mu$DataState trace and $\mu$ControlState trace. We describe how these outputs participate in the computation of loss functions in Section 3.3. When we finetune STATEFORMER for type inference, its output is the prediction of type labels defined in Section 3.4.

### 3.3 Pretraining with GSM

**Numerical representation module.** We treat each value $v_i$ in $\mu$DataState trace $v$ as an 8-byte sequence. To learn the inter-dependencies between high and low bytes in $v_i$, we develop a
learnable neural module with Neural Arithmetic Unit (NAU) [58], which is shown beneficial to capture the semantics of numerical values involved in arithmetic operations (Section 2.4). Formally, let \( \mathbf{v}_i = (\mathbf{v}_{i1}, \ldots, \mathbf{v}_{i8}) \) denote the 8-byte sequence of \( \mathbf{v}_i \), we denote the aggregated embedding \( \mathbf{E}_m \) as the representation of each \( \mathbf{µ}_\text{DataState} \): \( \mathbf{E}_m = \text{NAU}(\text{Emb}(\mathbf{v}_{i1}), \ldots, \text{Emb}(\mathbf{v}_{i8})) \), e.g., \( \text{Emb}(\mathbf{v}_{i1}) \) denote applying the embedding to the first byte token of \( \mathbf{v}_i \). Figure 3 briefly illustrates how a \( \mathbf{µ}_\text{DataState} \) 0x6 gets encoded by NAU. Note that \( \mathbf{v}_i \) in Figure 3 indicates the embedding \( \mathbf{E}_m \).

**Sampling subset of \( \mathbf{µ}_\text{DataState} \).** We sample a random subset of \( \mathbf{µ}_\text{DataState} \) and replace them with \(<\text{MASK}>\) (e.g., the grayed-out tokens as shown in Figure 3) tokens in the model input so that the model is trained to reconstruct the removed \( \mathbf{µ}_\text{DataState} \). We define \( P_{\text{mask}} \) as the percentage of the masked \( \mathbf{µ}_\text{DataState} \) and study the effect of different \( P_{\text{mask}} \) on type inference in Section 5.4.

**Multimodal encoding module.** We only apply NAU to each \( \mathbf{µ}_\text{DataState} \) sequence \( \mathbf{v} \). For other sequences, we apply regular embeddings. We end up with 5 embeddings for each token in each sequence: \( \mathbf{E}_c, \mathbf{E}_o, \mathbf{E}_p, \mathbf{E}_s, \mathbf{E}_a \). We then compute the vector sum of 5 embeddings and output a single embedding \( \mathbf{x}_i = \text{sum}((\mathbf{E}_c, \mathbf{E}_o, \mathbf{E}_p, \mathbf{E}_s, \mathbf{E}_a)) \). The vector sum operation aggregates the multiple modalities (e.g., instruction and state) of each token into a single embedding. When we compute attentions between these embeddings, i.e., dot product [93], the cross-modality (instruction-state) dependencies are naturally computed, following the distributive property of multiplication: \( \mathbf{x}_i \cdot \mathbf{x}_j = \mathbf{E}_{ci} \cdot \mathbf{E}_{cj} + \mathbf{E}_{oi} \cdot \mathbf{E}_{oj} + \ldots + \mathbf{E}_{ai} \cdot \mathbf{E}_{aj} \).

**Loss functions.** After encoding all the input sequences as a single sequence of embeddings \( \mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) \), we feed \( \mathbf{x} \) to self-attention layers. The output of self-attention layers are known as the contextual embeddings \( \mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n) \). We stack two independent 2-layer feedforward networks \( h_s \) and \( h_f \), that takes \( \mathbf{e} \) as input and output the predicted \( \mathbf{µ}_\text{DataState} \) and \( \mathbf{µ}_\text{ControlState} \). Formally, let \( f = \{0, 1\}^n \) denote the \( \mathbf{µ}_\text{ControlState} \) labels, and \( M \) a set of locations in the masked \( \mathbf{µ}_\text{DataState} \). We define the pretraining objective as:

\[
\min_{\mathbf{\mu}} \sum_{i \in M} \text{MSE}(\mathbf{v}_i, h_s(e_i)) + \alpha \sum_{i=1}^{n} \text{BCE}(f_i, h_t(e_i)) \quad (1)
\]

The first part of the objective function aims to minimize the Mean Squared Error (MSE) between the predicted 8-byte and the groundtruth 8-byte for the only masked \( \mathbf{µ}_\text{DataState} \). Note that MSE treats the output byte tokens as numerical values (as opposed to categorical as treated in the input). Such a setting encourages the loss to penalize predictions far from the groundtruth (e.g., predicts 0x00 but the groundtruth is 0xFF). The second part of the objective function aims to minimize the Binary Cross-Entropy (BCE) between the predicted \( \mathbf{µ}_\text{ControlState} \) and the groundtruth, for all input tokens. \( \alpha \) is the weighting hyperparameter that keeps the scale of both losses at roughly the same magnitude. As all the modules of STATEFORMER are differentiable, i.e., NAU, FFN used for aggregating input sequences, self-attention layers, and \( h_s \) and \( h_t \), optimizing Equation 1 can be efficiently solved by gradient descent.

### 3.4 Transfer Learning Type Inference

After pretraining with GSM, we transfer STATEFORMER’s learned knowledge by finetuning it for type inference. We define our considered types in Figure 4, which serves as the labels for STATEFORMER to predict. Notably, our considered types are much more fine-grained than the existing ML-based type inference approaches. For example, EKLAVYA [20] does not distinguish signedness of the primitive types. Debin [40] does not handle floating point. And both works treat the pointer as a single type (\( \text{ptr} \)), without inferring what the pointer refers (e.g., \( \text{predicting} \ \text{char}^* \text{or struct}^* \)).

As discussed in Section 2.3, we do not collect \( \mathbf{µ}_\text{State} \) by micro-executing the code during finetuning. Specifically, we replace each token in \( \mathbf{v} \) with the dummy token \( \$\$ \) (described in Section 3.2) and still follow the same steps to compute the embeddings \( \mathbf{x} \). We then stack a new prediction head \( h_{\text{type}} \), a 2-layer feedforward network, that takes as input \( \mathbf{e} \) (the output of the last self-attention layers), and predicts the type labels defined in Figure 4 for each input code token. Formally, let \( t_i \) denote the groundtruth type of code token \( e_i \), the objective function of finetuning task is defined as the Cross-Entropy between the predicted type \( h_{\text{type}}(e_i) \) and \( t_i \) for each token in an input sequence with length \( n \): \( \min_{\mathbf{\mu}} \sum_{i=1}^{n} \text{CE}(t_i, h_{\text{type}}(e_i)) \). During
We implement $S \langle \rangle$ (collected from the datasets described below) 9 times with different virtual cores, 188GB RAM, and 4 Nvidia RTX 2080-Ti GPUs. To reported numbers and use the same datasets to evaluate $S\langle \rangle$.

**FIGURE 4**: The types that STATEFORMER predicts as output. We define the type hierarchy using the production rules for clarity, but we concretize all types during prediction, resulting in 35-type labels. (prim), (agg), and (ptr) stand for primitive, aggregate, and pointer types.

finetuning, both $h_{type}$ and the pretrained model weights will be updated by gradient descent.

## 4 IMPLEMENTATION AND SETUP

We implement STATEFORMER using the Fairseq toolkit [65] based on PyTorch 1.6.0. All the experiments are run on a Linux server with Ubuntu 18.04, Intel Xeon 4214 at 2.20GHz with 48 virtual cores, 188GB RAM, and 4 Nvidia RTX 2080-Ti GPUs. To obtain ground-truth types for training and testing, we compile all the software projects with debugging information and parse the DWARF sections using pyelftools [12] and Ghidra [1].

**µState collection.** To log the program states ($µ$State) for pretraining STATEFORMER on GSM task, we implement micro-execution using Unicorn [76], a cross-architecture CPU emulator based on QEMU [11]. Specifically, we micro-execute each function binaries (collected from the datasets described below) 9 times with different randomized initial values for registers and memory, generating 9 sets of $µ$State for each function binary. To align the $µ$State with the corresponding assembly instructions (Section 3.2), we leverage Capstone [75] to disassemble the function binaries.

**Metrics.** As described in Section 2, we treat type inference as a classification task. As the datasets have highly imbalanced labels, where the majority of tokens do not possess any type, we use precision ($P$), recall ($R$), and F1 score to measure the actual performance of STATEFORMER and all other tools. Let $TP$ denote the number of correctly predicted types, $FP$ denote that of incorrectly predicted types, $TN$ denote the number of correctly predicted no-access, and $FN$ denote the number of incorrectly predicted no-access. $P = TP/(TP + FP)$, $R = TP/(TP + FN)$, and $F1 = 2 \cdot P \cdot R/(P + R)$.

**Baseline tools.** We compare STATEFORMER with 3 state-of-the-art ML-based type inference prototypes: EKLA VYA [20], Debin [40], and TypeMiner [59]. These tools have been demonstrated to outperform traditional type inference techniques. For example, EKLA VYA has been shown to outperform TypeArmor [91], which is based on principled dataflow analysis such as def-use and liveness analysis. EKLA VYA implements the function signature recovery task. The authors define the task as predicting the type of function arguments. Since EKLA VYA does not release their trained model, we use their reported numbers and use the same datasets to evaluate STATEFORMER’s accuracy in recovering types for function argument.

<table>
<thead>
<tr>
<th>ARCH</th>
<th>OPT/OBF</th>
<th># Variables</th>
<th># Instructions</th>
<th># Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>O0</td>
<td>26,173,242</td>
<td>12,511,100</td>
<td>821,191</td>
<td></td>
</tr>
<tr>
<td>O1</td>
<td>27,845,108</td>
<td>9,346,292</td>
<td>874,595</td>
<td></td>
</tr>
<tr>
<td>O2</td>
<td>27,829,459</td>
<td>9,279,857</td>
<td>898,930</td>
<td></td>
</tr>
<tr>
<td>O3</td>
<td>28,413,646</td>
<td>10,114,915</td>
<td>942,138</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>109,991,455</td>
<td>41,252,164</td>
<td>3,536,854</td>
<td>ARM</td>
</tr>
<tr>
<td>MIPS</td>
<td>O0</td>
<td>13,474,083</td>
<td>14,098,871</td>
<td>602,699</td>
</tr>
<tr>
<td>O1</td>
<td>15,081,503</td>
<td>10,559,297</td>
<td>652,769</td>
<td></td>
</tr>
<tr>
<td>O2</td>
<td>15,416,769</td>
<td>10,170,866</td>
<td>678,577</td>
<td></td>
</tr>
<tr>
<td>O3</td>
<td>15,457,561</td>
<td>11,021,417</td>
<td>721,519</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>59,159,916</td>
<td>45,848,451</td>
<td>2,655,564</td>
<td>x86</td>
</tr>
<tr>
<td>O0</td>
<td>187,621,379</td>
<td>53,057,850</td>
<td>6,735,347</td>
<td></td>
</tr>
<tr>
<td>O1</td>
<td>189,217,168</td>
<td>51,024,118</td>
<td>6,787,678</td>
<td></td>
</tr>
<tr>
<td>O2</td>
<td>189,220,382</td>
<td>51,410,490</td>
<td>6,810,321</td>
<td></td>
</tr>
<tr>
<td>O3</td>
<td>189,554,035</td>
<td>52,275,998</td>
<td>6,853,561</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>755,612,964</td>
<td>207,768,456</td>
<td>27,186,907</td>
<td></td>
</tr>
<tr>
<td>x64</td>
<td>O0</td>
<td>184,390,304</td>
<td>40,286,578</td>
<td>6,599,662</td>
</tr>
<tr>
<td>O1</td>
<td>186,140,724</td>
<td>38,196,269</td>
<td>6,636,821</td>
<td></td>
</tr>
<tr>
<td>O2</td>
<td>186,114,113</td>
<td>38,355,719</td>
<td>6,679,632</td>
<td></td>
</tr>
<tr>
<td>O3</td>
<td>186,425,557</td>
<td>39,179,302</td>
<td>6,723,296</td>
<td></td>
</tr>
<tr>
<td>bcf</td>
<td>714,892</td>
<td>12,960,798</td>
<td>119,706</td>
<td></td>
</tr>
<tr>
<td>eff</td>
<td>644,018</td>
<td>11,604,224</td>
<td>90,740</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>714,310</td>
<td>6,960,835</td>
<td>119,481</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>745,143,648</td>
<td>187,543,725</td>
<td>26,989,338</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>1,669,907,983</td>
<td>482,412,796</td>
<td>60,368,663</td>
<td></td>
</tr>
</tbody>
</table>

Debin recovers both variable types and names. As we do not study recovering source-level variable names but focus on obtaining variable types, we compare with Debin’s type prediction only. Since Debin has released their trained model, we run Debin on our datasets directly and compare against its attained accuracy.

TypeMiner considers much finer-grained type labels than the previous two works. For example, it further distinguishes the pointer type to struct and char, while the former two do not. As TypeMiner is not open-sourced, we have contacted the authors to obtain their reported F1 scores and compare them to STATEFORMER by running STATEFORMER on their dataset.

These tools vary in their definition of the target types (e.g., EKLA VYA is limited to predicting only function argument types) and the evaluated architectures (e.g., TypeMiner only handles x64, EKLA VYA handles x86 and x64). Hence, we adjust our setup accordingly when comparing with the baselines.

**Dataset.** We collect 33 open-sourced software projects in their latest versions, including popular and large projects such as OpenSSL, ImageMagic, and Coreutils. Due to the page constraints, we put the details of the datasets in our supplementary material. We compile these software projects to 4 instruction set architectures including x86, x64, MIPS, and ARM, each with 4 optimizations, i.e., O0-O3, using GCC-7.5, and 3 obfuscation strategies, including bogus control flow (bcf), control flow flattening (eff), and instruction substitution (sub), using Hikari [104] based on Clang-8. Table 1 summarizes the statistics of the datasets.

**Pretraining and finetuning setup.** We pretrain STATEFORMER (with GSM) on all datasets in Table 1. We sample a random 10% of the functions from the pretraining datasets as the validation set.

![Table 1: The statistics of our datasets, categorized by architecture (Arch), optimization (OPT), and obfuscation (OBF).](image-url)
Table 2: StateFormer’s precision, recall, and F1 score, for each architecture (ARCH), optimization (OPT), and obfuscation (OBF).

<table>
<thead>
<tr>
<th>ARCH</th>
<th>OPT/OBF</th>
<th>Precision</th>
<th>Recall</th>
<th>F1 score</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>O0</td>
<td>77.1</td>
<td>79.2</td>
<td>78.1</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>78</td>
<td>76.2</td>
<td>77.1</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>77.3</td>
<td>73.7</td>
<td>75.4</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>90.9</td>
<td>89.9</td>
<td>90.4</td>
</tr>
<tr>
<td>MIPS</td>
<td>O0</td>
<td>98.9</td>
<td>91.7</td>
<td>95.2</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>86.1</td>
<td>67.6</td>
<td>75.7</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>80</td>
<td>68</td>
<td>73.4</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>81.3</td>
<td>71.2</td>
<td>75.8</td>
</tr>
<tr>
<td>x86</td>
<td>O0</td>
<td>85.3</td>
<td>83.8</td>
<td>84.5</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>72.4</td>
<td>70.9</td>
<td>71.6</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>74.8</td>
<td>70.9</td>
<td>72.8</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>83.6</td>
<td>79.8</td>
<td>81.6</td>
</tr>
<tr>
<td>x64</td>
<td>O0</td>
<td>81.5</td>
<td>81.4</td>
<td>81.4</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>75.8</td>
<td>74</td>
<td>74.9</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>71.1</td>
<td>69.2</td>
<td>70.1</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>72.3</td>
<td>70.4</td>
<td>71.3</td>
</tr>
<tr>
<td></td>
<td>bcf</td>
<td>73.5</td>
<td>70.5</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>cff</td>
<td>73.2</td>
<td>71.1</td>
<td>72.1</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>75.6</td>
<td>69.1</td>
<td>72.2</td>
</tr>
</tbody>
</table>

We aim to answer the following research questions.

- **RQ1**: How accurate is StateFormer in type inference?
- **RQ2**: How does StateFormer compare to the state-of-the-art ML-based systems?
- **RQ3**: How fast is StateFormer compared to other tools?
- **RQ4**: How effective is pretraining with GSM in improving the type inference accuracy?
- **RQ5**: How well does StateFormer approximate the operational semantics by training with GSM?

5 EVALUATION

We first study the accuracy of StateFormer on all binaries. Following the setup described in Section 4, we report the results in Table 2. StateFormer achieves an average 77.9% F1 score across all architecture, optimization, and obfuscation.

On x86 and x64, we observe that StateFormer remains relatively robust for binaries with higher optimization and obfuscation.

For example, the F1 score for x86 O3 is only 2.9% lower than that of x86 O0. The F1 score for x64 O3 is only 3.6% lower than that of x64 O1. Regarding the performance across different architectures (with all optimizations/obfuscations), we notice no significant difference on average. These observations indicate that StateFormer is robust across architectures and optimizations with disparate operational semantics of their instructions.

**STATEFORMER achieves an average 77.9% F1 score across all architecture, optimization, and obfuscation and remains robust for binaries with higher optimization levels and obfuscations.**

5.2 RQ2: Comparison to Baseline

**Baseline comparison**. We compare StateFormer with 3 state-of-the-art type inference tools, namely EKLA VYA, Debin, and TypeMiner, as described in Section 4.

To compare with EKLA VYA, we evaluate StateFormer on the same 8 projects considered in their paper: Binutils, Coreutils, Findutils, sg3-utils, util-linux, Inetutils, Diffutils, and usbutils. We evaluate StateFormer on 7 types considered in EKLA VYA. EKLA VYA treats type inference for each argument (of multiple function arguments) as an independent classification task and reports the accuracy (instead of F1 score). We thus also evaluate StateFormer’s accuracy, defined as the number of correctly predicted types divided by all the number of tokens.

Figure 5 compares StateFormer to EKLA VYA side-by-side on two architectures (i.e., x86 and x64) and 4 optimizations (O0-O3) as EKLA VYA is evaluated with these settings. On average, StateFormer outperforms EKLA VYA by 13.3%. Notably, StateFormer remains robust across different optimization levels, while EKLA VYA has a clear drop when the optimization level is increased.

To compare with Debin, we run their released model on OpenSSL, which we have confirmed is not included in their training set. We compile OpenSSL into 3 architectures (x86, x64, and ARM) with 4 optimizations (O0-O3). As Debin considers only 17 types, we also restrict the prediction of StateFormer to the same 17 types. Figure 6 shows that StateFormer consistently outperforms Debin on all architectures and optimizations, achieving 14.6% higher F1 scores on average. We observe Debin has an apparent drop in F1 scores with higher optimizations (down to 46.1% for ARM), while StateFormer remains robust with at least 70% F1 scores.

Finally, we compare StateFormer to TypeMiner on the same datasets they have considered. We restrict our test on x64 with O3, as TypeMiner is evaluated only on x64. TypeMiner treats type inference
as a multi-stage classification task, training independent classifiers to predict types at different levels. For example, it first trains a binary classifier to predict whether a variable is a pointer or not and then trains a second classifier to predict the pointer type. Since they do not make complete predictions in one-shot, we compare STATEFORMER on 4 sub-tasks on which TypeMiner has been evaluated. Specifically, TypeMiner’s first prediction task is a binary classification task deciding whether a variable has a pointer type (<ptr>) or a primitive type (<prim>). Its second task is to predict the pointer types, including array*, struct*, char*, and other ptr. Its third task is to predict the primitive types, including int, long int, char, and double. Its fourth task is to predict the signedness, including signed and unsigned. We label these 4 tasks as Task 1-4.

Figure 7 demonstrates that STATEFORMER outperforms TypeMiner in 4 tasks by an average 8.2%. In particular, TypeMiner significantly fluctuates when predicting primitive types (Task 3) and pointer types (Task 2), but STATEFORMER is more robust.

STATEFORMER outperforms EKLAVA, Debin, and TypeMiner by 13.3%, 14.6%, and 8.2%, respectively, and is more robust than all baselines for different optimizations and type granularity.

5.3 RQ3: Inference Speed

We evaluate STATEFORMER’s inference speed on binary programs and compare it to Debin and Ghidra. Specifically, we consider 4 software projects with different sizes on x64 compiled with 00.

Table 3 shows the runtime performance of STATEFORMER, Debin, and Ghidra. STATEFORMER (based on GPUs) achieves 98.1x speedup on average than the second-best tool. Notably, while the authors of Debin have tried to optimize their underlying learning algorithms (conditional random field) with parallelized implementation [78], it performs 1023x and 35.8x slower than STATEFORMER GPU and CPU, respectively. We attribute the speedup of STATEFORMER to its underlying neural architecture, which is amenable to GPU acceleration, while neither Debin’s nor Ghidra’s underlying algorithms can be implemented using GPU efficiently.

STATEFORMER is 98.1x faster than the second-best tool.

5.4 RQ4: Effectiveness of GSM

In this section, we dig deeper into the effectiveness of GSM pretraining task by quantifying how much improvement that STATEFORMER achieves when pretrained with GSM.

Effectiveness of GSM. We compare STATEFORMER’s finetuning accuracy when it is (1) pretrained with GSM, (2) pretrained with only predicting μDataState, (3) pretrained with only predicting μControlState, and (4) not pretrained. Figure 8a shows STATEFORMER’s validation F1 score at each finetuning epoch when the masking percentages $P_{mask}$ in GSM are 0.8, 0.6, 0.4, or 0.2.

Figure 8b shows the state recovery task’s validation F1 score when it is (1) pretrained with GSM, (2) pretrained with only predicting μDataState, (3) pretrained with only predicting μControlState, and (4) not pretrained. We label these 4 tasks as Task 5-8.

Table 4 shows the state recovery task’s validation F1 scores on 5 of our datasets, with diverse number of instructions (measured in thousand).

Table 4: Execution time (in seconds) of STATEFORMER (on both CPU and GPU), Debin, and Ghidra on 4 of our datasets, with diverse number of instructions (measured in thousand).

<table>
<thead>
<tr>
<th>Project</th>
<th># Inst (k)</th>
<th>STATEFORMER</th>
<th>Debin</th>
<th>Ghidra</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageMagic</td>
<td>1,252</td>
<td>187.8</td>
<td>7.3</td>
<td>N/A</td>
<td>1664.3</td>
</tr>
<tr>
<td>PuTTY</td>
<td>969</td>
<td>146.0</td>
<td>5.6</td>
<td>5239.8</td>
<td>514.2</td>
</tr>
<tr>
<td>Findutils</td>
<td>157</td>
<td>23.7</td>
<td>0.9</td>
<td>8490.0</td>
<td>833.2</td>
</tr>
<tr>
<td>zlib</td>
<td>22</td>
<td>3.3</td>
<td>0.1</td>
<td>1190.0</td>
<td>11.7</td>
</tr>
</tbody>
</table>

*Debin terminates abruptly after running one of the binaries for 138 minutes.
predicting $\mu$DataState requires understanding instructions’ actual execution effect and computing the concrete values, while predicting $\mu$ControlState is only a binary classification task encoding approximate control flow. Nevertheless, we observe even pretraining with predicting only $\mu$DataState or $\mu$ControlState is still beneficial for type inference, as STATEFORMER pretrained on either of them obtains 69% and 62% F1 scores, respectively.

Masking percentage. Recall in GSM, we train STATEFORMER to reconstruct the masked $\mu$DataState, and we use default masking percentage $P_{\text{mask}} = 0.8$ throughout our experiments (Section 4). As masking less percentage of $\mu$DataState makes it easier to train on GSM, we study how varying $P_{\text{mask}}$ affects the type inference performance. Figure 8b shows the validation F1 scores achieved by STATEFORMER when we vary $P_{\text{mask}}$. We observe that the more we mask in GSM, the better it boosts the type inference performance, but the gap of improvement is not significant. One possible explanation is that even in one example, the masked states are less, many pretraining samples and the dynamic masking still introduce diverse enough cases for learning operational semantics.

STATEFORMER pretrained with GSM outperforms that without pretraining by 33% in F1 score. Masking percentage in pretraining GSM does not significantly affect the finetuning results: pretraining with 20% masking rate results in <2% decrease in F1 score compared to pretraining with 80% masking rate.

5.5 RQ5: STATEFORMER Performance on GSM

Pretraining losses with GSM. We also study the losses of pretraining STATEFORMER with GSM. Such a study directly validates whether pretraining with GSM indeed helps STATEFORMER to learn operational semantics. Low losses on unseen testing $\mu$State and function binaries indicates that STATEFORMER highly likely learns to generalize based on its learned knowledge of operational semantics.

Figure 9 shows the training and validation losses in 10 epochs of pretraining STATEFORMER with GSM. The validation set is constructed by sampling a random 10% functions from the projects used in pretraining (as described in Section 4). Specifically, Figure 9a shows the MSE loss of predicting $\mu$DataState and Figure 9b shows the BCE loss of predicting $\mu$ControlState. We observe that the validation MSE drops to 0.00011, which translates to average absolute distance (by taking the square root) between prediction and groundtruth as 0.011 ($\sqrt{0.00011} = 0.011$). As we normalize the byte values from $[0, 256]$ into $[0, 1]$ (see Section 3.2), $0.011 \times 256 \approx 2.8$ is the actual absolute error between the predicted byte and the groundtruth. The average error within the deviation of only 3-byte indicates that STATEFORMER learns to approximate the execution effect.

Effects of control and data flow pretraining. Concurrent to our work, Trex [68] also leverages transfer learning to learn program execution semantics. However, Trex completely ignores control and data flow modeling as it focuses on binary similarity detection. In contrast, STATEFORMER focuses on type inference; therefore, it requires precise data flow (type of output of an instruction depends on types of operands) and control flow (it must also infer types of values in the unexecuted portion of the code).

Because of these differences in the high-level requirements of the downstream tasks, STATEFORMER and Trex adopt significantly different pretraining approaches, i.e., generating control and data flow state (GSM) vs. code and trace token classification. In general, it remains an open challenge in transfer learning to determine which pretraining task is the most effective for which downstream task. Part of our contribution in STATEFORMER is to design a pretraining task that makes the downstream task of type inference precise. For example, Figure 10 shows that STATEFORMER outperforms Trex by around 10.9 percentage points in F1 score for type inference.

Probing STATEFORMER on real-world code. Besides quantifying the pretraining losses, we probe the pretrained STATEFORMER using a concrete binary example to study how it predicts $\mu$State.

Consider the example in Figure 11. We examine how STATEFORMER predicts registers esp, edi, ebp, and esi from input $\mu$DataState, in which we mask all registers except for their first appearances. The accurate prediction of esp at line 5 suggests that STATEFORMER is able to associate 0x0886644e with esp at line 1 and line 2 and understand the execution effect of sub. Further, to predict esi at line 6, STATEFORMER needs to understand xor’s execution effects at line 3. Since there is no other occurrence of esi in this code block, we can conclude that the prediction of esi is based solely on STATEFORMER’s understanding of xor.

Figure 9: MSE and BCE of predicting $\mu$DataState and $\mu$ControlState, respectively, during pretraining STATEFORMER with GSM.

Figure 10: Type inference F1 score between models pretrained by GSM and Trex’s pretraining objective.
To track the type propagation, these works often rely on expensive data/control dependency analysis [8, 9, 46, 47, 51, 52, 59, 72, 82]. By contrast, dynamic analysis uses accurate program states and memory access patterns observed during program execution [27] to define precise rules for type inference [5, 15, 22, 38, 48, 77, 79, 84] and propagation [17, 34–36, 54, 83, 102]. However, dynamic approaches suffer from low code coverage, leading to a high false negative rate [51]. Increasing code coverage requires collecting and combining dynamic traces from multiple program executions [15, 83], which incurs prohibitively high overhead.

**STATEFORMER** enjoys the benefits of both static and dynamic analysis as it automates learning instructions’ approximate operational semantics from *cheap micro-execution* and uses such semantics to learn type inference rules *without dynamic execution*. **ML-based approaches.** Recently, machine learning has been increasingly applied to type inference. Examples include inferring the type of function argument [20], recovering general variable type [41, 59, 60, 71, 97, 100], and other metadata (e.g., variable names) [2, 40, 56, 78, 80, 92, 94]. However, existing ML-based binary type inference approaches use only static code without any traces and suffer from similar limitations as static analysis. Concurrent to our work, Trex [68] also leverages transfer learning to learn program execution semantics. However, Trex is not control/data flow aware, resulting in a significant performance drop in the type inference task, as shown in Section 5.5.

More broadly, machine learning has shown significant success in learning generalizable representation that applies to many program analysis tasks [3, 13, 69]. **STATEFORMER** contributes a new generic framework to learn programs’ operational semantics. Therefore, we believe **STATEFORMER** has a great potential to apply to other downstream program analysis tasks beyond type inference.

**8 CONCLUSION**

We presented **STATEFORMER**, a neural architecture that uses the operational semantics of assembly code to recover type information from stripped binaries. We designed a novel pretraining task, Generative State Modeling, to help **STATEFORMER** to learn code operational semantics and transfers this knowledge to learn type inference rules. We showed that **STATEFORMER** is 14.6% more accurate than state-of-the-art tools, and our ablation studies showed that GSM improves type inference accuracy by 33%.

**ACKNOWLEDGMENT**

We thank the anonymous reviewers for their constructive and valuable feedback. This work is sponsored in part by NSF grants CNS-1842456, CNS-1801426, CNS-1617670, CCF-1822965, CCF-1845893, CNS-1563843, CNS-1564055, and IIS-2040961; ONR grants N00014-17-1-2010, N00014-16-1-2263, and N00014-17-1-2788; an NSF CAREER award; an ARL Young Investigator (YIP) award; a Google Faculty Fellowship; a J.P. Morgan Faculty research award; a DiDi Faculty research award; a Google Cloud grant; a Capital One research grant; and an Amazon Web Services grant. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, ONR, ARL, NSF, Capital One, Google, J.P. Morgan, DiDi, or Amazon.
REFERENCES


Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring

Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tris-


Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic

Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. 2015. Building

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural


Kangjie Lu and Hong Hu. 2019. Where does it go? refining indirect-call targets

Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Taeyoung Kim, Kisub Kim, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse


Michael Pradel, Georgios Gousios, Jason Liu, and Sathis Chandra. 2020. Type-


David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. 2020. Pass-


Dongrui Zeng and Gang Tan. 2018. From Debugging-Information Based Binary-Level Type Inference to CFG Generation. In Eighth ACM Conference on Data and Application Security and Privacy.

