2025 Design, Automation & Test in Europe Conference (DATE 2025) – Best Paper Award Candidate

*Re*BERT: LLM for Gate-Level to Word-Level Reverse Engineering

Lizi Zhang
University of Wisconsin-Madison
Madison, USA
lzhang697@wisc.edu

Azadeh Davoodi
University of Wisconsin-Madison
Madison, USA
adavoodi@wisc.edu

Rasit Onur Topaloglu
Adeia Inc.
Poughkeepsie, NY, USA
rasit.topaloglu@adeia.com

***Abstract*—In this paper, we introduce *Re*BERT, a specialized large language model (LLM) based on BERT, fine-tuned specifically for grouping bits into words within gate-level netlists. By treating the netlist as a form of language, we encode bits and their fan-in cones into sequences that capture structural dependencies. A novel contribution is augmenting BERT's embedding with a tree-based embedding strategy which mirrors the hierarchical nature of circuit designs in hardware. Leveraging the powerful representational learning capabilities of LLMs, we interpret hardware circuits at a higher level of abstraction. We evaluate *Re*BERT on various hardware designs, demonstrating that it significantly outperforms a state-of-the-art work based on partial structural matching in recovering word-level groupings. Our improvements are on average between 12.2% to 218.1% depending on degree of corrupting the structural patterns.**

***Index Terms*—reverse engineering, LLM, netlist, embedding**

I. Introduction

Reverse engineering of gate-level netlists is a crucial process in the field of hardware security and design analysis. As modern integrated circuits (ICs) become more complex and rely on third-party resources, such as third-party IP cores and overseas fabrication foundries, the security of these designs becomes increasingly difficult to guarantee. This concern is further heightened by the growing risk of hardware Trojans and malicious modifications, which can be inserted into designs at various stages of the supply chain. Hardware Trojans, for instance, can be small and subtle enough to avoid detection by traditional functional tests but can have significant security implications by introducing vulnerabilities or hidden malicious behaviors into the system [1]–[4].

Moreover, reverse engineering helps in security verification by allowing engineers to recover the high-level functionality from a gate-level netlist and verify that the design has not been tampered with during production, especially when original design documents are incomplete or unavailable. By analyzing the netlist, engineers can reconstruct the high-level behavior of the circuit, ensuring the system operates as intended [5]. This process is critical for ensuring the reliability and security of devices ranging from consumer electronics to critical infrastructure, where compromised hardware can have severe consequences. However, many existing tools require prior knowledge of the state registers, where the state registers are utilized to reconstruct parts of a chip's functionality [6], [7].

This work is supported by NSF Award No. 2322713.

Several approaches have been developed for reverse engineering gate-level netlists, primarily focusing on identifying high-level structures, such as word-level groupings from bit-level designs. Traditionally, methods such as pattern matching [8], [9] and structural analysis [10] have been widely used to detect word-level structures in circuits.

The work [10] attempts to find word-level structures by intersecting bounded fan-in subtrees of gates to identify control logic and functional units. While this method has shown success in some cases, it struggles in designs that are either heavily optimized or intentionally obfuscated. This is particularly challenging given the extensive optimizations, and occasional errors, introduced by synthesis tools [11]. The inability to detect words when subtrees are too small or control logic is distorted limits its effectiveness in complex circuits. Another set of techniques [12], [13] focuses on leveraging control signals to guide identification of word-level groupings. These methods capitalize on the structural similarities between groups of bits driven by the same control signals. However, they face challenges due to the vast number of control signals automatically inserted by the CAD tools. Identifying which signals are truly relevant is time-consuming and error-prone.

Additionally, methods such as register aggregation aim to group flip-flops into multi-bit registers by identifying shared enable signals and functional dependencies [14]–[16]. While these techniques have been effective in specific scenarios, they too struggle when applied to flattened or highly optimized netlists, where the original register structures may be difficult to recover. Furthermore, these methods are often reliant on predefined structural templates, limiting their flexibility to generalize across diverse circuit architectures.

In this work, we propose *Re*BERT, a novel approach that utilizes Large Language Models (LLMs), specifically a BERT-based architecture, to overcome the limitations of traditional *Re*verse Engineering techniques. BERT (Bidirectional Encoder Representations [17]) is a powerful deep learning model originally developed for natural language processing tasks. Its ability to understand the context and relationships between tokens in a sequence makes it highly suitable for the task of classifying sequences derived from circuit netlists. *Re*BERT treats the gate-level netlist as a structured sequence of operations, encoding each bit and its fan-in cone as a sequence of tokens.

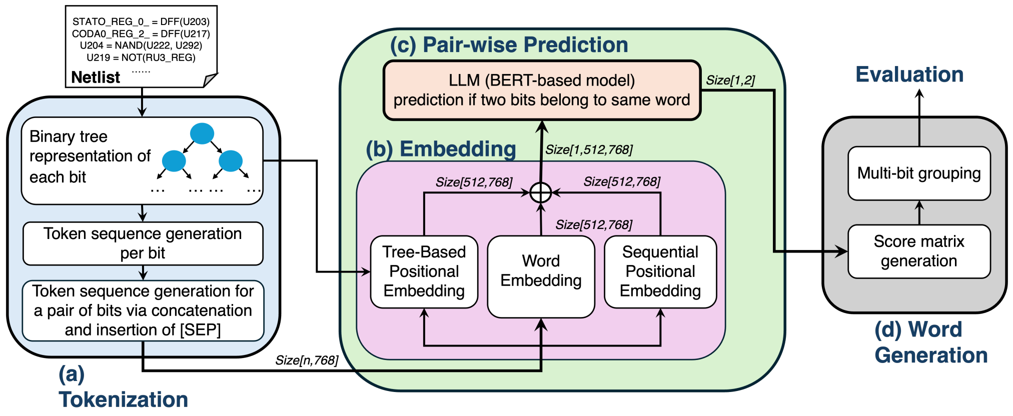


Fig. 1. Flowchart of *Re*BERT: (a) **Tokenization**: each pair of bits is represented as a sequence of tokens that is compatible by BERT (our used LLM); (b) **Embedding**: token sequences are processed into high-dimensional vectors; (c) **Pair-wise Prediction**: BERT predicts if a pair of bits belong to same word; (d) **Word Generation**: score matrix is generated for all bit pairs and examined to group bits into words. The sizes written on arrows are for one pair of bits.

By leveraging the contextual learning capabilities of Transformers, *Re*BERT can model relationships between bits and predict whether they belong to the same word. *Re*BERT's core strength lies in its robustness against corruption of structural patterns. *Re*BERT, does not depend on rigid structural templates. By encoding the netlist as sequences and utilizing a tree-based embedding strategy that mirrors the hierarchical nature of hardware circuits, *Re*BERT can capture the underlying dependencies between bits.

Besides the overall design of *Re*BERT and using LLM for reverse engineering, key contributions are summarized below:

- *Re*BERT features a novel approach to generate token sequences for bits from a gate-level netlist in a manner to be most effective for processing by the BERT model.
- *Re*BERT utilizes 3 schemes which are most appropriate for embedding token sequences, including a novel tree-based positional embedding to capture circuit hierarchy.

In our experiments we compare to a state-of-the-art approach based on structural matching [12]. We show average improvements ranging from 12.2% to 218.1% depending on degree of introduced netlist corruption.

II. Proposed Approach

Figure 1 shows overview of *Re*BERT which includes:

- **Tokenization**: First, binary trees representing the logical connections and operations implementing each 'bit' are produced. Bits are identified as signals feeding into sequential components in the gate-level netlist. Next, a pre-order traversal is performed on each binary tree to provide a sequence representation of the corresponding bit. Each entry in a sequence is referred as a token. Then for each pair of bits a token sequence is generated via concatenation and insertion of a special separation token. The tokens are also padded to make them appropriate format for BERT (our used LLM).
- **Embedding**: This step uses the token sequence representing a pair of bits and converts them into a dense vector. We utilize three embedding techniques: (1) word embedding, (2) sequential positional embedding, and (3) tree-based positional embedding. They allow capturing both content and positional context of token sequences, making them suitable as input to the BERT model.
- **Pair-wise Predictions**: Using the embedding representing two bits as input, BERT is used to estimate the likelihood if they belong to the same word.
- **Word Generation**: Based on the pair-wise predictions from BERT, a score matrix is generated which represents the likelihood that any two bits belong to the same word. A multi-bit grouping step then generates the words.

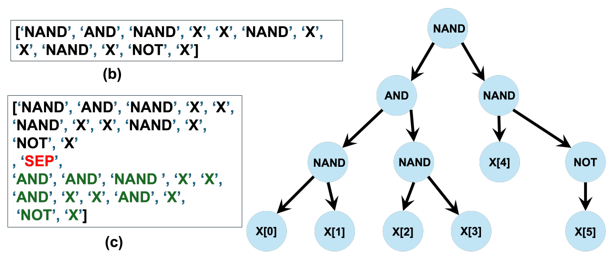
In the remainder of this section, we present the details of bit tokenization in Section II-A, embedding in Section II-B, pair-wise predictions in II-C, and word grouping in section II-D.

A. Tokenization

The transformation of a circuit netlist into tokens that can be effectively processed by a BERT model is essential, as the quality of these tokens directly impacts the model's performance. This process involves several critical steps which are discussed here. They include extracting binary tree and Boolean expression for each bit which is used to generate a sequence of tokens per bit. The next step is token sequence generation representing a pair of bits. We discuss them next.

1) Binary tree representation of a bit: We first start by converting all k-input gates (where $k > 2$) into equivalent 2-input gates based on predefined templates. This conversion is essential for standardizing the circuit into a *binary* tree format, enabling easier processing and analysis. For each bit, a binary tree is constructed to represent the subcircuit connected to that bit. The subcircuit is obtained by backtracing k levels, where k is an adjustable parameter (k=*6* in our implementation). In this binary tree, each gate is represented as a node. The leaf nodes represent the signal names feedings in the subcircuit. This transformation offers a consistent and hierarchical representation of the logical structure of the circuit.

2) Token sequence generation of a bit: After constructing the binary tree for each bit, the next step involves extracting a sequence per bit using its tree. This process is conducted through a pre-order traversal of the binary tree, which generates a unique ordering of the tree nodes.


['NAND', 'AND', 'NAND', 'X', 'X', 'NAND', 'X', 'X', 'NAND', 'X', 'NOT', 'X']

['NAND', 'AND', 'NAND', 'X', 'X', 'NAND', 'X', 'X', 'NAND', 'X', 'NOT', 'X' , 'SEP', 'AND', 'AND', 'NAND ', 'X', 'X', 'AND', 'X', 'X', 'AND', 'X', 'NOT', 'X']

Fig. 2. (a) Binary tree representation of a bit; (b) Corresponding token sequence; (c) Token sequence for a pair of bits (padding not shown).

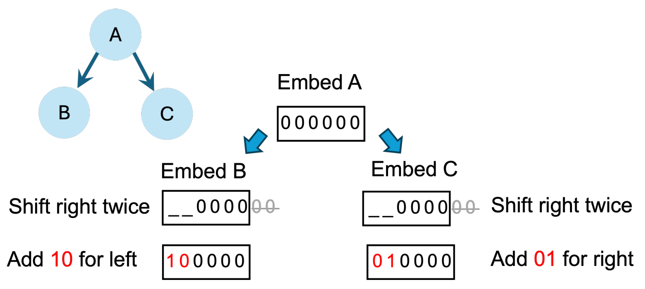


Fig. 3. Example showing our tree-based positional embedding.

Each entry in the sequence is referred to as a token. Each token stores the gate type of its corresponding node. Figure 2 shows binary tree for a bit extracted for *k=3* levels and its token sequence. It is important to note that the input bit names, denoted as *'X[i]'* in the tree are generalized to *'X'* in the sequence as the specific names contribute minimally to prediction accuracy but introduce unnecessary complexity into the vocabulary used in the BERT model.

3) Token sequence generation for two selected bits: For two bits, their corresponding token sequences are concatenated into a single token sequence, after inserting a special token '[SEP]' to separate them, as shown in Figure 2(c). This concatenation provides the model with the contextual information required to evaluate the relationship between the two bits within the circuit, thereby determining whether they belong to the same logical group. Next, for consistency and compatibility with BERT, each token in a sequence is padded to have a uniform length of 768. As shown in Figure 1, the size of a token sequence representing a pair of bits is $[n, 768]$ where n is sum of the number of tokens representing each bit plus one special separation token.

B. Embeddings for Netlist Sequences

The goal of embedding is to represent token sequences of a bit pair in a high-dimensional space. Specifically, *Re*BERT combines three embedding techniques as shown in Figure 1. These embeddings are critical for transforming the raw token sequences into a format that the model can effectively analyze, capturing both semantic and structural aspects of the circuit.

Our three embedding techniques are: (1) word embedding, (2) sequential positional embedding and (3) tree-based positional embedding. The first two embeddings are foundational to the BERT architecture. The third one (tree-based positional embedding) is a specialized approach designed to capture the hierarchical nature of circuit structures. By integrating these embeddings, the model gains a comprehensive representation of the circuit sequences. This rich, multidimensional framework allows the model to effectively process and analyze the intricate patterns and relationships within the circuit data.

1) Word embedding: Word embedding is a fundamental technique in natural language processing that maps each token (in this case, representing logic gates or circuit inputs) to a dense vector in a continuous vector space [18]. In our experiments, each token is represented as a vector of size 512. These vectors are learned during the BERT model's training process and capture the semantic relationships between different tokens. In the context of circuit sequences, word embedding allows the model to distinguish between various logic operations and inputs, providing a nuanced understanding of the circuit's components. Each token is mapped to a fixed-size vector through an embedding matrix, which is fine-tuned during training to optimize the model's performance.

2) Sequential positional embedding: While word embedding captures the identity of each token, sequential positional embedding encodes the position of each token within the sequence [19], [20]. This is crucial for preserving the order of operations in the circuit, as the sequence in which gates and inputs are arranged significantly impacts the circuit's behavior. We employ a learnable positional embedding layer, where each position in the sequence is mapped to a vector that is then added to the corresponding word embedding. This combination of word and positional embeddings ensures that the model can understand not only what operations are being performed but also the order in which they occur.

3) Tree-based positional embedding: Building on the standard sequential positional embedding, we use a tree-based positional embedding [21] which is specifically designed to capture the hierarchical structure of the circuit represented as a binary tree. Unlike linear sequences, circuits often exhibit more complex relationships between components, which are not fully captured by sequential positional embeddings.

The tree-based positional embedding resolves this limitation by embedding each node in the binary tree according to its hierarchical position. In this approach, each node (representing a gate or logic operation) is assigned a vector that encodes its relative position within the tree structure, specifically its parent-child relationships and its depth in the tree. To generate the positional encoding for each node, the encoding process begins with the root node, which is assigned a zero vector. The size of the vector is twice the number of nodes in the tree. For each subsequent child node, the encoding of the parent node is right-shifted by two digits, with the addition of '10' for a left child and '01' for a right child. Figure 3 illustrates this for a simple tree with 3 nodes.

Next, the encoding is done by following a pre-order traversal of the tree to maintain consistency with the token sequences. The final tree-based positional embedding is obtained by concatenating all the individual encodings. This hierarchical embedding allows the model to capture the deeper structural dependencies in the circuit, enabling it to more accurately understand and classify complex sequences.

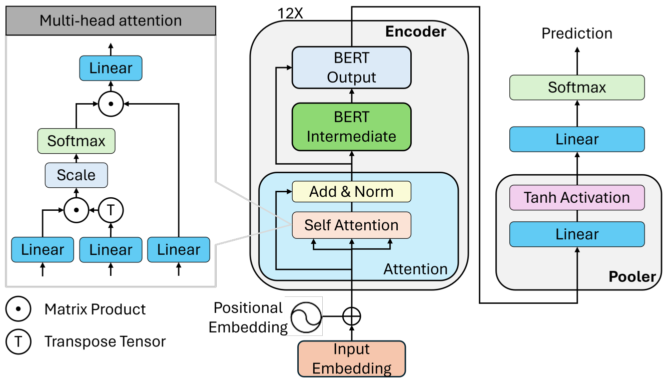


Fig. 4. Overview of the BERT model architecture [17].

C. LLM BERT-based Model for Pair-wise Prediction

Our BERT model makes a decision based on the embedded token sequence representing two bits, and generates a probability if they belong to the same word. Before passing the embedded token sequences to the BERT model, we apply a *filtering* step which decides if two bits are highly unlikely to belong to the same word. In that case, the next token sequence is processed. This filtering ensures that only bit pairs with a high similarity level are classified, effectively reducing computational efforts by early discarding of less relevant pairs.

Our filtering is done by applying the Jaccard similarity which is defined as $J(A, B) = |A \cap B|/|A \cup B|$ where $|A \cap B|$ is the cardinality of the intersection of sets A and B and $|A \cup B|$ is the cardinality of their union. Specifically, token sequence pairs with a Jaccard similarity score lower than 0.7 are filtered out, and their pairwise score is set to -1.

Next, we provide a brief overview of the BERT model architecture, and refer the reader to [17] for more details.

BERT is built on the Transformer architecture, which utilizes self-attention mechanisms to capture dependencies between tokens in a sequence, both from the left and right contexts. This bidirectional approach allows BERT to develop a deep understanding of the entire sequence, making it particularly effective in tasks where context is crucial, i.e., identifying relationships between different components of a circuit.

The BERT model comprises several key components, each playing a critical role in processing and analyzing input sequences. Figure 4 shows the architecture of the model.

Input Embedding (shown in Figure 4) is provided from the previous embedding step by combining the 3 discussed embedding techniques. Next, the Encoder is the core component of the BERT model, responsible for processing the embedded sequences and capturing the complex relationships between tokens. The encoder consists of following layers:

- **Attention**. This module includes *multi-head self-attention* and *Add & Norm* layer. The multi-head self-attention allows the model to consider the relationships between different tokens in the sequence, regardless of their distance from each other. The self-attention mechanism calculates attention scores between all pairs of tokens, enabling the model to weigh the importance of each token when forming a contextualized representation.

  The self-attention mechanism is enhanced through multi-head attention, which involves multiple attention layers running in parallel. Each "head" can focus on different parts of the sequence, capturing diverse aspects of the input data. In this work, we use 12 heads for every multi-head attention blocks. The outputs of these heads are then concatenated and linearly transformed to produce the final attention output. After the attention mechanism, the model applies an *Add & Norm* layer, which involves adding the original input to the attention output (residual connection) and then normalizing the result. This step helps stabilize the training process and ensures that the model can effectively learn from the input data.
- **BERT Intermediate**. Following the self-attention and normalization steps, the BERT model includes a feed-forward neural network (FFNN), often referred to as the 'BERT Intermediate' layer. This network consists of two linear transformations with a non-linear activation function (GELU) in between. The purpose of this layer is to further refine the representations learned from the attention mechanism.
- **BERT Output**. The final encoder layer produces the output of the BERT model. This output can be used for various downstream tasks, such as sequence classification. In the context of circuit analysis, the output is typically used to classify whether pairs of sequences or tokens belong to the same logical group within the circuit.

BERT also includes a 'pooler' layer, which is particularly useful for classification tasks. This layer applies a linear transformation followed by a Tanh activation to the first token in the sequence. The resulting vector can be used as a fixed-size representation of the entire sequence, suitable for feeding into a classifier.

D. Grouping Multiple Bits to Form Words

Using the previous steps, all possible bit pairs are processed which are either discarded in the filtering phase, or sent to the BERT model to predict a probability of belonging to the same word. Next, we generate a score matrix, which records a score for all possible bit pairs. The score is either the probability generated by BERT, or -1 if the pair is filtered.

We select a threshold to determine which bit pairs should be grouped together. Since the range and spread of the pairwise scores are not consistent across different matrices, the selection of the threshold is dynamically adjusted for each score matrix. Specifically, the threshold is defined as $\frac{1}{3}\max(\text{score matrix})$. This adaptive approach ensures that the threshold is responsive to the distribution of scores for each individual circuit, allowing the grouping method to remain effective regardless of the specific score distribution.

Once the threshold is determined, a graph is constructed in which each bit is represented as a node. If the score between two bits exceeds the threshold, an edge is added between the corresponding nodes. All connected bits—those linked by edges in the graph—are grouped together, forming a word. This graph-based approach ensures that bits with strong pairwise relationships are accurately grouped.

TABLE I
BENCHMARK CIRCUITS INFORMATION BASED ON ITC'99.

Benchmark	**b03**	**b04**	**b05**	**b07**	**b08**	**b11**	**b12**	**b13**	**b14**	**b15**	**b17**	**b18**
#gates	122	652	927	383	149	726	944	289	8367	9767	30777	111241
#FFs	30	66	34	49	21	31	121	53	449	245	1415	3320
#Words	7	9	5	7	5	5	46	7	32	8	98	212

III. EXPERIMENTAL RESULTS

We evaluated the performance of *Re*BERT across a variety of gate-level netlist benchmarks. The model is fine-tuned on an NVIDIA A100 GPU, using the Hugging Face transformers (4.42.2) and PyTorch (2.3.1).

A. Our Setup

1) Controlled Netlist Corruption: To introduce variability and challenge the model's robustness during our validation, we defined the following netlist corruption procedure. Each gate in the netlist was visited and replaced with functionally-equivalent gates (accordingly to a template) based on the value of a randomly-generated Gate Replacement Index (denoted by **R-Index**). For example, `A = NAND(B, C)` may be replaced by `A = OR(NOT(B), NOT(C))`. The R-index determines the probability of a gate being replaced. If the replacement index is set to 0, no corruption is applied. Conversely, if the replacement index is set to 1, all gates in the netlist are replaced. Intermediate values of the replacement index provide finer control over the degree of corruption, allowing for flexible testing of the model's robustness under varying levels of structural corruption. This step complicates the sequence classification task but also tests the model's ability to accurately group bits and understand the underlying logic in scenarios where structural similarity are no longer reliable.

2) Training Data Generation: Table I shows the statistics of the benchmarks used in our experiments. Before discussing training data generation, we note that we employed a leave-one-out cross-validation (LOO-CV) strategy to test each benchmark. In this process, all available benchmarks are used as training circuits, except for one, which is designated as the testing circuit. This approach ensures that each benchmark is used as the testing circuit once, providing a robust evaluation of the model's generalizability across different circuits.

To generate the training data, the bits were first found in each gate-level netlist by identifying the lines feeding in the sequential components. A token sequence was generated for each bit as discussed in Section II.A.2. All pair-wise bits were considered and token sequences were generated for each pair based on the procedure in Section II.A.3.

To further diversify the training dataset, we apply different R-Indexes to the benchmarks. For each design, six variations are considered by corrupting it with R-Indexes ranging from 0 (no gates replaced) to 1 (all gates replaced) with a 0.2 increment. Data are collected from both original benchmarks and corrupted circuits, resulting in an augmented dataset that encompasses a wide range of structural variations. This robust dataset enables the model to learn from circuits with differing levels of structural modification, improving its ability to generalize to unseen circuits.

A key aspect of data preparation is ensuring a balanced ratio between positive and negative samples. Positive samples consist of bit pairs that belong to the same group (word), while negative samples involve pairs from different groups. Since the dataset includes all possible bit pairings, negative samples naturally outnumber positive ones. To create a balanced dataset and minimize bias, we maintain a ratio of 1:1.2 between positive and negative pairs. This slight skew towards negative samples reflects the reality that most bit pairs do not belong to the same group, helping the model generalize by training on a wider variety of challenging negative cases. Additionally, due to the varying sizes of the circuits, the amount of data generated from different benchmark circuits differs significantly. To ensure that no single circuit dominates the training process, we limit the maximum number of samples from any circuit to 5,000. As a result, the final training set had 26,915 samples.

3) Evaluation Metric: The quality of the bit groupings is evaluated using the Adjusted Rand Index (ARI), which is a standard statistical measure for comparing the similarity between two clusters, i.e., the predicted groupings of bits and the true groupings of bits, while adjusting for chance. The ARI is defined as:

$$\text{ARI} = \frac{\text{Index} - \text{Expected Index}}{\text{Max Index} - \text{Expected Index}}$$

where Index is the number of pairs of bits that are either correctly grouped together or correctly separated into different groups. Expected Index is the expected value of the Index under random grouping. Max Index is the total number of bit pairs, representing the maximum possible value of the Index. The ARI score ranges from -1 (poor clustering) to 1 (perfect clustering), with a value of 0 indicating random grouping. By using the ARI, we can rigorously evaluate the accuracy of the model's grouping predictions, providing a reliable measure of its performance in reverse-engineering word-level groupings from gate-level information.

We note, the work [12] uses an accuracy metric for evaluation of quality for reverse engineering which in essence is very similar to this standard statistical measure.

B. Results

1) Comparison of the Grouping Quality: The performance of *Re*BERT was evaluated against a state-of-the-art structural method [12] which we additionally implemented based on the psuedo-code provided in the paper[1]. The results, shown in Table II, demonstrate that *Re*BERT outperforms the structural approach, both in terms of accuracy and robustness, with significantly better ARI in the presence of structural corruption.

[1]We also compared with [13] however their results were significantly worse than [12], in part because it relied on manual identification of control signals in each design. We did not include this comparison due to lack of space.

TABLE II
COMPARISON OF *Re*BERT WITH STRUCTURAL MATCHING IN TERMS OF ARI (OUR WORD GROUPING EVALUATION METRIC).

R-Index	**Method**	**ARI (Adjusted Random Index)**												**Average**
		b03	**b04**	**b05**	**b07**	**b08**	**b11**	**b12**	**b13**	**b14**	**b15**	**b17**	**b18**	
0	**Structural**	0.653	**0.584**	0.375	0.643	**0.860**	0.603	0.029	**0.721**	0.312	0.377	**0.257**	0.178	0.466
	***Re*BERT**	**0.728**	0.514	**0.980**	**0.742**	0.137	**0.847**	**0.419**	0.522	**0.392**	**0.508**	0.255	**0.231**	0.523 **(12.2%)**
0.2	**Structural**	0.465	0.300	0.103	0.218	0.523	0.090	0.080	0.236	0.259	0.190	0.251	0.174	0.241
	***Re*BERT**	**0.600**	**0.419**	**0.609**	**0.477**	**0.626**	**0.847**	**0.226**	**0.425**	**0.419**	**0.380**	**0.284**	**0.214**	0.461 **(91.3%)**
0.4	**Structural**	0.226	0.118	0.069	0.130	0.158	0.125	0.045	0.093	0.209	0.204	**0.194**	0.154	0.144
	***Re*BERT**	**0.335**	**0.370**	**0.283**	**0.444**	**0.474**	**0.644**	**0.229**	**0.221**	**0.301**	**0.402**	0.156	**0.180**	0.337 **(134.0%)**
0.6	**Structural**	0.091	0.034	0.067	0.129	0.091	0.028	0.021	0.039	0.201	0.224	0.164	0.141	0.103
	***Re*BERT**	**0.468**	**0.264**	**0.263**	**0.278**	**0.282**	**0.712**	**0.344**	**0.171**	**0.315**	**0.469**	**0.169**	**0.178**	0.326 **(218.1%)**
0.8	**Structural**	0.249	0.049	-0.004	0.220	0.037	0.153	0.009	0.125	0.264	0.179	0.187	0.114	0.132
	***Re*BERT**	**0.543**	**0.305**	**0.326**	**0.314**	**0.278**	**0.656**	**0.362**	**0.128**	**0.331**	**0.539**	**0.207**	**0.165**	0.346 **(162.6%)**
1	**Structural**	**0.527**	**0.044**	0.054	**0.342**	**0.126**	0.455	0.010	0.106	0.319	0.326	0.164	0.087	0.213
	***Re*BERT**	0.436	0.209	**0.093**	0.324	0.094	**0.707**	**0.349**	**0.401**	**0.341**	**0.548**	**0.195**	**0.122**	0.318 **(49.2%)**
Average	**Structural**	0.369	0.188	0.111	0.280	0.299	0.242	0.032	0.220	0.261	0.250	0.203	0.141	
	***Re*BERT**	**0.518**	**0.347**	**0.426**	**0.430**	**0.315**	**0.735**	**0.322**	**0.311**	**0.350**	**0.474**	**0.211**	**0.182**	
	Improv.	**40.6%**	**84.3%**	**284.7%**	**53.3%**	**5.4%**	**203.5%**	**894.4%**	**41.5%**	**39.9%**	**81.9%**	**4.1%**	**28.5%**	

TABLE III
COMPARISON OF AVERAGE RUNTIME ACROSS DIFFERENT REPLACEMENT INDEXES (R-INDEX).

	Average Runtime (s)											
	b03	**b04**	**b05**	**b07**	**b08**	b11	**b12**	**b13**	**b14**	**b15**	**b17**	**b18**
Structural	0.011	0.011	0.009	0.009	0.007	0.010	0.057	0.009	1.645	0.514	15.322	47.516
***Re*BERT**	0.009	0.011	0.009	0.010	0.008	0.009	0.059	0.009	1.672	0.638	15.201	120.968

When comparing performance across different R-Index values, *Re*BERT outperforms the structural method at all levels of corruption (as well as no corruption). The average improvements are shown in the last column. For instance, at R-Index = 0.2, *Re*BERT achieves significantly higher ARI scores, on-average 91.3% better across the benchmarks. As the replacement index increases, *Re*BERT maintains its high performance. For example, for `b11` at R-Index. = 0.6, *Re*BERT achieves an ARI score of 0.712, while the structural method declines sharply, scoring 0.028 under the same conditions. This shows *Re*BERT's robustness in handling structural corruption.

We observed that the structural method performs particularly worse for non-extreme R-index values. Recall that the R-Index indicates the likelihood of netlist gates being replaced by equivalent gates. When R-Index is close to 0 or 1, similar gates (i.e., corresponding gates in two similar bits) before replacement may remain similar after replacement because they are either both replaced in the same way (R-index=1), or both not replaced (R-index=0). In contrast, when R-Index is not at its extremes, the structural patterns are more heavily corrupted, and the structural method, which relies on consistent patterns, struggles the most.

In terms of benchmark-specific performance (last row), *Re*BERT achieves higher ARI scores across circuits of varying sizes and complexities, though not in every case. For example, in circuits such as `b15`, *Re*BERT performs better than structural method consistently for all R-index values. However, in other cases, like `b08`, the structural method achieves a higher ARI for R-index=0 but the average improvement in ARI across all R-index values is 5.4% for this case.

2) Runtime Comparison: While *Re*BERT delivers better grouping performance, it also maintains competitive runtime efficiency, as shown in table III, with only minor differences across most benchmarks. For example, in `b12`, on average, *Re*BERT takes 0.057 seconds to complete the task, slightly faster than the structural method (0.059 seconds). However, for `b18` which is one of the larger circuits, *Re*BERT's runtime overhead becomes more substantial. For instance, *Re*BERT requires 120.968 seconds, while the structural method completes the task in 47.516 seconds. This trend indicates that while *Re*BERT provides enhanced accuracy, it may come at the cost of an increase in computational time for some of the larger benchmarks. We have identified opportunities to accelerate *Re*BERT but have left these for exploration in the future.

IV. CONCLUSION

In this paper, we introduced *Re*BERT, a novel approach leveraging large language models (LLMs), specifically a BERT-based architecture, for reverse engineering gate-level netlists to recover word-level structures. By converting netlists as sequences and employing a tree-based embedding strategy, *Re*BERT effectively captures the hierarchical nature of circuits. Our approach significantly outperforms structure-based methods. Through comprehensive evaluations on various benchmarks, *Re*BERT demonstrated superior accuracy and robustness in identifying word-level groupings, even under significant structural corruption. These results highlight the potential of LLMs as a powerful tool for hardware reverse engineering, paving the way for more secure and reliable hardware design analysis in future applications.

References

[1] Y. Jin, N. Kupp, and Y. Makris, “Experiences in hardware trojan design and implementation,” in *HOST*, 2009, pp. 50–57.

[2] M. Oya, Y. Shi, M. Yanagisawa, and N. Togawa, “A score-based classification method for identifying hardware-trojans at gate-level netlists,” in *DATE*, 2015, pp. 465–470.

[3] Z. Huang, Q. Wang, Y. Chen, and X. Jiang, “A survey on machine learning against hardware trojan attacks: Recent advances and challenges,” *IEEE Access*, vol. 8, pp. 10 796–10 826, 2020.

[4] T. Zhang, J. Wang, S. Guo, and Z. Chen, “A comprehensive FPGA reverse engineering tool-chain: From bitstream to RTL code,” *IEEE Access*, vol. 7, pp. 38 379–38 389, 2019.

[5] J. Portillo, T. Meade, J. Hacker, S. Zhang, and Y. Jin, “RERTL: Finite state transducer logic recovery at register transfer level,” in *AsianHOST*, 2019, pp. 1–6.

[6] T. Meade, S. Zhang, and Y. Jin, “Netlist reverse engineering for high-level functionality reconstruction,” in *ASP-DAC*, 2016, pp. 655–660.

[7] W. Tang, Y.-T. Li, K.-P. Hsu, K.-L. Chou, Y.-C. Lin, C.-F. Chien, T.-L. Hsu, Y.-C. Chen, T.-C. Wang, S.-C. Chang, T. Hwang, and C.-Y. Wang, “A hybrid approach to reverse engineering on combinational circuits,” in *DATE*, 2024, pp. 1–2.

[8] M. Hansen, H. Yalcin, and J. Hayes, “Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering,” *IEEE Design Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.

[9] Z. He, Z. Wang, C. Bai, H. Yang, and B. Yu, “Graph learning-based arithmetic block identification,” in *ICCAD*, 2021, pp. 1–8.

[10] W. Li, A. Gascon, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia, “WordRev: Finding word-level structures in a sea of bit-level gates,” in *HOST*, 2013, pp. 67–74.

[11] Y. Herklotz and J. Wickerson, “Finding and understanding bugs in FPGA synthesis tools,” in *FPGA*, 2020, p. 277–287.

[12] T. Meade, Y. Jin, M. Tehranipoor, and S. Zhang, “Gate-level netlist reverse engineering for hardware security: Control logic register identification,” in *ISCAS*, 2016, pp. 1334–1337.

[13] E. Tashjian and A. Davoodi, “On using control signals for word-level identification in a gate-level netlist,” in *DAC*, 2015, pp. 1–6.

[14] V. Rao and Z. D. Sisco, “Register aggregation for hardware decompilation,” 2024. [Online]. Available: https://arxiv.org/abs/2409.03119

[15] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, “Reverse engineering digital circuits using functional analysis,” in *DATE*, 2013, pp. 1277–1280.

[16] S. D. Chowdhury, K. Yang, and P. Nuzzo, “ReIGNN: State register identification using graph neural networks for circuit reverse engineering,” in *ICCAD*, 2021, pp. 1–9.

[17] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: pre-training of deep bidirectional transformers for language understanding,” *CoRR*, vol. abs/1810.04805, 2018.

[18] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, “Google’s neural machine translation system: Bridging the gap between human and machine translation,” *CoRR*, vol. abs/1609.08144, 2016.

[19] A. Vaswani, “Attention is all you need,” *NIPS*.

[20] F. Luo, J. Zhang, and S. Xu, “Learning positional attention for sequential recommendation,” *arXiv preprint arXiv:2407.02793*, 2024.

[21] V. Shiv and C. Quirk, “Novel positional encodings to enable tree-based transformers,” in *NIPS*, vol. 32, 2019, pp. 12 081 – 12 091.