

TrojanLoC: LLM-based Framework for RTL Trojan Localization

Weihua Xiao^{†§}, Zeng Wang^{†§}, Minghao Shao^{†‡}, Raghu Vamshi Hemadri[†], Ozgur Sinanoglu[‡],
Muhammad Shafique[‡], Johann Knechtel[‡], Siddharth Garg[†], Ramesh Karri[†]

[†]NYU Tandon School of Engineering, New York, USA

[‡]NYU Abu Dhabi, Abu Dhabi, UAE

{wx2356, zw3464, shao.minghao, rh3884, ozgursin, muhammad.shafique, johann}@nyu.edu,
siddharth.j.garg@gmail.com, rkarri@nyu.edu

Abstract

Hardware Trojans (HTs) are a persistent threat to integrated circuits, especially when inserted at the *register-transfer level (RTL)*. Existing methods typically first convert the design into a graph, such as a gate-level netlist or an RTL-derived *dataflow graph (DFG)*, and then use a *graph neural network (GNN)* to obtain an embedding of that graph, which (i) loses compact RTL semantics, (ii) relies on shallow GNNs with limited receptive field, and (iii) is largely restricted to coarse, module-level binary HT detection. We propose **TrojanLoC**, an *LLM-based framework for RTL-level HT localization*. We use an RTL-finetuned LLM to derive *module-level* and *line-level embeddings* directly from RTL code, capturing both global design context and local semantics. Next, we train task-specific classifiers on these embeddings to perform *module-level Trojan detection*, *type prediction*, and fine-grained *line-level localization*. We also introduce *TrojanInS*, a large synthetic dataset of RTL designs with systematically injected Trojans from four effect-based categories, each accompanied by precise line-level annotations. Our experiments show that **TrojanLoC** achieves strong module-level performance, reaching 0.99 F1-score for Trojan detection, up to 0.68 higher than baseline, and 0.84 macro-F1 for Trojan-type classification. At the line level, TrojanLoC further achieves up to 0.93 macro-F1, enabling fine-grained localization of Trojan-relevant RTL lines.

Keywords

Hardware Trojan, Register Transfer Level, Large Language Model

1 Introduction

Machine learning (ML) has been increasingly applied to hardware security challenges [15–17], evolving from early classical ML methods [6, 21] to graph-based techniques that leverage *Graph Neural Networks (GNNs)* for modeling circuit structures [2, 13, 22]. While these approaches have advanced *hardware Trojan (HT)* detection, the emergence of *large language models (LLMs)* introduces a new paradigm [11, 14]. LLMs provide semantically rich representations of hardware description languages for HT analysis.

Most recent learning-based defenses follow a common pipeline: they first encode a hardware design into one or more vector *embeddings*, and then train a *classifier* on these embeddings to decide whether the design is Trojaned [8, 13, 20, 22]. Within this pipeline, the classifier is a separate component that operates on fixed-dimensional feature vectors, and the core technical challenge is how to construct embeddings that capture comprehensive RTL

semantics relevant for downstream HT detection. Existing methods typically address this by transforming RTL designs into intermediate graph representations and then using a GNN to extract graph-level embeddings [13, 20]. Two main types of intermediate graphs are commonly used: (i) gate-level netlists obtained after synthesis [8], and (ii) RTL-derived *dataflow graphs (DFGs)* [13, 20].

This graph-construction step has three key drawbacks. First, mapping RTL to gate-level netlists or DFGs discards RTL semantics, e.g., a single RTL statement can expand into thousands of gates. Second, GNNs are typically kept shallow to avoid over-smoothing, which limits their receptive field and capture the global semantic of a RTL code. Third, most existing methods focus on coarse, module-level binary detection and offer little support for fine-grained localization, while gate-level approaches also require full synthesis and incur significant overhead. In parallel, recent works have shown that LLMs can serve as general-purpose embedding extractors for input texts [24], software codes [1], and RTL codes analysis [5]. Additionally, another type of works directly prompts LLMs with RTL code and asks them to judge whether a given design contains a Trojan, effectively using the LLM itself as a Trojan detector [4].

In this work, we use LLMs to extract embeddings directly from RTL codes, which capture the design semantics needed for Trojan detection and localization. To realize this LLM-based approach, we develop *TrojanLoC: LLM-based Framework for RTL Trojan Localization*, which combines the TrojanInS dataset (Section 3.1), LLM-based embedding extraction (Section 3.2), and task-specific downstream classifiers (Section 3.3) for module-level and line-level Trojan analysis. *TrojanInS* is a synthetic RTL dataset with GPT-4.1 generated Trojans from four categories (T1–T4), providing module-level presence/type labels and line-level annotations after preprocessing (Section 3.1). TrojanLoC processes each RTL module at two levels. At the module level, the entire RTL text is encoded by an RTL-finetuned decoder-only LLM and mean-pooled a single module embedding (Section 3.2.1). At the line level, each RTL line is encoded to obtain a line embedding (Section 3.2.2). Then, both module-level and line-level embeddings are passed through two autoencoders for dimension reduction, which are finally used to train classifiers for both module-level and line-level Trojan tasks (Section 3.3). Overall, TrojanLoC uses LLM-derived RTL embeddings as a shared semantic backbone, while separate autoencoders and classifiers are trained on these embeddings for module-level detection, type classification, and fine-grained line-level localization.

Our main contributions are summarized as follows:

[§]Authors contributed equally to this research.

- (1) **TrojanInS Dataset.** We construct TrojanInS, a large-scale RTL dataset with 17k+ validated designs spanning four Trojan families (T1–T4), enabling comprehensive training and benchmarking of RTL-level Trojan detection and localization methods.
- (2) **TrojanLoC Framework.** We propose TrojanLoC, a unified LLM+classifier framework that generates module-level and line-level RTL embeddings and, together with autoencoders and classifiers, supports module-level Trojan detection, Trojan-type prediction, and fine-grained line-level localization.
- (3) **Extensive Evaluation.** We conduct detailed experiments and comparisons demonstrating that TrojanLoC preserves RTL semantics more effectively than graph-based methods and delivers significantly finer detection granularity, achieving higher precision, recall, and F1-score at both module and line levels.

2 Preliminaries

2.1 Hardware Trojans

HTs are malicious modifications intentionally inserted into integrated circuits. They are typically composed of two parts: a *trigger* and a *payload*. The trigger is designed to activate only under rare internal states or input conditions so that the Trojan is hard to be detected during normal functional verification and production testing. Once the trigger condition is met, the payload alters the circuit’s behavior or properties. In practice, both the trigger and payload are often inserted into low-activity, small modules of the RTL, where signal toggling is infrequent and coverage by simulation and testing is weaker. Trojan insertion can occur at different abstraction levels, including specification, RTL, gate-level netlists, or even during physical design and manufacturing, which makes early-stage detection particularly important.

HTs can be classified into four types according to their impacts [10], i.e., functionality modification (T1), information leakage (T2), denial of service (T3), and performance degradation (T4). This effect-based view highlights the diverse goals of HTs and motivates the need for detectors that can not only identify the presence of a Trojan but also distinguish its impact on the system.

2.2 Decoder-only Transformer-based LLMs

Decoder-only transformer-based LLMs have become the foundational architecture for modern generative and analytic AI systems. In Fig. 1(a), it shows the basic architecture of decoder-only transformer-based LLMs. Given an input text, a *tokenizer* first converts it into a sequence of t discrete tokens, where t is the sequence length. An *embedding layer* then maps each token to an embedding $\mathbf{e}_k^{(0)} \in \mathbb{R}^{d_{\text{model}}}$ for $k = 1, \dots, t$, where d_{model} is a parameter of an LLM. The input to the *decoder layer* can therefore be viewed as a vector of embeddings $\{\mathbf{e}_1^{(0)}, \dots, \mathbf{e}_t^{(0)}\}$ as shown in Fig. 1. The decoder layer consists of L identical *transformer blocks*, which process these embeddings in sequence (Fig. 1 (b)).

In transformer block ℓ (for $\ell = 0, \dots, L - 1$), the current embeddings $\{\mathbf{e}_1^{(\ell)}, \dots, \mathbf{e}_t^{(\ell)}\}$ are updated to $\{\mathbf{e}_1^{(\ell+1)}, \dots, \mathbf{e}_t^{(\ell+1)}\}$. For each position k , the block uses an attention module to mix information from positions j , and then applies a small *feed-forward network*. Which positions may interact is specified by an *attention mask*,

$$M \in \{0, 1\}^{t \times t},$$

generated by the tokenizer as shown in Fig. 1. Each row $M_{k,*}$ is a length- t vector that describes which tokens position k is allowed to use: if $M_{k,j} = 1$ then token j can contribute to the updated embedding at position k , and if $M_{k,j} = 0$ it cannot. In the causal setting, $M_{k,j} = 0$ for all $j > k$, so position k only uses information from tokens $1, \dots, k$. When several input texts are concatenated and processed together, entries of M are set to 0 whenever tokens k and j belong to different texts. In this way, one can think of M as t vectors (one per position), and the model can process multiple texts in a single forward pass while keeping them independent.

After the last (i.e., L -th) transformer block, final embeddings

$$\{\mathbf{e}_1^{(L)}, \dots, \mathbf{e}_t^{(L)}\},$$

are obtained as one embedding $\mathbf{e}_k^{(L)} \in \mathbb{R}^{d_{\text{model}}}$ for each token position k . Each $\mathbf{e}_k^{(L)}$ can be viewed as the final embedding of token k , because it encodes that token together with the context allowed by the mask. In a standard LLM, these embeddings are passed through a layer for linear projection to *vocabulary logits* \rightarrow a *softmax layer* that converts logits to probabilities over the next token \rightarrow a *sampling layer* that selects the next token. We use the final token embeddings $\{\mathbf{e}_k^{(L)}\}$ as the basis for higher-level embeddings in Section 3.

3 Methodology

The *TrojanLoC* framework has three components:

- (1) **TrojanInS dataset** (Section 3.1), which constructs a large-scale RTL corpus with systematically injected Trojans from four categories and fine-grained ground-truth annotations;
- (2) **LLM-based embedding extraction** (Section 3.2), which uses an RTL-finetuned LLM to derive module-level and line-level embeddings directly from TrojanInS RTL designs;
- (3) **Classifier training** (Section 3.3), which first trains an autoencoder to project these embeddings into a latent space to improve computational efficiency and robustness, and then trains task-specific classifiers on pairs of dimension-reduced embeddings and their corresponding labels for module-level Trojan detection/type prediction, and line-level Trojan localization.

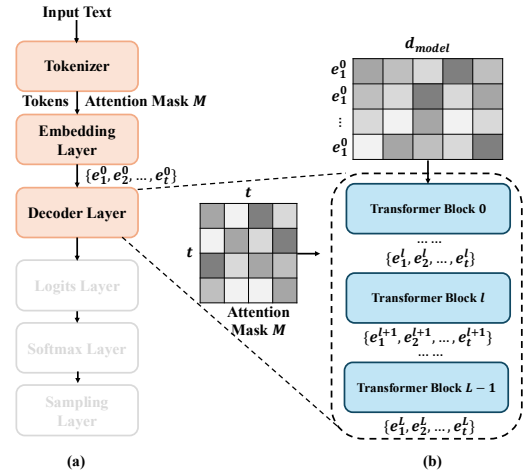


Figure 1: A decoder-only transformer-based LLM.

Table 1: Statistics of the TrojanInS dataset.

Property	Value
Base clean designs	~ 4,000
Trojaned designs	> 17,000
Modules (train / test)	17,658 / 4,407
Lines (train / test)	1,964,853 / 496,347
Class distribution	Clean: $\approx 20\%$; Trojan: $\approx 80\%$ (T1-T4 balanced)

3.1 TrojanInS Dataset

Reliable Trojan detection needs a large RTL corpus with systematic insertions and fine-grained labels, yet existing benchmarks remain small and lack line-level detail. We introduce *TrojanInS*, a dataset of 16000+ Trojaned Verilog designs with line-level ground truth, whose statistics are shown in Table 1.

3.1.1 Base Dataset. TrojanInS is constructed from VeriGen [12], a collection of over 4000 functionally correct Verilog designs gathered from public codebases and standard instructional materials. To ensure uniformity and avoid ambiguity in annotation, we restrict the dataset to designs containing a single top-level module, yielding roughly 4000 clean circuits that span combinational logic, sequential control, and mixed-structure designs. This filtering guarantees consistent evaluation granularity and simplifies the mapping between Trojaned regions and the original design structure.

3.1.2 Automated Trojan Insertion. Trojan variants are generated through an LLM-guided insertion process in which GPT-4.1 introduces malicious behavior based on established hardware security taxonomies while keeping the design’s normal functionality unchanged. Each clean RTL file is expanded into four Trojaned versions, one for each category described in Section 2.1. These categories cover the major RTL-level Trojans studied in prior work and capture the core malicious behaviors relevant to detection and localization. For each design, the LLM receives the original Verilog code and an explicit Trojan specification, ensuring that the injected variants exhibit clear semantic and structural malicious patterns.

3.1.3 Line-Level Annotation. Each Trojan-inserted design in TrojanInS includes precise line-level labels that mark trigger logic, payload behavior, and any auxiliary modifications, while all unchanged lines are labeled as clean. We first identify newly inserted or altered regions in the LLM’s output Trojaned design, then align them with the clean reference design to ensure accurate boundary marking. For every sample, the dataset provides (i) the clean RTL file, (ii) the Trojaned variant, (iii) a binary line-level mask, and (iv) metadata describing the Trojan type and its operational intent. These annotations allow us to train classifiers for multiple tasks within a unified framework, including module-level Trojan detection, Trojan-type classification, and fine-grained line-level localization, as shown in Section 3.

3.2 LLM-based Embedding Extraction

TrojanLoC uses a decoder-only transformer LLM as an embedding extractor for RTL code. We use the LLM that has been fine-tuned on a large amount of RTL codes, e.g., *Verilog* and *SystemVerilog*. This RTL fine-tuning exposes the model to hardware-specific syntax,

coding styles, and common structural patterns (such as always blocks, reset logic, state machines, and data-path operations). As a result, the final token embeddings $\mathbf{e}_k^{(L)}$ can capture richer semantics for hardware code than embeddings from a general LLM. Figure 2 illustrates the overall flow for processing an RTL module containing a T1 Trojan, with the Trojan-related lines highlighted by blue boxes. A Trojaned RTL module (the same procedure is applied to clean modules) is first split into its source lines. Both the full module text and each individual line are then fed into a *LLM architecture* consisting of a tokenizer, an embedding layer, a decoder layer with multiple transformer blocks, and an *average pooling* stage. The LLM produces a single d_{model} -dimensional *module-level embedding* and a set of d_{model} -dimensional *line-level embeddings*, which are paired with module-level and line-level labels and used as inputs to the autoencoders and classifiers in Section 3.3.

3.2.1 Module-level Embeddings. For the module-level branch in Figure 2, the entire RTL code of a module is treated as one input text. The tokenizer converts this text into a sequence of t tokens, and the embedding layer maps them to initial embeddings $\{\mathbf{e}_1^{(0)}, \dots, \mathbf{e}_t^{(0)}\}$. These embeddings are processed by the decoder layer described in Section 2.2, which consists of L transformer blocks and outputs final token embeddings $\{\mathbf{e}_1^{(L)}, \dots, \mathbf{e}_t^{(L)}\}$.

We then apply average pooling over all t final token embeddings to obtain a single vector that summarizes the module:

$$\mathbf{z}_{\text{mod}} = \frac{1}{t} \sum_{k=1}^t \mathbf{e}_k^{(L)} \in \mathbb{R}^{d_{\text{model}}}.$$

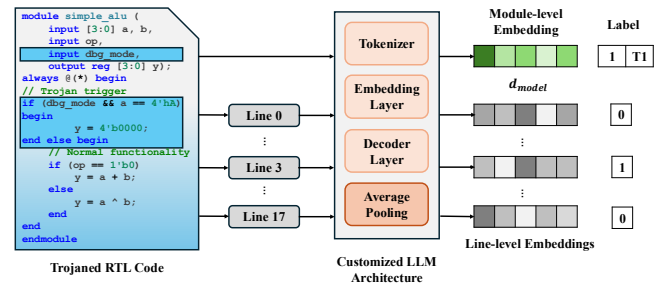
This \mathbf{z}_{mod} is shown as the green bar in Figure 2. In the figure, the module-level embedding is associated with a binary label of 1 and a type label T1, indicating that the module contains a T1 Trojan.

3.2.2 Line-level Embeddings. For line-level localization, we extract one embedding per RTL line, as shown in the lower branch of Figure 2. Given a module with L_{line} lines $\ell_0, \ell_1, \dots, \ell_{L_{\text{line}}-1}$, each line ℓ_i is treated as a text input. The tokenizer converts ℓ_i into t_i tokens, the embedding layer produces embeddings $\{\mathbf{e}_{i,1}^{(0)}, \dots, \mathbf{e}_{i,t_i}^{(0)}\}$, and the decoder layer produces final embeddings $\{\mathbf{e}_{i,1}^{(L)}, \dots, \mathbf{e}_{i,t_i}^{(L)}\}$ for line.

The line-level embedding is obtained by averaging its tokens:

$$\mathbf{z}_{\text{line},i} = \frac{1}{t_i} \sum_{k=1}^{t_i} \mathbf{e}_{i,k}^{(L)} \in \mathbb{R}^{d_{\text{model}}}.$$

In Figure 2, these are shown as the gray bars. For efficiency, our implementation does not run the LLM separately for each line. Instead,

**Figure 2: Overall flow of LLM-based embedding extraction.**

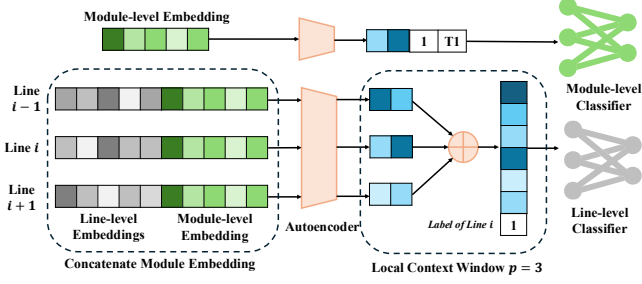


Figure 3: Training module- and line-level classifiers.

multiple lines from a module are concatenated into a longer token sequence and processed in a single forward pass, while an attention mask is used to prevent tokens from different lines from attending to one another (as described in Section 2.2). Conceptually, however, this procedure is equivalent to encoding each line independently and then applying average pooling over its tokens.

Finally, each line embedding $\mathbf{z}_{\text{line},i}$ is paired with a binary label $y_{\text{line},i} \in \{0, 1\}$ indicating whether line i belongs to Trojan logic. In the example in Figure 2, line embeddings are shown with labels 0 or 1 on the right, corresponding to clean and Trojan-related lines, respectively. These labeled embeddings are used by the line-level autoencoder and localization classifier in Section 3.3.

3.3 Classifier Training

Figure 3 gives the classifier training stage in TrojanLoC. The module-level embedding (top branch) and the line-level embeddings together with the module embedding (bottom branch) are first compressed by a shared autoencoder. The resulting low-dimensional representations are then used by two separate classifiers: a module-level classifier for Trojan detection and type prediction, and a line-level classifier for Trojan localization.

3.3.1 Autoencoders for Dimensionality Reduction. To reduce the dimensionality of LLM embeddings and remove redundancy, TrojanLoC uses two small autoencoders with the same architecture but different training data: module and line-level embeddings. The *module-level* autoencoder takes the module embedding

$$\mathbf{x}_{\text{mod}} = \mathbf{z}_{\text{mod}} \in \mathbb{R}^{d_{\text{model}}}.$$

For the *line-level* autoencoder, each input combines local and global information by concatenating the line embedding with the corresponding module embedding:

$$\mathbf{x}_{\text{line},i} = [\mathbf{z}_{\text{line},i}; \mathbf{z}_{\text{mod}}] \in \mathbb{R}^{2d_{\text{model}}},$$

as shown in the left dashed box of Figure 3.

In both cases, an encoder $f_{\text{enc}}(\cdot)$ maps the high-dimensional input \mathbf{x} to a lower-dimensional latent vector $\mathbf{h} \in \mathbb{R}^{d_{\text{enc}}}$ ($d_{\text{enc}} < d_{\text{model}}$), and a decoder $f_{\text{dec}}(\cdot)$ reconstructs $\hat{\mathbf{x}}$ from \mathbf{h} . Each autoencoder is trained with a standard reconstruction loss

$$\mathcal{L}_{\text{AE}} = \|\mathbf{x} - \hat{\mathbf{x}}\|_2^2,$$

minimized over all modules (for \mathbf{x}_{mod}) or all lines (for $\mathbf{x}_{\text{line},i}$). Both autoencoders are trained only on the training split of TrojanInS. No test modules or lines are used during autoencoder training. We include embeddings from both clean and Trojaned designs, since the autoencoders serve purely for dimensionality reduction rather

than Trojan detection. After training, we discard the decoders and keep only the encoders, obtaining compact features

$$\mathbf{h}_{\text{mod}} = f_{\text{enc}}^{\text{mod}}(\mathbf{x}_{\text{mod}}), \quad \mathbf{h}_{\text{line},i} = f_{\text{enc}}^{\text{line}}(\mathbf{x}_{\text{line},i}),$$

which correspond to the compressed vectors in the upper and lower branches of Figure 3. These reduced features are then used by the module-level and line-level classifiers described next.

3.3.2 Module-level and Line-level Classifiers.

Module-level classifiers. Each module-level embedding \mathbf{h}_{mod} is associated with two labels: (i) a binary label indicating whether the module is clean or Trojaned (0/1), and (ii) a categorical label specifying the Trojan type (T1–T4) when a Trojan is present. For example, in Figure 3, the module embedding is paired with the labels 1 and T1, meaning that the module is Trojaned and the inserted Trojan is type T1. To solve the two module-level tasks, we train two classifiers on \mathbf{h}_{mod} : one for Trojan detection (clean vs. Trojaned) and one for Trojan-type prediction (T1–T4). Both classifiers use the corresponding module-level labels during training.

Line-level classifier with local context. For line-level Trojan localization, we want to decide for each line i whether it is part of Trojan logic. Using only $\mathbf{h}_{\text{line},i}$ may miss important local patterns, so we augment it with a context window of neighboring lines, as shown in the lower branch of Figure 3.

Let p denote the context window size. For a target line index i , we take the reduced embeddings of lines $i - \frac{p-1}{2}, \dots, i, \dots, i + \frac{p-1}{2}$ and concatenate them to form a context-augmented embedding:

$$\mathbf{h}_{\text{line},i}^{\text{aug}} = [\mathbf{h}_{\text{line},i-\frac{p-1}{2}}; \dots; \mathbf{h}_{\text{line},i}; \dots; \mathbf{h}_{\text{line},i+\frac{p-1}{2}}].$$

In Figure 3, this is illustrated with a window $p = 3$. A line-level classifier takes $\mathbf{h}_{\text{line},i}^{\text{aug}}$ as input and predicts whether line i belongs to Trojan logic. During training of the line-level classifier, we use pairs of $\mathbf{h}_{\text{line},i}^{\text{aug}}$ and the corresponding binary label $y_{\text{line},i} \in \{0, 1\}$.

Overall, the pipeline in Figure 3 shows how a single autoencoder provides compact representations for both module- and line-level embeddings, and how these embeddings, combined with module-level information and local line context, are used to train classifiers to perform Trojan detection, Trojan-type prediction, and fine-grained line-level Trojan localization.

4 Experimental Evaluation

In this section, we evaluate TrojanLoC on both module-level and line-level HT tasks. We first describe the experimental setup, including datasets, embedding backbones, and classifiers. We report results for module-level Trojan detection and Trojan-type prediction, followed by a study of line-level Trojan localization under different design choices and hyperparameters.

4.1 Experimental Setup

Dataset. All experiments are conducted on the RTL dataset described in Section 3. We split the dataset into 80% for training and 20% for testing. Module-level tasks (Trojan detection and type prediction) are evaluated on modules, while line-level localization is evaluated on RTL lines with binary labels. To ensure our model learns semantic patterns rather than superficial cues, we preprocess the RTL code by removing Trojan-related comments and replacing

Trojan-indicative variable names with benign alternatives. This step is implemented by a Python script and does not use an LLM. This prevents the model from solving tasks through keyword matching.

Embedding backbones. Unless otherwise specified, TrojanLoC uses decoder-only transformer LLMs that have been fine-tuned on large RTL corpora (e.g., Verilog and SystemVerilog) to produce module-level and line-level embeddings, as described in Section 3.2. In experiments, we evaluate TrojanLoC using three finetuned LLMs: *CL-Verilog* 13B [9], *CodeV-QW* 7B [23], *HaVen-CodeQWen* 7B [19].

Dimensionality reduction. For all settings, we use the autoencoders described in Section 3.3 to reduce the dimensionality of module-level embeddings \mathbf{z}_{mod} and combined line-level embeddings $\mathbf{x}_{\text{line},i}$. The dimensions of \mathbf{h}_{mod} and $\mathbf{h}_{\text{line},i}$ are both 128.

Classifiers. We employ tree-based gradient boosting models, XGBoost [3] and LightGBM [7], as final classifiers due to their robustness and ability to handle mixed and redundant features. We use XGBoost [3] and LightGBM [7] for all three tasks: (i) binary module-level Trojan detection, (ii) multi-class module-level Trojan-type prediction, and (iii) binary line-level Trojan localization.

Baselines. For module-level tasks, we compare TrojanLoC against: (i) GNN4TJ [20], with a GNN to extract embeddings from the DFG of an RTL design, (ii) TrojanSAINT [8], with a GNN to extract embeddings from the gate-level netlist of an RTL design. We derive the gate-level netlist by synthesizing an RTL design with the Synopsys 90nm Generic Library through Yosys [18]¹, and (iii) LLM-based Trojan detection baseline [4], where a general LLM is prompted with the RTL module and asked whether the module is Trojaned.

Metrics. There are four primary evaluation metrics used in the experimental section: accuracy (denote as Acc), precision (denoted as P), recall (denoted as R), and F1. Given *true positives* (TP), *false positives* (FP), and *false negatives* (FN), they are defined as:

$$P = \frac{TP}{TP + FP}, \quad R = \frac{TP}{TP + FN},$$

$$F1_{\text{clean}} = \frac{2 \cdot P \cdot R}{P + R}, \quad \text{Acc} = \frac{TP + TN}{TP + FP + TN + FN}.$$

For module-level Trojan detection, we report precision, recall, and F1 for the Trojan class. For module-level Trojan-type prediction, we report accuracy, precision, recall, and F1 over the four Trojan categories (T1–T4). For line-level Trojan localization, we treat each line as a sample and report F1 for the non-Trojan and Trojan classes.

Hardware. All embedding extraction and classifier training in our experiments are performed on a single NVIDIA H100 GPU.

4.2 Module-Level Trojan Detection

We evaluate TrojanLoC on two module-level tasks: (i) binary Trojan detection (clean vs. Trojan), and (ii) Trojan-type prediction (T1–T4). Unless noted, all module embeddings \mathbf{z}_{mod} are projected through the module autoencoder to \mathbf{h}_{mod} and used as classifier inputs.

4.2.1 Comparison with Graph and Prompting Baselines. TrojanLoC yields strong improvements across all metrics. Table 2 compares TrojanLoC against graph-based baselines (GNN4TJ, TrojanSAINT) and prompting baselines using GPT-4o and GPT-4o-mini. Graph-based methods fail due to inherent limitations. GNN4TJ cannot

¹We use the Synopsys generic library created for educational purposes instead of a foundry-provided standard cell library, as using the latter may result in NDA violation.

Table 2: Module-level HT detection: baselines vs. TrojanLoC.

Model	Acc	P _{Trojan}	R _{Trojan}	F1 _{Trojan}	F1 _{clean}
Baseline					
GNN4TJ	0.27	0.66	0.20	0.31	0.23
TrojanSAINT	0.80	0.84	0.91	0.88	0.47
GPT-4o-mini	0.87	0.98	0.85	0.91	0.75
GPT-4o	0.94	0.99	0.94	0.96	0.87
TrojanLoC					
CLVerilog-XGB	0.98	0.99	0.99	0.99	0.96
CLVerilog-LGBM	0.99	0.98	0.99	0.99	0.96
CodeV-QW-XGB	0.97	0.98	0.99	0.98	0.93
CodeV-QW-LGBM	0.97	0.98	0.99	0.98	0.93
HaVen-CodeQWen-XGB	0.98	0.99	0.99	0.99	0.95
HaVen-CodeQWen-LGBM	0.98	0.99	1.00	0.99	0.96

process 14.09% of designs due to its DFG transformations and lack of submodule definitions; even for processed designs, it achieves only 0.27 accuracy, as DFG embeddings cannot capture Trojan patterns. TrojanSAINT performs node-level classification, and we mark a design as Trojaned if any node is predicted malicious. However, 21.24% of designs fail to synthesize because they are non-top-level modules lacking complete module hierarchy information required for synthesis. For processed designs, it achieves 0.80 accuracy.

We evaluate the direct LLM-based Trojan detection baseline over GPT-4o-mini and GPT-4o. Both GPT-4o-mini and GPT-4o achieve relative high accuracy 0.87 and 0.94 respectively, and high F1_{Trojan} 0.91 and 0.96 respectively. However, for the prediction of the clean class, they achieve a relative low performance with F1_{clean} 0.75 and 0.87 respectively, which indicates that generic code reasoning of general LLMs is insufficient for capturing subtle RTL trigger/payload patterns. In contrast, all RTL-finetuned variants of TrojanLoC achieve F1_{Trojan} = 0.98–0.99 and F1_{clean} = 0.93–0.96, showing that domain-adapted LLM embeddings provide the semantic granularity required for robust Trojan detection.

4.2.2 Effect of RTL-Finetuned LLMs and Classifiers. Table 2 presents an ablation across LLM backbones and gradient-boosted classifiers. RTL-finetuned LLMs (CLVerilog, CodeV-QW, HaVen-CodeQWen) consistently outperform direct prompting by a wide margin, confirming that exposure to hardware syntax, common coding patterns is crucial for learning Trojan-related semantics. Differences between the three RTL-finetuned LLMs are minor, suggesting that our TrojanLoC can be applicable to different LLMs finetuned using different training datasets. Classifier choice minimally affects binary detection (F1_{Trojan} ≈ 0.99 across all settings).

4.3 Module-Level Trojan Type Prediction

Beyond binary detection, TrojanLoC aims to classify the effect-based Trojan category (T1–T4) of the module. This task is more challenging, as it entails distinguishing subtle semantic differences in the trigger and payload logic that define a Trojan family.

4.3.1 Type Classification Performance. Table 3 summarizes TrojanLoC’s performance across all backbones and classifiers. Accuracy

and macro-F1 range from 0.80 to 0.84, with best results from LightGBM paired with CodeV-QW-7B or HaVen-CodeQWen-7B. The embeddings retain sufficient semantic resolution to separate the four categories, despite many Trojans modifying only a few RTL lines. Notably, the narrow performance range indicates minimal impact from model size, fine-tuning effectively captures normal design patterns, enabling strong clean/Trojan distinction (Table 2) while maintaining consistent type classification across model scales.

We observe moderate performance variation across backbones. Models with stronger RTL specialization yield more discriminative embeddings for type classification. This trend reflects each model’s training emphasis, HaVen-CodeQWen-7B prioritizes functional consistency and structural reasoning, directly benefiting anomaly-oriented pattern recognition such as HT types; CodeV-QW emphasizes instruction fidelity; while CL-Verilog-13B is largely syntax-driven, resulting in progressively weaker structural discrimination. Classifier choice has a secondary effect. LightGBM consistently outperforms XGBoost in macro-F1, suggesting its leaf-wise splitting captures finer boundaries between Trojan types.

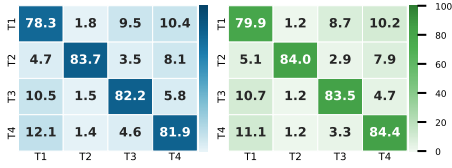


Figure 4: Average accuracy for XGB (L) and LGBM (R) on CLVerilog, CodeQWen and CodeV for module-level detection.

4.3.2 Per-Type Analysis and Confusion Patterns. A per-type analysis reveals performance patterns. Information leakage Trojans (T2) and performance degradation Trojans (T4) achieve the highest accuracies, $\approx 84\%$, driven by their distinctive routing and timing signatures. Functional-modification Trojans (T1) remain the most challenging, with accuracies $\approx 80\%$, while denial of service Trojans (T3) fall in range near 83%. As shown in Figure 4, both classifiers display comparable diagonal accuracy, though LightGBM offers sharper separation for T2 and T4. This advantage is evident for T4, where LightGBM reaches 84.4% compared to 81.9% for XGBoost, showing stronger sensitivity to timing and routing Trojans.

The confusion matrix highlights the structural proximity among categories. T1 is often mistaken for T3 and T4, consistent with their shared reliance on conditional control-flow changes that reshape datapath behavior. These overlaps blur semantic boundaries and make fine-grained discrimination difficult. In contrast, T2 exhibits the clear separation with minimal cross-type confusion, underscoring the distinctiveness of its leakage-oriented logic patterns.

4.4 Line-Level Trojan Localization

We now evaluate TrojanLoC on fine-grained Trojan localization, where the goal is to identify the specific RTL lines responsible for trigger or payload behavior. This task is considerably more demanding than module-level prediction, as Trojan lines are sparse, often resemble benign control or assignment statements, and may differ from clean logic only in subtle conditional dependencies.

Table 3: Module-level HT type prediction (T1–T4). Reported: overall accuracy, macro-averaged precision/recall/F1.

Model	Acc	P _{macro}	R _{macro}	F1 _{macro}
CLVerilog-XGB	0.80	0.81	0.80	0.80
CLVerilog-LGBM	0.81	0.82	0.81	0.81
CodeV-QW-XGB	0.82	0.83	0.82	0.82
CodeV-QW-LGBM	0.84	0.84	0.84	0.84
HaVen-CodeQWen-XGB	0.82	0.83	0.82	0.82
HaVen-CodeQWen-LGBM	0.84	0.85	0.84	0.84

Table 4: Ablation on line-level localization using CLVerilog + XGBoost, where m =module embedding, p =context window.

Model	Metric	$p=0$		$p=3$		$p=5$	
		$m=0$	$m=1$	$m=0$	$m=1$	$m=0$	$m=1$
CLVerilog-XGB	$F1_{\text{clean}}$	0.94	0.93	0.96	0.96	0.96	0.96
	$F1_{\text{Trojan}}$	0.81	0.81	0.87	0.88	0.88	0.89
	$F1_{\text{macro}}$	0.88	0.87	0.92	0.92	0.92	0.93
CodeV-QW-XGB	$F1_{\text{clean}}$	0.94	0.94	0.96	0.96	0.96	0.96
	$F1_{\text{Trojan}}$	0.80	0.82	0.88	0.88	0.87	0.88
	$F1_{\text{macro}}$	0.87	0.88	0.92	0.92	0.92	0.92
HaVen-CodeQWen-XGB	$F1_{\text{clean}}$	0.94	0.93	0.96	0.96	0.96	0.96
	$F1_{\text{Trojan}}$	0.81	0.80	0.89	0.88	0.88	0.88
	$F1_{\text{macro}}$	0.88	0.87	0.93	0.92	0.92	0.92

4.4.1 Effect of Module Embedding and Context. To understand design choices behind line-level performance, we perform controlled ablation using CLVerilog+XGBoost, varying (i) module-level embedding concatenation ($m \in \{0, 1\}$) and (ii) symmetric context window size p , shown in Tab. 4. **(1) Impact of context window size:** Context is crucial. Increasing p from $0 \rightarrow 3$ raises $F1_{\text{Trojan}}$ from 0.81 to 0.88, saturating around $p = 3-5$. This matches RTL coding practice: Trojans typically span short multi-line patterns (e.g., trigger condition followed by payload assignment) rarely extending beyond ± 5 lines. Larger windows provide diminishing returns and introduce noise. **(2) Impact of module embeddings:** Without context ($p = 0$), module embedding offers minimal benefit ($F1_{\text{Trojan}} = 0.81$ for both $m = 0, 1$). With context, module embeddings consistently help—at $p = 5$, $F1_{\text{macro}}$ rises from 0.92 to 0.93 when $m = 1$. This shows global semantics become useful only when combined with local context: line embeddings capture local behavior while module embeddings provide global consistency cues.

4.4.2 Localization Results. Table 4 shows the best configuration for each RTL-finetuned backbone. TrojanLoC achieves $F1_{\text{clean}} = 0.96$ and $F1_{\text{Trojan}} = 0.88-0.89$, yielding $F1_{\text{macro}} = 0.92-0.93$. These results confirm that RTL-tuned embeddings retain token-level semantics for line-level reasoning beyond coarse module classification. The ~ 8 -point gap reflects the difficulty of identifying minimally intrusive malicious lines differing from benign logic by only small conditionals or subtle assignments. TrojanLoC’s suspiciousness scores enable analysts to review only the top few percent of ranked lines, true Trojan lines consistently appear within the top 3–5%, reducing manual auditing by over an order of magnitude.

5 Conclusion

We presented TrojanLoC, an LLM-based framework for fine-grained HT detection and localization directly from RTL code. Built on TrojanInS, a dataset with 17k+ designs covering four major Trojan families, TrojanLoC extracts module and line level embeddings using RTL-finetuned LLMs and employs lightweight classifiers for unified binary detection, type prediction, and line-level localization. By avoiding lossy netlist or graph conversions, TrojanLoC preserves RTL semantics and achieves 0.99 F1-score for module-level detection and 0.92 macro-F1 for line-level localization, outperforming graph-based baselines. Its ranked suspiciousness scores surface true Trojan lines within the top few percent, reducing manual auditing effort by over an order of magnitude and offering a practical solution for scalable hardware security validation.

References

- [1] T. Bui, T. Vu, T. Nguyen, S. Nguyen, and H. Vo. 2025. Correctness Assessment of Code Generated by Large Language Models Using Internal Representations. *arXiv:2501.12934* [cs.SE]
- [2] Lihan Chen, Chen Dong, Qiaowen Wu, Ximeng Liu, Xiaodong Guo, Zhenyi Chen, Hao Zhang, and Yang Yang. 2024. Gnn4ht: A two-stage gnn based approach for hardware trojan multifunctional classification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).
- [3] T. Chen and C. Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *KDD*. 785–794.
- [4] V. T. Hayashi and W. V. Ruggiero. 2025. Hardware Trojan Detection in Open-Source Hardware Designs Using Machine Learning. *IEEE Access* 13 (2025), 37771–37788.
- [5] R. Hemadri, J. Bhandari, A. Nakkab, J. Knechtel, B. Gopalan, R. Narayanaswamy, R. Karri, and S. Garg. 2025. VeriLoC: Line-of-Code Level Prediction of Hardware Design Quality from Verilog Code. *arXiv:2506.07239*
- [6] Zhao Huang, Quan Wang, Yin Chen, and Xiaohong Jiang. 2020. A survey on machine learning against hardware trojan attacks: Recent advances and challenges. *IEEE Access* 8 (2020), 10796–10826.
- [7] G. Ke et al. 2017. LightGBM: a highly efficient gradient boosting decision tree. In *NIPS*. 3149–3157.
- [8] H. Lashen, L. Alrahis, J. Knechtel, and O. Sinanoglu. 2023. TrojanSAINT: Gate-Level Netlist Sampling-Based Inductive Learning for Hardware Trojan Detection. In *ISCAS*. 1–5.
- [9] A. Nakkab, S. Zhang, R. Karri, and S. Garg. 2024. Rome was Not Built in a Single Step: Hierarchical Prompting for LLM-based Chip Design. In *MLCAD*.
- [10] H. Salmani, M. Tehranipoor, S. Sutikno, and F. Wijitrisnanto. 2022. Trust-Hub Trojan Benchmark for Hardware Trojan Detection Model Creation using Machine Learning. *IEEE Dataport* (2022).
- [11] Minghao Shao, Abdul Basit, Ramesh Karri, and Muhammad Shafique. 2024. Survey of different large language model architectures: Trends, benchmarks, and challenges. *IEEE Access* (2024).
- [12] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg. 2024. VeriGen: A Large Language Model for Verilog Code Generation. *ACM Trans. Des. Autom. Electron. Syst.* 29, 3 (2024).
- [13] Kiran Thorat, Amit Hasan, Caiwen Ding, and Zhijie Shi. 2025. TROJAN-GUARD: Hardware Trojans Detection Using GNN in RTL Designs. *arXiv preprint arXiv:2506.17894* (2025).
- [14] Zeng Wang, Lilas Alrahis, Likhitha Mankali, Johann Knechtel, and Ozgur Sinanoglu. 2024. LLMs and the future of chip design: Unveiling security risks and building trust. In *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 385–390.
- [15] Zeng Wang, Minghao Shao, Jitendra Bhandari, Likhitha Mankali, Ramesh Karri, Ozgur Sinanoglu, Muhammad Shafique, and Johann Knechtel. 2025. VeriContaminated: Assessing LLM-driven verilog coding for data contamination. *arXiv preprint arXiv:2503.13572* (2025).
- [16] Zeng Wang, Minghao Shao, Rupesh Karn, Jitendra Bhandari, Likhitha Mankali, Ramesh Karri, Ozgur Sinanoglu, Muhammad Shafique, and Johann Knechtel. 2025. SALAD: Systematic Assessment of Machine Unlearning on LLM-Aided Hardware Design. *arXiv preprint arXiv:2506.02089* (2025).
- [17] Zeng Wang, Minghao Shao, Mohammed Nabeel, Prithwish Basu Roy, Likhitha Mankali, Jitendra Bhandari, Ramesh Karri, Ozgur Sinanoglu, Muhammad Shafique, and Johann Knechtel. 2025. Verileaky: Navigating ip protection vs utility in fine-tuning for llm-driven verilog coding. *arXiv preprint arXiv:2503.13116* (2025).
- [18] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, Vol. 97.
- [19] Y. Yang et al. 2025. HaVen: Hallucination-Mitigated LLM for Verilog Code Generation Aligned with HDL Engineers. *arXiv:2501.04908*
- [20] R. Yasaei, S. Yu, and M. Faruque. 2021. GNN4TJ: Graph Neural Networks for Hardware Trojan Detection at Register Transfer Level. In *DATE*. 1504–1509.
- [21] Farhath Zareen and Robert Karam. 2018. Detecting RTL trojans using artificial immune systems and high level behavior classification. In *2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 68–73.
- [22] Senjie Zhang, Shan Zhou, Panpan Xue, Lu Kong, and Jinbo Wang. 2025. GNN-MFF: A Multi-View Graph-Based Model for RTL Hardware Trojan Detection. *Applied Sciences* 15, 19 (2025), 10324.
- [23] Y. Zhao et al. 2024. CodeV: Empowering LLMs for Verilog Generation through Multi-Level Summarization. *arXiv:2407.10424*
- [24] Z. Zhou et al. 2024. How Alignment and Jailbreak Work: Explain LLM Safety through Intermediate Hidden States. In *EMNLP*. 2461–2488.