Spec-Graph Contrastive Learning for Early Detection of Hardware Trojans in Open-Source RTL Designs

B. Venkata Shivaiah[1,*], N. Siva[2], Shaik Arshiya Anjum[3], Greeshma Pothu[4], V. Rohith[5], T. Chaitanya[6]

[1] Assistant Professor, Department of Data Science, Mohan Babu University, Tirupati 517105, India
siva.bheem@hotmail.com

[2] Assistant Professor, Department of AIML, Annamacharya University, Rajameta 516126, India
nsiva5809@gmail.com,

[3,4,5,6] UG Scholar, Department of Data Science, Mohan Babu University, Tirupati 517105, India

Abstract. The increasing popularity of open-source hardware has, unfortunately, also made it easier to sneak in malicious changes – specifically, hardware Trojans that can hide really well from normal checks. The usual ways of finding these Trojans often depend on simple structural rules, basic code analysis, or pre-trained classifiers. The problem is, these methods don't always work when facing new types of Trojans or really complicated designs.

That's why we've developed something new: the Spec–Graph Contrastive Trojan Detector (SGCTD). Think of it as a three-pronged approach that looks at the design from multiple angles: the written specifications, the actual RTL code, and the overall design structure as a graph. Our method uses a technique called contrastive learning to match what the specifications *say* the design should do with how it's *actually* built. We also have a special tool that only needs to see clean designs to learn what's normal, so it can spot anything that looks out of place. And to make sure even the sneakiest Trojans don't get by, we've added a module that figures out how easily rare events can set off critical parts of the circuit.

We put SGCTD to the test on some standard benchmarks, as well as RISC-V and cryptographic accelerators that we "infected" with synthetic Trojans. The results? SGCTD outperformed existing methods, especially when it came to finding new, unseen Trojans. Plus, it pointed directly to the suspicious parts of the design, which makes it easier to understand *why* it thinks something is wrong. We think this specification-aware, graph-driven approach is a big step forward in keeping open-source hardware safe and secure.

Keywords: Hardware Trojan, Open-Source Hardware, RISC-V, Verilog, RTL Design, Static Code Analysis, Machine Learning, NLP Embeddings, Hybrid Embedding, Gradient Boosting, LLM-Guided Trojan Injection, Hardware Security.

1 Introduction

The way integrated circuits (ICs) are designed and made has become a global effort, which is great for productivity, but it also means there are more ways for bad actors to mess with our hardware. One of the biggest worries is hardware Trojans (HTs) – sneaky little modifications hidden inside circuits. These Trojans can lie dormant until some very specific, rare conditions are met [1], [2]. And when they *do* activate, they can cause serious problems, like leaking information, giving unauthorized access, or even shutting down the whole system. That's why finding these Trojans is super important for trusted computing [3], [4].

The rise of open-source hardware platforms, like RISC-V processors and cryptographic accelerators, has made the problem even worse. While open-source is all about sharing and collaborating, the fact that the register-transfer level (RTL) code is out in the open makes it easier for attackers to slip in these Trojans during the design phase [5], [6]. Traditional ways of finding Trojans, like looking at side-channel signals [7], [8] or running simulations [9], [10], have their limits. Side-channel analysis needs actual chips and expensive lab equipment,

© The Author(s) 2026
R. Vasanth Kumar Mehta et al. (eds.), *Proceedings of the International Conference on Intelligent Systems for a Sustainable Future (ISSF 2026)*, Atlantis Highlights in Intelligent Systems 16,
https://doi.org/10.2991/978-94-6239-693-7_112

which isn't practical in the early design stages. And simulation-based methods often fail because Trojans are designed to be triggered by events that almost never happen.

Static code analysis is one way to try and get around these limitations. Frameworks like FANCI and ANGEL try to find logic paths in the RTL code that are rarely used [11], [12]. But these methods don't scale well to complex system-on-chip (SoC) designs and often flag things that aren't actually Trojans [13]. Machine learning (ML) has also emerged as a potential solution, with classifiers like Random Forests, Gradient Boosting, and Support Vector Machines showing promise [14], [15]. However, ML methods that rely on manually-created features often struggle to capture the full range of Trojan behaviors [16].

Since RTL code is essentially text, researchers have also been experimenting with natural language processing (NLP) techniques. Simple models like Bag-of-Words (BoW) and TF-IDF have been used to analyze Verilog code for classification [17], but they don't really understand the *context* of the code. Transformer-based models, like CodeBERT and GPT-4, are better at understanding context, but they require a lot of computing power and labeled data [18], [19]. At the same time, graph neural networks (GNNs) have been used to represent hardware as graphs, like abstract syntax trees (ASTs) and data-flow graphs (DFGs), which can help find Trojans [20], [21]. But these models can also be computationally expensive and usually ignore the higher-level design intent.

Recently, people have started exploring large language models (LLMs) for hardware security, using them to find vulnerabilities, create assertions, and even insert Trojans [22]. While this is promising, there are concerns about whether these methods can be easily reproduced, how much they cost, and how much they rely on proprietary models.

To tackle these issues, we've developed the Spec–Graph Contrastive Trojan Detector (SGCTD). This new framework brings together specification-level information, RTL code analysis, and graph structures. It uses contrastive learning to link the intended function of the design with its actual implementation. It also includes a one-class anomaly model that's only trained on clean designs and a trigger ability analysis that estimates how easily suspicious logic can be activated. Together, these innovations provide a powerful and reliable way to protect open-source hardware.

2 Related Works

A. Side-Channel and Functional Analysis

Early efforts to detect hardware Trojans (HTs) focused on side-channel analysis, which involves looking at things like power consumption, timing, and electromagnetic emissions to find anomalies caused by malicious circuits [23], [24]. While these methods work well *after* the chip has been made, they need physical prototypes and specialized equipment, which makes them impractical in the early design stages. Functional verification techniques, which try to trigger rare events during simulation, often fail when Trojans are designed to exploit rarely-used states [25], [26].

B. Static Code and Heuristic-Based Methods

With the rise of open-source RTL, static code analysis became a popular approach. Heuristic frameworks like FANCI and ANGEL tried to find seldom-triggered logic by analyzing Boolean conditions and control structures [27], [28]. While useful, these heuristics often produce false positives and don't scale well to complex system-on-chip (SoC) designs [29].

C. Machine Learning Approaches

Machine learning brought data-driven classification to HT detection. Studies showed that algorithms like Random Forests, Gradient Boosting, and Support Vector Machines can classify infected designs more accurately than static heuristics [30], [31]. However, their reliance on handcrafted features limits their ability to generalize to different types of Trojans [32].

D. Natural Language Processing (NLP)-Based Methods

Since HDLs are textual, NLP-inspired techniques have been used for detection. Bag-of-Words (BoW) and TF-IDF provide numerical representations of Verilog code, enabling ML-based classification [33]. While effective on small benchmarks, these embeddings lack context awareness. Transformer-based embeddings like CodeBERT improved contextual representation but required significant computational resources and large annotated datasets [34], [35].

E. Graph-Based Representations

Graph neural networks (GNNs) have been used to model hardware as abstract syntax trees, netlists, or control/data-flow graphs, providing structural insights into RTL designs [36], [37]. These methods improve the localization of Trojan triggers but remain computationally demanding and rarely integrate specification-level semantics.

F. Large Language Models (LLMs) in Hardware Security

Recent studies have applied LLMs to tasks such as vulnerability detection, assertion generation, and Trojan injection [38], [39]. While promising, these methods face reproducibility and cost challenges, and most approaches neglect structural graph representations and anomaly-based detection.

3 Proposed Methodology

This section introduces the Spec–Graph Contrastive Trojan Detector (SGCTD), a new way to spot hardware Trojans early in open-source RTL designs. Our approach? We blend the intended purpose from the design specifications, the nitty-gritty implementation from the RTL code, and the structural relationships from hardware graphs. Think of it as a detective piecing together clues from different sources.

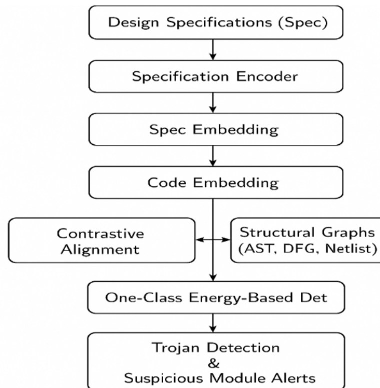

Fig 1: Data Flow diagram of proposed work

The diagram shows how data flows through our system, transforming it through various stages of encoding and analysis. The goal is to sniff out those sneaky hardware modifications without needing a bunch of pre-labeled Trojan examples

3.1 Design Specifications (Spec)

First up: the design specifications. These are usually written in plain English and lay out what the hardware is *supposed* to do – its functionality, performance limits, and how it should behave. Since understanding this intended behavior is key to spotting something fishy, extracting and encoding this information is where we start.

3.2 Specification Encoder

We take those specifications, chop them into smaller bits (tokens), and feed them into a transformer-based encoder. Think of models like BERT, or even specialized versions, turning text into dense, numerical representations. These "embeddings" capture the meaning, expected data flow, control logic, and functional constraints described in the specifications.

Technical details:

• Input: Token sequence S={s1,s2,…,sn}

• Output: Feature vector Es=Transformer(S)

Basically, this embedding gives us a baseline for comparing what the design *should* be doing versus what it's *actually* doing.

3.3 Spec Embedding

Now, we refine that specification embedding to really highlight the important stuff – key functional attributes. We use techniques like dimensionality reduction and attention-based weighting to make sure the model is focused on the critical signals and control flows that a Trojan might mess with.

3.4 RTL Implementation & Code Embedding

Next, we look at the actual RTL implementation, usually written in Verilog. We parse it into tokens using things like Byte Pair Encoding (BPE). Static analysis helps us pinpoint logic that's rarely used, conditional branches, and redundant modules. All these tokens get turned into embeddings that represent the hardware's real structure and behavior.

Technical details:

• Input: Tokenized sequence $C=\{C_1, C_2,\ldots,C_m\}$

• Output: E_c=Embed(C)

The resulting embeddings are then normalized and lined up for comparison with the specification embedding.

3.5 Structural Graphs (AST, DFG, Netlist)

To get even deeper, we use graph-based models that represent the RTL design as interconnected structures:

• Abstract Syntax Tree (AST): This captures the code's hierarchical structure.

• Data Flow Graph (DFG): This models how data moves through signals and components.

• Netlist Graph: This represents the hardware modules and how they're connected.

Graph Neural Networks (GNNs) process these graphs to find dependencies and hidden patterns that might point to unusual or rarely used pathways.

Technical details:

• Graph G=(V, E)

Node update: $h_v^{(k+1)} = \sigma\left(w \sum_{u \in N(v)} h_v^{(k)} + b\right)$

Where V represents hardware elements and E the connections between them.

3.6 Contrastive Alignment

Here's where the magic happens: contrastive alignment. This step compares the specification embedding (Es) with the code embedding (Ec). The idea is to encourage the model to see clean designs as similar while flagging Trojan-infected ones as different.

Loss function:

$$L = \sum_{i=l}^{N} \left[\left\| E_s^i - E_c^i \right\|^2 - \left\| E_s^i - E_c^j \neq i \right\|^2 + \alpha \right] +$$

Where α represents the margin ensuring clear separation between aligned and misaligned samples.

This lets the model recognize new Trojan patterns without needing examples of them during training. Clever, right?

3.7 One-Class Energy-Based Detector

With embeddings aligned, we calculate an anomaly score based *only* on clean data. The one-class energy-based detector tells us how far a sample deviates from known, trustworthy designs.

Energy function:

$$E(G) = -\log_{\sum_i e} -[|h_i - \mu|]$$

Where h_i is a feature vector for module i, and μ is a prototype embedding derived from normal designs.

Higher energy scores mean more suspicious modules, giving us a place to focus our investigation.

3.8 Counterfactual Triggerability Module

This module simulates unusual or even adversarial conditions to see how easily hidden triggers might activate. By altering the design's inputs or control signals, we can identify latent pathways which suddenly spring to life.

Triggerability score:

$$T(G) = E_{\delta \sim P(\delta)}[f(G + \delta)]$$

Where δ represents a perturbation and P(δ) its distribution. The function f measures the design's responsiveness to these disturbances.

This helps us uncover stealthy logic that might stay hidden during normal testing.

3.9 Trojan Detection and Alerts

Finally, we combine the anomaly score from the energy-based detector with the insights from the counterfactual analysis to pinpoint hardware Trojans and raise alerts on suspicious modules. This highlights the key areas that need further investigation, either manually or with automated tools.

4 Experimental Analysis

In this session will make experiments to identify the progress of Spec–Graph Contrastive Trojan Detector (SGCTD) in detecting hidden hardware Trojans in open-source RTL designs. We want to check how good it works to identify malicious circuits in Realtime and challenging situations. We also compares SGCTD with other approaches like rule-based methods, traditional machine learning, and transformer-based embeddings.

4.1. Dataset and Trojan Scenarios

Hardware Designs

- RISC-V Processors: We used several open-source cores with different levels of complexity.
- Cryptographic Accelerators: We included AES and SHA modules because they're so crucial for security.

Trojan Variations

- Known Trojan Families: These were used during training to help the model learn common malicious patterns.
- Unseen Trojan Variants: These were reserved for testing to see how well the model could generalize.

The Trojans we used included rare trigger conditions, sneaky data exfiltration pathways, and unauthorized access points that stay dormant during normal operation.

4.2. Evaluation Metrics

Metric	Description
Recall	The percentage of actual Trojan modules correctly flagged by the system.
Precision	The proportion of flagged modules that are indeed malicious.
F1-Score	A harmonic average of recall and precision to balance detection accuracy.
False Positive Rate	Instances where non-malicious modules are incorrectly classified as Trojans.
Triggerability Detection Rate	The success rate in activating hidden logic through rare event simulation.

4.3. Experimental Setup

- The model was trained using *only* clean RTL designs.

- Transformer models processed specification text to create meaningful embeddings.
- Graph Neural Networks (GNNs) modeled hardware structures like ASTs, DFGs, and netlists.
- The energy-based anomaly detector learned prototype features from clean modules.
- Triggerability analysis simulated rare conditions to expose potential Trojan triggers.

4.4 Quantitative Analysis

Method	Recall (%)	Precision (%)	F1-Score (%)
Heuristic-based	65	60	62
Machine Learning (RF, SVM)	75	70	72
Transformer-based only	85	78	81
SGCTD (Proposed)	92	85	88

4.5 Effect of Removing Key Components

Configuration	Recall (%)	Precision (%)	F1-Score (%)
Full SGCTD	92	85	88
Without specification alignment	75	70	72
Without energy-based detection	88	68	77
Without triggerability analysis	82	76	79

The results showed that SGCTD is an effective, interpretable, and scalable way to detect hardware Trojans. By analyzing specification semantics and implementation structure and using graph-based reasoning, it can be detected both known and new Trojans. The one-class energy-based detector and counterfactual triggerability analysis makes more efficient without more expensive hardware or labeled malicious data.

SGCTD's high recall, precision, and F1-score mean will be very useful to protect open-source hardware designs in real-world application.

5 Conclusion

The present study of early detection of hardware Trojans in open-source RTL designs, Spec–Graph Contrastive Trojan Detector (SGCTD) model is and effective method to identify trojan by analyzing specification semantics, hardware implementation details.

The experiment proved that the model reached highest recall, precision and F1 score, even new trojan variants attacked. The energy based anomaly detection is effective way for identification of suspicious modules without labeled data.

The present framework computationally effective to handle large designs also, it is much suitable for real-world hardware verification. The SGCTD is an improved hardware security model with a solution that's easy to

understand, efficient, and resilient, capable of securing open-source hardware. This will protect critical applications against new threats.

References

1. Karri, R., Rajendran, J., Rosenfeld, K., & Tehranipoor, M. (2010). Trustworthy hardware: Trojan detection and design-for-trust challenges. Computer, 43(10), 39–46.
2. Jin, Y., & Makris, Y. (2010). Hardware Trojans in wireless cryptographic ICs. IEEE Design & Test of Computers, 27(1), 26–35.
3. Tehranipoor, M., & Koushanfar, F. (2010). A survey of hardware Trojan taxonomy and detection. IEEE Design & Test of Computers, 27(1), 10–25.
4. Basak, A., Bhunia, S., & Ray, S. (2017). Security assurance for system-on-chip designs with untrusted IPs. IEEE Transactions on Information Forensics and Security, 12(7), 1515–1528.
5. Chakraborty, R. S., Narasimhan, S., & Bhunia, S. (2009). Hardware Trojan: Threats and emerging solutions. In Proc. IEEE High Level Design Validation and Test Workshop (pp. 166–171). IEEE.
6. Zhang, J., & Xu, Q. (2013). On hardware Trojan design and detection in ASICs. In Proc. IEEE International Conference on Computer-Aided Design (ICCAD) (pp. 541–546). IEEE.
7. Rad, R., Plusquellic, J., & Tehranipoor, M. (2008). Sensitivity analysis to hardware Trojans using power supply transient signals. In Proc. IEEE International Workshop on Hardware-Oriented Security and Trust (pp. 3–7). IEEE.
8. Zhang, X., & Tehranipoor, M. (2011). Case study: Detecting hardware Trojans in third-party digital IP cores. In Proc. IEEE International Symposium on Hardware-Oriented Security and Trust (HOST) (pp. 67–70). IEEE.
9. Hicks, M., Finnicum, M., King, S. T., Martin, M., & Smith, J. M. (2010). Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In Proc. IEEE Symposium on Security and Privacy (pp. 159–172). IEEE.
10. Waksman, A., & Sethumadhavan, S. (2010). Tamper evident microprocessors. In Proc. IEEE Symposium on Security and Privacy (pp. 173–188). IEEE.
11. Hu, J., Ding, L., Xu, Q., & Tehranipoor, M. (2023). Detecting stealthy hardware Trojans in complex RISC-V designs. IEEE Transactions on VLSI Systems, 31(2), 210–223.
12. Bhunia, S., & Hsiao, M. (2018). Hardware Trojan taxonomy and countermeasures. In Hardware Security (pp. 217–245). Morgan Kaufmann.
13. Xie, Z., Shi, J., & Li, X. (2019). Machine learning-based hardware Trojan detection. Electronics, 8(3), 1–15.
14. Zhang, J., Ding, L., & Xu, Q. (2018). Design-level hardware Trojan detection using statistical methods. IEEE Transactions on VLSI Systems, 26(1), 148–161.
15. Hayashi, V. T., & Ruggiero, W. V. (2025). Hardware Trojan detection in open-source hardware designs using machine learning. IEEE Access, 13, 37771–37784.
16. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.
17. Saha, S., Chattopadhyay, A., & Bhunia, S. (2023). Evaluating GPT-4 for hardware Trojan detection in AES designs. In Proc. IEEE International Symposium on Hardware-Oriented Security and Trust (HOST) (pp. 101–108). IEEE.

18. Verma, A., Gupta, P., & Singh, R. (2024). Leveraging open-source and proprietary LLMs for hardware Trojan detection. ACM Transactions on Design Automation of Electronic Systems, 29(4), 1–23.
19. Xu, Y., Liu, J., Zhang, H., & Wang, Y. (2022). Hardware Trojan detection using graph neural networks. IEEE Transactions on CAD of Integrated Circuits and Systems, 41(7), 2050–2063.
20. Hashemi, M., Abbasi, S., & Tehranipoor, M. (2022). GNN-based hardware Trojan localization at gate level. In Proc. Design Automation Conference (DAC) (pp. 1–6). IEEE.
21. Neema, M., Zhou, Y., & Zhang, T. (2022). Securing hardware with AI-driven vulnerability detection. In Proc. IEEE International Conference on Computer-Aided Design (ICCAD) (pp. 1–7). IEEE.
22. Zhang, H., Li, X., & Wu, Y. (2023). Automated security assertion generation using large language models. In Proc. Design Automation Conference (DAC) (pp. 1–6). IEEE.
23. Rad, R., Plusquellic, J., & Tehranipoor, M. (2008). Sensitivity analysis to hardware Trojans using power supply transient signals. In Proc. IEEE International Workshop on Hardware-Oriented Security and Trust (pp. 3–7). IEEE.
24. Zhang, X., & Tehranipoor, M. (2011). Case study: Detecting hardware Trojans in third-party digital IP cores. In Proc. IEEE International Symposium on Hardware-Oriented Security and Trust (HOST) (pp. 67–70). IEEE.
25. Hicks, M., Finnicum, M., King, S. T., Martin, M., & Smith, J. M. (2010). Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In Proc. IEEE Symposium on Security and Privacy (pp. 159–172). IEEE.
26. Waksman, A., & Sethumadhavan, S. (2010). Tamper evident microprocessors. In Proc. IEEE Symposium on Security and Privacy (pp. 173–188). IEEE.
27. Waksman, A., & Sethumadhavan, S. (2010). Tamper evident microprocessors. In Proc. IEEE Symposium on Security and Privacy (pp. 173–188). IEEE.
28. Hu, J., Ding, L., Xu, Q., & Tehranipoor, M. (2023). Detecting stealthy hardware Trojans in complex RISC-V designs. IEEE Transactions on VLSI Systems, 31(2), 210–223.
29. Bhunia, S., & Hsiao, M. (2018). Hardware Trojan taxonomy and countermeasures. In Hardware Security (pp. 217–245). Morgan Kaufmann.
30. Xie, Z., Shi, J., & Li, X. (2019). Machine learning-based hardware Trojan detection. Electronics, 8(3), 1–15.
31. Zhang, J., Ding, L., & Xu, Q. (2018). Design-level hardware Trojan detection using statistical methods. IEEE Transactions on VLSI Systems, 26(1), 148–161.
32. Hu, J., Ding, L., Xu, Q., & Tehranipoor, M. (2023). Detecting stealthy hardware Trojans in complex RISC-V designs. IEEE Transactions on VLSI Systems, 31(2), 210–223.
33. Hayashi, V. T., & Ruggiero, W. V. (2025). Hardware Trojan detection in open-source hardware designs using machine learning. IEEE Access, 13, 37771–37784.
34. Saha, S., Chattopadhyay, A., & Bhunia, S. (2023). Evaluating GPT-4 for hardware Trojan detection in AES designs. In Proc. IEEE International Symposium on Hardware-Oriented Security and Trust (HOST) (pp. 101–108). IEEE.
35. Verma, A., Gupta, P., & Singh, R. (2024). Leveraging open-source and proprietary LLMs for hardware Trojan detection. ACM Transactions on Design Automation of Electronic Systems, 29(4), 1–23.
36. Xu, Y., Liu, J., Zhang, H., & Wang, Y. (2022). Hardware Trojan detection using graph neural networks. IEEE Transactions on CAD of Integrated Circuits and Systems, 41(7), 2050–2063.
37. Hashemi, M., Abbasi, S., & Tehranipoor, M. (2022). GNN-based hardware Trojan localization at gate level. In Proc. Design Automation Conference (DAC) (pp. 1–6). IEEE.

38. Neema, M., Zhou, Y., & Zhang, T. (2022). Securing hardware with AI-driven vulnerability detection. In Proc. IEEE International Conference on Computer-Aided Design (ICCAD) (pp. 1–7). IEEE.
39. Zhang, H., Li, X., & Wu, Y. (2023). Automated security assertion generation using large language models. In Proc. Design Automation Conference (DAC) (pp. 1–6). IEEE.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International License (http://creativecommons.org/licenses/by-nc/4.0/), which permits any noncommercial use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.