Detecting Vulnerability in Hardware Description Languages: Opcode Language Processing

Alaaddin Goktug Ayar, *Graduate Student Member, IEEE*, Abdullah Sahruri, *Graduate Student Member, IEEE*, Sercan Aygun, *Member, IEEE*, Mehran Shoushtari Moghadam, *Graduate Student Member, IEEE*, M. Hassan Najafi, *Senior Member, IEEE*, and Martin Margala, *Senior Member, IEEE*

***Abstract*—Detecting vulnerable code blocks has become a highly popular topic in computer-aided design, especially with the advancement of natural language processing (NLP). Analyzing hardware description languages (`HDLs`), such as Verilog, involves dealing with lengthy code. This letter introduces an innovative identification of attack-vulnerable hardware by the use of `opcode` processing. Leveraging the advantage of architecturally defined `opcodes` and expressing all operations at the beginning of each code line, the word processing problem is efficiently transformed into `opcode` processing. This research converts a benchmark dataset into an intermediary code stack, subsequently classifying secure and fragile codes using NLP techniques. The results reveal a framework that achieves up to 94% *accuracy* when employing sophisticated convolutional neural networks (CNNs) architecture with extra embedding layers. Thus, it provides a means for users to quickly verify the vulnerability of their `HDL` code by inspecting a supervised learning model trained on the predefined vulnerabilities. It also supports the superior efficacy of `opcode`-based processing in Trojan detection by analyzing the outcomes derived from a model trained using the `HDL` dataset.**

***Index Terms*—Convolutional neural networks (CNNs), deep learning, hardware description language (HDL), hardware security, natural language processing (NLP), opcode.**

I. Introduction

DEVELOPING secure code is of paramount importance not only in software engineering but also in hardware engineering. While the literature has seen considerable research on software-only vulnerabilities, the design of *hardware* necessitates a higher-level consideration and a prudent approach. Ensuring the secure coding of hardware modules using hardware description languages (**`HDLs`**) during design space exploration is crucial before reaching the register-transfer level (RTL). Identifying code vulnerabilities at this stage is critical and cost-effective before the actual hardware fabrication. The assessment of code written in **`HDL`** for potential adversarial scenarios gives rise to the need for either testing through a testbench to engage in heuristic analysis or employing machine learning (ML) techniques to detect known attack patterns. However, comprehensively testing the integrity of the code flow presents challenges, especially considering the complex syntax. A new concept of an intermediary code, similar to low-level code, can be applied to the **`HDL`** code. This is where the simplicity of **`assembly`** language comes into play. The hardware–software defined in the **`HDL`** is translated into an equivalent intermediary code, preserving the logic and flow of the original code. This process ensures that each individual operation is guaranteed as a single instruction, and the register transfers following the **`opcodes`**.

This study applies conventional ML and neural network (NN) approaches to the preliminary unification procedure in the **`assembly`** language. To test and evaluate the effectiveness of the methodology, a widely recognized benchmark, Trust-Hub [1], is employed. Notably, this study explores ML techniques when applied to a problem transferred into a different code space: from **`HDL`** to **`assembly`** via **`C++`**. The experimental results demonstrate fruitful findings for the benchmark. The key contribution of this letter lies in pioneering the processing of **`assembly`** codes from **`HDL`** for the detection of malign attacks. Furthermore, a novel ML exploration is presented, focusing on performance evaluation using various data-mining metrics based on the confusion matrix. Ultimately, the main objective is to introduce design steps that offer an intermediary security checkpoint within the integrated circuit (IC) supply chain. The structure of this letter is as follows. Section II provides the background and motivation of the study. Section III delves into the detailed presentation of our proposed methodology. For a comprehensive understanding of the conventional versus NN performance, Section IV offers a thorough ML-based design space exploration. In addition, Section V addresses the assumptions and limitations of this letter. Finally, the conclusions are outlined in Section VI.

II. Background and Motivation

Hardware trojans (HTs) are malicious modifications or insertions in the design or fabrication of ICs. These Trojans are intentionally introduced to compromise the security, reliability, or functionality of hardware systems without the knowledge of the end users. The insertion of HTs can occur at various stages of the IC's lifecycle, including specification, design, fabrication, testing, and packaging (please see Fig. 1). Conventional VLSI manufacturing, testing, and verification approaches demonstrate ineffectiveness in detecting HTs due to the unique and unmodeled characteristics of these malicious modifications. Moreover, the constraints in chip manufacturing and the absence of confidence in third-party foundries for chip fabrication further undermine the suitability of these

Manuscript received 21 August 2023; revised 18 October 2023; accepted 24 October 2023. Date of publication 21 November 2023; date of current version 30 May 2024. This work was supported in part by the National Science Foundation under Grant 2019511; by the University of Louisiana at Lafayette Computer Science Endowed Chair Fund; by the Louisiana Board of Regents Support Fund under Grant LEQSF(2020–23)-RD-A-26; by Cisco; by Xilinx; and by NVIDIA. This manuscript was recommended for publication by F. Merchant. *(Corresponding author: Sercan Aygun.)*

The authors are with the School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, LA 70503 USA (e-mail: alaaddin.ayar1@louisiana.edu; abdullah.sahruri1@louisiana.edu; sercan.aygun@louisiana.edu; m.moghadam@louisiana.edu; najafi@louisiana.edu; martin.margala@louisiana.edu).

Digital Object Identifier 10.1109/LES.2023.3334728

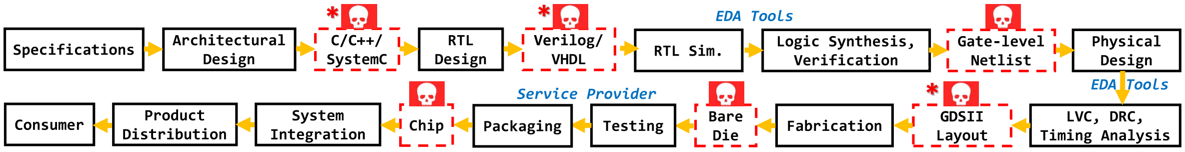


Fig. 1. IC supply chain, as defined by Yasaei et al. [9], involves the insertion of HTs during the early phases of design, either in C/C++/SystemC-related designs or in **HDL** designs. Detecting and inspecting these malicious actions early on can effectively prevent potential cumulative security breaches. Notably, third-party IP providers are marked with *, which can trigger security issues. Any participant in the digital design process is considered a potential attacker with the ability to introduce a module for altering the design as an integral component of the overall architecture rather than as a separate IP. (LVC: layout versus schematics, DRC: design rule checking, and GDSII: Graphical Database System II.)

conventional techniques [2]. The field of HTs detection encompasses two main categories: 1) *presilicon* detection and 2) *post-silicon* detection [3]. The detection strategies for *presilicon* Trojan detection can be categorized into static and dynamic approaches [4], [5] incorporating various ML and deep learning techniques [6]. In the *post-silicon* detection step, the methods can be further categorized into destructive and nondestructive detection approaches. The former involve techniques like reverse engineering to reconstruct the design and compare it with a reference model (also known as golden chip) [3]. The latter uses logic testing and side-channel analysis approaches that compare the design outputs with the correct reference results [7]. Dong et al. [4] proposed the *MHTtext* model, which serves as a method for detecting HTs using *TextCNN* as a deep learning approach. The model automatically extracts HT components from netlist files without comparing them with the golden chip. The state-of-the-art (SOTA) approaches employ ML techniques to detect HTs. These methods concentrate on the timing model at design time incorporating voltage noise and the trained NN as a watchdog tracking process [2].

The literature reveals various hardware-related techniques for Trojan detection, such as ❶ full Trojan activation; ❷ power-based Trojan detection; and ❸ delay-based Trojan detection [8]. Our study aims to shift the focus from examining the physical state of case-by-case HT analysis to deriving conclusions from the software aspect. The ultimate goal is to attain a generalizable code flow, enabling code analysis through ML models without resorting to heuristic searches.

III. Opcode Language Processing

A. From Hardware Descriptions to Machine Code

Assembly language serves as a universal representation for a human-readable machine language. Its structure is more straightforward compared to high-level programming languages. Despite having limited operational codes and reserved bits, it clearly defines the source and destination registers for each equal-structured **opcode**, yielding a simple and predetermined flow. As illustrated in Fig. 1, the initial design implementation in the IC chain is already in C/C++/SystemC, establishing a relative relationship between **HDL** and these languages. Given the vulnerabilities associated with third-party intellectual properties (IPs) in each programming approach, a unique solution can be applied to address both. This solution lies in the **assembly** code-based analysis, leveraging its well-structured format.

Fig. 2 illustrates the proposed approach, which addresses the vulnerability of **HDL** design to attacks, focusing on this specific aspect of the IC production chain. The central contribution of this letter lies in the proposing and integration of **Opcode** language processing (OLP), necessitating an initial intermediary step of language conversion, followed by the application of classifiers through ML methodologies.

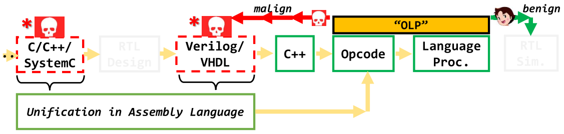


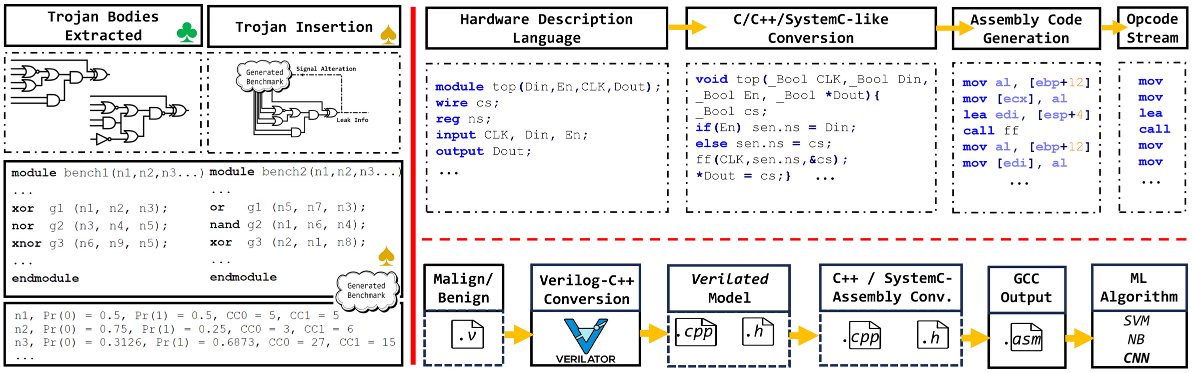
Fig. 2. Proposed approach is positioned within the IC supply chain, with low-level language design, such as **C++**, serving as a preliminary step before RTL design. Reverting the code flow back to **C++** makes it possible to inspect the system's behavior more effectively. To achieve a unified and generalized code flow for any module design, the lowest-level platform-dependent **assembly** code comes into play, bridging the gap between the intermediate **C++** form and the final RTL design. A unique proposition in this letter involves **opcode** processing, which facilitates the inspection and classification of behavior as either "malign" or "benign."

Generating **opcode** requires translating a **C/C++** code from the Verilog module. However, manually creating a **C/C++** equivalent code from Verilog presents significant challenges. As a result, a practical and automated Verilog to **C/C++** translation/conversion tool becomes imperative. A popular tool for automatic conversion is `Verilator` [10]. It converts synthesizable Verilog into **C++**. Notably, `Verilator` possesses properties that render it particularly suitable for our application, including its ability to handle large amounts of gates and create a **C++** wrapper file with all the necessary inclusion of header files. Our approach comprises three main steps: ① inputting Trojan-infected and Trojan-free Verilog modules into `Verilator`; ② `Verilator` execution: generating a header and a **C++** file; and ③ compiling the **C++** file, along with the header, using GNU compiler collection (GCC) to produce the **assembly** equivalent. Fig. 3 illustrates the overall steps.

B. Exploration of Netlist Vulnerabilities

We begin by investigating netlist analysis for detecting HTs within an ML framework. This involves generating multiple benchmarks to identify vulnerabilities and inserting various HTs using a specialized Trojan insertion tool [11]. A synthetic benchmark generator is then employed, allowing the creation of circuit benchmarks with user-defined parameters, such as depth levels, logical units, inputs/outputs, and wire counts [12].

Following the extraction of HTs from benchmarks provided by [11], a stochastic approach is employed to insert these HTs into the benchmarks. This insertion process is executed by targeting nodes identified as "rare nodes" [13], characterized by a signal probability below a defined threshold. These rare nodes are identified through a vulnerability score (VS) computation, where $VS = Pr(0)(1 - CC_0/CC_0 + CC_1)$, with $Pr(0)$ representing the probability of a node transitioning to "0," and CC_0 and CC_1 indicating the efforts to guide the node to "0" and "1," respectively. The higher $Pr(0)$ signifies a greater chance of transitioning to an undesired state,



```
module bench1(n1,n2,n3...)
...
xor  g1 (n1, n2, n3);
nor  g2 (n3, n4, n5);
xnor g3 (n6, n9, n5);
...
endmodule
```

```
module bench2(n1,n2,n3...)
...
or   g1 (n5, n7, n3);
nand g2 (n1, n6, n4);
xor  g3 (n2, n1, n8);
...
endmodule
```

```
n1, Pr(0) = 0.5, Pr(1) = 0.5, CC0 = 5, CC1 = 5
n2, Pr(0) = 0.75, Pr(1) = 0.25, CC0 = 3, CC1 = 6
n3, Pr(0) = 0.3126, Pr(1) = 0.6873, CC0 = 27, CC1 = 15
...
```

```
module top(Din,En,CLK,Dout);
wire cs;
reg ns;
input CLK, Din, En;
output Dout;
...
```

```
void top(_Bool CLK,_Bool Din,
_Bool En, _Bool *Dout){
_Bool cs;
if(En) sen.ns = Din;
else sen.ns = cs;
ff(CLK,sen.ns,&cs);
*Dout = cs;}   ...
```

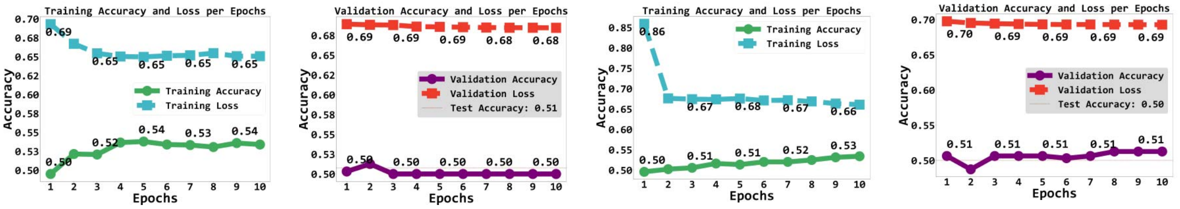
```
mov al, [ebp+12]
mov [ecx], al
lea edi, [esp+4]
call ff
mov al, [ebp+12]
mov [edi], al
   ...
```

```
mov
mov
lea
call
mov
mov
...
```

(a)

(b)

Fig. 3. Procedures for creating opcode and HDL datasets. (a) Methodology for generating Trojan-injected and Trojan-free instances. (b) Preliminary steps to unify the **HDL** code written in any description language. The first step in this process is the conversion to **C/C++**. The Verilator tool is employed, allowing for seamless **C++** conversion. The obtained **C++** code undergoes conversion to **assembly** language, following thorough checks on the compiler side. The architecture chosen for this purpose is Intel x86. Ultimately, the result is an **opcode** stream, which is the final outcome of this unification process. (SVM: support vector machine and NB: naive Bayes.)



(a) (b) (c) (d)

Fig. 4. Training, test, and validation results based on the **HDL** dataset. (a) and (b) Exhibit model trained with an embedding size of 48. (c) and (d) Exhibit an embedding size of 96. The model encounters difficulty when dealing with complex patterns within **HDL** code, particularly when attempting to identify indications of Trojan injections. Its tendency to consider typical code structures more common leads to classifying all test cases as Trojan-free. Essentially, the model learns to generalize more confidently toward Trojan-free (negative) predictions.

countered by lower controllability preventing "0" transitions. Subsequently, stochastic HT templates are linked to these identified nodes [14].

Upon completion of the procedural stages outlined in Fig. 3(a) for the HT injection process, a dataset consisting of 2000 samples, all containing **HDL** code, is obtained. This dataset includes 1000 Trojan-infected **HDL** files and an equal number of Trojan-free **HDL** files. The rationale behind initially evaluating **HDL** code before **opcode** analysis is rooted in the notion that **HDL** can offer insights into Trojan contamination. In contrast to the standardized instructions present in **opcodes**, the utilization of **HDL** code introduces several complexities. These complexities encompass extended code lengths, intricate code lines dedicated to expressing singular operations, and extensive keywords in **HDL**. These attributes augment the intricacy of representing individual inputs during embedding **HDL** model in the training phase. Furthermore, diverse keywords within the code expand the vocabulary size, a significant parameter influencing the model's training process. As indicated by the results illustrated in Fig. 4, the convolutional neural network (CNN) model exhibits limitations in solely utilizing **HDL** code processing for detecting the Trojan-infected status of a module. Consequently, an imperative arises to explore alternative approaches more amenable to the application of NLP algorithms, with a focus on **opcode** analysis.

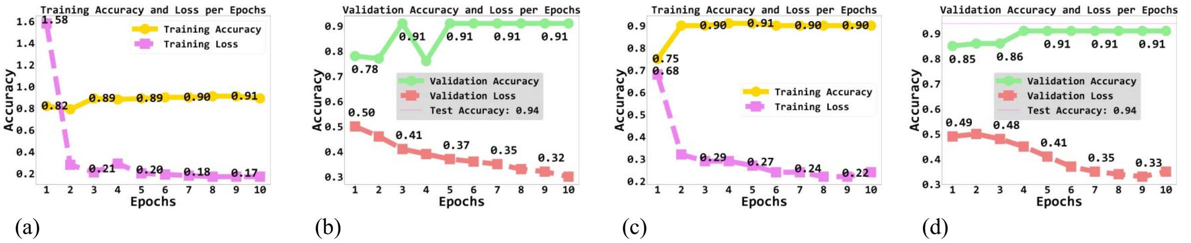
C. Learning the Mnemonics

We treat the Trojan detection problem as *sentence classification* in our proposed approach. Instead of using regular words from human language, we employ **opcodes** to train our data. Thus, our sentences consist of sequential **opcodes** converted from Verilog modules, while our words represent the *instruction set* for computer architecture. The fundamental distinction between classifying a sentence in human language and classifying **opcodes** lies in the structure of the inputs. While sentences typically consist of 15–20 words, **opcodes** can result in extremely long sentences containing up to 4000–5000 words. Consequently, our problem poses a unique challenge, as it involves classifying long sentences with a fixed number of words. To address this issue, we employed 1) ML and 2) deep learning algorithms to determine the most accurate model based on respective performances. The proposed CNN model can learn intricate patterns within Trojan-infected and Trojan-free inputs. In 1), raw text data are encoded into a transformed dataset using the term frequency-inverse document frequency transformer (TF-IDF). This encoding method down-weights the significance of frequently occurring words while emphasizing the value of rare words within a document. Since **opcodes** often consist of repetitive instructions, vectorizing the raw text data based on their importance becomes essential. In 2), the proposed OLP leverages the capabilities of CNNs, requiring vectorized inputs. Traditional methods, such as Bag-of-Words and TF-IDF, adopt frequency-based approaches to vectorize data into fixed-length input documents. Our approach adopts a different strategy by training our model with its own embedding layer. This approach offers the advantage of localizing our embeddings for **opcode** analysis. We can modify the embedding dimension according to different test cases by training our

TABLE I
PERFORMANCE MEASUREMENT METRICS

Trained Model	TP	FP	TN	FN	TPR	FPR	Accuracy	Precision	Recall	F1-Score	Matthews Correlation Coeff.
SVM	19	0	29	26	0.422	0	0.648	1.000	0.422	0.593	0.471
NBC	8	0	33	33	0.195	0	0.554	1.000	0.195	0.326	0.312
EMBEDDING-I + CNN	34	4	25	0	1.000	0.137	0.936	0.894	1.000	0.944	0.878
EMBEDDING-II + CNN	34	4	25	0	1.000	0.137	0.936	0.894	1.000	0.944	0.878

CNN details: The model architecture consists of an *Embedding Layer* with an input size of 96, followed by a 1-D Convolutional layer with 32 filters and a kernel size of 8, then a MaxPooling 1-D layer with a pool size of 2. The model also includes BatchNormalization, a Flatten layer, a Dense layer with 10 units using ReLU activation, a Dropout layer with a dropout rate of 0.15, and finally, a Dense output layer with 1 unit and sigmoid activation for binary classification. The model is trained using the Adam optimizer with an initial learning rate of 0.001. Early stopping and learning rate reduction are applied with a factor of 0.2, reducing the learning rate to a minimum of 10^{-6} to prevent overfitting. ***Embedding Size***: *EMBEDDING*-I: 48, *EMBEDDING*-II: 96. (TP: True Positive, FP: False Positive, TN: True Negative, FN: False Negative, TPR, FPR: True Positive and False Positive Rates.) The overall time complexity of our entire pipeline is dominantly influenced by the input length within the CNN architecture and is approximately $O(N \times F \times F \times K)$, where '$N$' is input sequence length ($N = 5389$), 'F' is the filter size ($F = 3$), and 'K' is the number of filters ($K = 32$). Our open-source development environment can be accessed from [15].



(a) (b) (c) (d)

Fig. 5. Training, test, and validation results based on the **opcode** dataset which contains 490 samples. Each class has an equal number of samples. (a) and (b) Exhibit model trained with an embedding size of 48. (c) and (d) Exhibit an embedding size of 96.

embeddings. Consequently, our architecture becomes adept at discerning patterns within embedded datasets, leveraging shift-invariance, automatic feature extraction, and local feature learning.

IV. EXPERIMENTS AND RESULTS

We utilized datasets sourced from Trust-Hub [16], [17] for our experiments. These datasets comprise Verilog gate-level netlist files, each appropriately labeled to distinguish between Trojan-free and Trojan-infected modules. Our focus lies in model exploration and hyperparameter tuning strategies, which are crucial for optimizing the proposed CNN architecture. By carefully evaluating critical parameters during training, we aim to obtain reliable results.

A. Model Exploration

We first employ support vector machine (SVM) and naive Bayes classifier (NBC) due to the limited size of the available Trojan benchmarks compared to SOTA sentence classification datasets [18]. While ML algorithms find sentence classification in human language relatively straightforward, the **opcode** analysis presents a challenging task due to the complex nature of the input dataset. To overcome this challenge, we consider CNN-based architectures. These architectures are conventionally used in NLP for text classification and next-sentence prediction tasks on diverse datasets [19].

Conventional ML Models: SVM and NBC are trained and tested using the generated **opcode** dataset. Both of these algorithms face challenges stemming from the *Curse of Dimensionality* problem, which arises due to the extensive dimensions of the input data. Traditional vectorization approaches in ML algorithms lack the capability to assign importance weights to each input based on location embedding. Consequently, simple frequency-based encoding techniques fail to yield meaningful values compared to location embeddings. As a result of these limitations, both SVM and NBC exhibit poor performance on the test data; their predictions appear to be nearly random, as evident from the outcomes presented in the first two rows of Table I. This underscores the need for more sophisticated approaches to handle the complexities of the **opcode** dataset.

B. Proposed CNN Architecture

The proposed sequential model encompasses an embedding layer, 1-D convolutional, and dense layers. Regularization techniques, such as batch normalization, dropout, and early stopping, have been incorporated to mitigate overfitting concerns. (Please refer to Table I footnote for further architectural details.)

Data Augmentation: Due to the limited size of the **opcode** dataset in benchmarks, we resorted to employing data augmentation techniques to enrich the available data. Specifically, we utilized random swapping and random deletion augmentation techniques [20]. Leveraging the shift-invariance feature of CNNs, we could modify the order of instructions, striking a valuable balance between accuracy and data augmentation.

Model Exploitation: The process of selecting hyperparameters was carried out through a series of tests conducted on the dataset. The vectorized version of each input token can have varying effects depending on the chosen embedding size and other hyperparameters. In our experimentation, we evaluated two embedding sizes: 48 (*EMBEDDING*-I) and 96 (*EMBEDDING*-II); each is tested on a dataset with a vocabulary size of 513. The results of the training model with two embedding sizes are presented in Fig. 5. The final experiment with an embedding size of 96 achieves an accuracy of 94%.

The ultimate results were obtained with an embedding size of 96 using the **opcode** dataset. All evaluation metrics on the **opcode** data are presented in the last two rows of Table I. The results, including F1-score, recall, and Matthews correlation coefficient, affirm that an embedding size around 48–96 can be chosen, thereby facilitating yet efficient training process. In conclusion, an embedding size of at least 48 guarantees classification accuracy above 93%. This comprehensive exploration and selection of the appropriate hyperparameters significantly contributes to the overall success of the model.

TABLE II
ACCURACY COMPARISON OF THE SOTA METHODS

Reference	**Architecture**	**Accuracy**
[21]	Random Forest	76.8%
[22]	Neural Network	92.2%
[23]	Clustering	87.0%
[24]	XGBoost	90.0%
This work	**Embedding + CNN**	**93.6%**

Table II compares our method's accuracy results to other ML-related HT detection SOTA studies.

V. ASSUMPTIONS AND LIMITATIONS

The assumptions and limitations of this letter are as follows.

Assumptions:

1) Our study employs a supervised learning approach within an ML framework.
2) We treat **opcodes** as analogous to NLP challenges.
3) Our dataset analysis relies on rare nodes and VS.
4) We initially assumed that the CNN architecture constitutes the most suitable solution for the problem.

This assumption was made based on rigorous scientific evaluation involving a comparative analysis with conventional architectures and subsequent experimental validation, affirming the redundancy of more intricate NN designs.

Limitations:

1) The preprocessing step for converting **HDL** to **assembly** language has minimal computational impact, albeit constituting a limitation.
2) Converting **opcodes** to **assembly** language may result in longer code lines, notwithstanding its benefits in enhancing structural learning in artificial intelligence (AI).
3) Our primary focus centers on vulnerabilities within the hardware-related software aspect of the supply chain, while further stages, including gate-level netlists, layout, fabricated chips, and third-party soft IPs, hold potential for future ML-driven vulnerability analysis.

Future efforts can unveil new opportunities to analyze vulnerabilities and malign-benign scoring for black box IP cores using unsupervised learning methods. Furthermore, an expanded dataset encompassing **HDL** to **opcode** conversion could be generated through AI-driven generative data techniques, such as those involving pretrained transformers. This generated data could serve as an opportunity to assess the performance of our proposed approach on a broader scale.

VI. CONCLUSION

This letter evaluated the HT detection problem considering **assembly** and **HDL** code, employing ML and deep learning methods. The extensive **HDL** dataset, which is fourfold larger than the **opcode** dataset, serves to affirm that the analysis of **opcode** is pivotal in comprehending the underlying patterns of Trojans. Despite a smaller **opcode** dataset compared to the **HDL** dataset, the model extracts more patterns, yielding better accuracy. The new approach proved to be instrumental in the CNN integration for **opcode** processing, as it enabled us to distinguish between Trojan-infected and Trojan-free hardware description modules. The insights gained from this analysis offer valuable implications for future studies aimed at attack classification.

REFERENCES

[1] H. Salmani et al., "Trust-hub trojan benchmark for hardware trojan detection model creation using machine learning," Dataset, IEEE DataPort, New York, NY, USA, 2022.

[2] K. I. Gubbi et al., "Hardware trojan detection using machine learning: A tutorial," *ACM TECS*, vol. 22, no. 3, pp. 1–26, Apr. 2023.

[3] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, and M. Tehranipoor, "Benchmarking of hardware trojans and maliciously affected circuits," *J. Hardw. Syst. Secur.*, vol. 1, no. 1, pp. 85–102, Mar. 2017.

[4] C. Dong et al, "A cost-driven method for deep-learning-based hardware trojan detection," *Sensors*, vol. 23, no. 12, p. 5503, 2023.

[5] L. Shen, D. Mu, G. Cao, M. Qin, J. Zhu, and W. Hu, "Accelerating hardware security verification and vulnerability detection through state space reduction," *Coumpt. Security*, vol. 103, Apr. 2021, Art. no. 102167.

[6] T. Kurihara, K. Hasegawa, and N. Togawa, "Evaluation on hardware-trojan detection at gate-level IP cores utilizing machine learning methods," in *Proc. IEEE IOLTS*, 2020, pp. 1–4.

[7] S. Yang, T. Hoque, P. Chakraborty, and S. Bhunia, "Golden-free hardware trojan detection using self-referencing," *IEEE Trans. Very Large Scale Integr. Sys.*, vol. 30, no. 3, pp. 325–338, Mar. 2022.

[8] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Des. & Test Comput.*, vol. 27, no. 1, pp. 10–25, Jan./Feb. 2010.

[9] R. Yasaei, S. Faezi, and M. A. Al Faruque, "Golden reference-free hardware trojan localization using graph convolutional network," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 30, no. 10, pp. 1401–1411, Oct. 2022.

[10] W. Snyder, "Verilator and SystemPerl," in *Proc. North Amer. SystemC Users' Group Meeting, DAC*, 2004, p. 1–14.

[11] J. Cruz, Y. Huang, P. Mishra, and S. Bhunia, "An automated configurable trojan insertion framework for dynamic trust benchmarks," in *Proc. IEEE Des. Autom. & Test Eur. Conf. & Exhibition (DATE)*, 2018, pp. 1598–1603.

[12] S. Amir and D. Forte, "Adaptable and divergent synthetic benchmark generation for hardware security," in *Proc. ICCAD*, 2020, pp. 1–9.

[13] S. Narasimhan, R. S. Chakraborty, and S. Chakraborty, "Hardware IP protection during evaluation using embedded sequential trojan," *IEEE Des. Test Coumput.*, vol. 29, no. 3, pp. 70–79, Jun. 2012.

[14] J. Cruz, P. Slpsk, P. Gaikwad, and S. Bhunia "TVF: A metric for quantifying vulnerability against hardware trojan attacks," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 31, no. 7, pp. 969–979, Jul. 2023.

[15] A. G. Ayar, A. Sahruri, S. Aygun, M. S. Moghadam, M. H. Najafi, and M. Margala, "Detecting vulnerability in hardware description languages," early access, Nov. 21, 2023, doi: 10.1109/LES.2023.3334728. https://github.com/goktug16/Opcode-Language-Processing

[16] Trust-HUB. Trust-hub, 2023. [Online]. Available: https://www.trust-hub.org//benchmarks/chip-level-trojan Accessed: Jul. 1, 2023.

[17] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in *Proc. ICCD*, 2013, pp. 471–474.

[18] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proc. ACL*, 2011, pp. 142–150.

[19] Y. Luan and S. Lin. "Research on text classification based on CNN and LSTM," in *Proc. ICAICA*, 2019, pp. 352–355.

[20] B. Li et al. "Data augmentation approaches in natural language processing: A survey," *AI Open*, vol. 3, pp. 71–90, Aug. 2022.

[21] K. Hasegawa, M. Yanagisawa, and N. Togawa, "Trojan-feature extraction at gate-level netlists and its application to hardware-trojan detection using random forest classifier," in *Proc. IEEE ISCAS*, 2017, pp. 1–4.

[22] K. Hasegawa, M. Yanagisawa, and N. Togawa, "Hardware trojans classification for gate-level netlists using multi-layer neural networks," in *Proc. IEEE IOLTS*, 2017, pp. 227–232.

[23] P. Zhao and Q. Liu, "Density-based clustering method for hardware trojan detection based on gate-level structural features," in *Proc. AsianHOST*, 2019, pp. 1–4.

[24] Y. Wang, T. Han, X. Han, and P. Liu, "Ensemble-learning-based hardware trojans detection method by detecting the trigger nets," in *Proc. IEEE ISCAS*, 2019, pp. 1–5.