LLMPirate: LLMs for Black-box Hardware IP Piracy

Vasudev Gohil, Matthew DeLorenzo, Veera Vishwa Achuta Sai Venkat Nallam, Joey See, Jeyavijayan Rajendran
Texas A&M University
vgohil.research@gmail.com, {matthewdelorenzo, nallamsaiv, joeysee, jv.rajendran}@tamu.edu

***Abstract*—The rapid advancement of large language models (LLMs) has enabled the ability to effectively analyze and generate code nearly instantaneously, resulting in their widespread adoption in software development. Following this advancement, researchers and companies have also begun integrating LLMs across the hardware design and verification process. However, these highly potent LLMs can also induce new attack scenarios upon security vulnerabilities across the hardware development process. One such attack vector that has not been explored so far is intellectual property (IP) piracy. Given that this attack can manifest as rewriting hardware designs to evade piracy detection, it is essential to thoroughly evaluate LLM capabilities in performing this task and assess the mitigation abilities of current IP piracy detection tools.**

Therefore, in this work, we propose *LLMPirate*, the first LLM-based technique able to generate pirated variations of circuit designs that successfully evade detection across multiple state-of-the-art piracy detection tools. We devise three solutions to overcome challenges related to integration of LLMs for hardware circuit designs, scalability to large circuits, and effectiveness, resulting in an end-to-end automated, efficient, and practical formulation. We perform an extensive experimental evaluation of *LLMPirate* using eight LLMs of varying sizes and capabilities and assess their performance in pirating various circuit designs against four state-of-the-art, widely-used piracy detection tools. Our experiments demonstrate that *LLMPirate* is able to consistently evade detection on 100% of tested circuits across every detection tool. Additionally, we showcase the ramifications of *LLMPirate* using case studies on `IBEX` and `MOR1KX` processors and a `GPS` module, that we successfully pirate. We envision that our work motivates and fosters the development of better IP piracy detection tools.

I. Introduction

Recent advancements within artificial intelligence and computing performance have greatly accelerated the development of large language models (LLMs), with state-of-the-art models (including OpenAI's ChatGPT [60] and Google's Bard [68]) achieving groundbreaking performance in natural language processing and gaining mass popularity [38]. With the ability to effectively interpret text prompts and generate human-like responses [56], LLMs have proven effective across a variety of tasks, such as language translation [44], text summarization [67], and generating code [26]. This widespread applicability has resulted in the rapid adoption of LLMs across various industries, serving as chat-bots for customer service [67], documentation aids in healthcare [48], and coding assistants for programmers [26]. These applications have prompted companies and researchers to further explore the most effective ways in which LLMs can be tailored and utilized to automate specified tasks and processes, including the software and hardware design workflow.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.242059
www.ndss-symposium.org

A. LLMs for Code Generation

Given the success of LLMs in natural language processing, many models are also extensively trained on large datasets of open-source code with the specific purpose of generating functionally correct programs based upon a prompt description [49]. These programming-oriented LLMs are utilized in a variety of applications within the software and hardware development processes. Through Microsoft's Github CoPilot, the advantages of LLMs are applied directly to the software development environment, providing context-aware code suggestions and refactoring recommendations [28]. In fact, Microsoft reported that the first versions of CoPilot tools substantially increase productivity on common enterprise information worker tasks [14]. Similarly, OpenAI's widely utilized GPT-4 model also has strong performance in software engineering tasks, including the ability to generate programs from pseudocode and explain its results in natural language [11].

Following advancements in the software domain, semiconductor companies have also begun utilizing generative artificial intelligence (AI), specifically LLMs, within various stages of hardware integrated circuit design process, including the generation of register-transfer level (RTL) code. Semiconductor giant NVIDIA's ChipNeMo explores fine-tuning smaller LLMs for industrial chip-design, in which their 70-billion parameter model was able to outperform OpenAI's GPT-4 in electronic design automation (EDA) tools' script generation [46]. ChipGPT from Cadence demonstrated the first proof-of-concept LLM technology in chip design, able to load architecture and design specifications to accelerate test-bench creation and RTL code generation [10]. Cadence has also developed the Cadence.AI generative AI platform for applications in digital circuit design, analog circuit design, debug and verification, and printed circuit board design [13]. Likewise, Synopsys, another EDA enterprise, has developed Synopsys.ai Copilot that harnesses generative AI with LLMs throughout their EDA suite, aiding in tedious workflow tasks including test pattern generation, verification coverage, and design space exploration [69]. RapidGPT from Rapid Silicon provides similar auto-complete capabilities tailored to field-programmable gate array design [20].

LLMs can also be offensively leveraged by threat actors to execute attacks that exploit various software and hardware security vulnerabilities. For instance, RatGPT utilizes GPT-4 as a proxy method to distribute malicious software through

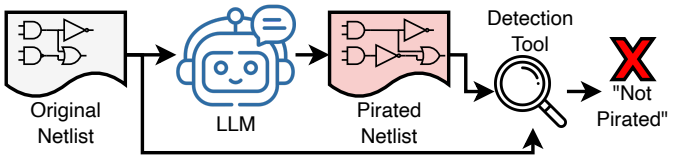

Fig. 1: High-level overview of our proposed technique.

the use of openly accessible LLM plugins, enabling access to the victim's machine [6]. Additionally, GPThreats-3 [8] explores how LLMs (GPT-3) can be utilized to generate malware itself, demonstrating success through a building-block prompting strategy. To attack hardware systems, LLMs can enhance side-channel attacks, in which unintentional data is extracted based upon the physical implementation of the circuit design (e.g., the secret key from a cryptographic algorithm). AgentSCA demonstrates that through fine-tuning LLMs with human feedback upon side-channel datasets, the model can effectively interpret side-channel statistics and provide correct decisions based upon the test system [80]. However, **a crucial attack vector that can be orchestrated using LLMs that has not been explored so far is the piracy of intellectual property (IP) within the hardware domain.**

B. Impact of IP Piracy

The theft of hardware design IP, or IP piracy, is a significant concern within the system-on-chip design flow [16]. This can be attributed to the globalization of the integrated circuits supply chain, where semiconductor companies outsource their IP design to (potentially untrusted) fabrication entities to reduce the cost and time of chip production [37]. This has caused an increased risk of theft of IP assets shared by vendors (including RTL designs), causing significant security and economic consequences. A recent instance is observed within the dynamic random-access memory (DRAM) market. Micron, who held 20-25% of the global DRAM market share, reported estimated losses of $8.75 billion to IP piracy alone in 2018, demonstrating significant economic impacts [58]. Additionally, the semiconductor industry has been largely impacted, with the U.S. Trade Representative reporting losses between $225 to $600 billion as a result of Chinese theft of American IP [59], [19], [53].

To address this threat across the hardware design process, a number of hardware IP protection techniques [43] as well as piracy detection tools such as GNN4IP are utilized [81]. However, piracy detection tools such as GNN4IP have not been thoroughly tested. In this work, we show how LLMs can be used to pirate IPs and successfully evade tools such as GNN4IP.

C. Our Goals and Contributions

We propose an end-to-end automated LLM-based IP piracy scheme, *LLMPirate*, using which we can rewrite circuits, i.e., Verilog netlists, such that they evade detection by various IP piracy detection tools. Figure 1 illustrates the high-level idea. This requires designing an appropriate task for a given LLM, such that prompting it with a target netlist results in a design that is functionally equivalent to the original circuit, but is also different enough to not be flagged by piracy detection tools.

However, several challenges exist in designing such an end-to-end automated IP piracy framework. First, LLMs are trained on extremely limited hardware designs, i.e., circuits described in hardware description languages such as Verilog or VHDL, as opposed to software codes such as C, C++, Java, or Python [77]. In fact, our experiments indicate that even advanced LLMs such as OpenAI's GPT-3.5 are unable to understand and rewrite simple Verilog designs. Second, these LLMs face scalability issues: they are unable to work with larger Verilog designs. Another scalability issue also stems from the limited context windows (i.e., the amount of information that an LLM can take into account to generate responses without losing context) of LLMs. This is especially important since typical Verilog designs have hundreds of thousands of characters, which is far more than the context windows of all LLMs available today. Third, LLMs' responses are not deterministic, so often, simply prompting an LLM to rewrite/pirate circuit designs, i.e., netlists, results in error-prone/incorrect responses. This needs to be alleviated in order to realize a practical IP piracy technique.

We overcome the challenge of limited Verilog data used to train LLMs by devising Solution A: translating the syntax of Verilog netlists to a more generic format of Boolean functions before prompting the LLMs. Doing so makes it easier for LLMs to understand the prompt details and respond accordingly. To overcome the challenges related to scalability, we devise Solution B, which characterizes large netlists to extract all unique gate types and then using a divide-and-conquer approach to not exceed the context window sizes of LLMs. Finally, we overcome the challenge related to error-prone responses, we devise Solution C, which combines the interactive capabilities of LLMs with fine-grained feedback related to syntactical and functional correctness of the circuit generated by the LLM. Sec. IV-A contains more details about these challenges and our solutions. By solving these challenges, we develop an end-to-end automated technique, *LLMPirate*, for successfully pirating hardware IPs. Our primary contributions are:

- We present a first-of-its-kind LLM-based technique, *LLMPirate*, that successfully pirates hardware IPs and evades detection by state-of-the-art detection tools. This is also the first work to provide a detailed comparative study of the efficacy of four different piracy detection tools that use different algorithms.
- We overcome unique challenges related to integration of LLMs for Verilog netlists, scalability, and effectiveness by designing custom solutions based on domain knowledge.
- We provide a comparative evaluation between various popular LLM models across various IP piracy detection tools, with results indicating that overall, GPT-4 is the most effective in successfully pirating Verilog netlists. Other large closed-source LLMs such as CoPilot and GPT-3.5 also perform very well.
- Our results also demonstrate that our feedback-guided interactive formulation greatly improves the performance of all LLMs, and most notably of smaller, open-source LLMs such as the recently released Llama3.
- We demonstrate the practical ramifications of our LLM-based IP piracy technique through case studies on modern, real-world designs: `IBEX` and `MOR1KX` processors and

TABLE I: Overview of *LLMPirate* against piracy detection tools.

Detection Tool	GNN4IP [81]	MOSS [2]	Jplag [41]	SIM [35]
Algorithm Used	Graph Neural Network	Winnowing	Greedy String Tiling	Tokenization and Longest Common Subsequence
Key Features	High Accuracy, Designed for Verilog	Widely-used, Supports Verilog	Widely-used, Robust Against Obfuscation	Widely-used, Efficient
***LLMPirate* (This Work)'s Evasion Rate**	100%	100%	100%	81.25%[1]

a `GPS` module. We successfully pirate them and evade multiple state-of-the-art detection tools.

D. Why LLMs?

A natural question here could be about the need of LLMs for hardware IP piracy. LLMs have shown tremendous improvement over the past few years, and today's LLMs are proficient at a variety of tasks including programming [40]. However, as we demonstrate in Sec. IV-A, LLMs still struggle with understanding simple Verilog netlists. Given this limitation of LLMs, a potential approach for a malicious human developer can be to manually rewrite netlists. However, such an approach would not be scalable. Additionally, the limitations of this manual approach are exacerbated by the fact that different piracy detection techniques use different types of algorithms (e.g., graph structural similarity, "fingerprints" of hashed Verilog netlist structures, and text-based comparisons). Another limitation of the manual approach is the requirement of additional effort for every new piracy detection technique. In contrast, *LLMPirate* offers an end-to-end automated flow to easily and quickly pirate hardware circuit netlists, enabling proper evaluation of existing and new piracy detection techniques.

II. Background

A. Large Language Models

In recent years, large language models (LLMs) have emerged as powerful tools in natural language processing and related fields. These models, often based on deep learning architectures, exhibit remarkable capabilities in tasks such as text generation, translation [44], and sentiment analysis [45]. LLMs learn to represent language patterns and context by training on massive amounts of text data. Notable examples include GPT-3.5 (used in ChatGPT) [62], GPT-4 [61], Gemini [31], and Llama [50], among others. The remarkable success of LLMs can be attributed to their architectural innovation, in which transformer architectures are leveraged to enable parallel processing of sequential data, thereby efficiently capturing complex linguistic patterns and dependencies within text [78].

B. Code Generation with Large Language Models

Among other avenues, LLMs have also brought a paradigm shift in code generation [40]. LLMs have demonstrated remarkable proficiency in generating code across different programming languages [15]. By leveraging the vast knowledge encoded in their pre-trained parameters, these models can understand natural language prompts describing desired functionalities or requirements and produce corresponding code snippets with high fidelity. This capability has shown promise for accelerating software development processes, facilitating rapid prototyping, and reducing the burden on programmers by automating routine coding tasks [75], [28]. Additionally, fine-tuning these models on domain-specific codebases further enhances their proficiency in generating contextually relevant and syntactically correct code [71]. For instance, researchers and corporations have devised custom LLMs for generating codes in hardware description languages, such as Verilog and VHDL, which are used to create digital integrated circuits [77], [27], [22], [21].

C. IP Piracy Detection

Although there are many noteworthy works for measuring similarity, we choose four techniques for our evaluation, as explained next. Our selection of target similarity detection techniques ranges from the earliest tools with high popularity, MOSS [2], to the most recent, GNN4IP [81], which uses machine learning. We also select other tools, SIM [35] and Jplag [41], based on their high accuracy, impact, and popularity (see Table I for an overview). MOSS is arguably the most widely-used similarity measurement tool for codes. It has been used globally for decades [74], [76], [9], [18], has over 300K active accounts [23], and is also used in (or for the basis of) commercial tools for similarity detection, such as Gradescope [57] and Codequiry [17]. GNN4IP is the most powerful similarity measurement tool for Verilog, as it was developed with the specific objective of detecting IP piracy in Verilog code. Jplag, like MOSS, is also a widely-used similarity measurement tool that is used in universities [18]. Overall, our selection represents a set of similarity detection tools that use different frameworks, demonstrate excellent performance, and have been used extensively.

GNN4IP is a piracy detection tool developed with the specific purpose of detecting hardware IP piracy [81]. It converts Verilog descriptions of hardware IP into graph representations and performs graph convolutions on those graphs in order to extract their node embeddings. By finding the cosine similarity between the embeddings of different IPs, it then becomes possible to determine if IP piracy has occurred.

MOSS is a piracy detection tool developed by Stanford University [2], primarily utilized for detecting plagiarism in code across students in college-level computer science courses. MOSS uses the winnowing algorithm [72], which first breaks down code into tokens and hashes them using a hash function,

[1] *LLMPirate* does not have 100% evasion rate against SIM because SIM has high false-positive rate as it does not support Verilog natively.

then applies a sliding window over the hashes, and lastly selects the minimum hash value from each window. These values become the "fingerprint" of the code, which are then utilized to evaluate the similarity percentage between two target codes.

JPlag is a Java-based software similarity detection tool [41]. It utilizes the Greedy-String-Tiling algorithm on tokenized entries to systemically break entries up into "tiles" of matching token strings, prioritizing the longest strings first. The number of tiles found are then compared to the overall length of entries to assess similarities. Originally developed in 1996, it has JPlag has received consistent updates and improvements by universities and other community members who continue to use it today [18].

SIM is another piracy detection tool, utilized to detect plagiarism in writing assignments and software projects across college students [35]. SIM firstly breaks the given code/text into 16-bit tokens using a hash function and normalizes them in order to minimize superficial differences (such as comments, white spaces, etc). Then the algorithm scans for overlapping blocks of tokens that appear in both files. Finally, the similarity percentage is found by dividing the number of matching tokens by the total number of tokens [36].

III. Threat Model

We consider a standard black-box attacker model applicable to piracy detection or similarity measurement techniques such as GNN4IP, MOSS, Jplag, and SIM. In this context, we establish the following assumptions about the attacker:

Attacker's Knowledge. We assume a black-box setting, where the attacker lacks access to the detection tool's internal parameters (e.g., ML model's parameters or training labels, or internal parameters used in algorithms of the detection tool). The attacker can only make black-box queries to obtain output similarity scores or predicted labels (in case of ML-based techniques).

Attacker's Capacity. The piracy attack occurs after the detection tool is finalized. Especially in case of machine learning (ML)-based techniques, the attack occurs after the model has undergone training. The detection tool remains fixed, and the adversary lacks the ability to alter its parameters or structure. For instance, the attacker cannot introduce model poisoning (for ML-based techniques) or inject backdoors.

Attacker's Abilities. The attacker can rewrite the netlist arbitrarily, but not alter the netlist's functionality. Additionally, the attacker must adhere to circuit design rules.

Attacker's Goal. The attacker's objective is to generate netlists that lead to misclassification by the target detection tool(s). For instance, when the target detection tool is GNN4IP, the attacker aims to create a pirated version of an original netlists such that GNN4IP incorrectly classifies the pirated netlist as "not pirated". Or, when the target detection tool is MOSS, the attacker aims to pirate an original netlist such that MOSS returns a low enough similarity score (determined by a threshold, explained in Sec. V).

```
// Can you refactor it so that functionality
 remains the same, but the gates used and
their interconnections are different from
the original structure?
module top (input a, input b, output c);
or U1 (c, a, b);
endmodule
```

Listing 1: Prompt for rewriting Verilog netlist.

```
module top (input a, input b, output c);
nand n1 (w1, a, b);
nand n2 (c, w1, w1);
endmodule
```

Listing 2: GPT-3.5's response to the prompt in Listing 1.

IV. Methodology

In this section, we first provide a preliminary formulation to pirate firm hardware intellectual property (IP), i.e., gate-level Verilog netlists, using LLMs. Then, we show that this preliminary formulation only works to an extent and doesn't help us successfully pirate IPs. Then we delve into the details of the limitations and describe the different challenges that need to be overcome to achieve our goal. We also devise solutions to address these challenges and build our framework, which successfully pirates hardware IP and evades detection by state-of-the-art piracy detection tools solely through black-box LLM access.

A. LLMs for Pirating IPs - Formulation, Challenges, and Solutions

Here, we devise a preliminary formulation using LLMs to pirate IPs in the form of gate-level netlists. To that end, consider the example prompt shown in Listing 1. Here, we simply ask the LLM to rewrite a Verilog gate-level netlist for an `OR` gate. Mathematically, this formulation can be represented as follows: The response R is obtained from the underlying distribution

$$p(R|Q,\theta), \tag{1}$$

where θ denotes the parameters of the LLM, Q denotes the query, and p represents the probability (since LLMs' responses are not deterministic).

Listing 2 contains the code portion of an example response from the LLM.[2] The generated netlist is a valid gate-level netlist, but it is not functionally equivalent to the original (as it implements an `AND` gate, not an `OR` gate). Similar results hold true for other simple netlists as well, which leads us to the first challenge in pirating IPs.

Challenge 1: Difficulty Understanding and Rewriting Simple Hardware Circuit Netlists. Although LLMs understand the syntax of gate-level netlists and generate syntactically correct netlists that compile successfully, the generated netlists do not maintain the same functionality even for extremely small and simple modules. This is likely because most of the Verilog codes available on GitHub and other sources for

[2]Although the listing shows GPT-3.5's response, we also tested GPT-4 and observed similar results.

```
Can you refactor this circuit so that
functionality remains the same but the
Boolean operators are different? Return the
circuit in the same format.
c = OR(a,b)
```

Listing 3: Updated prompt for rewriting Verilog netlist.

```
c = NAND(NOT(a), NOT(b))
```

Listing 4: GPT-3.5's response to the prompt in Listing 3.

LLMs' training data is at the RTL abstraction and not at the gate-level netlist abstraction.

Solution A: Prompt Syntax Translation For Hardware Netlists. To address this challenge, we revise the formulation to (i) extract only the relevant parts (i.e., the gates and not the `module` declarations, `endmodule` declaration, etc.) from the Verilog netlist, and (ii) translate the syntax of the extracted gates into a more generic format of Boolean functions (as opposed to gate declarations in the standard Verilog syntax). For instance, the standard Verilog syntax of "`or U1 (c, a, b);`" would be translated into a generic Boolean function format as "`c = OR(a,b)`". Mathematically, in this updated formulation, the response R is obtained from the underlying distribution

$$p(R|\mathcal{T}(Q), \theta), \tag{2}$$

where $\mathcal{T}(Q)$ denotes that the query, Q, is processed to extract the relevant parts (i.e., the gates). These gates are then translated (denoted by $\mathcal{T}(\cdot)$) into a generic Boolean function format, which assists the LLM in generating better responses. Finally, note that the response R is also post-processed using $\mathcal{T}^{-1}$, the inverse of $\mathcal{T}$, to translate the generic Boolean function syntax back to the standard Verilog syntax. We omit this in the formulation for the sake of clarity. For additional information regarding the translation process, see Sec. VII-F of the Appendix.

Listings 3 and 4 show the translated prompt, $\mathcal{T}(Q)$, according to this updated formulation and the corresponding response, R, from GPT-3.5, respectively. As shown, the LLM is not only able to understand the provided circuit in the generic Boolean function format, but it actually rewrites the circuit correctly using the `NAND` and `NOT` Boolean functions while maintaining the overall functionality. Thus, theoretically, LLMs can be used to modify gates with the objective of evading evade piracy detection tools. However, in practice, when we use the above formulation (i.e., the one in Eq. (2)), we face challenges related to scalability and effectiveness. Next, we describe these challenges and how we overcome them.

Challenge 2: Lack of Scalability to Large Netlists. Although the LLM successfully rewrites the netlist in the example above, that example contains a toy netlist with just one gate. Real-world netlists contain several thousands, if not hundreds of thousands, of gates. To check the formulation's capability in scaling to larger netlists, we test for a small standard benchmark circuit, `c17`, which contains 6 gates, as shown in Listing 5. Following the formulation in Eq. (2), we query the LLM with the prompt shown in Listing 6. The LLM's response in Listing 7 shows that although it follows the instruction and uses different Boolean operators, the resulting circuit is

```
module c17 (N1,N2,N3,N6,N7,N22,N23);
input N1,N2,N3,N6,N7;
output N22,N23;
wire N10,N11,N16,N19;
nand U1 (N10, N1, N3);
nand U2 (N11, N3, N6);
nand U3 (N16, N2, N11);
nand U4 (N19, N11, N7);
nand U5 (N22, N10, N16);
nand U6 (N23, N16, N19);
endmodule
```

Listing 5: `c17` benchmark Verilog netlist.

```
Can you refactor this circuit so that
functionality remains the same but the
Boolean operators are different? Return the
circuit in the same format.
N10 = NAND(N1, N3)
N11 = NAND(N3, N6)
N16 = NAND(N2, N11)
N19 = NAND(N11, N7)
N22 = NAND(N10, N16)
N23 = NAND(N16, N19)
```

Listing 6: Prompt corresponding to Listing 5.

not functionally equivalent to the original circuit (because `AND(NOT(a), NOT(b))` $\neq$ `NAND(a, b)`).

Challenge 3: Limited Token Context Windows of LLMs. Another limitation of the above formulation is that rewriting netlists by simply providing all gates to the LLM is not possible. This is because all LLMs have finite input token context windows, meaning that the prompt size cannot be too large. For example, OpenAI's GPT-3.5 LLM (more specifically, `gpt-3.5-turbo-0125`) has a context window of 16,385 tokens [63], which, assuming ≈4 characters per token [66], translates to ≈65,540 characters. However, practical netlists containing thousands or more gates have hundreds of thousands of characters. Thus, it is not possible to rewrite realistic netlists by providing all gates to LLMs.

Solution B: Pre-characterization and Divide-and-conquer. To address these challenges, we modify the formulation by characterizing the given netlist, as explained next. Suppose we wish to rewrite a given Verilog gate-level netlist. Instead of simply extracting all the gates and translating them to create one big prompt (as shown in Listings 5 and 6), we first analyze the netlist and extract all the different gate types (e.g., 2-input `AND` gates, 3-input `AND` gates, `XOR` gates, etc.). Then, for each unique gate type, we create a representative circuit in a generic Boolean function format, as explained in Solution A above. Finally, for each representative circuit in the generic Boolean function format, we independently prompt the LLM to rewrite that circuit. Note that, for each representative circuit (i.e., gate type), we devise lists of specific Boolean operators (different from the gate in the original circuit) and instruct

```
N10 = AND(NOT(N1), NOT(N3))
N11 = AND(NOT(N3), NOT(N6))
N16 = AND(NOT(N2), NOT(N11))
N19 = AND(NOT(N11), NOT(N7))
N22 = AND(NOT(N10), NOT(N16))
N23 = AND(NOT(N16), NOT(N19))
```

Listing 7: GPT-3.5's response to the prompt in Listing 6.

TABLE II: Allowed Boolean operators for different gates crafted to achieve structural differences.

Gate	AND	OR	NAND	NOR	XOR	XNOR
Allowed Operators For Transformation	[NAND]/[NOR]/[OR, NOT]	[NAND]/[NOR]/[AND, NOT]	[NOR]/[AND, NOT]/[OR, NOT]	[NAND]/[AND, NOT]/[OR, NOT]	[NAND]/[NOR]	[NAND]/[NOR]

```
1 Can you refactor this circuit following
  these instructions? 1) Use only OR and/or
  NOT Boolean operators. 2) Ensure that the
  final functionality remains the same. 3)
  Just give me the new circuit and nothing
  else. 4) Generate your response in the
  following format: <output> = <gate type> (<
  inputs>).
2 c = AND(a,b)
```

Listing 8: Example prompt for instructing the LLM to use only specific Boolean operators from Table II according to Solution B.

the LLM to only use operators from that list in order to achieve structural differences and result in successful piracy. For instance, if the representative circuit is for an OR gate, the list of Boolean operators the LLM is allowed to use is one of the following: [NAND], [NOR], or [AND, NOT].[3] Table II shows the different Boolean operators we allow for each type of gate. Also, Listing 8 shows an example prompt when we want the LLM to rewrite a 2-input AND gate. From the three available operator options for the AND gate, in this example, we choose the [OR, NOT] operators. The last two instructions in the prompt are provided for ease of parsing the generated response.

In essence, we use a divide-and-conquer approach where instead of asking the LLM to rewrite the entire netlist, we analyze the netlist, extract different types of gates, and then ask the LLM to individually rewrite the circuits corresponding to the different types of gates. Doing so (i) allows us to successfully scale to large netlists since we characterize them and focus on different gate types individually, and (ii) overcomes the issue of limited token context windows of LLMs since the prompt for each unique gate type is independent of others. Mathematically, the formulation is updated to obtain response R_i for the i^{th} unique gate type from the underlying distribution

$$p(R_i|\mathcal{T}(Q_i),\theta), \quad \forall i \in \{1,2,\ldots,|G|\}, \tag{3}$$

where G is the set of unique gate types and $\mathcal{T}(Q_i)$ denotes the translated query (from Verilog gate format to generic Boolean function format with instructions about allowed Boolean operators) for the i^{th} unique gate type.

Challenge 4: Error-prone Single-shot Netlists. The final issue with the formulation described so far is that it only allows the LLMs one chance to generate a functionally equivalent circuit that uses different Boolean operators. However, due to randomness in the LLMs' responses and differing training processes (including number of parameters, size and quality of training data, and training processes such as pre-training, fine tuning, instruction tuning, reinforcement learning, etc.), different LLMs have varying amounts of success in rewriting circuits. This implies that, with this single-shot formulation, an LLM's response could be classified as a failure even though it might only generate a slightly incorrect circuit.

Solution C: Feedback-guided Interactive Formulation. To overcome this issue and ensure that LLMs are not penalized for minor mistakes, we leverage the interactive capabilities of LLMs by combining them with multi-level fine-grained feedback. More specifically, we allow the target LLM M attempts for each different gate type in G, the set of unique gate types. For each attempt, we first check if the generated response results in a valid circuit, i.e., the syntax adheres to the generic format of Boolean functions in which the input circuit is provided. If the response does not pass this check, we provide the LLM feedback about its incorrect format and ask it to try again. On the other hand, if the response passes this check, we then check if the generated circuit follows our instructions regarding the allowed set of Boolean operators (i.e., the allowed operators specified in the prompt instruction). If the circuit fails this second check, we provide the LLM feedback about the use of operators that are not allowed and ask it to try again. However, if the circuit passes this second check, we move on to the third check where we evaluate the functional equivalence of the generated circuit to the original circuit. Again, if the generated circuit is not functionally-equivalent to the original circuit, we provide the LLM feedback about the non-equivalence and ask it to try again. Whereas, if the generated circuit passes this third check, we save the generated circuit (for later use in pirating circuits) and move on to the next gate type. Note that, for each gate type, if any of the three checks fails, we count it as a failed attempt (and increment the counter for the number of attempts), so each LLM has M attempts to pass all three checks combined. As evidenced by our results, such interactive feedback-guided approach significantly improves LLMs performance (see Sec. V-H).

The updated mathematical representation for this formulation is as follows: The final response (after potentially up to M attempts) R_i^M for the i^{th} unique gate type is obtained from the underlying distribution

$$p(R_i^j|\mathcal{C}(R_i^{j-1})\oplus\mathcal{T}(Q_i^j),\theta), \quad j \in \{1,2,\ldots,M\}, \quad \forall i \in \{1,2,\ldots,|G|\}, \tag{4}$$

where R_i^j denotes the LLM's response in the j^{th} attempt for the i^{th} unique gate type, and $\mathcal{C}(\cdot)$ denotes a function that analyzes the response for the three checks mentioned above (syntax, allowed operators, and functionality) and produces a feedback according to the result of the checks. Additionally, $\oplus$ denotes concatenation, which combines the feedback with the original circuit, creating the query for the next attempt. We use this final formulation to generate functionally-equivalent but structurally different circuits for all unique gate types in the target netlist.

Next, we describe our end-to-end flow of pirating Ver-

[3]We create these lists to ensure that operators in each list form a "complete set", i.e., it is possible to rewrite the given representative circuit gate using only the operators in the list.

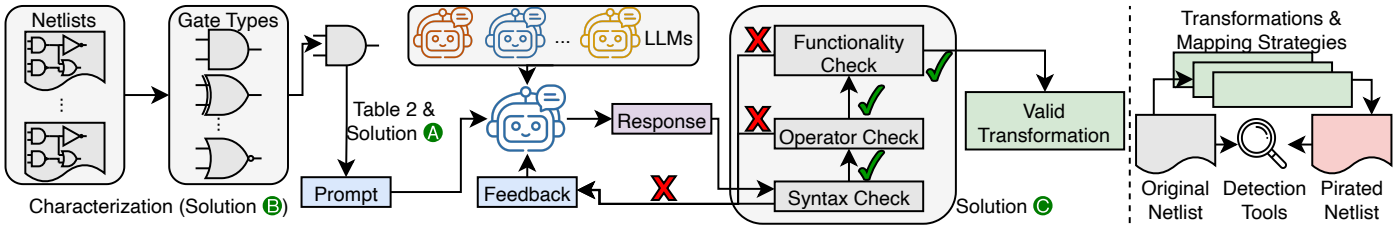

Fig. 2: *LLMPirate*'s end-to-end automated flow. All steps, including characterization, prompt syntax translation, syntax, operator and functionality checks, feedback, and the generation of pirated netlists using the LLM-generated transformations are end-to-end automated, and no manual intervention is needed.

ilog netlists, which includes A prompt Syntax translation For hardware netlists, B pre-characterization and divide-and-conquer, and C a feedback-guided interactive approach.

B. Putting It All Together

Figure 2 illustrates the end-to-end flow. Given a netlist (or a set of netlists) to be pirated, we first perform pre-characterization (Solution B), which analyzes the netlist(s) to extract the different gate types. Then, for each different gate type, we use the list of allowed Boolean operators in Table II to create prompts following the generic Boolean operator syntax (Solution A). For instance, for each of the different `AND` gate types in the target netlist(s) (e.g., 2-input `AND` gate, 3-input `AND` gate, etc.), we create three prompts, one for each of the allowed Boolean operators: [`NAND`], [`NOR`], [`OR`, `NOT`]. This way, we create prompts for all different gate types for each of the corresponding allowed Boolean operators. Then, we pick a target LLM and query it for responses to these prompts, one after another. Additionally, as explained in Solution C, after each response, we perform a series of checks (for syntax, use of only allowed Boolean operators, and functional equivalence). If any of these checks fail, we provide appropriate feedback to the LLM using a follow-up prompt. For instance, if the generated circuit fails the functionality check, we provide the following feedback "`This is not correct because the functionality is not the same as the original circuit. Can you try again? Below is the original circuit:`", followed by the original circuit in the generic Boolean function format provided in the initial prompt. In this way, we provide the LLM M attempts to generate a circuit that passes all three checks. If, during any attempt, the LLM is successful, we save the generated circuit as a valid transformation of the original circuit so we can later use it for pirating netlists. For example Boolean transformations, see Sec. VII-H of the Appendix. On the other hand, even after M attempts, if the LLM is unable to generate a a circuit that passes all three checks, we exit the loop and move on to the next allowed Boolean operator or to the next gate type. Thus, at the end, we obtain a dictionary of functionally equivalent transformations for all (or some, depending on the success of the LLM) different gate types using all (or some) of the different allowed Boolean operators for the corresponding gate type. For further information regarding the contribution of each solution within the framework, see the ablation study in Sec. VII-D of the Appendix.

Next, we describe how to pirate a given netlist using this dictionary of transformations. Recall that, for each different gate type, we have multiple transformations in the dictionary. In order to select the exact transformation to apply for a given gate when pirating a netlist, we devise five *mapping strategies*: *AND_NOT*, *NAND*, *NOR*, *OR_NOT*, and *random*. The *NAND* mapping strategy only uses the [`NAND`] transformation, the *AND_NOT* mapping strategy only uses the [`AND`, `NOT`] transformation, and so on.[4] Finally, we pirate a given netlist using each of the five mapping strategies (one by one) by replacing the gates in the original netlist according to the transformation determined by the mapping strategy. Additionally, to overcome randomness, we repeat this process N times and evaluate each of the $N \times 5$ pirated versions using the piracy detection tools to obtain the similarity scores.

Note that, to ensure ease-of-use and wide application, the entire *LLMPirate* flow described above is automated end-to-end, from characterizing netlists, to creating prompts, querying LLMs, performing the three checks, providing feedback to the LLMs, creating pirated versions of netlists, and finally evaluating them using the detection tools. Additionally, we ensure that the pirated netlists are functionally equivalent to the original netlists through exhaustive simulation-based testing. We further validate equivalence using *Cadence Conformal Equivalence Checker* [12], an industry-standard commercial formal equivalence checker. We describe this in more detail in Sec. VII-C of the Appendix. Next, we demonstrate *LLMPirate*'s efficacy in pirating netlists and evading a variety of detection tools.

V. Results

We conduct a detailed experimental investigation of the capabilities of different LLMs to pirate hardware IPs. Next, we detail our experimental setup.

A. Experimental Setup

We implement *LLMPirate* using *Python*. We set M, the maximum number of attempts available to the LLMs, to be 5. We set N, the number of pirated netlists created for each mapping strategy to capture the effect of randomness (Sec. IV-B), to be 5. We use a dataset of 31 different Verilog netlists from the GNN4IP repository for our experiments [1]. We choose these netlists because of two reasons: (i) GNN4IP is trained on them, making this a more difficult setting for our attack than the typical setting where one pirates netlists that the detection tool has not seen before, e.g., netlists from the testing set of GNN4IP. We adhere to this challenging scenario to highlight *LLMPirate*'s remarkable proficiency in effectively

[4]Since it is possible that some transformations might not exist for a given gate type, in that case, we pick a random transformation available in the dictionary for that gate type.

TABLE III: Details of detection tools used in our evaluation.

Detection Tool	GNN4IP [81]	MOSS [2]	Jplag [41]	SIM [35]
Source	GNN4IP Repository [1]	Internet Submission Method [2]	Jplag Repository [41]	Source Code [35]
Similarity Scores Range	[-1,1]	[0,1]	[0,1]	[0,1]
Detection Threshold	0	0.2	0.3	0.3
Notes	Designed for Verilog, Most accurate tool for our case	Supports Verilog, Stricter threshold as it was primarily designed for software code	Doesn't support Verilog, Relatively strict threshold as we use it in text mode	Doesn't support Verilog, Relatively strict threshold as we use it in text mode

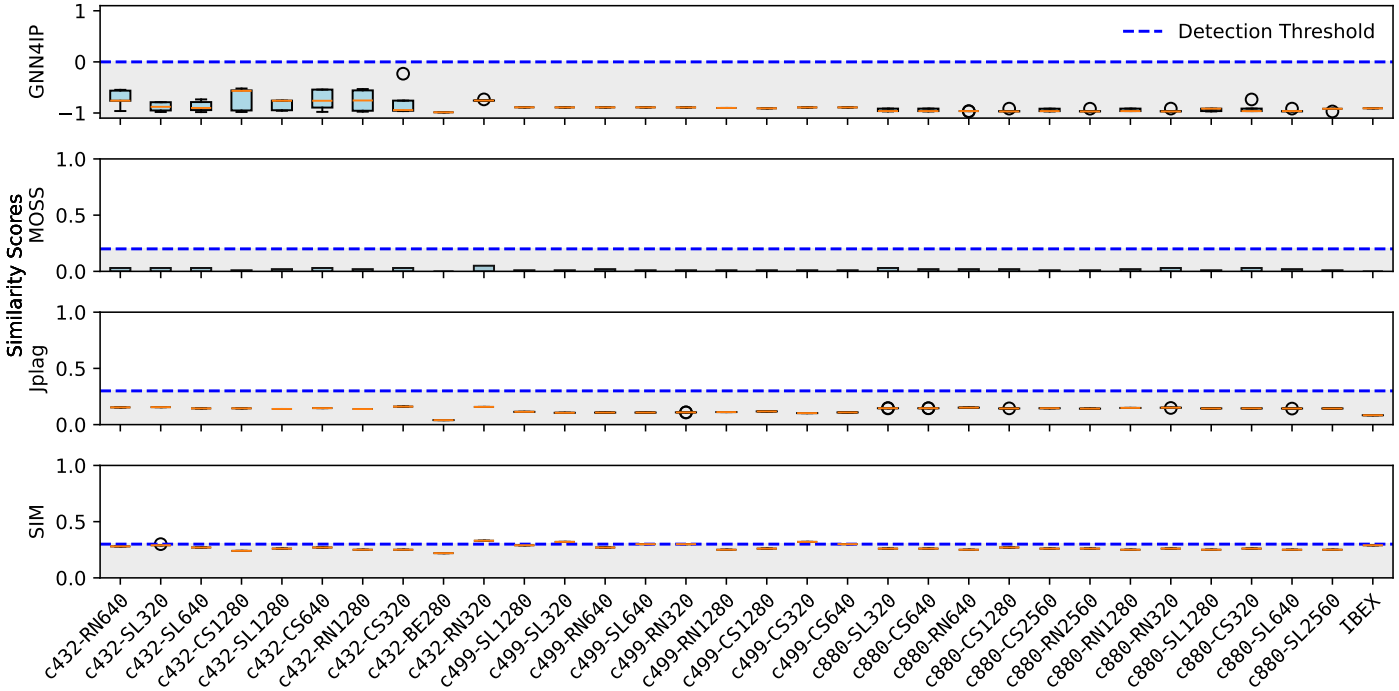

Fig. 3: *LLMPirate*'s best performance against GNN4IP [81], MOSS [2], Jplag [41], and SIM [35].

pirating netlists. (ii) Additionally, we are constrained by these available netlists in GNN4IP to perform a fair evaluation of GNN4IP. However, these netlists are small, and small netlists are difficult to pirate since there is a limited set of gates to work with and detection tools perform very well on them. However, to showcase *LLMPirate*'s scalability and ramifications, we also test it on large netlists, `IBEX` [47] and `MOR1KX` [5] processors, and a `GPS` [55] module. We chose these netlists because of their significant design complexity, practical relevance, and widespread adoption across various applications. Our target netlists range from a few hundred gates to hundreds of thousands of gates. We provide the netlist size metrics in Table IV in Sec. VII-A of the Appendix.

LLMs. For a thorough analysis, we select eight representative LLMs for our evaluations:

- **CoPilot** from Microsoft uses the Prometheus model, and iteratively generates search queries, to combine Bing search results with OpenAI's GPT-4 and GPT-4 Turbo LLMs to produce responses [54]. We use CoPilot via http://copilot.microsoft.com.
- **GPT-3.5** models from OpenAI that can understand and generate natural language or code and have been optimized for chat and instruction based tasks [64]. We use the `gpt-3.5-turbo-16k` model in our experiments.
- **GPT-4** models improve on GPT-3.5 and can understand as well as generate natural language or code with greater accuracy than any of the previous GPT models [65]. Also, GPT-4 is one of the most advanced general-purpose LLMs available today. We use the `gpt-4-turbo` model, the most advanced GPT-4 model, in our experiments.
- **Claude** models from Anthropic can perform complex analysis, tasks with multiple steps, and higher-order math and coding tasks [4]. Also, they have low hallucination rates [4]. We use the `claude-3-opus-20240229` model, the most advanced Claude model, in our experiments.
- **Gemini** models from Google are built for reasoning across text, images, audio, video, and code [33]. Gemini 1.0 was the first model to outperform human experts on the Massive Multitask Language Understanding benchmark. [32]. We use the `gemini-1.0-pro` model in our experiments.
- **Llama2** is a set of open-source LLMs developed by Meta. At the time of release, Llama 2 outperformed the other open-source models across all benchmarks [52]. CodeLlama is a code-specialized version of Llama2 that was created by further training Llama2 on its code-specific datasets [49]. In our experiments we use two variants of CodeLlama: `CodeLlama-7b-Instruct-hf` and `CodeLlama-13b-Instruct-hf` (denoted as CL-7B and CL-13B in our evaluations) from HuggingFace [39].

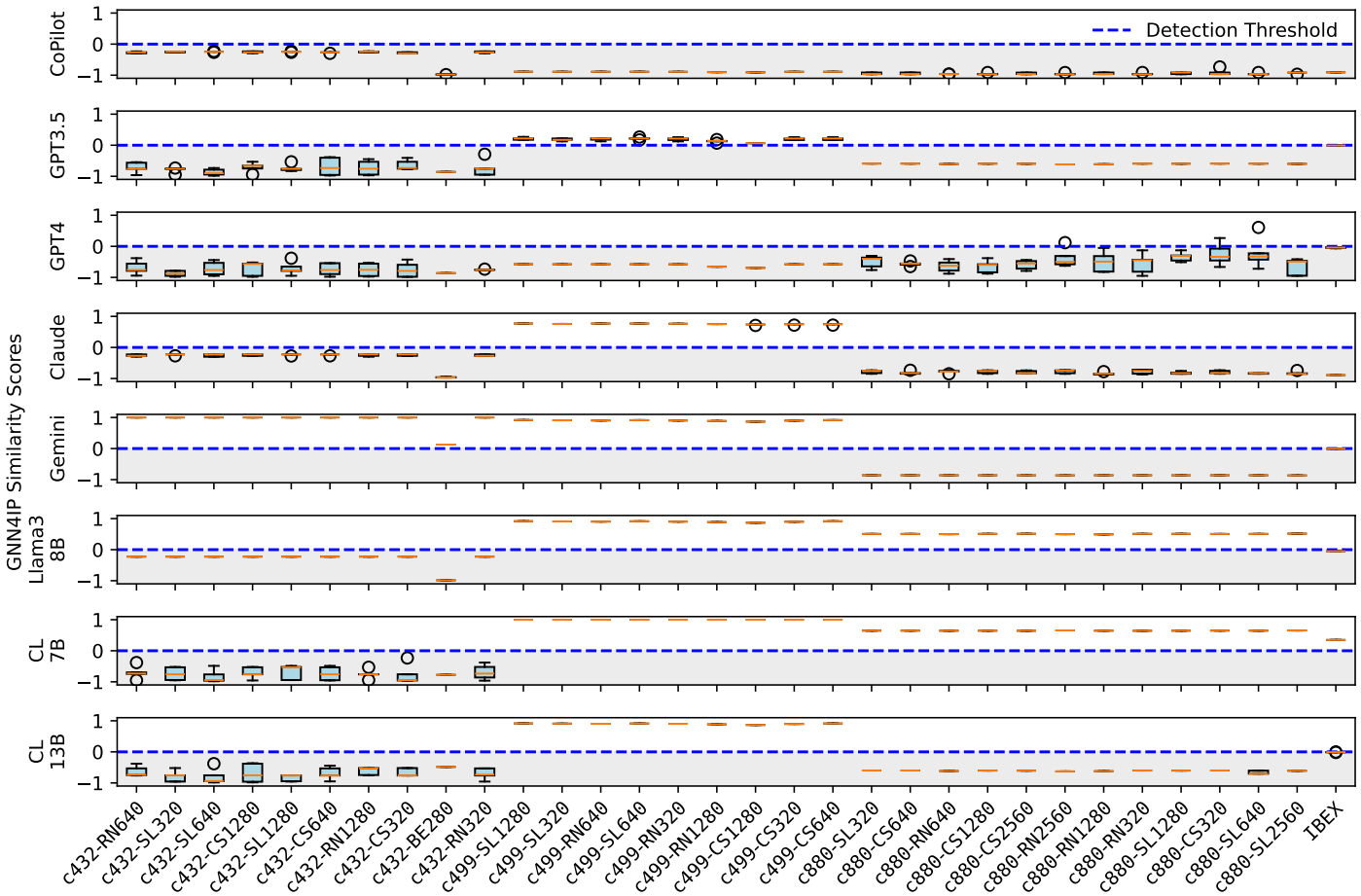

Fig. 4: Distribution of GNN4IP [81] similarity scores for different LLMs in *LLMPirate*'s framework.

- **Llama3** is a recently released and highly capable openly available set of LLMs [50]. These models have greatly improved capabilities like reasoning, code generation, and instruction following. We use the `Meta-Llama-3-8B-Instruct` (denoted as Llama3-8B) model in our experiments.

Our selection of LLMs represents some of the most advanced LLMs available today from a variety of organizations, as well as current state-of-the-art and recently released publicly available LLMs capable of performing code-related tasks.

Detection Tools. To evaluate the success of these LLMs in pirating Verilog netlists, we use the piracy/similarity measurement tools described in Table III. As explained in Sec. II-C, our selection consists of highly accurate, widely-used, and popular techniques ranging over multiple decades. The remainder of the section is organized as follows: First, we provide the main piracy results against these detection tools (Sec. V-B). Then, we provide further results and analysis of the performance of the LLMs against each detection tool separately (Secs. V-C-V-F). We also perform more analyses of the different mapping strategies and the number of attempts available to the LLMs (Secs. V-G, V-H). Then, we demonstrate the ramifications of *LLMPirate* through case studies on practical netlists, `IBEX` and `MOR1KX` processors and a `GPS` module (Secs. V-I, V-J). Finally, we summarize the key characteristics of LLMs in an effort to understand their performances in Sec. V-K.

B. Main Piracy Results

Figure 3 shows the distribution of best (i.e., the lowest) similarity scores from the four detection tools for the 32 netlists in our dataset.[5] It is clear that using *LLMPirate*, we are successfully able to pirate all 32 netlists against all four detection tools with very limited variance in performance. Note that since MOSS limits use to 100 queries per day per user [2], we randomly picked one of the $N = 5$ pirated netlists for each mapping strategy and queried MOSS for similarity. Hence, the plots for MOSS are bar plots showing the single similarity scores instead of box plots for the distribution of similarity scores. Also note that the Jplag and SIM similarity scores are higher (compared to MOSS) because Verilog netlists have keywords (e.g., `and`, `nand`, etc.) that are repeated frequently, and since these detection tools are used in text mode, such repeated keywords contribute to high similarity scores. Nonetheless, *LLMPirate* successfully pirates all netlists. Additionally, due to our divide-and-conquer approach and saving of generated transformations, the runtime of *LLMPirate* for any given LLM is in the order of a few minutes. Furthermore, the performance overheads of our pirated netlists are also reasonable (see Sec. VII-E in the Appendix).

[5]Note that, to showcase the best results achieved using *LLMPirate*, the similarity scores plotted are the best (i.e., the lowest) scores over all mapping strategies and LLMs. We provide more fine-grained results of the performance of different LLMs and mapping strategies in the subsequent subsections.

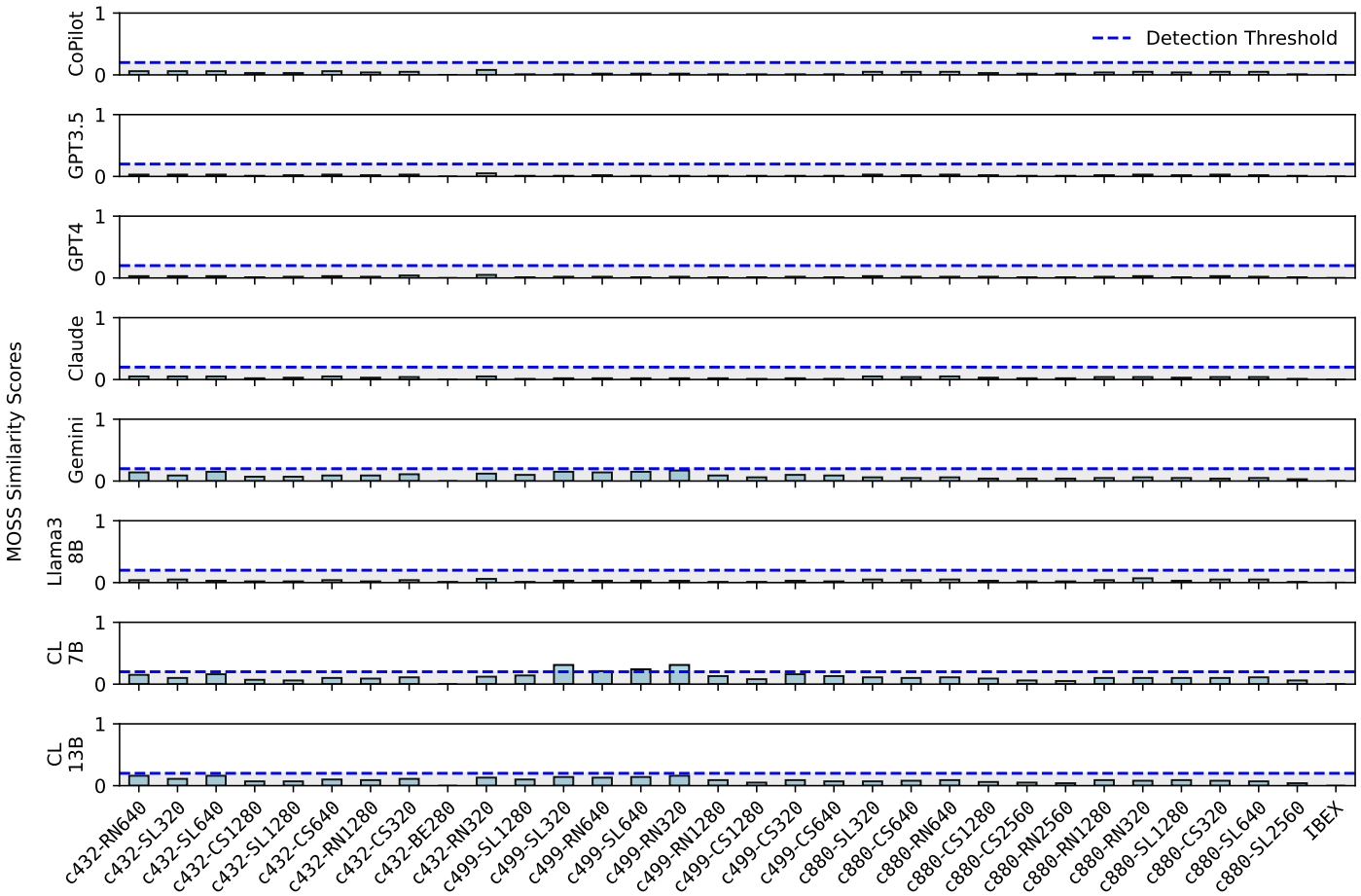

Fig. 5: MOSS [2] similarity scores for different LLMs in *LLMPirate*'s framework.

C. LLMs Against GNN4IP

To evaluate the LLMs' ability to pirate Verilog netlists, we first analyze GNN4IP's similarity scores between each of the pirated netlists and the original netlists. Figure 4, summarizes these values across all 32 netlists for each LLM. Note that the distribution of similarity scores plotted for each netlist for each LLM are for the best mapping strategies for that netlist and LLM.

Here are the key takeaways from the figure: (i) Most LLMs (CoPilot, GPT-3.5, GPT-4, Claude, and CL-13B, i.e., CodeLlama-13B) are successfully able to evade GNN4IP detection for the majority of the netlists. (ii) Some LLMs (Gemini, Llama3-8B, and CL-7B) are unable to pirate the majority of netlists (the reason behind this in explained in Sec. V-H). The worst performing LLMs are CL-7B and Llama3-8B, only evading detection on 10 and 11 of the 32 netlists, respectively. (iii) CL-13B performed significantly better (23 successes) than its smaller version, CL-7B (10 successful netlists). (iv) Overall, GPT-4 and CoPilot (which uses GPT-4 internally) achieve the best performance, i.e., lowest GNN4IP similarity scores.

D. LLMs Against MOSS

Here, we repeat the evaluation process using MOSS as the piracy detection tool. As explained in Sec. V-B, due to restrictions on the number of queries, the plots for MOSS in Figure 5 are bar plots showing the single similarity score instead of box plots showing the distribution of similarity scores. Note that these single similarity scores still provide enough information to analyze the performance of LLMs against MOSS. Also, as in Sec. V-C, the similarity score plotted for each netlist for each LLM is for the best mapping strategy for that netlist and LLM.

Here are the key takeaways: (i) All LLMs except CL-7B evade MOSS for all netlists. (ii) As with GNN4IP, CL-13B performs better than the smaller CL-7B. (iii) Notably, Llama3-8B is at par with larger models (e.g., GPT-3.5 and GPT-4).

E. LLMs Against JPlag

Here, we use the same evaluation procedure using Jplag as the piracy detection tool. To adhere to the page limit, we only provide key takeaways here and refer the reader to the Appendix of the extended version of this work in [29] for comprehensive results and analyses. These key takeaways are as follows: (i) Most closed-source LLMs (CoPilot, GPT-3.5, GPT-4, Claude) successfully evade detection across all 32 netlists. (ii) As seen before, CL-13B performs better than the smaller CL-7B. (iii) As with MOSS, the open-source Llama3-8B performs almost as well as the larger models, successfully bypassing JPlag for all but one netlists.

F. LLMs Against SIM

The evaluation is again repeated using SIM as the piracy detector. Similar to Sec. V-E, key takeaways are described as follows, with additional results included in the extended

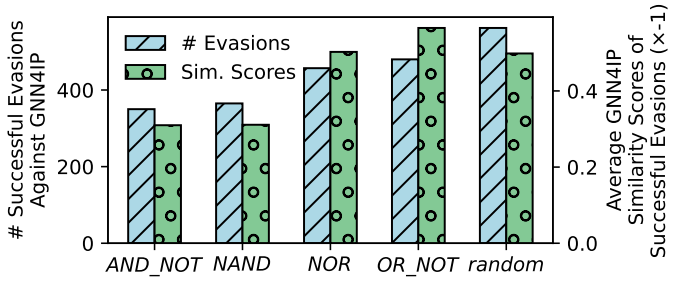

Fig. 6: Performance of mapping strategies against GNN4IP.

work's [29] Appendix: (i) Due to the lack of compatibility with Verilog and the text mode of operation of SIM, it results in unusually high similarity scores because Verilog keywords (e.g., `nand`, `and`, etc.) are repeated frequently in the netlists. (ii) Nonetheless, GPT-3.5 and GPT-4 still evade SIM for 25 netlists. (iii) Llama3-8B, with 11 successes, performs the best among open-source models.

Finding 1. Overall, GPT-4 and CoPilot achieve the best performance in successfully pirating netlists against all four detection tools.

Finding 2. Overall, CodeLlama-13B performs significantly better than the smaller CodeLlama-7B. More often than not, Llama3-8B performs better (for our task) than the CodeLlama models (which are based on Llama2).

G. Analysis of Mapping Strategies

So far, we analyzed the main piracy results against four detection tools, and the performance of different LLMs against different tools. Now, we take a closer look at the performance of the five mapping strategies, *AND_NOT*, *NAND*, *NOR*, *OR_NOT*, and *random*. More specifically, to understand the relative performance of these mapping strategies, we analyze them in terms of the number of successful instances of evasions (over all netlists and all LLMs) and the average similarity scores of those instances against GNN4IP (Figure 6). It is evident that the *random* strategy yields the largest number of successful evasions. This makes intuitive sense because, with the *random* strategy, a given gate can be replaced with any of its transformations, leading to different structures in the pirated netlist, whereas, with other strategies, the pirated netlist is likely to have similar structures due to the possibility of more deterministic replacements. This is also reflected in the low (note that the second y-axis is inverted, i.e., multiplied by -1) average GNN4IP similarity scores compared to most other strategies. We observe similar results against MOSS, Jplag, and SIM (see Sec. VII-B in the Appendix).

Finding 3. All five mapping strategies result in successful pirated netlists against all detection tools, with the *random* mapping strategy showing the best performance in terms of the similarity scores.

H. LLMs' Performance Comparison

Next, we compare the performance of the LLMs in generating successful transformations according to the allowed Boolean operators in Table II (e.g., `AND` gate using `NOR` operators, etc.). Recall that we allow each LLM a maximum of

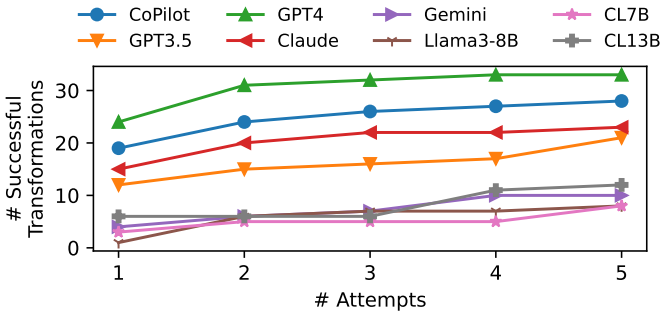

Fig. 7: Comparison of number of attempts for successful transformations using different LLMs.

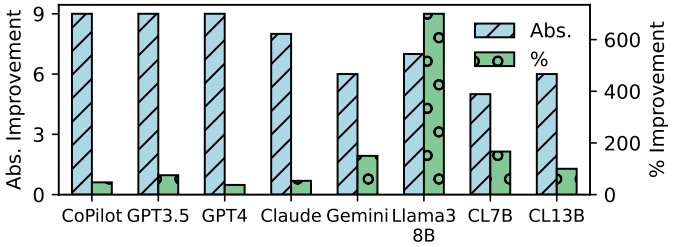

Fig. 8: Absolute and percentage improvements (attempt 5 vs. attempt 1) in successful transformations.

$M = 5$ attempts for each different gate type. Additionally, if an attempt fails, we also provide fine-grained feedback about syntax, use of correct Boolean operators, or functionality in allow the LLM to fix its mistakes. To that end, Figure 7 shows the total number of successful transformations generate by different LLMs as a function of the number of attempts. It is evident that, after 5 attempts, there are two classes of LLM in terms of number of successful transformations. The first class consists of GPT-4, CoPilot, Claude, and GPT-3.5, with 33, 28, 23, and 21 successful transformations, respectively. The second class consists of CL-13B, Gemini, Llama3-8B, and CL-7B, with significantly fewer successful transformations. This explains why the LLMs from the second class are sometimes unable to evade some detection tools. A surprising observation is that Gemini (one of the closed-source LLM that performs similar to the GPT models on other common tasks) struggles with our task of rewriting circuits. The exact reason behind this is difficult to know, however, a possible reason could be a lack of enough Verilog/circuit training data. Another observation from the figure is that all LLMs improve with more attempts and feedback. This validates our Solution C of devising a feedback-guided interactive formulation for our task.

We also analyze the impact of multiple attempts through the absolute and percentage improvements (attempt 5 vs. attempt 1) in the number of successful transformations in Figure 8. Overall, all LLMs benefit from the multiple attempts, with larger and more capable LLMs (CoPilot, GPT-3.5, GPT-4, and Claude) showing the most absolute improvement, i.e., most improvement in number of successful mappings at attempt 5 compared to attempt 1. However, smaller LLMs (such as Llama3-8B, CL-7B, and CL-13B), especially Llama3-8B, really leverage the multiple attempts with feedback and improve drastically (e.g., 700% improvement in Llama3-8B) over their poor performance in the first attempt.

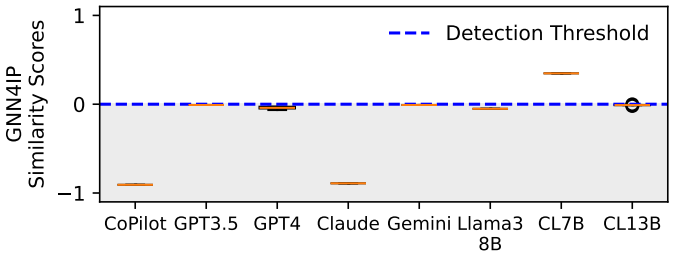

Fig. 9: Comparison of GNN4IP similarity scores for IBEX processor pirated using different LLMs.

Finding 4. We find two classes of LLMs in terms of the number of successful transformations they generate, explaining why LLMs from the second class are sometimes unable to evade detection.

Finding 5. Our interactive formulation with multiple attempts and fine-grained feedback improves the performance of all LLMs, especially smaller, less capable LLMs, such as Llama3-8B.

I. Case Study on the IBEX Processor: Ramifications of LLMPirate

In this subsection, we demonstrate and discuss the performance of *LLMPirate* on a real-world netlist, the IBEX processor [47] in more detail. Specifically, we run our end-to-end automated flow of *LLMPirate* to pirate the processor using our mapping strategies and the corresponding transformations obtained from the eight LLMs. Then, we query GNN4IP to get the similarity scores between our pirated netlists and the original netlist. Note that, similar to related works, we assume full-scan access to ensure compatibility of the netlists with GNN4IP [30]. Figure 9 compares the distribution of the GNN4IP similarity scores for the eight LLMs. Note that, as earlier, the distributions are for the best-performing strategy. We observe that seven out of the eight LLMs (all except CL-7B) successfully evade GNN4IP. Thus, *LLMPirate* easily fools GNN4IP into classifying pirated versions of IBEX as not pirated. Moreover, the distribution of the GNN4IP similarity scores is extremely low for CoPilot and Claude, meaning that not only is GNN4IP evaded, the magnitude of the incorrect detection is extremely high. This case study demonstrates the capabilities of *LLMPirate*, which can lead to piracy of practical netlists, and failure of the state-of-the-art piracy detection tool in catching it.

J. Case Study on Larger Netlists: Scalability of LLMPirate

Recall that *LLMPirate* first generates and caches (i.e., saves) valid gate transformations. Then, for any given target netlist, it creates a pirated netlist gate-by-gate. So, working with a larger netlist will only increase the runtime linearly with the number of gates, and thus will be easily manageable. We validate this scalability of *LLMPirate* by further experimenting with even larger netlists, containing hundreds of thousands of gates. More specifically, we target two open-source netlists: a GPS module from Common Evaluation Platform [55], containing $\approx$ 193K gates, and an MOR1KX processor [5], containing $\approx$ 158K gates. We observed that *LLMPirate* generates pirated netlists within seconds. Additionally, these netlists successfully evade MOSS [2], Jplag [41], and SIM [35], but surprisingly, GNN4IP [81] always classifies these *LLMPirate*-generated netlists as pirated. This unusual result might lead one to believe that GNN4IP thwarts *LLMPirate* for larger netlists. However, a closer evaluation reveals a surprising observation. To validate the efficacy of GNN4IP for these large netlists, we query GNN4IP to predict the similarity between the original GPS and MOR1KX netlists, and observe that GNN4IP yields an extremely high similarity score of 0.97. This means that GNN4IP has a high bias towards classifying large netlists as pirated, which explains why *LLMPirate*-generated netlists are unable to evade GNN4IP. This bias of GNN4IP makes it unsuitable for evaluating such netlists.

Finding 6. GNN4IP, the current state-of-the-art hardware IP piracy detector, struggles against *LLMPirate* for large netlists.

K. LLMs' Characteristics

In the previous sections, we evaluated the performance of *LLMPirate* in evading detection tools. Here, we delve deeper and provide some insights about the characteristics of LLMs that make some LLMs perform better than others in the context of hardware IP piracy.

- Model Size Matters: Large LLMs, with potentially trillions of parameters, perform best.
- Training Data Size Matters: Latest version of Llama (Llama3-8B) outperforms the older Llama2 models, again, potentially due to its $> 7\times$ training data size [51], [52].
- Open vs. Closed Source LLMs: Closed source LLMs still have a fairly decent margin compared to open source LLMs for our task.
- Potential of Smaller LLMs: With proper feedback and multiple attempts, smaller LLMs correct their mistakes.

Summary of Results. *LLMPirate* successfully pirates all given netlists (including the IBEX and MOR1KX processors and the GPS module) and evades all detection tools with only minutes of runtime cost.

VI. Related Work and Discussion

Here, we first discuss the need for *LLMPirate* over simply using Verilog-fine-tuned LLMs. Then we discuss related works on hardware and software intellectual property (IP) piracy and how our work is different from them. We also discuss applicability to other detection tools and potential countermeasures against *LLMPirate*. We also discuss the impact of netlist obfuscation in Sec. VII-G of the Appendix.

A. Evaluating Verilog-fine-tuned LLMs

As observed in Challenge 1 in Sec. IV-A, various general-purpose open-source and closed-source LLMs struggle in understanding and rewriting simple netlist written in Verilog hardware description language. So, a natural question could be about the possibility of using LLMs fine-tuned on Verilog data instead of using the solutions we described. To address this, first, note that fine-tuning LLMs is computationally much more expensive than our current approach. Nonetheless, we tested VeriGen (an LLM fine-tuned on a large corpus of

Verilog dataset) [77] by asking it to rewrite the simple gate-level netlist shown in Listing 1. To account for the LLMs' non-deterministic responses, we repeated the experiment 10 times, but VeriGen was unsuccessful in either generating a syntactically correct code, i.e., a valid gate-level netlist, or a functionally-equivalent netlist even once out of the 10 times. This demonstrates that even fine-tuned LLMs struggle in the context of our task of rewriting gate-level netlists and emphasize the need for *LLMPirate* to successfully pirate netlists.

B. Evading Hardware IP Piracy Detection

Various works have demonstrated strategies to evade hardware IP detection tools, primarily targeting GNN4IP [30], [3]. `PoisonedGNN`, exploits the susceptibility of graph neural networks (GNNs) to data poisoning attacks by injecting backdoor triggers at the register and gate-level of circuit designs within GNN4IP's training dataset [3]. The resulting accuracy of the target GNN model is then reduced, enabling pirated circuits to successfully evade GNN4IP detection. However, unlike `PoisonedGNN`, our work (i) does not require access to the GNN's training procedure, (ii) evades detection without changing the GNN's parameters or training dataset, (iii) assumes only black-box access to the target GNN, and (iv) does not design backdoors specific to target detection tools, but evades multiple types of detection tools, including GNN4IP.

AttackGNN, another recent technique, is perhaps the closest to our work in terms of evading GNN4IP [30]. By training a reinforcement learning agent, *AttackGNN* learns to perturb netlists that evade GNN4IP detection. However, there are critical differences in terms of methods, results, and impact. Unlike *AttackGNN*, *LLMPirate* (i) lowers the technical expertise barrier: our work does not need detailed understanding of reinforcement learning methods; rather, it simply requires prompting off-the-shelf LLMs, (ii) does not require time-consuming training procedure, and generates pirated netlists directly using LLMs assisted with a quick feedback-guided formulation, and (iii) is not tailored to only generating adversarial examples that evade machine-learning-based detectors such as GNN4IP; rather, *LLMPirate* evades a variety of detection tools.

C. Evading Software IP Piracy Detection

Previous works have investigated strategies in which software IP piracy detection tools can be successfully evaded. In particular, MOSSAD [23] details an automated program transformation algorithm that is able to successfully evade MOSS and JPlag. Through combining genetic programming techniques and domain-specific knowledge, MOSSAD is able to effectively generate multiple semantically equivalent variants of a given program which are evaluated as no more suspicious than a non-plagiarized counterpart [23]. However, a key component in MOSSAD is adding code lines that do not affect the final output. Using such an approach for hardware IPs is not feasible since unused gates are trivially optimized out in Verilog netlists. In fact, MOSSAD highlights this as a weakness in its approach since when working with compiled and optimized assembly code, MOSSAD fails.

LLMs have also been evaluated in their ability to generate software that bypasses plagiarism detection tools. Researchers determined that when utilized by students on programming assignments, GPT-J (a 6-billion parameter LLM) is able to generate functional code that evades MOSS detection [7]. However, such techniques focus on simply generating new software programs (not pirating existing programs) using an LLM, which is clearly not feasible for hardware codes because of LLMs' terrible performance in understanding and generating netlists (as evidenced in Sec. IV).

D. Other Detection Tools and Potential Countermeasures

Other Detection Tools. Since a detection tool (other than the ones we evaluated) can be built on different principles, it is difficult to guarantee *LLMPirate*'s success against new tools. However, since *LLMPirate* evades detection tools based on a wide variety of principles (e.g., GNN, winnowing, greedy string tiling, etc.), we are hopeful that similar results would hold for new detection tools too. Moreover, although the actual evasion performance could vary, the techniques we developed, i.e., prompt syntax translation, pre-characterization and divide-and-conquer, and different kinds of feedback, would still likely be helpful in improving piracy performance.

Potential Countermeasures. There can be a few different potential countermeasures against our work. (i) Re-training models like GNN4IP with our pirated netlists included in the training set to increase the robustness of detection. However, research has shown limitations of such approaches for GNNs [34]. (ii) Another potential countermeasure could be watermarking to identify LLM-generated text [42], however, since we don't directly use LLMs to generate pirated netlists, but process the output from LLMs to aid the piracy process, the applicability of watermarking as a countermeasure against our attack is unclear and needs further investigation.

VII. Conclusion

Large language models have become increasingly capable of understanding and generating code, leading to their adoption into the hardware design industry. However, we observe that these models also can be maliciously employed to attack vulnerabilities within this design process, particularly resulting in additional security concerns regarding hardware IP piracy.

To demonstrate this, we devised *LLMPirate*, a first-of-its-kind end-to-end automated, efficient, and practical framework that leverages the logical capabilities of LLMs to successfully pirate circuit designs in the form of Verilog netlists. Since LLMs are trained on very limited Verilog data, their off-the-shelf performance in pirating netlists is poor, so we formulate various solutions to achieve successful piracy. In particular, we perform syntax translation from netlists to generic Boolean function format, allowing the LLMs to better understand the circuit design. We use pre-characterization and divide-and-conquer techniques to overcome context window limitations and ensure scalability to large netlists. We also incorporate a fine-grained feedback-guided iterative flow to mitigate error-prone responses, ensuring reliability.

Our experimental results confirm that *LLMPirate* is able to evade detection against four state-of-the-art piracy detection tools across every tested netlist. We test on the netlists from the GNN4IP repository as those netlists are seen by GNN4IP during training and are relatively small in size, both factors making them difficult to pirate successfully. Despite

this, overall, GPT-3.5 and GPT-4 demonstrate the best ability to pirate the netlists. We also observe that the smaller LLMs (Llama3-8B, CodeLlama-7B, CodeLlama-13B) derive the most relative improvement from our feedback-guided flow. Finally, we highlight the ramifications of our work through case studies on `IBEX` and `MOR1KX` processors and a `GPS` module, demonstrating both, the capabilities of *LLMPirate* and the limitations of current piracy detectors.

ACKNOWLEDGMENT

The authors acknowledge the support from the Purdue Center for Secure Microelectronics Ecosystem – CSME#210205. This work was also partially supported by the National Science Foundation (NSF CNS–1822848 and NSF DGE–2039610).

REFERENCES

[1] AICPS, "HW2VEC: A Graph Learning Tool for Automating Hardware Security," https://github.com/AICPS/hw2vec, 2021, [Online; last accessed 9-Jul-2024].

[2] A. Aiken, "A System for Detecting Software Similarity," https://theory.stanford.edu/~aiken/moss/, [Online; last accessed 9-Jul-2024].

[3] L. Alrahis, S. Patnaik, M. A. Hanif, M. Shafique, and O. Sinanoglu, "Poisonedgnn: Backdoor attack on graph neural networks-based hardware security systems," *IEEE Transactions on Computers*, 2023.

[4] Anthropic, "Meet Claude," https://www.anthropic.com/claude, 2024, [Online; last accessed 9-Jul-2024].

[5] Baxter, Julius and Kristiansson, Stefan, "mor1kx - an OpenRISC processor IP core," https://github.com/openrisc/mor1kx, 2022, [Online; last accessed 27-Oct-2024].

[6] M. Beckerich, L. Plein, and S. Coronado, "Ratgpt: Turning online llms into proxies for malware attacks," *arXiv preprint arXiv:2308.09183*, 2023.

[7] S. Biderman and E. Raff, "Fooling moss detection with pretrained language models," in *Proceedings of the 31st ACM international conference on information & knowledge management*, 2022, pp. 2933–2943.

[8] M. Botacin, "Gpthreats-3: Is automatic malware generation a threat?" in *2023 IEEE Security and Privacy Workshops (SPW)*, 2023, pp. 238–254.

[9] K. W. Bowyer and L. O. Hall, "Experience using "MOSS" to detect cheating on programming assignments," in *FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No. 99CH37011*, vol. 3. IEEE, 1999, pp. 13B3–18.

[10] S. Brown, "Cadence creates industry's first LLM Technology for Chip Design," https://community.cadence.com/cadence_blogs_8/b/corporate/posts/cadence-creates-industry-s-first-llm-technology-for-chip-design, 2023, [Online; last accessed 9-Jul-2024].

[11] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," *arXiv preprint arXiv:2303.12712*, 2023.

[12] Cadence, "Cadence - Logic Equivalence Checking," https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/logic-equivalence-checking.html, [Online; last accessed 27-Oct-2024].

[13] ——, "Cadence.AI Generative AI Platform," https://www.cadence.com/en_US/home/solutions/cadence-ai-platform.html, 2024, [Online; last accessed 9-Jul-2024].

[14] A. Cambon, B. Hecht, B. Edelman, D. Ngwe, S. Jaffe, A. Heger, M. Vorvoreanu, S. Peng, J. Hofman, A. Farach, M. Bermejo-Cano, E. Knudsen, J. Bono, H. Sanghavi, S. Spatharioti, D. Rothschild, D. G. Goldstein, E. Kalliamvakou, P. Cihon, M. Demirer, M. Schwarz, and J. Teevan, "Early llm-based tools for enterprise information workers likely provide meaningful boosts to productivity," Microsoft, Tech. Rep. MSR-TR-2023-43, December 2023. [Online]. Available: https://www.microsoft.com/en-us/research/publication/early-llm-based-tools-for-enterprise-information-workers-likely-provide-meaningful-boosts-to-productivity/

[15] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman *et al.*, "Multipl-e: a scalable and polyglot approach to benchmarking neural code generation," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3675–3691, 2023.

[16] R. S. Chakraborty and S. Bhunia, "Rtl hardware ip protection using key-based control and data flow obfuscation," in *2010 23rd International Conference on VLSI Design*, 2010, pp. 405–410.

[17] codequiry, "How to detect plagiarism in source code," https://codequiry.com/resources/how-to-detect-code-plagiairsm, 2021, [Online; last accessed 9-Jul-2024].

[18] ——, "Related technologies," https://www.monash.edu/learning-teaching/teachhq/Assessment/academic-integrity/related-technologies, 2024, [Online; last accessed 9-Jul-2024].

[19] M. Cohen, "The 600 Billion Dollar China IP echo chamber," https://chinaipr.com/2019/05/12/the-600-billion-dollar-china-ip-echo-chamber/, 2019, [Online; last accessed 9-Jul-2024].

[20] CorporateTeam, "Rapid silicon announces RapidGPT's official availability," https://rapidsilicon.com/rapid-silicon-announces-rapidgpts-official-availability/, 2023, [Online; last accessed 9-Jul-2024].

[21] M. DeLorenzo, A. B. Chowdhury, V. Gohil, S. Thakur, R. Karri, S. Garg, and J. Rajendran, "Make every move count: Llm-based high-quality rtl code generation using mcts," *arXiv preprint arXiv:2402.03289*, 2024.

[22] M. DeLorenzo, V. Gohil, and J. Rajendran, "CreativEval: Evaluating Creativity of LLM-Based Hardware Code Generation," *arXiv preprint arXiv:2404.08806*, 2024.

[23] B. Devore-McDonald and E. D. Berger, "Mossad: defeating software plagiarism detection," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428206

[24] DfX-NYUAD, "Breaking_CAS-Lock," https://github.com/DfX-NYUAD/Breaking_CAS-Lock/tree/main, 2021, [Online; last accessed 15-Nov-2024].

[25] S. Engels, M. Hoffmann, and C. Paar, "The end of logic locking? a critical view on the security of logic locking," *Cryptology ePrint Archive*, 2019.

[26] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," *arXiv preprint arXiv:2310.03533*, 2023.

[27] W. Fu, S. Li, Y. Zhao, H. Ma, R. Dutta, X. Zhang, K. Yang, Y. Jin, and X. Guo, "Hardware Phi-1.5 B: A Large Language Model Encodes Hardware Domain Specific Knowledge," *arXiv preprint arXiv:2402.01728*, 2024.

[28] GitHub, "GitHub CoPilot," https://github.com/features/copilot, 2024, [Online; last accessed 9-Jul-2024].

[29] V. Gohil, M. DeLorenzo, V. V. A. S. V. Nallam, J. See, and J. Rajendran, "LLMPirate: LLMs for Black-box Hardware IP Piracy," *arXiv preprint arXiv:2411.16111*, 2024.

[30] V. Gohil, S. Patnaik, D. Kalathil, and J. Rajendran, "AttackGNN: Red-Teaming GNNs in Hardware Security Using Reinforcement Learning," *arXiv preprint arXiv:2402.13946*, 2024.

[31] Google, "Supercharge your creativity and productivity," https://gemini.google.com/, 2024, [Online; last accessed 9-Jul-2024].

[32] Google DeepMind, "Meet the first version of Gemini — our most capable AI model." https://deepmind.google/technologies/gemini/#gemini-1.0, 2024, [Online; last accessed 9-Jul-2024].

[33] ——, "Welcome to the Gemini era," https://deepmind.google/technologies/gemini/#introduction, 2024, [Online; last accessed 9-Jul-2024].

[34] L. Gosch, S. Geisler, D. Sturm, B. Charpentier, D. Zügner, and S. Günnemann, "Adversarial Training for Graph Neural Networks," *arXiv preprint arXiv:2306.15427*, 2023.

[35] D. Grune, "The software and text similarity tester SIM," https://dickgrune.com/Programs/similarity_tester/, [Online; last accessed 9-Jul-2024].

[36] ——, "SIM Manual," https://dickgrune.com/Programs/similarity_tester/sim.pdf, 2017, [Online; last accessed 9-Jul-2024].

[37] V. Hassija, V. Chamola, V. Gupta, S. Jain, and N. Guizani, "A survey on supply chain security: Application areas, security threats, and solution architectures," *IEEE Internet of Things Journal*, vol. 8, no. 8, pp. 6222–6246, 2020.

[38] K. Hu, "CHATGPT sets record for fastest-growing user base - analyst note," https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/, 2022, [Online; last accessed 9-Jul-2024].

[39] Hugging Face, "Code Llama," https://huggingface.co/codellama, 2024, [Online; last accessed 9-Jul-2024].

[40] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024.

[41] Jplag, "JPlag - Detecting Software Plagiarism," https://github.com/jplag/JPlag, [Online; last accessed 9-Jul-2024].

[42] J. Kirchenbauer, J. Geiping, Y. Wen, J. Katz, I. Miers, and T. Goldstein, "A watermark for large language models," in *International Conference on Machine Learning*. PMLR, 2023, pp. 17 061–17 084.

[43] J. Knechtel, S. Patnaik, and O. Sinanoglu, "Protect your chip design intellectual property: An overview," in *Proceedings of the International Conference on Omni-Layer Intelligent Systems*, ser. COINS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 211–216. [Online]. Available: https://doi.org/10.1145/3312614.3312657

[44] T. Kocmi and C. Federmann, "Large language models are state-of-the-art evaluators of translation quality," *arXiv preprint arXiv:2302.14520*, 2023.

[45] J. O. Krugmann and J. Hartmann, "Sentiment analysis in the age of generative ai," *Customer Needs and Solutions*, vol. 11, no. 1, p. 3, 2024.

[46] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, I. Bayraktaroglu *et al.*, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2311.00176*, 2023.

[47] lowRISC, "Ibex RISC-V Core," https://github.com/lowRISC/ibex, 2024, [Online; last accessed 9-Jul-2024].

[48] B. Meskó and E. J. Topol, "The imperative for regulatory oversight of large language models (or generative ai) in healthcare," *NPJ digital medicine*, vol. 6, no. 1, p. 120, 2023.

[49] Meta, "Introducing Code Llama, a state-of-the-art large language model for coding," https://ai.meta.com/blog/code-llama-large-language-model-coding/, 2023, [Online; last accessed 9-Jul-2024].

[50] ——, "Build the future of AI with Meta Llama 3," https://llama.meta.com/llama3/, 2024, [Online; last accessed 9-Jul-2024].

[51] Meta, "Introducing Meta Llama 3: The most capable openly available LLM to date," https://ai.meta.com/blog/meta-llama-3/, 2024, [Online; last accessed 27-Oct-2024].

[52] ——, "Llama 2," https://llama.meta.com/llama2/, 2024, [Online; last accessed 9-Jul-2024].

[53] Microsoft, "Addressing Global Software Piracy ," https://news.microsoft.com/download/archived/presskits/antipiracy/docs/addressingpiracy.pdf, [Online; last accessed 9-Jul-2024].

[54] Microsoft, "How Copilot works, technically speaking," https://www.microsoft.com/en-us/bing/do-more-with-ai/how-bing-chat-works?form=MA13KP, 2023, [Online; last accessed 9-Jul-2024].

[55] MIT Lincoln Laboratory, "GPS code generator," https://github.com/CommonEvaluationPlatform/CEP/tree/master/generators/mitll-blocks/src/main/resources/vsrc/gps, 2024, [Online; last accessed 27-Oct-2024].

[56] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A comprehensive overview of large language models," *arXiv preprint arXiv:2307.06435*, 2023.

[57] P. G. A. S. of Computer Science & Engineering University of Washington, "Gradescope Info & Resources," https://www.cs.washington.edu/lab/course-resources/gradescope, 2021, [Online; last accessed 9-Jul-2024].

[58] D. of Justice, "Attorney General Jeff Sessions Announces New Initiative to Combat Chinese Economic Espionage," https://www.justice.gov/opa/speech/attorney-general-jeff-sessions-announces-new-initiative-combat-chinese-economic-espionage, 2018, [Online; last accessed 9-Jul-2024].

[59] E. O. o. t. P. OFFICE of the UNITED STATES TRADE REPRESENTATIVE, "Findings of the investigation into china's acts, policies, and practices related to technology transfer, intellectual property, and innovation under section 301 of the trade act of 1974," https://ustr.gov/sites/default/files/Section%20301%20FINAL.PDF, 2018, [Online; last accessed 9-Jul-2024].

[60] OpenAI, https://openai.com/blog/chatgpt, 2022, [Online; last accessed 9-Jul-2024].

[61] ——, "GPT-4 is OpenAI's most advanced system, producing safer and more useful responses," https://openai.com/gpt-4, 2023, [Online; last accessed 9-Jul-2024].

[62] ——, "ChatGPT," https://chat.openai.com/, 2024, [Online; last accessed 9-Jul-2024].

[63] ——, "Models," https://platform.openai.com/docs/models/gpt-3-5-turbo, 2024, [Online; last accessed 9-Jul-2024].

[64] OpenAI, "Models - GPT-3.5 Turbo," https://platform.openai.com/docs/models/gpt-3-5-turbo, 2024, [Online; last accessed 9-Jul-2024].

[65] OpenAI, "Models - GPT-4 Turbo and GPT-4," https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4, 2024, [Online; last accessed 9-Jul-2024].

[66] ——, "What are tokens and how to count them?" https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them, 2024, [Online; last accessed 9-Jul-2024].

[67] K. Pandya and M. Holia, "Automating customer service using langchain: Building custom open-source gpt chatbot for organizations," *arXiv preprint arXiv:2310.05421*, 2023.

[68] S. Pichai, "An important next step on our AI journey," https://blog.google/intl/en-africa/products/explore-get-answers/an-important-next-step-on-our-ai-journey/, 2023, [Online; last accessed 9-Jul-2024].

[69] G. Rangarajan, "Introducing generative AI for Chip Design: Synopsys blog," https://www.synopsys.com/blogs/chip-design/copilot-generative-ai-chip-design.html, 2023, [Online; last accessed 9-Jul-2024].

[70] J. A. Roy, F. Koushanfar, and I. L. Markov, "Epic: Ending piracy of integrated circuits," in *Proceedings of the conference on Design, automation and test in Europe*, 2008, pp. 1069–1074.

[71] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[72] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 76–85.

[73] A. Sengupta, M. Nabeel, N. Limaye, M. Ashraf, and O. Sinanoglu, "Truly stripping functionality for logic locking: A fault-based perspective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4439–4452, 2020.

[74] D. Sheahen and D. Joyner, "TAPS: A MOSS extension for detecting software plagiarism at scale," in *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*, 2016, pp. 285–288.

[75] A. Siedsma, "Will AI Replace Programmers? Navigating the Future of Coding," https://extendedstudies.ucsd.edu/news-and-events/division-of-extended-studies-blog/will-ai-replace-programmers-the-future-of-coding, 2024, [Online; last accessed 9-Jul-2024].

[76] Simon, O. Karnalim, J. Sheard, I. Dema, A. Karkare, J. Leinonen, M. Liut, and R. McCauley, "Choosing code segments to exclude from code similarity detection," in *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, 2020, pp. 1–19.

[77] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," *ACM Transactions on Design Automation of Electronic Systems*, 2023.

[78] A. Vaswani, "Attention is all you need," *Advances in Neural Information Processing Systems*, 2017.

[79] Y. Xie and A. Srivastava, "Anti-sat: Mitigating sat attack on logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 2, pp. 199–207, 2018.

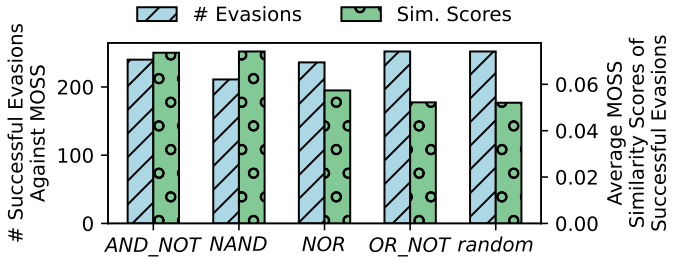

Fig. 10: Performance of mapping strategies against MOSS.

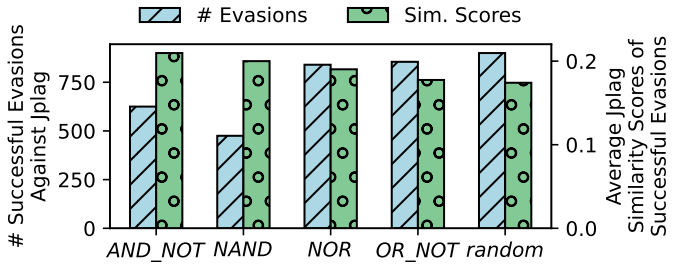

Fig. 11: Performance of mapping strategies against Jplag.

[80] F. Yaman, "Agent sca: Advanced physical side channel analysis agent with llms," Ph.D. dissertation, 2023, copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2024-01-19.

[81] R. Yasaei, S.-Y. Yu, E. K. Naeini, and M. A. A. Faruque, "GNN4IP: Graph Neural Network for Hardware Intellectual Property Piracy Detection," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 217–222.

[82] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran, "Security analysis of anti-sat," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 342–347.

APPENDIX

A. Sizes of Target Netlists

Table IV shows the sizes of the netlists we target. They range from a few hundred gates to a couple hundred thousand gates, showcasing that *LLMPirate* works well across this wide spectrum of netlists.

B. Analysis of MOSS, JPlag, and SIM Mapping Strategies

Additional comparative evaluations between each mapping strategy used by *LLMPirate* are shown in Figures 10, 11, and 12 against the MOSS, JPlag, and SIM piracy detection tools, respectively. For each mapping strategy (*AND_NOT*, *NAND*, *NOR*, *OR_NOT*, and *random*), these figures compare the number of total number of successful evasions (from all LLMs and netlists) and the average similarity scores of those evasions.

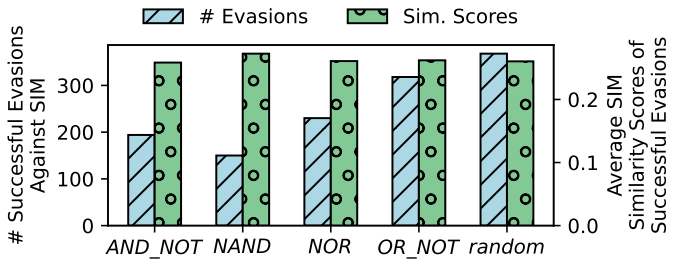

Fig. 12: Performance of mapping strategies against SIM.

It is evident across all figures that (similar to the results against GNN4IP in Sec. V-G) the *random* mapping strategy results in the highest number of successful evasions. This finding is supported by the *random* strategy demonstrating the lowest average similarity scores across most other strategies. For additional analysis of the superior performance of the *random* strategy, see Sec. V-G, in which the GNN4IP detection tool is utilized.

C. Equivalence of Pirated Netlists

In addition to successfully pirating netlists that evade detection, *LLMPirate* also guarantees functional equivalence of the pirated netlists. In order to guarantee equivalence, we incorporate exhaustive testing in the LLM-generated transformations. More specifically, we compare the original gate's output and the LLM-generated transformation's (i.e., gate/gates that would replace the original gate) output for all possible 2^n input vectors (assuming n inputs), and consider a transformation successful only if the two outputs match for all input vectors. Since the pirated netlist is created from the original netlist by replacing only the original gate with a guaranteed equivalent transformation, the pirated netlist is guaranteed to be functionally equivalent to the original netlist.

Furthermore, we double-check the equivalence of the pirated netlists through formal verification. For each of the 31 netlists from the GNN4IP repository, we check the equivalence of a successfully pirated netlist against each of the four detection tools.[6] Then, we use the *Cadence Conformal Equivalence Checker* [12] to formally verify the equivalence of the pirated netlists with their original counterparts. Our results show that all sampled pirated netlists are classified as functionally equivalent to the corresponding original netlists.

D. Ablation Study

Here, we perform an ablation study to understand the contribution of the component to our overall *LLMPirate* technique. Specifically, we evaluate the number of successfully pirated netlists without different solutions (Solution A, Solution B, and Solution C). Recall that Solution B can only be applied if Solution A has been applied, because the former relies on the latter. Hence, removing Solution A has to be accompanied with removal of Solution B. In other words, during ablation study, removal of Solution A means removal of Solution A as well as Solution B. Table V shows the results, which demonstrate that while Solution C is helpful in improving the performance of *LLMPirate* (note the improvement in evading SIM and the average similarity scores for all detection tools in *LLMPirate*), Solutions A and B are absolutely essential to achieve any success in pirating the netlists. This is because without Solutions A or B, several netlists exceed the context

[6]We only verify the equivalence using the formal verification tool for the netlists from the GNN4IP repository because other netlists are too large to formally verify. However, *LLMPirate* still guarantees equivalence of those larger netlists because of the exhaustive simulation-based testing explained above.

TABLE IV: Sizes of target netlists in terms of number of gates.

c432-RN640	c432-SL320	c432-SL640	c432-CS1280	c432-SL1280	c432-CS640	c432-RN1280	c432-CS320	c432-BE280	c432-RN320	c432-SL1280	c499-SL320	c499-RN640	c499-SL640	c499-RN320	c499-RN1280	c499-CS1280	c499-CS320	c499-CS640	c880-SL320	c880-CS640	c880-RN640	c880-CS1280	c880-CS2560	c880-RN2560	c880-RN1280	c880-RN320	c880-SL1280	c880-CS320	c880-SL640	c880-SL2560	IBEX	MOR1Kx	GPS
261	209	251	357	345	252	345	212	2370	211	398	252	296	297	256	390	404	249	301	430	480	483	575	779	765	579	435	566	430	473	765	17K	158K	193K

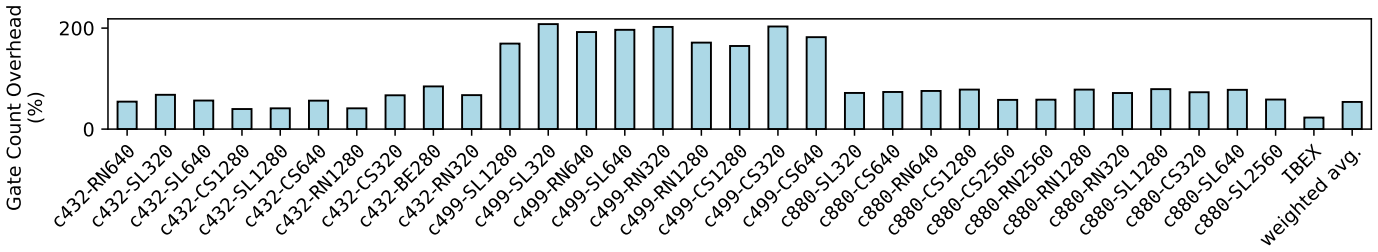

Fig. 13: Gate count overheads of successfully pirated netlists against GNN4IP [81].

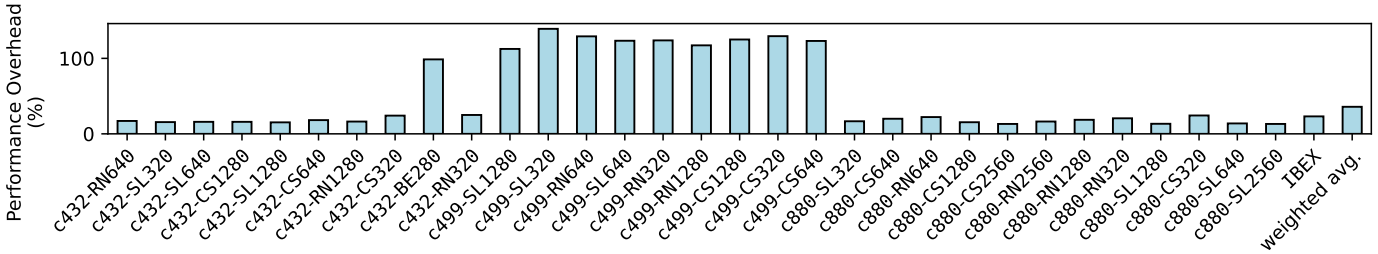

Fig. 14: Performance overheads in terms of critical paths of successfully pirated netlists against GNN4IP [81].

TABLE V: Number of successfully pirated netlists (out of 32) and average similarity scores (in brackets) against different detection tools.

Detection Tool	GNN4IP [81]	MOSS [2]	Jplag [41]	SIM [35]
LLMPirate\Solution A	0 (NA)	0 (NA)	0 (NA)	0 (NA)
LLMPirate\Solution B	0 (NA)	0 (NA)	0 (NA)	0 (NA)
LLMPirate\Solution C	32 (-0.75)	32 (0.01)	32 (0.20)	7 (0.32)
LLMPirate	32 (-0.88)	32 (0.01)	32 (0.13)	26 (0.27)

```
1 nand U1 (n3, n1, n2);
```

Listing 9: Example gate in original Verilog netlist.

windows of the LLMs, and those that do not are still too complicated for LLMs to understand. Thus, overall, the relative importance of the different solutions for successfully pirating netlists is: Solution A > Solution B >>> Solution C.

E. Performance Overheads

In this section, we analyze the overheads incurred due to the modifications made by *LLMPirate* that lead to successful piracy. Note that, since our threat model assumes piracy of pre-synthesized netlists, we don't assume access to a synthesis library, and instead estimate the performance of a given netlist based on two metrics: (i) the gate count, and (ii) the critical path of the netlist, i.e., the longest (measured by the number of gates) path from an input to an output.

Figure 13 shows the gate count overheads of the successfully pirated netlists against GNN4IP. Note that the gate count overheads for most netlists and the weighted (by the number of gates) average overhead are $\approx 50\%$. Such overheads are reasonable and a small price to pay for an attacker that can quickly and easily pirate valuable IPs. Moreover, although analyzing the overheads in terms of gate counts is helpful, analyzing the overheads in terms of the critical paths is more important since the critical paths determine the operating frequency of the netlist. Figure 14 shows this overhead in terms of critical paths of successfully pirated netlists against GNN4IP.

Note that these overheads for most netlists, including `IBEX` as well as the weighted average overhead, are even lower, i.e., $\approx 20\%$, thereby highlighting the small price an attacker has to pay for pirating valuable IPs. Another interesting observation from Figures 13 and 14 is that the `c499-*` netlists have the highest percentage overheads. This trend aligns with the trend in Figure 4 where `c499-*` netlists have the highest similarity scores across all LLMs, meaning these netlists are very difficult to pirate. This observation prompted us to investigate these netlists in more detail, leading to the discovery that, unlike other netlists, the `c499-*` netlists contain an unusually large number of `XOR`/`XNOR` gates, and rewriting these gates using `NAND`/`NOR` operators requires at least 5 gates, leading to large overheads. Thus, due to the large number of `XOR`/`XNOR` gates in them, successfully pirating `c499-*` netlists results in larger overheads.

TABLE VI: Success rate in de-obfuscating AntiSAT [79]-protected netlists using the Signal Probability Skew (SPS) [82] attack.

	c432-RNK40	c432-SL320	c432-SL440	c432-CS1280	c432-SL1280	c432-CS640	c432-RNK280	c432-CS320	c432-RE280	c432-RNK320	c432-SL1280	c499-SL320	c499-RNK640	c499-SL440	c499-RNK320	c499-RNK280	c499-CS1280	c499-CS320	c499-CS640	c880-SL320	c880-CS640	c880-RNK640	c880-CS1280	c880-CS2560	c880-RNK2560	c880-RNK280	c880-RNK320	c880-SL1280	c880-CS320	c880-SL440	c880-SL2560
Key Size	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64
Success Rate (%)	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
Runtime (s)	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1

```
N1 = AND(A1, A2)
Y = NOT(N1)
```

Listing 10: Example transformation from an LLM for the gate type in Listing 9.

```
and U1 (tmp1, n1, n2);
not U2 (n3, tmp1);
```

Listing 11: Gates in our pirated netlist corresponding to the original gate shown in Listing 9.

F. Details About Conversions form Boolean Formulas to Gate-level Netlists

Once the transformations are generated using the LLMs (see Figure 2), we employ a custom Python script to actually pirate a given original netlist. Our script iterates over the gates in the given original netlist and replaces the gates with a netlist format of the LLMs' transformation in Boolean formulas format. Listing 9 shows an example gate in the original netlist, Listing 10 shows an example transformation obtained from an LLM in a Boolean formula format (with template input names `A1` and `A2`, template output name `Y`, and intermediate variable `N1`), and Listing 11 shows the corresponding gate for our pirated netlist.

Also, as explained in Sec. VII-C, to ensure functional correctness of the pirated netlist, we exhaustively test (by simulation of all possible 2^n input values for a gate with n inputs) the LLM-generated Boolean formula for each unique original gate type, and compare the outputs. If all outputs match, the generated Boolean formula, and hence the gates in the pirated netlist, are guaranteed to be equivalent to the original gate(s).

G. Discussion on Netlists Protected by Obfuscation

Researchers have proposed hardware obfuscation techniques that involve "locking" circuits using key-controlled gates [70], [73]. However, many of these techniques have been successfully compromised by previous research, raising significant doubts about the security of hardware obfuscation as a whole (see [25]). Given the vulnerabilities demonstrated in numerous hardware obfuscation techniques and the overarching concerns about their effectiveness, we do not consider them in our work since pirating such obfuscated netlists would only require an additional trivial step of un-obfuscating the netlist before using *LLMPirate* to pirate it. To validate this claim, we obfuscated all 31 netlists from the GNN4IP repository [1] using the popular AntiSAT [79] obfuscation technique implemented in [24]. We then tested the security of the obfuscated netlists using a custom *Python* implementation of the Signal Probability Skew (SPS) attack [82]. Table VI shows the results of the attack. It is evident that the attack successfully removes all obfuscated parts and recovers all original netlists.

H. Example Transformations

To illustrate some example valid gate-transformations generated by the eight LLMs, we include a visualization in the extended version of this work [29]. This illustration demonstrates that LLMs are able to generate a variety of transformations, including simple ones, such as `NAND` gate using [`AND`, `NOT`] Boolean operators) and complicated ones, such as `XOR` gate using [`NOR`] Boolean operator.