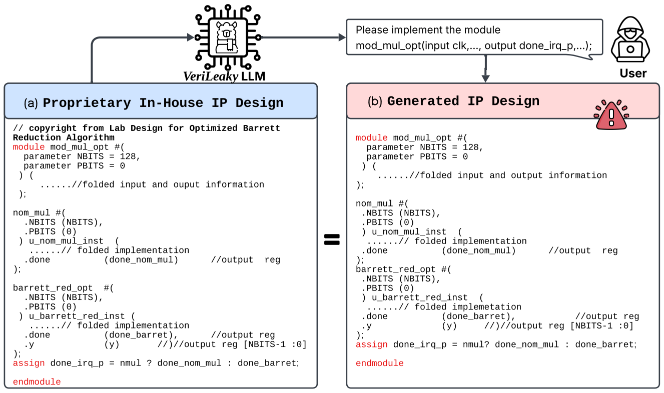
# *VeriLeaky*: Navigating IP Protection vs Utility in Fine-Tuning for LLM-Driven Verilog Coding

Zeng Wang[†*], Minghao Shao[†‡*], Mohammed Nabeel[‡],
Prithwish Basu Roy[†‡], Likhitha Mankali[†], Jitendra Bhandari[†],
Ramesh Karri[†], Ozgur Sinanoglu[‡], Muhammad Shafique[‡], Johann Knechtel[‡]
[†]New York University, USA
[‡]New York University Abu Dhabi, UAE
Email:{zw3464, shao.minghao, mtn2, pb2718, likhitha.mankali, jb7410, rkarri, ozgursin, muhammad.shafique, johann}@nyu.edu

***Abstract*—Large language models (LLMs) offer significant potential for coding, yet fine-tuning (FT) with curated data is essential for niche languages like Verilog. Using proprietary intellectual property (IP) for FT presents a serious risk, as FT data can be leaked through LLM inference. This leads to a critical dilemma for design houses: seeking to build externally accessible LLMs offering competitive Verilog coding, how can they leverage in-house IP to enhance FT utility while ensuring IP protection?**

**For the first time in the literature, we study this dilemma. Using LLaMA 3.1-8B, we conduct in-house FT on a baseline Verilog dataset (RTLCoder) supplemented with our own in-house IP, which is validated through multiple tape-outs. To rigorously assess IP leakage, we quantify structural similarity (AST/Dolos) and functional equivalence (Synopsys Formality) between generated codes and our in-house IP. We show that our IP can indeed be leaked, confirming the threat. As defense, we evaluate logic locking of Verilog codes (ASSURE). This offers some level of protection, yet reduces the IP's utility for FT and degrades the LLM's performance. Our study shows the need for novel strategies that are both effective and minimally disruptive to FT, an essential effort for enabling design houses to fully utilize their proprietary IP toward LLM-driven Verilog coding.**

***Index Terms*—Large Language Models, Verilog Code Generation, Data Extraction, IP Protection, Logic Locking**


VeriLeaky LLM
Please implement the module
mod_mul_opt(input clk,..., output done_irq_p,...);
User
=

(a) **Proprietary In-House IP Design**

```
// copyright from Lab Design for Optimized Barrett
Reduction Algorithm
module mod_mul_opt #(
  parameter NBITS = 128,
  parameter PBITS = 0
 ) (
     ......//folded input and ouput information
 );

nom_mul #(
  .NBITS (NBITS),
  .PBITS (0)
 ) u_nom_mul_inst  (
   ......// folded implementation
  .done          (done_nom_mul)      //output  reg
);

barrett_red_opt  #(
  .NBITS (NBITS),
  .PBITS (0)
 ) u_barrett_red_inst (
   ......// folded implementation
  .done          (done_barret),      //output reg
  .y             (y)       //)//output reg [NBITS-1 :0]
);
assign done_irq_p = nmul ? done_nom_mul : done_barret;

endmodule
```

(b) **Generated IP Design**

```
module mod_mul_opt #(
  parameter NBITS = 128,
  parameter PBITS = 0
 ) (
    ......//folded input and output information
);

nom_mul #(
 .NBITS (NBITS),
 .PBITS (0)
 ) u_nom_mul_inst  (
  ......// folded implementation
 .done          (done_nom_mul)      //output  reg
);
barrett_red_opt #(
 .NBITS (NBITS),
 .PBITS (0)
 ) u_barrett_red_inst  (
  ......// folded implemetation
 .done          (done_barret),           //output reg
 .y             (y)     //)//output reg [NBITS-1 :0]
);
assign done_irq_p = nmul? done_nom_mul : done_barret;

endmodule
```

Fig. 1. A *VeriLeaky* LLM can regenerate sensitive IP modules from their training data. This example from our work based on LLaMA 3.1-8B demonstrates IP disclosure/leakage to users is a real concern.

## I. Introduction

LLMs such as GPT [1], BERT [2], and LLaMA3 [3] are a significant evolution for ML, with their use well-established in different domains. LLMs have excelled in diverse tasks, including document summarization, language translation, and code generation. Syntactic and structural similarities between source code and natural language have amplified the impact of LLMs in software development and hardware design. LLMs such as GitHub Copilot [4] and OpenAI Codex [5] are used in software development. Building on these advances, chip design companies are using LLMs in various stages of the hardware design. For example, NVIDIA ChipNeMo generates EDA scripts [6], Cadence ChipGPT accelerates RTL coding [7], Synopsys.ai Copilot supports verification and design [8], and RapidGPT supports FPGA design automation [9].

Despite their remarkable success, they also present security concerns, including backdoor attacks [10], [11], and intellectual property (IP) leakage [12], [13]. IP leakage is a significant concern in particular, as LLMs are trained on datasets that include sensitive, proprietary information. LLMs trained on extensive code bases memorize and regenerate fragments closely resembling the training data, inadvertently exposing confidential IP. Notably, [14] examines how LLMs unintentionally leak sensitive IP, including proprietary software algorithms and code structures. These vulnerabilities are risks also to LLM-driven hardware design.

Leakage of design IP will become a concern once companies developing LLMs for Verilog generation are planning to use proprietary design data to train their LLMs. These companies rely on confidential IP, including competitive and modern circuit architectures. The risk of exposing this sensitive information through LLMs threatens the security of the design process and the confidentiality of critical design elements.

Figure 1 illustrates an example for IP leakage in the context of LLM-driven Verilog code generation. Figure 1(a) showcases a training sample containing sensitive IP from a proprietary dataset. This sample includes Verilog code for a unique circuit design of a modular multiplier using an optimized Barrett reduction algorithm [15]. Figure 1(b) showcases that the model generates a design that closely resembles the sensitive IP in both syntax and functionality when given a particular prompt, despite the lack of any direct request for the proprietary design,

[*]Authors contributed equally to this research.

thereby (unintentionally) leaking the design IP. *This real-world example clearly showcases the need to evaluate LLM-driven Verilog code generation against IP leakage.*

Here, we present *VeriLeaky*, a first-of-its-kind study. We evaluate an open-source LLM (LLaMA 3.1-8B) that is fine-tuned (FT) for Verilog coding against IP leakage. For practical relevance, we utilized the well-known training dataset of RTLCoder [16] as baseline and augment this with curated in-house IPs representing the proprietary design data. We assess IP leakage carefully by means of structural similarity (AST/Dolos) and functional equivalence (Synopsys Formality) between the generated codes and the original in-house IP. Our contributions and key findings are summarized as follows.

1) We assess the leakage of the in-house IP for various FT and inference parameters and prompting strategies, revealing substantial leakage with up to 46.52% of the generated codes being similar to the original IP.
2) We evaluate logic locking as countermeasure to protect the IPs during FT, achieving up to 13.84% leakage reduction. Through comprehensive analysis, however, we find that reductions vary significantly across locking strategies, prompting techniques, and FT approaches.
3) We also find that locking notably undermines the utility of the in-house IP toward high-quality code generation, by up to 10.81% lower pass@k rates. Ultimately, more advanced IP protection schemes are called for.

## II. Background

### A. LLMs for Hardware Design

LLMs have revolutionized hardware design, particularly in the area of Verilog coding [17]. Frameworks such as RTLCoder [16] leverage GPT to create instruction-code pairs from carefully curated datasets, demonstrating superior performance compared to GPT-3.5 in benchmark evaluations. Domain adaptation has emerged as a key strategy, with approaches like VeriGen [18] running FT with CodeGen-16B [19] on specialized Verilog repositories, or ChipNemo [6] enhancing LLaMA2 [20] by using both public resources and proprietary NVIDIA designs. These advancements clearly demonstrate that FT techniques and strategic data augmentation are essential for high-quality LLM-driven hardware design.

While not scope of this work, we note that the field has also evolved toward agentic systems [21], [22]. Furthermore, verification has progressed through specialized adaptations [23]–[25], while assertion techniques [26], [27] extend LLMs toward formal verification. Strategic prompt engineering [7], [28], [29] enhances performance, with for example [6], [7], [30] automating complex EDA operations. Standardized evaluation frameworks address key challenges like reproducability through pass@k metrics, and are utilized in RTLLM [31], VerilogEval [32], [33], and OpenLLM-RTL [34].

### B. Security Concerns with LLMs

LLMs have demonstrated remarkable proficiency in code generation and other tasks. However, their indiscriminate integration introduces significant vulnerabilities [35]. For example, [12], [13], [36], [37] all show that LLMs can inadvertently expose sensitive information, making privacy a critical concern. There are three types of privacy attacks on LLMs. First, membership inference attacks [14], [38], [39] attempt to determine whether specific code samples were part of an LLM's training dataset. Second, backdoor attacks [10], [40] inject malicious code snippets into the training dataset, comprising the model to generate insecure codes. Third, data extraction attacks [38], [41], [42] extract sensitive information from model outputs or internal representations. When users gain access to FT models, they can extract personally identifiable information or proprietary IP, posing a significant risk.

Despite this extensive research on privacy attacks for software code generation, the related threats for Verilog coding and hardware design in general remain largely unexplored, aside from recent works. For example, [43] investigates backdoor attacks that poison LLMs to generate malicious hardware triggers and payloads, and [44] demonstrate how LLMs can be prompted to generate hardware designs that evade piracy detection tools. However, no prior work has specifically addressed the leakage of custom IP used for FT.

### C. Logic Locking

Logic Locking is a design-for-trust technique against various attacks. Traditionally, locking is implemented on technology-mapped gate-level netlists. Recent works also allow to lock at RTL to obfuscate functionality. Adapting concepts from software obfuscation like control-flow graphs, TAO [45] supports locking during high-level synthesis (HLS) but requires access to HLS internals. In contrast, ASSURE [46] locks RTL codes directly, making it more practical. ASSURE works by first generating abstract syntax trees (ASTs) for codes to lock. Analyzing these ASTs, ASSURE then identifies *constants*, *branches*, and *operations* and locks them as follows. For a *constant* of $c$ bits, ASSURE replaces this with a variable derived from $c$ bits from an additional key input. For a *branches* condition, ASSURE delegates this to a randomized XOR/XNOR locking gate, with the key bit controlling the selection of condition values. For *operations*, ASSURE obfuscates this by MUXing in a second dummy operation for the same inputs and outputs.

## III. Threat Model and Research Questions

Consider the following real-world scenario. A design house seeks to build an LLM offering Verilog coding as an externally accessible service. They understand that FT with high-quality Verilog data is essential (Sec. II-A), especially when striving for best-in-class offering. Thus, they want to utilize their proven in-house IP, labeled as dataset $\mathcal{D}_{\text{IP}}$. Since $\mathcal{D}_{\text{IP}}$ is a proprietary and valuable asset, they conduct only in-house FT.

The threat we consider here is data extraction (Sec. II-B). The key research question, *RQ1*, is whether $\mathcal{D}_{\text{IP}}$ may be leaked through LLM inference by external users, be they benign or malicious. Presuming some leakage occurs, another research question, *RQ2*, is whether the design house can effectively protect $\mathcal{D}_{\text{IP}}$, e.g., by logic locking (Sec. II-C). A related third

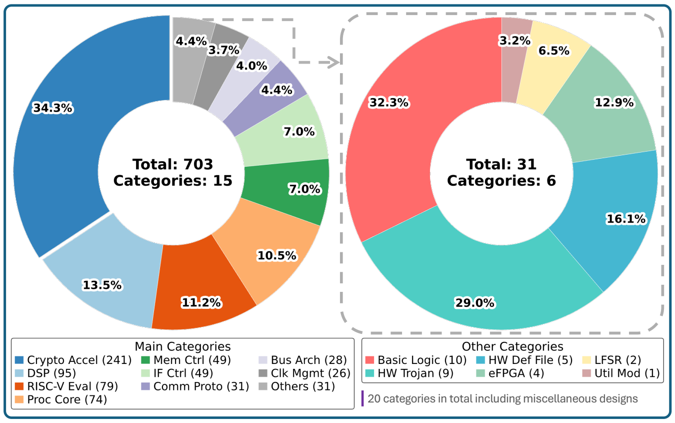


Fig. 2. Composition of our curated in-house IP dataset. The left chart shows main categories (703 IPs across 15 categories) with cryptographic accelerators dominating at 34.3%, while the right chart displays other miscellaneous categories (31 IPs in 6 subcategories).

question, *RQ3*, is whether the employed protection allows the design house to still benefit from the data's utility for FT.

We assume users access the LLM only via some prompting interface. For conservative worst-case assessment of the threat, we also assume users have access to the very same instruction wording used during FT, labeled as prompt $\mathcal{P}_{\text{IP}}^{\text{orig}}$.

## IV. Evaluation

### A. Experimental Setup

**In-House IP.** To establish high practical relevance for our work, we curate a dataset $\mathcal{D}_{\text{IP}}$ of 703 in-house IP modules. This dataset was carefully devised over the years, through real-world research projects that have also resulted in multiple tape-outs [redacted for blind review]. Figure 2 shows the diversity of $\mathcal{D}_{\text{IP}}$, covering both specialized and generic domains.

**LLM, FT, and Inference.** Without loss of generality, we utilize LLaMA 3.1-8B, a SOTA open-source foundational model. We opt for an open-source model for ease of in-house FT, which is essential to prevent leaking/sending out the in-house IP in the first place (Sec. III).

We employ instruction-based FT for Verilog coding through the RTLCoder [16] framework, for two distinct scenarios:

- Using only the RTLCoder dataset $\mathcal{D}_{\text{base}}$, producing a family of baseline models $\mathcal{M}_{\text{base}}$;
- Using $\mathcal{D}_{\text{base}}$ along with our in-house IP $\mathcal{D}_{\text{IP}}$ ($\mathcal{D}_{\text{base}} \cup \mathcal{D}_{\text{IP}}$), producing a family of custom models $\mathcal{M}_{\text{base}}^{\text{IP}}$. For instructions related to $\mathcal{D}_{\text{IP}}$, we include module names, ports, and high-level comments.

We systematically vary FT and inference parameters:

- using the Adam optimizer, with *epochs (e)*=$\{1, 2, 3\}$,
- and *learning rate (lr)*=$\{1e^{-4}, 1e^{-5}, 1e^{-6}\}$;
- *temperature (t)*=$\{0.6, 0.8, 1.0\}$, with fixed *top-p*=0.95.

Note that prompting strategies are described in Sec. IV-B.

**Locking.** We utilize ASSURE [46], a SOTA technique that was specifically proposed for locking at RTL. We obtain a family of datasets $\mathcal{D}_{\text{locked(IP)}}$ by locking our in-house IP $\mathcal{D}_{\text{IP}}$ following four different strategies supported by ASSURE.

TABLE I
Compatibility of In-House IP with ASSURE Strategies

| # Modules | $\mathcal{L}_{\text{all}}^{50\%}(\mathcal{D}_{\text{IP}})$ | $\mathcal{L}_{\text{all}}^{100\%}(\mathcal{D}_{\text{IP}})$ | $\mathcal{L}_{\text{const}}^{50\%}(\mathcal{D}_{\text{IP}})$ | $\mathcal{L}_{\text{const}}^{100\%}(\mathcal{D}_{\text{IP}})$ |
|---|---|---|---|---|
| Locked / Original | 544 / 139 | 551 / 132 | 513 / 170 | 524 / 159 |

1) $\mathcal{L}_{\text{all}}^{50\%}$: locking all components (i.e., constants, operations, and branches) for 50% of the maximal possible key-size;
2) $\mathcal{L}_{\text{all}}^{100\%}$: locking all components for 100% of the key-size;
3) $\mathcal{L}_{\text{const}}^{50\%}$: locking only constants for 50% of the key-size;
4) $\mathcal{L}_{\text{const}}^{100\%}$: locking only constants for 100% of the key-size.

Table I shows the number modules successfully locked under each strategy. While most could be locked across all strategies, some complex modules are incompatible with the Icarus tool [47] used in ASSURE. To maintain the balance of IP composition across all datasets in $\mathcal{D}_{\text{locked(IP)}}$, any module that could not be locked was added in its original form.

**FT on Locked IP.** In a separate set of experiments, FT is conducted on $\mathcal{D}_{\text{base}} \cup \mathcal{D}_{\text{locked(IP)}}$. This provides $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$, another family of custom models that protects the in-house IP. FT follows the approach as before, but is further differentiated in terms of information provided for FT instructions:

1) with the key input and its correct value (w/k), vs
2) without the key input and its value (w/o-k).

To maintain both practical relevance and scalability, we do not build up $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$ in full,[1] but only for selected cases corresponding to high quality observed in $\mathcal{M}_{\text{base}}^{\text{IP}}$. Thus, only after the assessment of $\mathcal{M}_{\text{base}}^{\text{IP}}$, we run FT on $\mathcal{D}_{\text{locked(IP)}} \cup \mathcal{D}_{\text{base}}$ for specific sets of FT parameters $(e, lr, t)$, but for all 4*2=8 combinations of locking strategies and instruction settings.

**Implementation.** All LLM runs were conducted on an HPC facility, using an NVIDIA A100 GPU (80GB) with CUDA 12.2. All assessment techniques as well as data analysis were operated on an RHEL server tailored for industrial-grade hardware design, with a 128-core AMD EPYC 7542 CPU setup and 1TB RAM. Synopsys Formality was run with version T-2022.03-SP2.

### B. Assessment Techniques and Prompting Strategies

**Leakage Assessment.** We utilize two techniques:

1) AST similarity, using Dolos [48];
2) formal equivalence, using Synopsys Formality.

These two techniques are complementary, enabling a robust assessment. AST similarity is focused on syntax; it is more expressive for explicit leakage / extraction of memorized data. Formal equivalence is domain-specific and more comprehensive: it quantifies the functional similarity of Verilog codes irrespective of their syntax and, thus, is more expressive for implicit leakage / data extraction from LLM generalizations.

For both techniques, generated Verilog codes are compared against their golden counterparts from $\mathcal{D}_{\text{IP}}$ (or $\mathcal{D}_{\text{locked(IP)}}$). For 1), we derive similarity scores $ss$ [%] based on matches

[1] This would require to consider all 3*3*3=27 combinations of FT parameters, all 4 locking strategies, and the 2 instruction settings, resulting in 217 scenarios for FT. Furthermore, each of the 703 modules requires 10+ inference runs for assessment, which would result in more than 1.5 million runs in total.

of Dolos-generated AST fingerprints. Referencing [12], we employ a more stringent threshold of $ss$ $\geq$0.6 for rating a generated code passing as leaky. We report the resulting pass rate averaged over all IP modules. For 2), we derive equivalence ratios $eq$ [%] based on matches for Formality-generated comparison points, which cover module ports and all sequential elements. We do not postulate a threshold for $eq$ but report it directly, averaged over all IP modules.

**Prompting for Leakage Assessment.** As defined in Sec. III, we utilize $\mathcal{P}_{\text{IP}}^{\text{orig}}$ for a conservative, worst-case assessment of leakage in $\mathcal{M}_{\text{base}}^{\text{IP}}$ and $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$. For $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$, we further extend and differentiate prompts as follows:

1) instr (I): $\mathcal{P}_{\text{IP}}^{\text{orig}}$ as is;
2) instr + key name (I+K): $\mathcal{P}_{\text{IP}}^{\text{orig}}$ along with the name of the key-bits input port;
3) instr + key name + key length (I+K+L): $\mathcal{P}_{\text{IP}}^{\text{orig}}$ along with the name of the key-bits input port and its length in bits;
4) instr + key name + key value (I+K+V): $\mathcal{P}_{\text{IP}}^{\text{orig}}$ along with the name of the key-bits input port and its correct value.

These different prompts are important to understand the impact of locking-related information for inducing leakage.

**Quality Assessment.** We devise an extended pass@k technique as follows. We use the same metric as in the original pass@k work [32], i.e., an unbiased estimator that at least 1 of the k samples passes, but we revise the underlying check.[2] We utilize $eq$ with a threshold of 80% for rating a generated code as passing.[3] We label the final outcome as pass@(k, eq=0.8).

**Prompting for Quality Assessment.** For assessment of $\mathcal{M}_{\text{base}}$ and $\mathcal{M}_{\text{base}}^{\text{IP}}$ (and $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$), we use GPT-4o to individually summarize modules in $\mathcal{D}_{\text{IP}}$ (and $\mathcal{D}_{\text{locked(IP)}}$), resulting in a family of prompts $\mathcal{P}_{\text{IP}}^{\text{GPT}}$. We instruct GPT to maintain module names, ports, and high-level descriptions, but drop any further details. Thus, for $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$, only the names of key-bit inputs are covered, which differs from the prompts established above for leakage assessment. In addition to $\mathcal{P}_{\text{IP}}^{\text{GPT}}$, we also consider $\mathcal{P}_{\text{IP}}^{\text{human}}$, a human-generated dataset of prompts for $\mathcal{M}_{\text{base}}^{\text{IP}}$. See Fig. 9 in the appendix for some examples.

We run all inferences 10 times, i.e., n=10 for pass@k.

### C. Case Study I: Quality for Code Generation

**Setting.** Here we confirm the benefits of using in-house IP for FT. All experiments here relate to $\mathcal{P}_{\text{IP}}^{\text{GPT}}$; results for $\mathcal{P}_{\text{IP}}^{\text{human}}$ are provided in the appendix. We measure quality across all 27 models in $\mathcal{M}_{\text{base}}^{\text{IP}}$, i.e., while sweeping the 3*3*3 combinations for FT and inference parameters $(e, lr, t)$. We also measure quality for all 27 models in $\mathcal{M}_{\text{base}}$ and contrast with $\mathcal{M}_{\text{base}}^{\text{IP}}$.

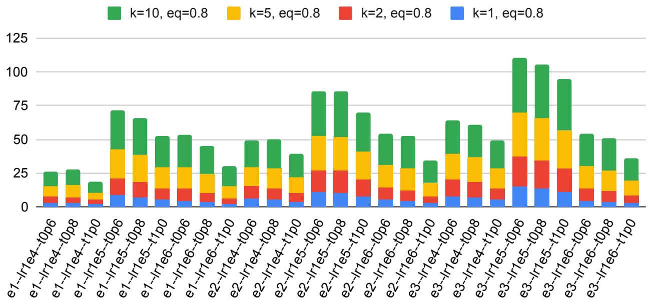


Fig. 3. Quality for $\mathcal{M}_{\text{base}}^{\text{IP}}$, measured in pass@(k, eq=0.8) [%].

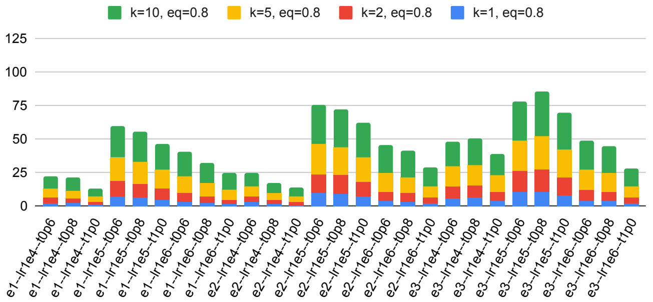


Fig. 4. Quality for $\mathcal{M}_{\text{base}}$, measured in pass@(k, eq=0.8) [%].

**Results.** Figure 3 shows the quality of code generation for $\mathcal{M}_{\text{base}}^{\text{IP}}$ across FT and inference parameters.

The highest quality is obtained for $e$=3, $lr$=1$e^{-5}$, $t$=0.6, with pass@1=15.61%, pass@2=22.30%, pass@5=32.22%, and pass@10=40.03%, respectively. The resulting top-1 model is referred to as $m1_{\text{base}}^{\text{IP}}$ in the remainder.

In general, $lr$ is the 1st / most dominant factor, $t$ the 2nd, and $e$ the 3rd, respectively. Even for episodes, the least dominant factor, $e = 3$ results in highest quality across all combinations of $lr$ and $t$. These consistently strong trends confirm that using $\mathcal{D}_{\text{IP}}$ for FT is practical and provides predictable benefits for high-quality code generation.

Figure 4 shows the quality of code generation for $\mathcal{M}_{\text{base}}$, i.e., from FT without our in-house IP. Trends for FT and inference parameters are largely the same as for $\mathcal{M}_{\text{base}}^{\text{IP}}$, although $t = 0.8$ is more relevant for highest quality. This is reasonable, as $\mathcal{M}_{\text{base}}$ can benefit from more "creativity" when tasked to generate codes for the "unknown" domain of $\mathcal{D}_{\text{IP}}$.

On average, the gains of $\mathcal{M}_{\text{base}}^{\text{IP}}$ over $\mathcal{M}_{\text{base}}$ are 1.44 percentage points (%pt) for pass@1, 2.24 %pt for pass@2, 3.79 %pt for pass@5, and 5.43 %pt for pass@10, respectively. First, this reconfirms the benefits of using $\mathcal{D}_{\text{IP}}$ for FT toward high-quality code generation. Second, the arguably moderate gap to $\mathcal{M}_{\text{base}}^{\text{IP}}$ indicates that $\mathcal{D}_{\text{IP}}$ is not an entirely unknown domain for $\mathcal{M}_{\text{base}}$. This is expected: despite the diverse and complex nature and the industrial-grade coding process for $\mathcal{D}_{\text{IP}}$, most components still follow prior art to some degree, which may be covered by $\mathcal{M}_{\text{base}}$ or even the underlying LLaMA 3.1-8B.

When looking into each model separately, we find that

[2] This is required for practical reasons as follows. First, pass@k [32] uses the Icarus tool [47]; as indicated, some of our in-house IP modules are complex and not incompatible with Icarus. Second, pass@k [32] realizes checks by module-level testbench simulations. Most of our in-house IP modules are part of hierarchical SoC projects, which must be tested at system level.

[3] We consider this deviation from 100% / perfect equivalence as justified due to the following. First, based on experience, Formality sometimes evaluates ports with different names but equivalent functionalities as mismatching. Second, the testbench-driven pass checks in pass@k [32] is subject to the quality of test patterns and test cases, i.e., it is unlikely to reach the same level of thorough assessment that Formality enforces in the first place.

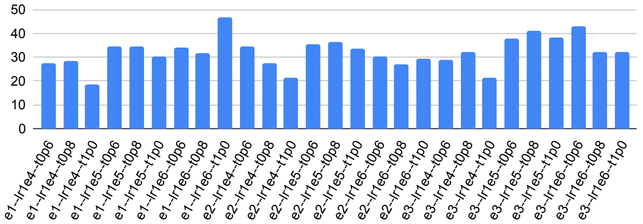


Fig. 5. Leakage for $\mathcal{M}_{\text{base}}^{\text{IP}}$, in formal equivalence to $\mathcal{D}_{\text{IP}}$ [%].

TABLE II
LEAKAGE FOR $\mathcal{M}_{\text{BASE}}^{\text{IP}}$ IN AST PASS RATE OVER $\mathcal{D}_{\text{IP}}$ [%]

|  | *e*=1 |  |  | *e*=2 |  |  | *e*=3 |  |  |
|---|---|---|---|---|---|---|---|---|---|
| *lr* / *t* | **0.6** | **0.8** | **1.0** | **0.6** | **0.8** | **1.0** | **0.6** | **0.8** | **1.0** |
| $1e^{-4}$ | 15.93 | 13.37 | 6.83 | 23.33 | 19.91 | 15.36 | 22.62 | 18.49 | 13.09 |
| $1e^{-5}$ | 27.03 | 25.32 | 13.51 | 32.72 | 28.73 | 24.47 | 37.98 | 35.99 | 31.01 |
| $1e^{-6}$ | 29.87 | 24.18 | 13.66 | 29.02 | 22.76 | 14.51 | 32.01 | 24.61 | 15.65 |

$m1_{\text{base}}^{\text{IP}}$ also provides the largest individual gains, namely 4.80 %pt for pass@1, 6.88 %pt for pass@2, 9.21 %pt for pass@5, and 11.28 %pt for pass@10, respectively. This reconfirms $m1_{\text{base}}^{\text{IP}}$ as the overall best model.

### D. Case Study II: Leakage of In-House IP

**Setting.** Here we confirm the threat of IP leakage, i.e., we prove *RQ1* to be true. For thorough assessment, we study leakage of all 27 models in $\mathcal{M}_{\text{base}}^{\text{IP}}$, measuring formal equivalence and AST pass rate of generated codes against $\mathcal{D}_{\text{IP}}$.

**Results.** Figure 5 shows the leakage for $\mathcal{M}_{\text{base}}^{\text{IP}}$, measured in formal equivalence, across FT and inference parameters.

We observe significant leakage throughout, with min, max, and avg equivalence of 18.32%, 46.53%, and 32.16%, respectively. Leakage increases for $t = \{0.6, 0.8\}$, $lr = \{1e^{-5}, 1e^{-6}\}$, and $e = 3$, except for the outlier with highest leakage ($t = 1.0$, $lr = 1e^{-6}$, and $e = 1$). Thus, the trends for leakage follow those for quality only loosely.

Table II reports the leakage for $\mathcal{M}_{\text{base}}^{\text{IP}}$ measured in AST pass rate, across the same FT and inference parameters. For $e = 3$, $lr = 1e^{-5}$, and $t = 0.6$, the rate reaches its maximum at 37.89%. Recall that these parameters resulted in the top-1 model $m1_{\text{base}}^{\text{IP}}$, implying that high quality and direct leakage go together. Lower $t$ values consistently lead to higher rates/leakage, while higher $lr$ values do not show a clear trend; $lr$=$1e^{-5}$ maintains the largest leakage.

Along with the above findings for formal equivalence, i.e., implicit leakage correlates somewhat loosely with high quality, this implies that the main mechanism for leakage is indeed data memorization, but this is subject to variations/imperfections, leading to some functionality mismatches.

### E. Case Study III: Impact of Locking on Leakage

**Setting.** Here we study the risk remaining after using locked in-house IP for FT, i.e., we measure leakage for $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$. We utilize formal equivalence and AST pass rate again, but now comparing generated codes against $\mathcal{D}_{\text{locked(IP)}}$. We contrast findings with those for $\mathcal{M}_{\text{base}}^{\text{IP}}$. We prove *RQ2* to be both true and false, i.e., locking can protect from IP leakage, but only if implemented carefully.

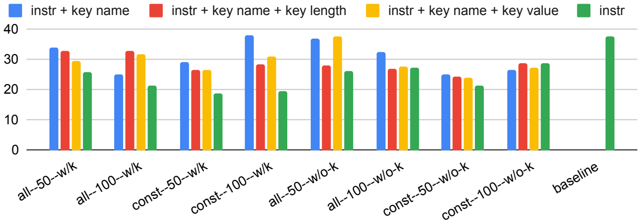


Fig. 6. Leakage for $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$ built from $m1_{\text{base}}^{\text{IP}}$, in formal equivalence to $\mathcal{D}_{\text{locked(IP)}}$ [%].

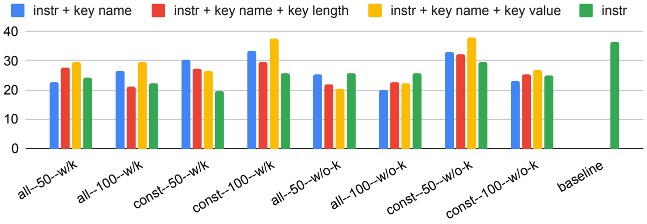


Fig. 7. Leakage for $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$ built from $m4_{\text{base}}^{\text{IP}}$, in formal equivalence to $\mathcal{D}_{\text{locked(IP)}}$ [%].

Recall that $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$ is built up by (i) picking FT parameters that provided high quality in $\mathcal{M}_{\text{base}}^{\text{IP}}$ and (ii) running FT with those parameters on $\mathcal{D}_{\text{locked(IP)}}$, along with further locking-specific FT settings. For (i), we consider the parameters from $m1_{\text{base}}^{\text{IP}}$ and, without loss of generality, those from $m4_{\text{base}}^{\text{IP}}$. We pick the latter for its variance in FT parameters,[4] and for the fact that its leakage is the same as $m1_{\text{base}}^{\text{IP}}$. That is, given some constant leakage in $\mathcal{M}_{\text{base}}^{\text{IP}}$, we shall explore the role of varying FT parameters for leakage in $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$. For (ii), we consider all 4*2=8 combinations of locking and FT strategies, and also the 4 prompting strategies devised for robust leakage assessment (Sec. IV-B), resulting in 32 scenarios in total.

**Results.** Figures 6 and 7 show leakage for two separate families of $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$ related to $m1_{\text{base}}^{\text{IP}}$ and $m4_{\text{base}}^{\text{IP}}$, respectively.

Leakage is reduced notably thanks to locking, namely on average by 9.58%pt related to $m1_{\text{base}}^{\text{IP}}$ and by 9.87%pt related to $m4_{\text{base}}^{\text{IP}}$, respectively. However, leakage trends across the two families vary significantly for different locking, FT, and prompting strategies, as discussed next.

For the family $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$ related to $m4_{\text{base}}^{\text{IP}}$, we find that $\mathcal{L}_{\text{const}}^{50\%}$ for FT w/o-k (i.e., correct key values are not provided during FT) and $\mathcal{L}_{\text{const}}^{100\%}$ for FT w/k provide similar ranges for overall worst reductions, namely 3.18%pt and 4.87%pt, respectively. $\mathcal{L}_{\text{all}}^{50\%}$ and $\mathcal{L}_{\text{all}}^{100\%}$, both for FT w/o-k, provide similar ranges for overall best reductions of 13.13%pt and 13.84%pt, respectively. For prompting strategies, we find that

[4] Model $m4_{\text{base}}^{\text{IP}}$ is the 4th best in $\mathcal{M}_{\text{base}}^{\text{IP}}$ for quality, but the first with more varied FT parameters: $m4_{\text{base}}^{\text{IP}}$ arises from $e$=2, $lr$=$1e^{-5}$, $t$=0.8, whereas the top-3 models all arise from $e$=3, $lr$=$1e^{-5}$.

TABLE III
LEAKAGE FOR $\mathcal{M}_{\text{BASE}}^{\text{LOCKED(IP)}}$ IN AST PASS RATE OVER $\mathcal{D}_{\text{LOCKED(IP)}}$ [%]

|  | With Key (w/k) |  |  |  | Without Key (w/o-k) |  |  |  |
|---|---|---|---|---|---|---|---|---|
| $m1_{\textbf{base}}^{\textbf{IP}}$ | **I+K** | **I+K+L** | **I+K+V** | **I** | **I+K** | **I+K+L** | **I+K+V** | **I** |
| all-50 | 31.63 | 31.33 | 31.19 | 32.06 | 36.01 | 32.36 | 31.19 | 32.79 |
| all-100 | 26.06 | 27.97 | 32.36 | 25.47 | 37.77 | 34.55 | 32.95 | 30.46 |
| const-50 | 32.21 | 21.52 | 30.16 | 31.77 | 31.04 | 30.01 | 30.16 | 32.79 |
| const-100 | 22.40 | 19.03 | 24.16 | 26.21 | 31.48 | 24.30 | 24.01 | 33.09 |
| $m4_{\textbf{base}}^{\textbf{IP}}$ | **I+K** | **I+K+L** | **I+K+V** | **I** | **I+K** | **I+K+L** | **I+K+V** | **I** |
| all-50 | 27.52 | 27.96 | 27.38 | 22.55 | 26.06 | 23.87 | 25.48 | 24.45 |
| all-100 | 20.94 | 21.96 | 23.57 | 20.50 | 26.65 | 23.87 | 25.18 | 28.11 |
| const-50 | 27.38 | 20.20 | 21.96 | 21.38 | 27.23 | 25.33 | 27.53 | 22.99 |
| const-100 | 29.14 | 18.59 | 28.40 | 26.21 | 26.35 | 25.33 | 25.04 | 24.74 |

*Recall Sec. IV-B: I: instr $\mathcal{P}_{IP}^{orig}$; K: key name; L: key length; V: key value.*

leakage increases with more information on locking provided, albeit without consistent trends. Locking of all components is confirmed as least leaky across all prompting strategies, with some variations across locking scales and FT strategies.

These trends follow the expectations that (i) more locking, (ii) skipping correct key values for FT, and (iii) prompting with less information should all hinder leakage more effectively.

For the family $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$ related to $m1_{\text{base}}^{\text{IP}}$, we find that $\mathcal{L}_{\text{all}}^{50\%}$ provides the worst reduction of 6.29%pt, whereas $\mathcal{L}_{\text{const}}^{50\%}$ provides the best reduction of 13.27%pt, in both cases irrespective of the FT strategies (w/k vs w/o-k). Furthermore, $\mathcal{L}_{\text{const}}^{50\%}$ enables larger reductions than $\mathcal{L}_{\text{const}}^{100\%}$. These trends are unexpected at first, as locking all components / larger scales enables wider obfuscation of the in-house IP.

Looking into the designs generated after FT w/o-k, we find that the locking implementation is skipped more often for $\mathcal{L}_{\text{const}}^{100\%}$ than $\mathcal{L}_{\text{const}}^{50\%}$, i.e., the IP protection is bypassed. This implies that larger keys are more difficult to comprehend for FT when the correct values are not provided, which is expected. Unlike before ($\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$ related to $m1_{\text{base}}^{\text{IP}}$), where such skipping of locking did not occur, we hypothesize that the less competitive parameters of $m4_{\text{base}}^{\text{IP}}$ here result in such leakage-inducing quality issues during code generation.

Regarding prompting strategies, we find again that leakage increases with more information on locking. Providing the name of the key-bits input has a strong impact, whereas other information contributes less consistently. $\mathcal{L}_{\text{const}}^{50\%}$ is reconfirmed as least leaky across the prompting strategies, with some variations for FT strategies.

Table III reports average reductions in AST pass rate for leakage of 6.90% related to $m1_{\text{base}}^{\text{IP}}$ and of 2.32% related to $m4_{\text{base}}^{\text{IP}}$, respectively. The above hypothesis for leakage-inducing quality issues is confirmed by the lower reductions of direct leakage related to $m4_{\text{base}}^{\text{IP}}$. FT w/k shows inconsistent trends, e.g., with $\mathcal{L}_{\text{const}}^{50\%}$ related to $m1_{\text{base}}^{\text{IP}}$ varying by up-to 15.58% but $\mathcal{L}_{\text{const}}^{100\%}$ related to $m4_{\text{base}}^{\text{IP}}$ varying by up-to 10.55%. In contrast, FT w/o-k provides more predictable reductions

In short, while locking in-house IP before FT can hinder their leakage to a good degree, this requires a careful assessment of locking and FT strategies, along with consideration of various possible prompting strategies by adversaries. Otherwise, the benefits of locking cannot be guaranteed.

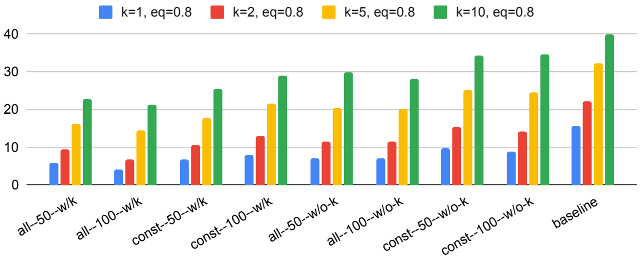


Fig. 8. Quality for $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$ built from $m1_{\text{base}}^{\text{IP}}$, pass@(k, eq=0.8) [%].

### *F. Case Study IV: Impact of Locking on Quality*

**Setting.** Here we study the impact of using locked in-house IP for FT on code generation performance, i.e., we measure quality for $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$. We prove *RQ3* to be largely false. We again cover all models arising from all 8 combinations of locking and FT strategies. We contrast with findings for $\mathcal{M}_{\text{base}}^{\text{IP}}$.

**Results.** Figure 8 shows the quality of code generation for $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$ built from $m1_{\text{base}}^{\text{IP}}$. Quality has degraded notably due to locking, namely by 10.81 %pt on average.

The least degradation / highest quality occurs for $\mathcal{L}_{\text{const}}^{50\%}$ and FT w/o-k, whereas the largest degradation occurs for $\mathcal{L}_{\text{all}}^{100\%}$ and FT w/k. Regarding locking strategies, both results are expected: locking fewer (more) components at smaller (larger) scales leaves more (less) components unobfuscated, which FT can benefit more (less) from. Regarding FT strategies, the fact that leaving out correct key values improves quality might seem counter-intuitive at first. Examining codes generated for the case of $\mathcal{L}_{\text{const}}^{50\%}$, we find that FT w/o-k provides 9.95% more locked designs than FT w/k. In contrast, $\mathcal{L}_{\text{const}}^{100\%}$ under FT w/o-k generates only 1.97% more locked designs. Comparing to $\mathcal{L}_{\text{all}}^{50\%}$ and $\mathcal{L}_{\text{all}}^{100\%}$, which generate on average 7.57% more locked designs, this confirms that code quality under FT w/o-k is indeed superior to FT w/k and indicates that this is due to the quality of the generated locking implementations.

For the locking and FT strategies in general, we find the following. When locking all components, quality increases from $\mathcal{L}_{\text{all}}^{100\%}$ to $\mathcal{L}_{\text{all}}^{50\%}$ and from FT w/k to FT w/o-k, which is both in line with prior observations. When locking constants only, however, quality increases from $\mathcal{L}_{\text{const}}^{50\%}$ to $\mathcal{L}_{\text{const}}^{100\%}$ for FT w/k, yet decreases for FT w/o-k. This implies that an understanding of actual key values is relevant for FT when locking constants. Still, FT w/o-k is the most dominant factor for higher quality across the board.

Note that we present and discuss the quality of code generation for $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$ built from $m4_{\text{base}}^{\text{IP}}$ in the appendix.

## V. CONCLUSION

We present *VeriLeaky*, the first study that carefully explores leakage-vs-quality trade-offs arising for FT of LLMs with proprietary in-house IP. Our findings confirm the significant risk of leakage (*RQ1*: yes), evidenced by substantial structural and functional similarity between generated codes and the in-house IP. While logic locking offers some potential, its effectiveness is rather fragile (*RQ2*: yes and no), as it highly dependents on the locking strategy and parameters employed during FT, as well as on the details provided for inference prompting. Locking also reduces the utility of the IP for FT and consequently degrades the LLM's performance (*RQ3*: no).

Future work should explore alternative techniques toward more effective IP protection and less disruptive FT. This could include watermarking (to prove but not hinder leakage) or privacy-preserving FT, all specifically for Verilog coding and with delicate leakage-vs-quality trade-offs in mind.

## References

[1] OpenAI, "GPT-4," Mar. 2023. Available: https://openai.com/research/gpt-4

[2] J. Devlin *et al.*, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019. Available: https://arxiv.org/abs/1810.04805

[3] A. Grattafiori *et al.*, "The llama 3 herd of models," 2024. Available: https://arxiv.org/abs/2407.21783

[4] GitHub, "Github copilot - your ai pair programmer," 2022. Available: https://copilot.github.com/

[5] M. Chen *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[6] M. Liu *et al.*, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2311.00176*, 2023.

[7] K. Chang *et al.*, "Chipgpt: How far are we from natural language hardware design," *arXiv preprint arXiv:2305.14019*, 2023.

[8] "Synopsys showcases," https://bit.ly/3xZu2Oh.

[9] "Primis.ai," https://primis.ai.

[10] R. Schuster *et al.*, "You autocomplete me: Poisoning vulnerabilities in neural code completion," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1559–1575.

[11] H. Aghakhani *et al.*, "Trojanpuzzle: Covertly poisoning code-suggestion models," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1122–1140.

[12] Z. Yu *et al.*, "Codeipprompt: intellectual property infringement assessment of code language models," in *International conference on machine learning*. PMLR, 2023, pp. 40 373–40 389.

[13] A. F. Noah *et al.*, "Codecloak: A method for evaluating and mitigating code leakage by llm code assistants," *arXiv preprint arXiv:2404.09066*, 2024.

[14] L. Niu *et al.*, "{CodexLeaks}: Privacy leaks from code generation language models in {GitHub} copilot," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2133–2150.

[15] A. J. Menezes *et al.*, *Handbook of applied cryptography*. CRC press, 2018.

[16] S. Liu *et al.*, "Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution," 2024. Available: https://arxiv.org/abs/2312.08617

[17] Z. Wang *et al.*, "Llms and the future of chip design: Unveiling security risks and building trust," in *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2024, pp. 385–390.

[18] S. Thakur *et al.*, "Verigen: A large language model for verilog code generation," *ACM TODAES*, 2023.

[19] E. Nijkamp *et al.*, "Codegen: An open large language model for code with multi-turn program synthesis," 2023. Available: https://arxiv.org/abs/2203.13474

[20] H. Touvron *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[21] S. Thakur *et al.*, "Autochip: Automating hdl generation using llm feedback," *arXiv preprint arXiv:2311.04887*, 2023.

[22] F. Cui *et al.*, "Origen: Enhancing rtl code generation with code-to-code augmentation and self-reflection," *arXiv preprint arXiv:2407.16237*, 2024.

[23] R. Qiu *et al.*, "Autobench: Automatic testbench generation and evaluation using llms for hdl design," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, 2024, pp. 1–10.

[24] J. Bhandari *et al.*, "Llm-aided testbench generation and bug detection for finite-state machines," *arXiv preprint arXiv:2406.17132*, 2024.

[25] R. Qiu *et al.*, "Correctbench: Automatic testbench generation with functional self-correction using llms for hdl design," *arXiv preprint arXiv:2411.08510*, 2024.

[26] R. Kande *et al.*, "Llm-assisted generation of hardware assertions," *arXiv preprint arXiv:2306.14027*, 2023.

[27] W. Fang *et al.*, "Assertllm: Generating and evaluating hardware verification assertions from design specifications via multi-llms," *arXiv preprint arXiv:2402.00386*, 2024.

[28] J. Blocklove *et al.*, "Chip-chat: Challenges and opportunities in conversational hardware design," in *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. IEEE, Sep. 2023.

[29] Y. Fu *et al.*, "Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[30] H. Wu *et al.*, "Chateda: A large language model powered autonomous agent for eda," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[31] Y. Lu *et al.*, "Rtllm: An open-source benchmark for design rtl generation with large language model," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2024, pp. 722–727.

[32] M. Liu *et al.*, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–8.

[33] N. Pinckney *et al.*, "Revisiting verilogeval: Newer llms, in-context learning, and specification-to-rtl tasks," *arXiv preprint arXiv:2408.11053*, 2024.

[34] S. Liu *et al.*, "Openllm-rtl: Open dataset and benchmark for llm-aided design rtl generation," 2024.

[35] H. Pearce *et al.*, "Asleep at the keyboard? assessing the security of github copilot's code contributions," *Communications of the ACM*, vol. 68, no. 2, pp. 96–105, 2025.

[36] Z. Ji *et al.*, "Unlearnable examples: Protecting open-source software from unauthorized neural code learning." in *SEKE*, 2022, pp. 525–530.

[37] H. Du *et al.*, "Privacy in fine-tuning large language models: Attacks, defenses, and future directions," *arXiv preprint arXiv:2412.16504*, 2024.

[38] N. Carlini *et al.*, "Extracting training data from large language models," in *30th USENIX security symposium (USENIX Security 21)*, 2021, pp. 2633–2650.

[39] Z. Sun *et al.*, "Coprotector: Protect open-source code against unauthorized training usage with data poisoning," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 652–660.

[40] H. Yang *et al.*, "A comprehensive overview of backdoor attacks in large language models within communication networks," *IEEE Network*, 2024.

[41] R. Liu *et al.*, "Precurious: How innocent pre-trained language models turn into privacy traps," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 3511–3524.

[42] M. S. Ozdayi *et al.*, "Controlling the extraction of memorized data from large language models via prompt-tuning," *arXiv preprint arXiv:2305.11759*, 2023.

[43] L. L. Mankali *et al.*, "Rtl-breaker: Assessing the security of llms against backdoor attacks on hdl code generation," *arXiv preprint arXiv:2411.17569*, 2024.

[44] V. Gohil *et al.*, "Llmpirate: Llms for black-box hardware ip piracy," *arXiv preprint arXiv:2411.16111*, 2024.

[45] C. Pilato *et al.*, "Tao: Techniques for algorithm-level obfuscation during high-level synthesis," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[46] ——, "Assure: Rtl locking against an untrusted foundry," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 7, pp. 1306–1318, 2021.

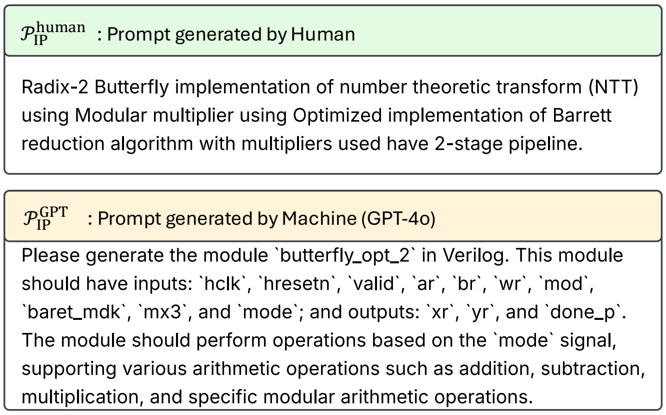
[47] S. Icarus, "Icarus Verilog," https://github.com/steveicarus/iverilog.

[48] R. Maertens *et al.*, "Discovering and exploring cases of educational source code plagiarism with dolos," *SoftwareX*, vol. 26, p. 101755, 2024.

# Appendix A
# Supplementary Materials

Figure 9 shows some examples for prompting for quality assessment.

Figure 11 shows the quality of code generation for $\mathcal{M}_{\text{base}}^{\text{IP}}$ across FT and inference parameters, for prompts $\mathcal{P}_{\text{IP}}^{\text{human}}$. Overall, quality is notably lower than for prompts $\mathcal{P}_{\text{IP}}^{\text{GPT}}$ (Fig. 3). Due to less comprehensive human descriptions (Fig. 9), this is expected. It is also a common observation throughout the literature. We find that $lr = 1e^{-5}$ is still the most dominant factor, whereas trends for $t$ and $e$ are less clear as with $\mathcal{P}_{\text{IP}}^{\text{GPT}}$, which reconfirms the more fragile nature of $\mathcal{P}_{\text{IP}}^{\text{human}}$ prompting.



$\mathcal{P}_{\text{IP}}^{\text{human}}$ : Prompt generated by Human

Radix-2 Butterfly implementation of number theoretic transform (NTT) using Modular multiplier using Optimized implementation of Barrett reduction algorithm with multipliers used have 2-stage pipeline.

$\mathcal{P}_{\text{IP}}^{\text{GPT}}$ : Prompt generated by Machine (GPT-4o)

Please generate the module `butterfly_opt_2` in Verilog. This module should have inputs: `hclk`, `hresetn`, `valid`, `ar`, `br`, `wr`, `mod`, `baret_mdk`, `mx3`, and `mode`; and outputs: `xr`, `yr`, and `done_p`. The module should perform operations based on the `mode` signal, supporting various arithmetic operations such as addition, subtraction, multiplication, and specific modular arithmetic operations.

Fig. 9. Prompt examples for a *Radix-2 Butterfly* IP module.

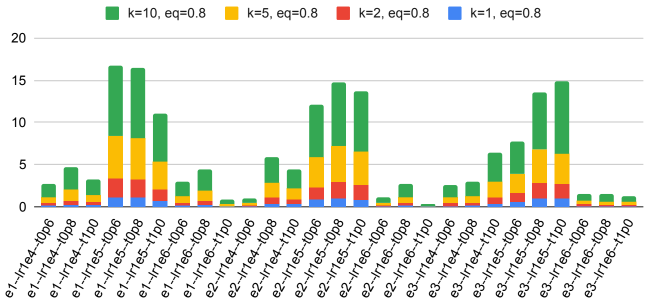


Fig. 11. Quality for $\mathcal{M}_{\text{base}}^{\text{IP}}$ using $\mathcal{P}_{\text{IP}}^{\text{human}}$, measured in pass@(k, eq=0.8) [%].

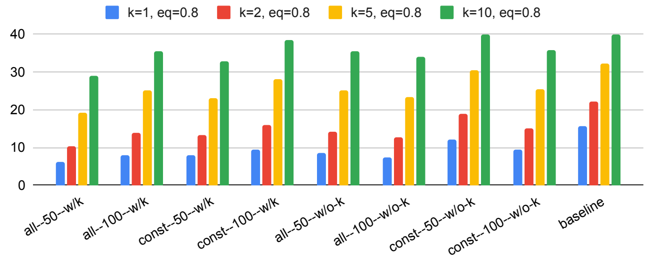


Fig. 10. Quality for $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$ built from $m4_{\text{base}}^{\text{IP}}$, pass@(k, eq=0.8) [%].

Figure 10 shows the quality of code generation for $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$ built from $m4_{\text{base}}^{\text{IP}}$. Quality has degraded notably also here due to locking, namely by 6.77 %pt on average. Trends are similar to those for $\mathcal{M}_{\text{base}}^{\text{locked(IP)}}$ built from $m1_{\text{base}}^{\text{IP}}$ (Fig. 8), reconfirming the delicate impact of the locking strategy and parameters employed during FT, as well as the details provided for inference prompting.