# LockForge: **Automating Paper-to-Code for Logic Locking with Multi-Agent Reasoning LLMs**

Akashdeep Saha[†], Zeng Wang[‡], Prithwish Basu Roy[‡], Johann Knechtel[†], Ozgur Sinanoglu[†], Ramesh Karri[‡]
[†]New York University Abu Dhabi
Abu Dhabi, UAE
[‡]New York University Tandon School of Engineering
New York, USA
{as19360,zw3464,pb2718,johann,ozgursin,rkarri}@nyu.edu

## Abstract

Despite rapid progress in logic locking (LL), reproducibility remains a challenge as codes are rarely made public. We present LockForge, a first-of-its-kind, multi-agent large language model (LLM) framework that turns LL descriptions in papers into executable and tested code. LockForge provides a carefully crafted pipeline realizing forethought, implementation, iterative refinement, and a multi-stage validation, all to systematically bridge the gap between prose and practice for complex LL schemes. For validation, we devise (i) an LLM-as-Judge stage with a scoring system considering behavioral checks, conceptual mechanisms, structural elements, and reproducibility on benchmarks, and (ii) an independent LLM-as-Examiner stage for ground-truth assessment. We apply LockForge to 10 seminal LL schemes, many of which lack reference implementations. Our evaluation on multiple SOTA LLMs, including ablation studies, reveals the significant complexity of the task. We show that an advanced reasoning model and a sophisticated, multi-stage framework like LockForge are required. We release all implementations and benchmarks, providing a reproducible and fair foundation for evaluation of further LL research.

## 1 Introduction

In a globalized IC supply chain, cost and time-to-market pressures come with risks. These include IP piracy, counterfeiting, malicious modifications, and reverse engineering. Logic locking (LL) addresses these risks by inserting key-dependent logic, enabling correct IC behavior only for the right key(s).

A major concern for (LL) research is the lack of reproducibility and independent evaluation. Figure 1 shows that public codes are sparse, mostly <10% of SOTA in the last five years.[1] Likewise, in most research domains, reference implementations are rare, algorithmic details are distributed across papers, key experimental settings are missing, and data/tool access is restricted or proprietary [19]. As a result, researchers have to painstakingly reconstruct methods from papers, slowing the community's progress and complicating independent validation. Concurrently, large language models (LLMs) have advanced in natural language processing, including coding. Agentic LLMs can extract structure and concepts from technical PDFs, generate non-trivial codes, and repair it through execution-guided debugging and verification [12, 22].

In a timely and first-of-its-kind effort, we present LockForge, a multi-agent LLM framework that converts LL papers into executable codes. The LockForge pipeline has four stages/phases as follows. Forethought ingests papers and extracts concepts. Implementation performs prompted code synthesis with runtime feedback by compilation and local execution, also producing small-scale locked circuits. Refinement conducts concept-to-code alignment via content mining, verifying that key mechanisms from the paper are realized in code, and runs smoke tests on the benchmark circuits. Validation employs independent agents to (i) probe conceptual fidelity and functional correctness, checking the code explanations against the paper overview, through a rigorous scoring system, and (ii) pose reference-grounded true/false queries for final assessment.

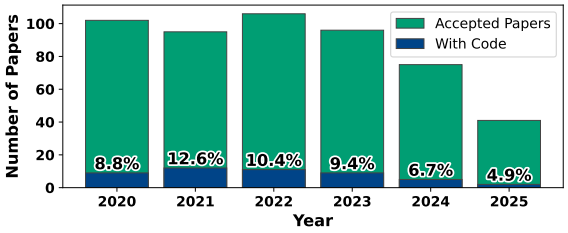


**Figure 1: Open source code for logic locking (2020–2025).**

We apply LockForge to 10 diverse LL schemes [1, 5–8, 11, 20, 24, 27, 28], spanning point-function schemes, SAT-hard schemes, FSM-gated schemes, and membership-logic schemes, most lacking public implementations. For end-to-end execution of generated codes on benchmarks, we observe consistent pass rates for simulation, equivalence checks, and our formal score criteria (which verify all claimed paper features). Ablations show that paper-grounded content mining/alignment, local execution, and independent examination all significantly contribute to fidelity and successful code execution. We release all LL implementations, locked benchmarks, and checklists to support reproducibility and standardized comparative evaluation. Our contributions can be summarized as follows:

- We propose LockForge, a multi-agent, multi-stage LLM workflow with role isolation for LL coding and evaluation. Key to LockForge is a formalized, objective, and reference-free validation of codes against papers through a systematic scoring system, coupled with agentic explanation alignment and paper-grounded true/false examination.
- We evaluate LockForge on 10 LL schemes, most lacking public implementations, producing executable and verified codes. We open-source all LL implementations, locked benchmarks, and checklists to enable standardized and reproducible evaluation for the community at [2].

[1]Obtained from https://openalex.org/ and https://github.com/

- We benchmark multiple SOTA LLM backbones and run ablations on key stages. A key insight is that tackling this paper-to-code challenge for complex LL schemes requires not only a sophisticated agentic framework but also a SOTA reasoning engine, all carefully orchestrated.

## 2 Background

### 2.1 Logic Locking

Early methods relied on random insertion of key gates for high corruption [13, 14, 16], offering limited security guarantees. The SAT attack [21] changed the goal of LL schemes from ad-hoc corruption to principled defenses, through SAT-hard structures [18], controllable point functions [20, 26, 27], etc. A post-SAT scheme SFLL-HD [27] introduced perturb-and-restore units. TriLock [28] allows tuning corruptibility while preserving resilience, DECOR [8] formalizes membership logic to permit multiple correct keys, SARO [1] and AutoLock [24] target scalability and automation, Entangle [6] couples perturb and restore networks. DK-Lock [11] and HARPOON [5] add counters or FSMs to decouple sequential activation from combinational behaviour. *Missing reference implementations for most schemes hinders independent evaluation.*

**Representative prompts used in** LockForge

**Forethoughts — LLM-A (PDF only)**
- Read the paper and explain the logic locking scheme in it briefly.
- Can you list the crucial concepts of the logic locking scheme in this paper?

**Implementation — LLM-A**
- Can you make a Python implementation of the logic locking and validate it using this input circuit? You should follow the paper's implementation exactly.

**Refinement Loop — LLM-A**
- Generate the YAML file for the BCSRP checklist for this logic locking scheme.
- Have you implemented all the concepts in your Python code?
- Validate your implementation with the BCSRP checklists.

**Validation — LLM-B and LLM-C (Code only)**
- Explain the logic locking scheme implemented in this code.
- There are two descriptions of the logic locking scheme. Are these two explanations similar? *(also ask LLM-A)*
- Use these BCSRP checklists in the YAML file to evaluate this implementation and score it (1 - 10).
- Generate 10 conceptual questions regarding the implementation with only true/false answers. **(prompt LLM-A)**
- Answer these questions with only T/F.

### 2.2 LLM-Assisted Coding

Early works on LLM-based coding targeted short snippets for isolated, toy programming tasks [4]. As long-context reasoning and code synthesis have improved, the scope has shifted toward more demanding, end-to-end problems, using multi-agent or role-based workflows that mirror real-world development pipelines [10]. General-purpose AI platforms like [3] support data analysis, PDF processing, and interactive notebooks. In hardware security, LLMs are being recently used [17, 23, 25]. *Ours is the first work to translate security schemes described in papers into codes for reproducibility.*

## 3 The LockForge Framework

### 3.1 Motivation

While a naïve, single-pass approach—uploading a paper and asking an LLM to '*write the code*', may succeed for simple schemes [16], we find in exploratory experiments that this quickly fails for contemporary LL works, due to the following challenges. First, there is a significant semantic gap between narrative exposition and implementation, often because algorithmic details are spread throughout the paper, implementation parameters/settings are missing, test data is unavailable, etc. Second, the context of LLMs is still limited when facing complex domains like LL [19]. While an LLM may readily compile some locked circuit, even with the correct key, it may not comprehend and implement the actual LL mechanisms.

To tackle these challenges, we devise LockForge, a self-contained, multi-agent, multi-stage framework which implements LL schemes of various complexity that have no public reference codes. LockForge comprises supervised stages/tasks, for validation, and assigns them to specialized LLM roles. This staged workflow improves grounding, allows feedback, and makes acceptance criteria explicit.

### 3.2 Multi-Stage Pipeline

Next, we describe the stages of LockForge; refer Fig. 2. Also see the list above for representative prompts.

**Forethoughts.** This stage transforms unstructured text as obtained from the paper into implementation-level abstractions. LLM-A ingests the PDF, produces a concise overview, enumerates core mechanisms, and derives artifact-level expectations. Thus, forethoughts distills a persuasive narrative into a structured plan by extracting, say, $N$ concepts of the paper that are precise enough to guide coding and checks.

**Implementation.** Next, LLM-A generates initial code, targeting benchmark formats and standard scripts. Simulation on small circuits are run to ensure executability and reveal immediate errors, yielding self-contained codes ready for refinements.

**Refinement Loop.** This stage first performs *content mining*, by instructing the same LLM-A to generate the so-called $\mathcal{B}, C, \mathcal{S}, \mathcal{R}, \mathcal{P}$ checklist for similarity scoring of the code against the paper.[2] To do so, we initialize LLM-A with (i) a definition of the checklist, (ii) the scoring rules and computation, and (iii) some templates for the actual checks. We instruct LLM-A to compile its findings into a simple YAML format. Based on the latter, LLM-A is also instructed to iteratively patch the code until the $N$ derived concepts are realized and, thus, the self-evaluation targets for $\mathcal{B}, C, \mathcal{S}$ are met (or until a feedback/iteration limit is reached). Second, this stage performs *local execution* on benchmarks to validate $\mathcal{R}$, including both simulation runs and equivalence checks. As before, feedback from this stage is returned to the same LLM-A for iterative revisions. Finally, if all $\mathcal{B}, C, \mathcal{S}, \mathcal{R}$ thresholds are met, the code passes the smoke test and advances to the validation phase.

**Validation of the Generated Code.** LockForge certifies that generated codes implement the key mechanism described in LL papers even when no reference implementations exist. We utilize role isolation—LLM-A reads the paper and performs iterative coding,

[2] $\mathcal{B}$ refers to behavioural checks, $C$ to conceptual mechanisms, $\mathcal{S}$ to structural elements, and $\mathcal{R}$ to reproducibility on benchmarks. Further, $\mathcal{P}$ is a non-compensatory penalty for critical omissions. See Sec. 3.3 for more details.

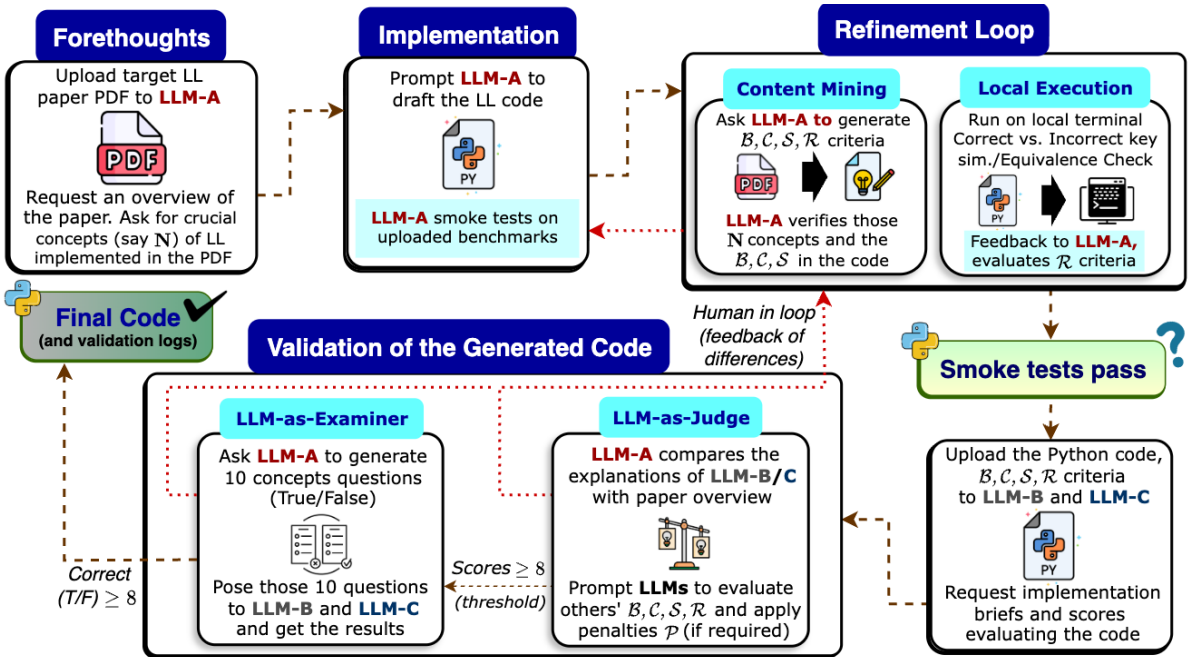


**Figure 2:** LockForge **is a multi-agent LLM workflow. It (i) parses a paper, (ii) drafts and refines code via concept mining and local execution/testing, and (iii) validates via independent LLMs and optional human assessment. LLM-A is a coder with PDF access; LLM-B/C are judges/examiners with no PDF access.**

while LLM-B and LLM-C analyze the code—along with similarity scoring and independent true/false examination, as follows.

First, LLM-B and LLM-C independently judge, by summarizing and scoring the code on the $\mathcal{B}, C, \mathcal{S}, \mathcal{R}, \mathcal{P}$ criteria, using the same YAML file compiled by LLM-A. The aggregated and weighted criteria yield the final Score (Sec. 3.3, Eq. 1). Only codes above an acceptance threshold, i.e., Score $\geq$ 8, advance. Second, to avoid false positives by overfitting to the derived criteria, an independent examination of paper-grounded true/false statements is conducted. For that, LLM-A derives a fixed set of statements from the paper, which LLM-B and LLM-C answer separately while inspecting only the code. Codes advance for #(correct true/false) $\geq$ 8. Finally, optional human-in-the-loop steps can audit the $\mathcal{B}, C, \mathcal{S}, \mathcal{R}, \mathcal{P}$ checklists against the paper to complement automated scoring and review anomalies, if any, like low judge/examiner pass rates.

**Final Code.** Passing validation proves that the paper's concepts are correctly realized and that the generated code executes as specified. For reproducibility and independent verification, we release all LL implementations along with the $\mathcal{B}, C, \mathcal{S}, \mathcal{R}, \mathcal{P}$ checklists [2].

**Summary.** LockForge engages multiple LLMs for different roles/-tasks, which are operated independently but orchestrated via agentic interactions. Content mining enables targeted code revisions, local execution runs help to cross-check the revisions and also uncover practical coding shortcomings like I/O issues, etc. For validation, the LLM-as-Judge role captures breadth (concepts, structure, behaviour), while the LLM-as-Examiner role cross-checks for any missing semantics from the paper.

## 3.3 Similarity Scoring

The similarity Score $\in [1, 10]$ is defined in Eq. 1. It is computed from the $\mathcal{B}, C, \mathcal{S}, \mathcal{R}, \mathcal{P}$ criteria using multi-criteria decision analysis (MCDA) [9].[3] As indicated, we consider four criteria: behavioural checks $\mathcal{B}$, conceptual mechanisms $C$, structural elements $\mathcal{S}$, and reproducibility $\mathcal{R}$. Details are provided next, and Tab. 1 provides the $\mathcal{B}, C, \mathcal{S}, \mathcal{R}, \mathcal{P}$ checklists with rationales for exemplary LL schemes.[4]

$$\text{Score}(x) = \max\Big\{1,\ \Big\lfloor 10\Big(0.4\,\mathcal{B}(x) + 0.3\,C(x) + 0.2\,\mathcal{S}(x) + 0.1\,\mathcal{R}(x)\Big)\Big\rfloor - \mathcal{P}(x)\Big\} \tag{1}$$

Using empirical insights and without loss of generality, we use weights of 0.4, 0.3, 0.2, and 0.1, respectively, for $\mathcal{B}, C, \mathcal{S}, \mathcal{R}$.

**Behavioural checks** $\mathcal{B}$ quantifies the ratio of verification checks satisfied, including correct-key equivalence to a golden model, wrong-key behaviour, and temporal/FSM semantics.

**Conceptual mechanisms** $C$ measures the paper-specific mechanisms realized in code. That is, $N$ core concepts are extracted from the paper, defining set $\mathcal{F}_{\text{req}}$, and their implementations are verified against the code ($\mathcal{F}_{\text{impl}}$).

$$C(x) = \frac{\left|\mathcal{F}_{\text{impl}}(x) \cap \mathcal{F}_{\text{req}}\right|}{\left|\mathcal{F}_{\text{req}}\right|}. \tag{2}$$

[3] MCDA is a family of methods that converts performance on multiple criteria into a single ranking by normalizing, weighting, and aggregating those criteria linearly. We apply a small non-compensatory penalty so that *hard failures* cannot be averaged away, realizing a veto-style safeguard [15].

[4] Checklists for all other LL schemes are provided in our release [2].

**Table 1: $\mathcal{B}, C, \mathcal{S}, \mathcal{P}$ checklists with brief descriptions for exemplary LL schemes**

| Scheme | $\mathcal{B}$ (Behaviour) | $C$ (Conceptual) | $\mathcal{S}$ (Structural) | $\mathcal{P}$ (Penalty; severity in brackets) |
|---|---|---|---|---|
| CAS-Lock[20] | **Equivalence after insertion:** with the correct key, locked = golden (all inputs). **Wrong-key corruption:** wrong key should yield output mismatches (some inputs). | **Primary inputs keyed by XOR/XNOR:** each tapped signal randomly XOR/XNORed with a key bit. **AND/OR cascade from keyed inputs:** keyed literals folded into 2-input serial cascades. **Cascade output drives internal nets:** trigger $Y$ gates/perturbs selected nets. **Cascade depth/fan-in match paper:** linear 2-input stages; intent depth $N-1$. | **Key-gate count equals key bits:** inserted gates match declared key size. **Cascade depth within expected range:** realized depth of $N-1$, comparable across chains. **Cascade output reaches outputs:** live path from $Y$ to at least one primary output. **No bypass under wrong key:** no alternate path restoring golden when $Y$=1. | **Cascade absent (3):** no cascades detected. **Cascade not connected (2):** trigger does not influence outputs. **Key-bit count mismatch (1):** declared key size ≠ number of key gates. |
| Auto-Lock [24] | **Equivalence after insertion:** with the correct key, locked = golden (all inputs). **Wrong-key corruption:** any incorrect key causes non-trivial deviation. **Search loop integrates with netlist:** iterations read/modify circuit to score/apply changes. | **Genotype defined:** candidate encodes edges/bits (e.g., $f_i, f_j, g_i, g_j$ with key). **Fitness mixes corruption/overhead/constraints:** robustness, cost, validity combined. **Iterative selection/mutation/crossover:** GA evolves population each generation. **Insertion operator places key gates:** deterministic routine rewires chosen edges. | **Inserted gate count matches budget:** equals requested key length. **Insertion points valid in graph:** nets exist, distinct, and keep graph acyclic/legal. **Fitness wired into search:** scores drive ranking and selection. **No bypass under wrong key:** only keyed path restores correct behaviour. | **Insertion operator absent (3):** no effective rewiring. **Fitness unused (2):** score computed but not applied (random search used). **Budget violation (1):** too many/few key gates relative to target. |

**Structural elements** $\mathcal{S}$ quantifies the share of essential structural elements present and correctly implemented, e.g., key gates, restore/perturb modules, comparators, and FSM states.

$$\mathcal{S}(x) = 1 - \frac{|\mathcal{M}(x)|}{|J|} \tag{3}$$

where $J$ is the set of structural components considered, and $\mathcal{M}(x) \subseteq J$ covers the implementation mismatches in $x$.

**Reproducibility** $\mathcal{R}$ measures fidelity of code execution on paper-specified benchmarks. $\mathcal{R}$ rates end-to-end successful runs, runs with minor code tweaks, and fails as 1.0, 0.5, 0.0.

**Penalty** $\mathcal{P}$ is the sum of severity grades for critical omissions, with each minor, major, and severe omission rated as 1, 2, and 3, respectively. Note that the categorization is based on the specifics for the LL scheme as derived by LLM-A.

## 4 Experimental Investigation

### 4.1 Setup

**LL Schemes and Benchmarks.** We apply LockForge to 10 LL schemes [1, 5–8, 11, 20, 24, 27, 28], covering different approaches and implementation styles for LL. We use benchmarks from these papers; ISCAS-85, ITC-99, and MCNC are common to many schemes. We utilize key-bit sizes as specified in these papers.

**Models and Roles.** We benchmark ChatGPT-5 Thinking, DeepSeek-V3, and Gemini-2.5pro in combinations of distinct LLM-A/B/C roles. An LLM may behave differently as coder vs evaluator and cross-model judging reduces single-model bias. Without loss of generality, we apply thresholds Score ≥ 8 and T/F ≥ 8, and a budget of 10 iterations for refinement.

**Metrics.** We report $\mathcal{B}, C, \mathcal{S}, \mathcal{R}, \mathcal{P}$, the derived Score, T/F accuracy, and PASS/INCORRECT/FAIL for local execution. PASS indicates that the generated code runs successfully and matches the expected outputs for correct vs wrong keys on at least 90% of benchmarks. INCORRECT indicates that the code runs but outputs deviate significantly for one or more key settings/benchmarks. FAIL indicates code failing to execute.

### 4.2 Results

**Overall Performance.** Shown in Fig. 3, ChatGPT-5 acting as coder achieves the highest similarity scores, best examination pass rate, and lower $\mathcal{P}$ (shown later), across all 10 LL schemes. In contrast,

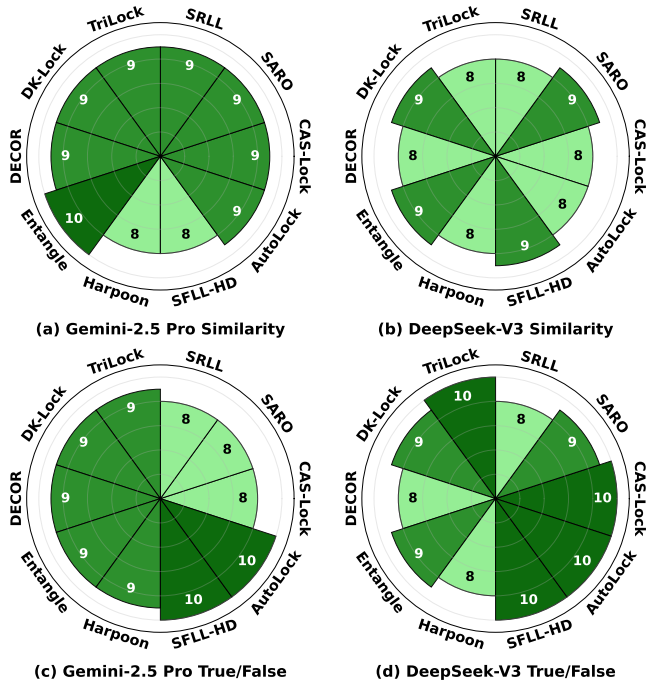


**Figure 3: Scoring ChatGPT-5 codes by Gemini-2.5pro and DeepSeek-V3.**

DeepSeek-V3 and Gemini-2.5pro acting as coders require more refinement and saturate earlier (also shown later). *Takeaways:* Thorough reasoning, enabled by the latest ChatGPT-5 model integrated into the agentic pipeline of LockForge, yields better code quality.

**Scoring Details for Cross-Model Evaluation.** Figure 4 shows the $\mathcal{B}, C, \mathcal{S}, \mathcal{R}, \mathcal{P}$ scoring for DeepSeek-V3 and Gemini-2.5pro generated codes as evaluated by ChatGPT-5. Vice versa, Fig. 5 shows scoring for ChatGPT-5 generated codes as evaluated by DeepSeek-V3 and Gemini-2.5pro. Again, ChatGPT-5 codes consistently show the highest scoring, with minimal $\mathcal{P}$, and are reproducible. *Takeaways:* The significant performance gap between coders highlights the task's complexity—success is not just about code generation, but about high-fidelity conceptual understanding, a trait only observed in the latest reasoning models. For fair evaluation, we do not use same models as coder, judge, and examiner; cross-model evaluation curbs any model-specific 'blind spots'.

**Coding Refinement.** Table 4 details coding outcomes for DeepSeek-V3, with ChatGPT-5 and Gemini-2.5 Pro each serving as judge and

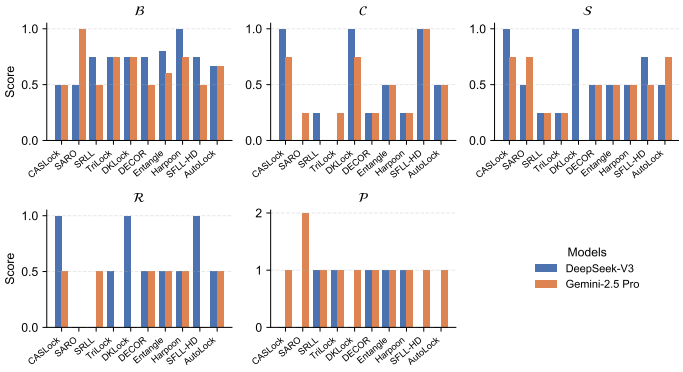


**Figure 4: Gemini-2.5pro, DeepSeek-V3 coding; evaluation by ChatGPT-5.**

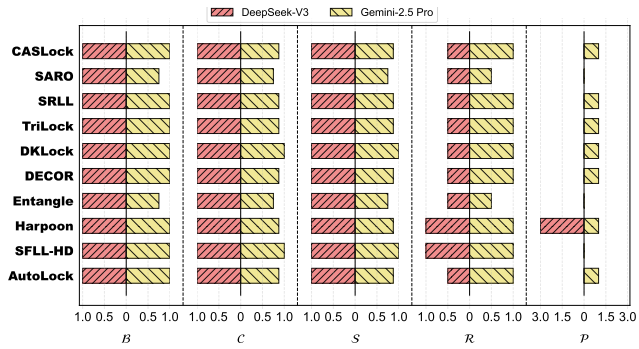


**Figure 5: ChatGPT-5 coding; evaluation by Gemini-2.5pro, DeepSeek-V3.**

examiner. Notably, only two implementations pass local execution. Even after iterative agentic feedback, various crucial concepts of most LL schemes were not realized by DeepSeek-V3. Likewise, Tab. 5 details coding outcomes for Gemini-2.5pro, where none of the implementations realize fully correct code before exhausting the 10-iteration limit. Refinement often saturates early, with no further gains observed for $\mathcal{B}, C, \mathcal{S}, \mathcal{R}$ scorings. In contrast, codes generated by ChatGPT-5 always pass local execution, with only minor implementation issues, if any. *Takeaways:* Refinement cannot compensate for concept omissions. Content mining and quality of the checklist, which informs the concepts to be implemented, are a first-order priority. Clearly, thorough reasoning is crucial to fully understand the semantics of the LL domain.

**Comparison with Reference Implementations.** Most LL schemes offer no publicly available implementations. For LL schemes with code released,[5] we compare them to LockForge using ChatGPT-5 coder in Fig. 6. *Takeaways:* LockForge performs on par with reference codes, confirming the faithful nature of generated artifacts.

**Comparison with Related Work.** ATLAS [3] and P2C [19] are commercial and academic offerings, respectively, for paper-to-code tasks. We followed their setup instructions to implement four LL schemes. As Tab. 2 shows, both underperformed. ATLAS often produced tangential code (misidentified mechanisms, pattern-matched

[5]DK-Lock at https://github.com/cars-lab-repo/DKL, SFLL-HD at https://github.com/circuitgraph/logiclocking, and CAS-Lock and AutoLock via [2].

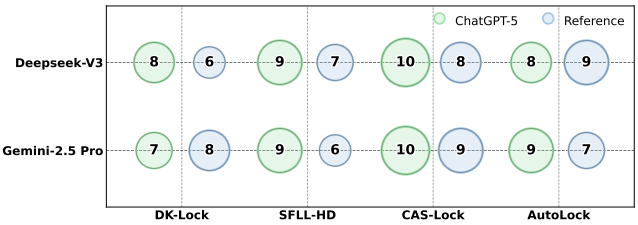


**Figure 6: Overall scores (computed via Eq. 1) comparing ChatGPT-5–generated code with available reference codes.**

**Table 2: ChatGPT-5 evaluation scores for related works.**

| Platform | SARO | AutoLock | Harpoon | DK-Lock |
|---|---|---|---|---|
| ATLAS [3] | 2/10 | 2/10 | 4/10 | 1/10 |
| P2C [19] | 3/10 | 7/10 | 3/10 | 4/10 |

scaffolds) and lacked hardware-awareness. P2C was paper-focused yet struggled with hardware-centric aspects, resulting in incomplete netlist integration. Notably, P2C handled the ML-based scheme AutoLock [24] relatively well, as expected due to its ML-focused pipeline [19]. *Takeaways:* The complex task of paper-to-code for LL requires hardware-aware reasoning and agentic feedback.

## 4.3 Ablation Studies

Using the best-performing ChatGPT-5 coder configuration, we conduct an ablation study to evaluate the individual contributions of the different stages in LockForge.

**Content Mining.** Here, we ablate the content mining process in which the paper-grounded checklist is extracted. We tailor local execution and judge/examiner LLMs to score only by (i) running codes and (ii) cross-checking against the paper without checklists. All other stages remain as is. As Tab. 3(left) shows, scheme-specific concepts and structures are missing in the implementation and related checks are failing. Since missions are not explicitly addressed early on, penalties increase. *Takeaways:* Content mining is essential for capturing paper-specific concepts, structures, and behaviour. Without content mining, reasoning reduces to weak proxies, even for ChatGPT-5. Subsequent evaluation and feedback cannot compensate for such foundational shortcomings.

**Local Execution.** Here, we ablate local execution, i.e., the behaviour of the generated code is only judged by static inspection. Thus, scoring and related agentic feedback acts without the $\mathcal{R}$ criteria. Once this ablated pipeline of LockForge finishes, we run the codes as before, and we use $\mathcal{R}$ again for final scoring. Results are shown in Tab. 3(right). *Takeaways:* Codes may be structurally sound, i.e., pass all static checks, yet fail in practice. Local execution exposes related issues and instructs the coder LLM for revisions. Compared to content mining, local execution is less impactful.

**Independent Examination.** Here, we ablate the LLM-as-Examiner process. Thus, final codes are only required to pass against the checklist. Exemplary results are provided in Tab. VI. While the ablated code generation passed (i.e., scored $\geq 8$), some key semantics are missing. Re-introducing the examiner flagged all these cases and prompted revisions of the checklists. *Takeaways:* Examination

**Table 3: Ablation results. Scoring by Gemini-2.5pro and DeepSeek-V3 on ChatGPT-5 generated codes.**

| LL Scheme | Content Mining Ablation | | | Local Execution Ablation | | |
|---|---|---|---|---|---|---|
| | Gemini | DeepSeek | Common Remarks | Gemini | DeepSeek | Common Remarks |
| **CAS-Lock** [20] | 2 | 2 | missing dual cascades, complementary patterns single cascade only; fixed AND-only cascade | 7 | 7 | Output misalignment (_LOCKED_wire/name) reuse; bench-specific; fixed key for all |
| **DECOR** [8] | 2 | 3 | Single correct key; DECOR structural transform; `is_correct` logic; output correction mechanism | 5 | 6 | Local execution step absent; conceptual-only check; reproducibility unstable; several benches show mismatches |
| **DK-Lock** [11] | 4 | 3 | M-cycle activation requirement; sequential counter; simple XOR locking; output clamping; M-cycle activation counter | 6 | 7 | Keys auto-padded/truncated, hence, mismatches despite "correct" JSON; output reordering can flip results |
| **TriLock** [28] | 3 | 4 | Structural obfuscation; EF trigger; no ES/EF mechanisms; no state obfuscation; no $k_s/k_f$ separation | 5 | 6 | Key pin order not exported, hence, "correct key" mismatches |
| **SRLL** [7] | 5 | 6 | No CS calculation; no alternative blocks; no combinational-cycle insertion; no LUT-based withholding with TT keys | 6 | 6 | Local run inconsistent per-bench (some fail); key format mismatch (JSON expected); script prints CSV, parser fail |
| **Entangle** [6] | 4 | 4 | No signal entanglement; no dummy logic/cones; faulty restore unit; external-signal entanglement | 5 | 7 | Sim accepted incorrect/correct keys across random inputs; no golden/locked equivalence check done |
| **SARO** [1] | 2 | 2 | Rename-then-wrap; avoids cycling; no correct key identity guarantee; XOR corruption | 3 | 4 | Not generic to all bench formats; local run skipped |
| **Harpoon** [5] | 4 | 6 | Missing sequential entrance lock; no clear obfuscate/authenticate phases; no state-conditioned corrupt/restore; no reuse of unreachable states | 5 | 7 | Local execution failed due to parsing; reproducibility not evaluated across benches |
| **AutoLock** [24] | 3 | 2 | Attack-in-loop fitness; cycle, conflict detection & repair; genotype with per-gene embedding | 5 | 4 | Correct key mismatch from key-order mismatch; no wrong-key sims or miter checks |
| **SFLL-HD** [27] | 5 | 6 | No stripped functionality; incorrect restore unit | 7 | 6 | Correct key check sample-based only (no miter); occasional small mismatches possible |

**Table 4: DeepSeek-V3 codes; Gemini-2.5pro, ChatGPT-5 evaluate. Scores are binned. Generated remarks cross-verified.**

| Logic Locking Schemes | Similarity Check (1–10) | | Local Simulation? | Common remarks from Gemini-2.5 and ChatGPT-5 |
|---|---|---|---|---|
| | Gemini-2.5 | ChatGPT-5 | | |
| **CAS-Lock** [20] | 8 | 8 | ✓ PASS | Initial cascade wrong; same lock gates for all inputs. Corrected with feedback. |
| **SARO** [1] | 2 | 3 | ✗ FAIL | Concepts not implemented. No Rename-then-Wrap. Incorrect T3 transform implementation. |
| **SRLL** [7] | 3 | 3 | ✗ FAIL | Wrong implementation; No LUT-based withholding w/ truth-table keys, key-controlled entanglement with cone-external signals. Random XOR/XNOR LL. |
| **TriLock** [28] | 5 | 3 | △ INCORR | Concepts not implemented. No temporal ES/phase gating; no k-deep/EF modeling. |
| **DK-Lock** [11] | 9 | 9 | ✓ PASS | Most crucial concepts implemented. |
| **DECOR** [8] | 4 | 4 | △ INCORR | inserts XOR/XNOR key-gates; does not implement UDC/cofactor altering; no is_correct+G/L correction logic. |
| **Entangle** [6] | 5 | 5 | △ INCORR | Incorrect control logic and restore unit implementation. |
| **Harpoon** [5] | 6 | 5 | △ INCORR | No state-space expansion implemented. |
| **SFLL-HD** [27] | 8 | 8 | △ INCORR | FSC built via truth-table enumeration. Simulation fails on large circuits. |
| **AutoLock** [24] | 6 | 6 | △ INCORR | wrong fitness function implementation. Locking insertion flawed. |

**Table 5: Gemini-2.5pro: codes; DeepSeek-V3, ChatGPT-5: evaluate. Scores are binned. Generated remarks cross-verified.**

| Logic Locking Schemes | Similarity Check (1–10) | | Local Simulation? | Common remarks from DeepSeek-V3 and ChatGPT-5 |
|---|---|---|---|---|
| | DeepSeek-V3 | ChatGPT-5 | | |
| **CAS-Lock** [20] | 6 | 5 | △ INCORR | Self-loop/combinational cycle introduced when rewiring. Cascading of AND-OR not present initially – corrected with iterative feedback. |
| **SARO** [1] | 4 | 4 | ✗ FAIL | No explicit partition TT transforms. |
| **SRLL** [7] | 3 | 2 | △ INCORR | Implements random XOR/XNOR locking with key inputs. |
| **TriLock** [28] | 4 | 3 | ✗ FAIL | EF and thus ESF = ES ∨ EF. Mod-$k$/phase gating not implemented. |
| **DK-Lock** [11] | 5 | 4 | ✗ FAIL | Activation not enforced for $m$ cycles. No global enable gating counter by full key match each cycle. |
| **DECOR** [8] | 4 | 3 | △ INCORR | No `is_correct`/membership logic to promote keys as correct. |
| **Entangle** [6] | 5 | 4 | △ INCORR | Wrong implementation of perturb unit. |
| **Harpoon** [5] | 6 | 4 | △ INCORR | No separate obfuscation/authentication modes; just a linear key-sequence gate with reset-on-mismatch. |
| **SFLL-HD** [27] | 3 | 5 | △ INCORR | Not SFLL-HD implementation. Incorrect FSC construction. |
| **AutoLock** [24] | 6 | 5 | △ INCORR | Incorrect genotype/candidate definition and fitness function. |

**Table 6: Examination ablation induces shortcomings for ChatGPT-5 coding. Initial criteria lists those from $\mathcal{B}, \mathcal{C}, \mathcal{S}, \mathcal{R}, \mathcal{P}$ flagged.**

| LL Scheme | Initial Criteria | Some T/F questions that had failed |
|---|---|---|
| **SFLL-HD** | **B:** equivalence; restore cancels flip; wrong-key perturbation<br>**S:** restore reaches outputs; no bypass with wrong key | In the *stripped* design, outputs differ from golden iff HD(input,key)= $h$.<br>The HD comparator implements equality (=$h$), not a threshold.<br>The HD checker is fully wired to designated input subsets and corresponding $k$ key bits. |
| **TriLock** | **S:** FSM state count; ES drives phase mux/gate-enables on the corruption/ restore path | The required input sequence must appear contiguously to trigger ES.<br>On any symbol mismatch, the FSM resets to the start state on the next cycle.<br>A phase-gating mux enforces mutual exclusivity between ES and the restore/safe path. |

provides paper-anchored semantic checks exposing scoring 'blind spots'. These include issues with sequence contiguity, reset semantics, and predicate equality vs. threshold. Examination acts as an independent overfitting guard for alignment with the LL schemes. While essential for faithful code generation, its benefits vary across LL schemes with different underlying concepts and complexity.

## 5 Conclusion

LockForge is a reference-free framework that turns, for the first time, LL papers into executable code. Addressing this challenge requires more than a simple coding assistant, given the significant semantic gap and domain-specific complexity of LL schemes. Key stages of LockForge are paper-grounded concept extraction and scoring, local execution, and independent final examination. Ablation studies show that each stage is crucial for guiding LLMs

through this complex task. LockForge 's cross-model evaluation improves fidelity of the codes and mitigates potential model-specific bias. LockForge yields faithful artifacts on ten seminal schemes, spanning a diverse range of LL techniques and security philosophies. All generated codes and results all released at [2]. Our results show that only recent reasoning models such as ChatGPT-5 reliably meet the thresholds for this complex task, underscoring the timeliness of our work. Building on this milestone, future work will extend LockForge to more schemes and incorporate established LL attacks to further validate resilience of SOTA LL schemes.

## References


[1] Abdulrahman Alaql, Siqiao Li, Yuanqi Shen, and Hai Zhou. 2021. SARO: Scalable Attack-Resistant Logic Locking. *IEEE Transactions on Information Forensics and Security* 16 (2021), 3724–3739. doi:10.1109/TIFS.2021.3092135

[2] Anonymous. 2025. LockForge (Anonymous Artifact). https://github.com/codesanonymousgit-sudo/LockForge.git.

[3] Atlas Research: AI-Powered Research Platform. 2025. https://atlas-research.io/.

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[5] Rajat Subhra Chakraborty and Swarup Bhunia. 2009. HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 10 (2009), 1493–1502. doi:10.1109/TCAD.2009.2028166

[6] Armin Darjani, Nima Kavand, Shubham Rai, Mark Wijtvliet, and Akash Kumar. 2022. ENTANGLE: An Enhanced Logic-Locking Technique for Thwarting SAT and Structural Attacks. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI '22)*. 107–112. doi:10.1145/3526241.3530371

[7] Mona Hashemi, Siamak Mohammadi, and Trevor E. Carlson. 2025. SRLL: Improving Security and Reliability with User-Defined Constraint-Aware Logic Locking. *ACM Journal on Emerging Technologies in Computing Systems* (2025). doi:10.1145/3709139

[8] Yinghua Hu, Kaixin Yang, Subhajit Dutta Chowdhury, and Pierluigi Nuzzo. 2024. DECOR: Enhancing Logic Locking Against Machine Learning-Based Attacks. In *2024 25th International Symposium on Quality Electronic Design (ISQED)*. 1–8. doi:10.1109/ISQED60706.2024.10528687

[9] Ching-Lai Hwang and Kwangsun Yoon. 1981. Methods for multiple attribute decision making. In *Multiple attribute decision making: methods and applications a state-of-the-art survey*. Springer, 58–191.

[10] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974* (2024).

[11] Jordan Maynard and Amin Rezaei. 2023. DK Lock: Dual Key Logic Locking Against Oracle-Guided Attacks. In *2023 24th International Symposium on Quality Electronic Design (ISQED)*. 1–7. doi:10.1109/ISQED57927.2023.10129368

[12] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[13] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. 2012. Security analysis of logic obfuscation. In *Proceedings of the 49th Annual Design Automation Conference*. Association for Computing Machinery, 83–89. doi:10.1145/2228360.2228377

[14] Jeyavijayan Rajendran, Huan Zhang, Chi Zhang, Garrett S. Rose, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. 2015. Fault Analysis-Based Logic Encryption. *IEEE Trans. Comput.* 64, 2 (2015), 410–424. doi:10.1109/TC.2013.193

[15] Bernard Roy. 1991. The outranking approach and the foundations of ELECTRE methods. *Theory and decision* 31, 1 (1991), 49–73.

[16] Jarrod A Roy, Farinaz Koushanfar, and Igor L Markov. 2008. EPIC: Ending piracy of integrated circuits. In *Proceedings of the conference on Design, automation and test in Europe*. 1069–1074.

[17] Akashdeep Saha, Prithwish Basu Roy, Johann Knechtel, Ramesh Karri, Ozgur Sinanoglu, and Lilas Alrahis. 2025. GLLaMoR: Graph-based Logic Locking by Large Language Models for Enhanced Robustness. In *2025 IEEE 43rd VLSI Test Symposium (VTS)*. IEEE, 1–5.

[18] Akashdeep Saha, Sayandeep Saha, Siddhartha Chowdhury, Debdeep Mukhopadhyay, and Bhargab B Bhattacharya. 2020. Lopher: Sat-hardened logic embedding on block ciphers. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[19] Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. 2025. Paper2code: Automating code generation from scientific papers in machine learning. *arXiv preprint arXiv:2504.17192* (2025).

[20] Bicky Shakya, Xiaolin Xu, Domenic Forte, and Mark Tehranipoor. 2019. CAS-Lock: A Security-Corruptibility Trade-off Resilient Logic Locking Scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020 (2019), 175–202. doi:10.13154/tches.v2020.i1.175-202

[21] Pramod Subramanyan, Sayak Ray, and Sharad Malik. 2015. Evaluating the security of logic encryption algorithms. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 137–143.

[22] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530* (2024).

[23] Zeng Wang, Lilas Alrahis, Likhitha Mankali, Johann Knechtel, and Ozgur Sinanoglu. 2024. Llms and the future of chip design: Unveiling security risks and building trust. In *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 385–390.

[24] Zeng Wang, Lilas Alrahis, Dominik Sisejkovic, and Ozgur Sinanoglu. 2023. AutoLock: Automatic Design of Logic Locking with Evolutionary Computation. In *2023 IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 25–32. doi:10.1109/DSN-W58399.2023.00062

[25] Zeng Wang, Minghao Shao, Mohammed Nabeel, Prithwish Basu Roy, Likhitha Mankali, Jitendra Bhandari, Ramesh Karri, Ozgur Sinanoglu, Muhammad Shafique, and Johann Knechtel. 2025. Verileaky: Navigating ip protection vs utility in fine-tuning for llm-driven verilog coding. *arXiv preprint arXiv:2503.13116* (2025).

[26] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan JV Rajendran, and Ozgur Sinanoglu. 2016. SARLock: SAT attack resistant logic locking. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 236–241.

[27] Muhammad Yasin, Abhrajit Sengupta, Mohammed T. Nabeel, Mohammed Ashraf, Jeyavijayan Rajendran, and Ozgur Sinanoglu. 2017. Provably-Secure Logic Locking: From Theory to Practice. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. 1601–1618. doi:10.1145/3133956.3133985 Introduces SFLL and the SFLL-HD variant.

[28] Yuke Zhang, Yinghua Hu, Pierluigi Nuzzo, and Peter A. Beerel. 2022. TriLock: IC Protection with Tunable Corruptibility and Resilience to SAT and Removal Attacks. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1329–1334. doi:10.23919/DATE54114.2022.9774649