MPC-PATCH-BENCH: BEYOND FUNCTIONAL TESTS FOR LLM REPAIR OF MPC SOFTWARE

Anonymous authors
Paper under double-blind review

ABSTRACT

Repairing secure multi-party computation (MPC) software requires more than producing outputs that pass existing tests. Patches must also respect intended information disclosure and numerical behavior. We introduce **MPC-Patch-Bench**, a repository-level benchmark containing 205 executable repair tasks from five MPC frameworks. Its construction combines MPC-specific pull-request filtering with expert-reviewed synthesis of missing problem statements and regression tests. Evaluation supplements functional tests with differential checks against plaintext references and static rules targeting potential MPC-specific violations. Across eight evaluated LLMs, Sonnet 4.6 achieves the highest functional resolution rate, 22.9%, but falls to 16.1% after the additional checks; Opus 4.6 then ranks highest at 17.1%. Across models, these checks reject 11–40% of functionally passing patches. The results show that functional-test performance and performance under additional MPC-specific checks are distinct evaluation signals. The benchmark supports studying this gap without treating check-passing patches as formally secure.

1 INTRODUCTION

Secure multi-party computation (MPC) lets parties jointly compute a function of their private inputs without revealing those inputs to one another, and it is used in privacy-preserving machine learning (Mohassel & Zhang, 2017; Wagh et al., 2019; Zheng et al., 2019), biomedical collaboration (Hie et al., 2018), and institutional analytics (Bogdanov et al., 2016). MPC programs compute on secret shares through interactive protocols and fixed-point encodings over finite rings, so a correct repair must preserve not only outputs but also what is disclosed and how numbers are approximated. LLMs are increasingly evaluated on repository-level repair (Jimenez et al., 2024), but we are not aware of a benchmark that measures this ability on MPC software. Building one raises three construction challenges, each addressed by one step of MPC-Patch-Bench[1] (Figure 1).

Finding relevant repairs. Being in an MPC repository does not make a task about MPC: many merged pull requests (PRs) change CI configuration, packaging, or high-level API wrappers. Our curation agent examines each PR against three tiers of the MPC stack—cryptographic primitives, protocol logic, and MPC-specific constraints such as data-obliviousness and fixed-point truncation—and, with expert review of borderline cases, retains 1,175 of 7,305 PRs from five frameworks (Section 3.2.1).

Making repairs executable. Useful historical fixes often lack a clear problem statement or runnable regression tests. PySyft, our largest source, links only 6.2% of its PRs to issues, and our SWE-bench-style extraction (Jimenez et al., 2024), which requires a linked issue and developer-written tests, retained 42 instances (Appendix A). In our completion stage, a construction agent drafts missing problem statements and Fail-to-Pass (F2P) / Pass-to-Pass (P2P) tests and an expert reviews them; this yields 205 validated executable tasks (Section 3.2.2).

Assessing more than tested outputs. An MPC patch should also avoid unintended reveals of secret values, respect data-oblivious control flow and public/private type discipline, and keep fixed-point error within tolerance (Rastogi et al., 2014; Zahur & Evans, 2015; Hastings et al., 2019); ordinary repair tests may not cover these properties. We therefore supplement the functional tests with

[1]Anonymous code and dataset for review: `https://anonymous.4open.science/r/MPC_bench-D496/`.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

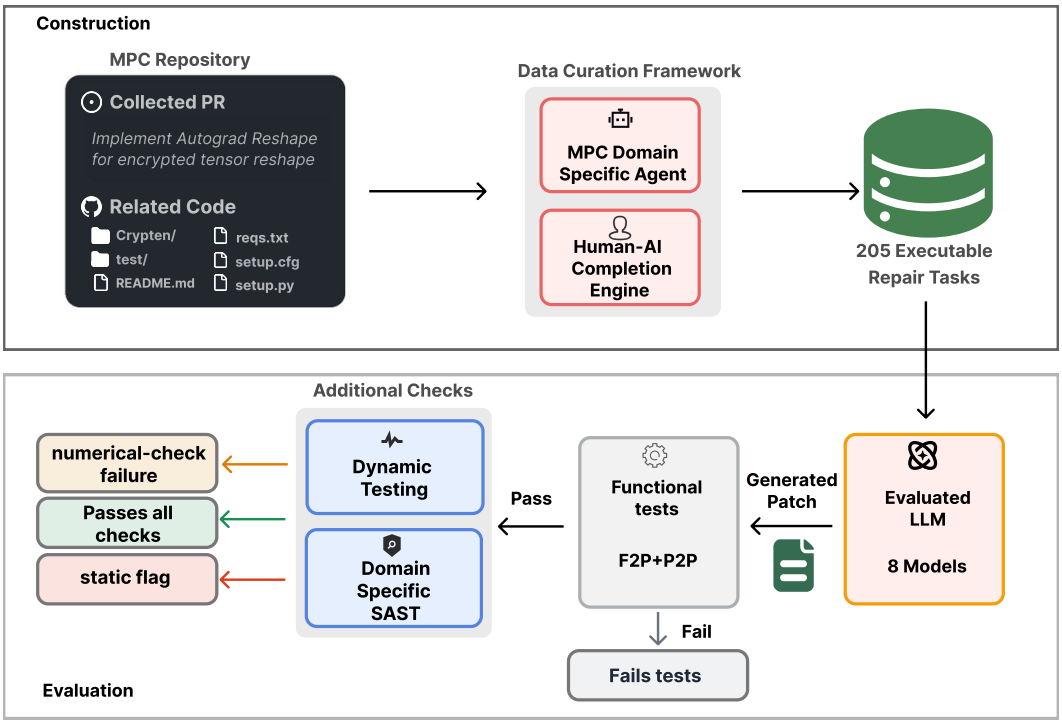

Figure 1: MPC-Patch-Bench. *Construction:* the MPC domain-specific agent (the LLM construction agent) filters merged PRs, and the Human-AI Completion Engine, in which the agent works with an expert, turns them into 205 executable repair tasks. *Evaluation:* each evaluated LLM produces a patch that is run against the functional tests; patches that pass then undergo dynamic testing and MPC-specific static checks (SAST), which separate patches that pass all checks from numerical-check failures and static flags.

differential checks against plaintext references (Pang et al., 2024) and static rules that flag potential MPC-specific violations (Section 3.3). Passing these checks means passing the benchmark's checks; it is not a proof of protocol security.

Evaluating eight LLMs shows that a patch can pass the repair tests yet fail the additional MPC-specific checks, and that evaluating both can change which model appears strongest. Sonnet 4.6 has the highest functional resolution rate (47/205, 22.9%) but passes all checks on 33/205 (16.1%), while Opus 4.6 goes from 43 to 35 (17.1%) and ranks first, although by a small margin. Across models, the checks reject from 2 of 19 (GPT-4o-mini) to 16 of 40 (GPT-5.4) functionally passing patches (Section 5.3).

2 Background and Related Work

2.1 Secure Multi-Party Computation

Building on the introduction, we briefly fix terminology before discussing related work. Practical MPC implementations split each input into additive secret shares over a finite ring; parties evaluate linear operations locally and realise non-linear operations through interactive sub-protocols built on Beaver triples and oblivious transfer (Hastings et al., 2019). Composed, these primitives support secure pipelines from linear regression to deep neural-network inference, and have driven the emergence of a mature open-source ecosystem.

Our benchmark draws repository-level tasks from five widely-used MPC frameworks. **CrypTen** (Knott et al., 2021) provides a PyTorch-style API for secure tensor computation focused on privacy-preserving machine learning. **tf-encrypted** (Dahl et al., 2018) offers an analogous TensorFlow-native interface for encrypted ML. **MP-SPDZ** (Keller, 2020) is a protocol-level toolkit supporting dozens of MPC protocols (semi-honest and malicious, two-party and multi-party) compiled from a domain-specific language. **SecretFlow** (Ma et al., 2023) integrates MPC with federated and hybrid privacy-preserving computation across heterogeneous data silos. **PySyft** (Ziller et al., 2021) is the OpenMined privacy-preserving deep-learning framework, supporting federated learning, differential privacy, and secure aggregation alongside secret-sharing-based MPC.

Table 1: Raw PR statistics.

Repository	Total PRs
crypten	236
secretflow	658
MP-SPDZ	110
PySyft	5,840
tf-encrypted	461
Total	**7,305**

2.2 MPC-Specific Code Synthesis and Verification

Existing efforts targeting LLM-based MPC code synthesis or MPC-aware verification address slices of the MPC software stack rather than repository-level repair. SPDZCoder (Dong et al., 2024) combines expert-derived transformation rules with an LLM to translate high-level Python into the MP-SPDZ (Keller, 2020) domain-specific language without dedicated training data. While SPDZCoder reduces the manual effort of writing MPC code, it differs from MPC-Patch-Bench in two respects: it operates at the operator/protocol level rather than across full repository contexts, and it targets a single MPC framework (MP-SPDZ), whereas real MPC engineering spans a heterogeneous library ecosystem. CrypTorch (Liu et al., 2025) is an auto-tuning compiler that selects operator approximations for MPC-based ML workloads to balance performance and accuracy; like SPDZCoder, it is a code-generation tool rather than an LLM evaluation, and its objective is performance-vs-accuracy rather than security correctness.

For verification, MPCDiff (Pang et al., 2024) performs differential testing between secure and plaintext executions to surface numerical-precision deviations in MPC-hardened ML models, HawkEye (Ruan et al., 2025) statically profiles the communication cost of models in multi-party-learning frameworks, and Li et al. (2024) use metamorphic testing to uncover logic vulnerabilities in MPC compilers. These tools are complementary to ours: MPC-Patch-Bench's checks include a differential stream similar in spirit to MPCDiff and static rules for potential MPC-specific violations (e.g., reveals inside loops, public/private casts, non-cryptographic randomness), applied to LLM-generated patches as part of a benchmark rather than as standalone tools on hand-written code or compiled protocols.

2.3 General-Purpose Code Repair Benchmarks

LLM coding benchmarks have evolved from synthesising isolated functions—e.g., HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021)—toward repository-level repair. EvalPlus (Liu et al., 2023) and DS-1000 (Lai et al., 2023) strengthened test rigour and broadened domain coverage to data-science libraries; CoderEval (Yu et al., 2024) extended evaluation to function-level generation under realistic repository dependencies. The current standard for repository-level evaluation is SWE-bench (Jimenez et al., 2024), which assembles real GitHub issues paired with developer-authored fail-to-pass tests, marking an instance resolved only when the model's patch passes those tests while preserving pass-to-pass behaviour. SWE-smith (Yang et al., 2025) extends this paradigm by synthetically generating bug-fix instances at scale via test-breaking mutations on existing codebases.

MPC-Patch-Bench adopts SWE-bench's task format and its F2P/P2P resolution criterion. Applying the SWE-bench construction procedure to MPC repositories nonetheless raises the three challenges of Section 1: domain-agnostic selection admits generic maintenance tasks, requiring a linked issue and developer-written tests left 42 of 7,305 PRs in our SWE-bench-style extraction (Appendix A), and functional tests alone may not cover disclosure or numerical properties.

3 MPC-PATCH-BENCH

3.1 OVERVIEW

MPC-Patch-Bench converts merged pull requests into executable repair tasks and evaluates generated patches in two stages. The **Data Curation Framework** (Section 3.2) filters PRs with a domain-specific curation agent and completes the retained PRs with an expert-reviewed completion engine that synthesises missing problem statements and F2P/P2P tests, yielding 205 validated executable instances from 7,305 PRs. The **MPC Verifier** (Section 3.3) supplements the functional tests with MPC-specific static checks and differential checks against plaintext references. Throughout, the *construction agent* denotes the LLM used to build the benchmark, and the *evaluated models* are the LLMs under test.

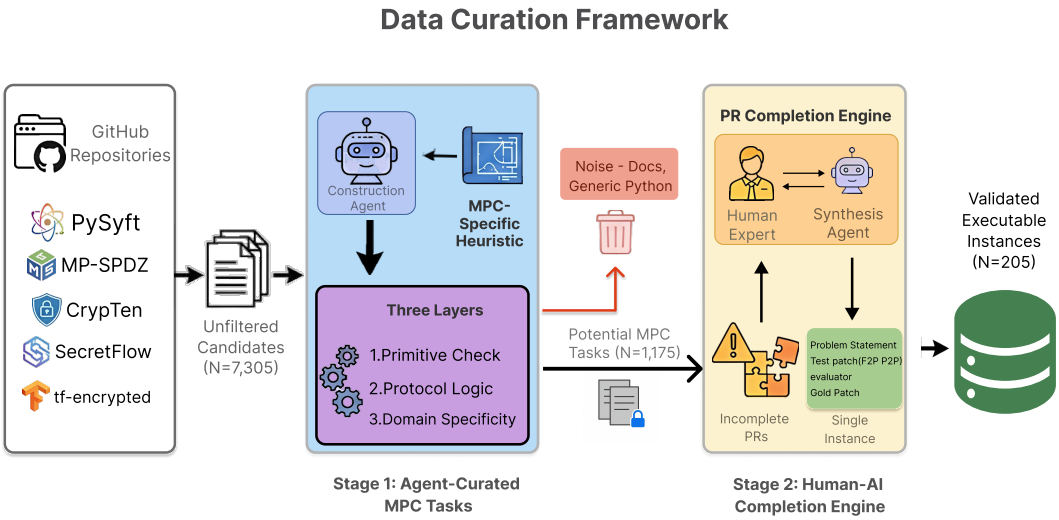

Figure 2: The Data Curation Framework: from 7,305 raw GitHub PRs through agent-driven curation and human-AI completion to 205 validated executable MPC task instances.

3.2 DATA CURATION FRAMEWORK

We crawl the five MPC frameworks introduced in Section 2.1 through the GitHub API following the SWE-bench protocol (Jimenez et al., 2024), yielding 7,305 candidate PRs (Table 1). The framework's two stages, depicted in Figure 2, are described next.

3.2.1 DOMAIN-SPECIFIC CURATION AGENT

The objective of this stage is to retain, at scale and with principled coverage of the MPC software stack, only those pull requests whose repair involves MPC-specific code, while emitting a graded confidence so that scarce human review can focus on ambiguous cases rather than obvious noise. Our key observation is that practical MPC software stacks are organised hierarchically—the architectural blueprints of frameworks such as CrypTen (Knott et al., 2021) expose a bottom-up structure of cryptographic primitives (secret shares, oblivious transfer), interactive protocol logic (e.g., A2B conversion, Beaver-triple synchronisation), and domain-specific secure tensor operations. We therefore align our curator with this hierarchy via three stack-aware layers, each examining one tier of the stack, and combine their outputs into a graded confidence that drives autonomous routing.

Concretely, an AI Agent processes each candidate's diff and natural-language metadata through a fixed Domain-Specific Heuristic prompt that decomposes into the three layers:

- **Layer 1: Primitive Check.** Verifies direct manipulation of core cryptographic primitives. As highlighted in standard Systematization of Knowledge (SoK) studies (Hastings et al., 2019), practical MPC implementations inherently rely on foundational mathematical constructs at

the base layer, such as arithmetic/binary secret sharing and Oblivious Transfer (OT). This layer ensures the code modification directly operates at this cryptographic bedrock.

- **Layer 2: Protocol Logic.** Distinguishes fixes addressing mathematical MPC correctness from generic system wrappers or API formatting. State-of-the-art MPC frameworks heavily rely on the combination of multiple protocols and the secure, efficient conversions between them (e.g., Arithmetic-to-Binary or A2B conversions) (Braun et al., 2022). This layer evaluates whether the issue tackles the logic and state machines of these multi-party interactive protocols.

- **Layer 3: Domain Specificity.** Ensures the resolution demands specialized cryptographic reasoning rather than general software engineering. This explicitly checks for MPC-exclusive engineering constraints that a standard developer would not encounter, such as maintaining strict data-obliviousness during tensor computations (e.g., within a `CrypTensor` (Knott et al., 2021) structure), managing fixed-point truncation errors over finite rings, or optimizing multi-party communication rounds.

Each layer maps onto a distinct tier of the MPC stack, so a candidate that activates one or more layers is likely to involve MPC-specific code rather than generic Python. The Agent emits a calibrated confidence from the layer outputs, and we apply the autonomous routing of Algorithm 1: candidates below the noise threshold $\tau_{med} = 0.75$ are discarded as generic noise; those above $\tau_{high} = 0.9$ are admitted to the potential pool $\mathcal{C}_{potential}$ directly; and the band $[\tau_{med}, \tau_{high})$ is routed to a review pool $\mathcal{C}_{review}$ for human adjudication.

Result. Applied to the 7,305 raw candidates, the curator discards 6,130 PRs as not MPC-specific and retains 1,175 candidates (596 high-confidence accepted automatically and 579 medium-confidence routed for human review), which then enter the human-AI completion stage of Section 3.2.2.

3.2.2 Human-AI Completion Engine

The objective of this stage is to convert the 1,175 candidates retained by curation into standardised, executable benchmark instances, including those whose original pull requests ship without developer-authored Fail-to-Pass (F2P) and Pass-to-Pass (P2P) tests—a population that our SWE-bench-style extraction discards, retaining 42 instances out of the same 7,305 PRs (Appendix A). Our observation is that the missing artefacts (clean problem statement, F2P/P2P test patch, executable harness) are tractable to synthesise from the raw PR diff plus expert cryptographic intuition: the diff itself encodes the intended fix, and an expert can identify which behaviours the F2P test must isolate. We therefore introduce a **Human-AI Collaborative Completion** paradigm in which a human expert and a synthesis agent jointly reconstruct the missing artefacts.

As illustrated in the third stage of Figure 2, the 1,175 candidates are routed through a PR Completion Engine that operates as a tightly coupled human-AI workflow. Rather than discarding cryptographically sound but incomplete PRs, a human expert co-pilots with a synthesis agent: the expert provides domain knowledge and reviews each generated artefact against the upstream fix, while the agent structures and generates the missing programmatic components. PRs flagged as medium-confidence by curation first pass through a manual offline adjudication step to confirm validity before being admitted to the completion pipeline.

Through this bidirectional collaboration the engine constructs the components of a standard, executable instance—a clear problem statement, the missing F2P/P2P test patch, an evaluator execution harness, and a gold patch that passes the tests. The resulting instances treat test-incomplete PRs as completable rather than disqualifying, are reviewed by the expert against the upstream fix, and can be scored against LLM patches automatically without further human intervention.

Result. The Data Curation Framework distils the 7,305 raw candidates into **205 validated executable instances**, compared with 42 from our SWE-bench-style extraction on the same corpus (Appendix A).

3.3 MPC VERIFIER

The functional tests check a patch's behaviour on the tested inputs. The MPC Verifier adds two screens for properties those tests may not cover, while remaining cheap enough to run on every generated patch—a constraint that rules out heavyweight formal verification. The failure modes we target fall into two classes: *structural* patterns visible in the source (e.g., reveals inside loops, non-cryptographic randomness, public/private casts) and *behavioural* discrepancies that appear only when the secure computation runs (e.g., fixed-point drift or edge-case divergence from a plaintext reference). The verifier therefore has a *static* stream and a *dynamic* stream (Figure 3); a functionally resolved patch that passes both *passes the benchmark's checks*. This is not a proof of protocol security: the dynamic checks cover finitely many inputs, and a static flag marks a potential violation for review rather than a confirmed vulnerability.

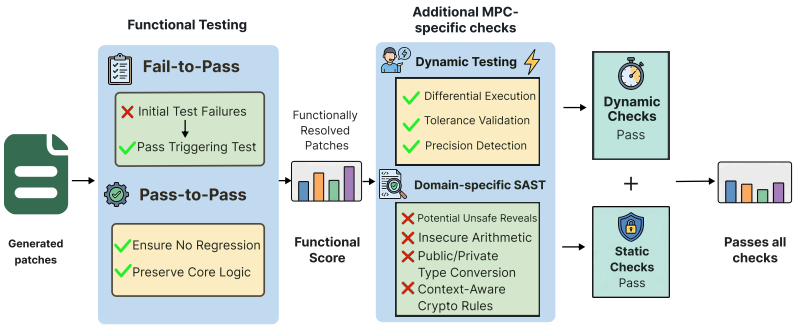

Figure 3: The MPC Verifier. Functionally resolved patches are screened by (1) dynamic checks that compare secure outputs with a plaintext reference, including boundary inputs and a fixed-point error tolerance, and (2) MPC-specific static rules that flag potential violations without execution.

3.3.1 DYNAMIC TESTING

The Dynamic Testing stream orchestrates multi-party code execution and emits a **Dynamic Score** for execution fidelity through three mechanisms: *differential execution* (MPCDiff-style comparison of secure outputs against a plaintext reference), *tolerance validation* (injecting strict boundary values such as 0, -1, MAX_INT and complementary inequalities to probe algebraic edge cases), and *precision detection* (monitoring fixed-point truncations and cumulative tensor-operation error against a fixed tolerance, atol $= 0.005$).

3.3.2 DOMAIN-SPECIFIC SAST

The Domain-Specific Static Application Security Testing (SAST) module applies MPC-specific static checks to the patched source and emits a **Static Score**. Its rules flag four classes of potential violation: *unsafe-reveal patterns* (e.g., `reveal()` or `get_plain_text()` inside a loop, which may expose intermediate secret values), *insecure arithmetic* (raw floating-point coercion on secret-shared types or non-cryptographic randomness such as `np.random.rand()`), *public/private type conversions* that may cross the intended data-oblivious boundary, and *context-dependent rule violations* (e.g., missing synchronisation steps). A flag identifies a pattern that warrants review; whether it constitutes an unauthorised disclosure depends on what the code is intended to reveal, and to whom.

Effect. Section 5.3 reports how often these checks reject functionally passing patches and how this changes the model ranking.

4 DATASET

4.1 DATASET OVERVIEW

The MPC-Patch-Bench dataset comprises **205 validated executable task instances**, curated from five foundational open-source MPC libraries: CrypTen (Knott et al., 2021), tf-encrypted (Dahl et al., 2018), MP-SPDZ (Keller, 2020), SecretFlow (Ma et al., 2023), and PySyft (Ziller et al., 2021). As illustrated in Figure 4, the dataset spans a diverse range of MPC paradigms: CrypTen (29.8%, $n = 61$) and tf-encrypted (21.0%, $n = 43$) represent tensor-based, ML-oriented secret sharing frameworks; MP-SPDZ (20.0%, $n = 41$) covers protocol-level MPC with multi-party arithmetic; SecretFlow (12.7%, $n = 26$) addresses federated and hybrid MPC computation; and PySyft (16.6%, $n = 34$) represents privacy-preserving deep learning infrastructure. This cross-library diversity ensures that MPC-Patch-Bench evaluates LLMs against a broad and representative spectrum of real-world MPC engineering challenges, rather than overfitting to any single framework or cryptographic paradigm.

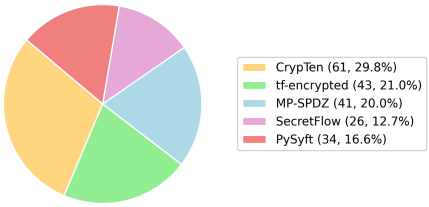

Figure 4: Distribution of MPC-Patch-Bench's 205 task instances across five MPC libraries. CrypTen dominates (29.8%, $n = 61$), while SecretFlow contributes the smallest share (12.7%, $n = 26$).

Table 2: Statistics of MPC-Patch-Bench task instances (micro-averages across all 205 instances).

		Mean	Max
Problem Statement	Length (Words)	47.8	481
	Length (Chars)	372.6	3,557
Gold Patch	# Lines edited	583.5	74,717
	# Files edited	9.7	842
Tests	# F2P tests	2.2	51
	# P2P tests	1.4	64

4.2 COMPLEXITY AND DIVERSITY OF TASK CHARACTERISTICS

To comprehensively evaluate the reasoning span required by our benchmark, we decompose each MPC-Patch-Bench instance into a structured tuple: a *problem statement* (the developer's natural language intent), a *code context* (the repository state), a *test patch* (Fail-to-Pass and Pass-to-Pass verifiers), and a *gold patch* (the empirical solution). Figure 5 visualizes the empirical cumulative distribution functions (CDFs) across three critical structural dimensions, demonstrating that MPC-Patch-Bench exhibits both the structural diversity of real-world software engineering and the unique, exacting complexity of cryptographic development.

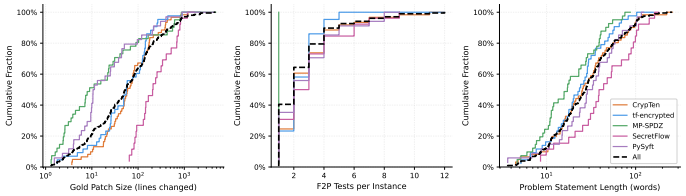

Figure 5: Cumulative distribution functions (CDFs) of MPC-Patch-Bench tasks: (left) gold patch size, (centre) Fail-to-Pass test count per instance, and (right) problem statement word count. See main text for analysis.

The CDFs reveal three things at once. **Gold patch size** (left) ranges from surgical 10–50-line edits typical of MP-SPDZ to long tails beyond 200 lines (prominent in SecretFlow), confirming that MPC-Patch-Bench captures the full spectrum of cryptographic-software complexity rather than isolated functions. **Fail-to-Pass test count** (centre) clusters tightly across all five frameworks—more than half of all instances (55%) are gated by a single rigorously constrained F2P test, exposing a domain-universal hallmark: MPC correctness is verified against precise mathematical invariants rather than

expansive behavioural suites, imposing a steep penalty on iterative guessing. **Problem statement length** (right) varies widely (under 10 words to over 100), spanning terse bug reports requiring deep repository-level context to extensive cryptographic discussions demanding step-by-step reasoning, preventing models from overfitting to a specific prompt formulation.

A complementary analysis of how task difficulty—measured by per-instance resolution rate—varies with gold-patch size and Fail-to-Pass test count is presented in Appendix F.

5 Evaluation

5.1 Experimental Setup

Models and prompting. We evaluate eight contemporary LLMs spanning Anthropic, OpenAI, and Google (Sonnet 4.6, Opus 4.6, Haiku 4.5, GPT-5.4, GPT-4.1, GPT-4o-mini, Gemini 2.5 Pro, Gemini 2.5 Flash-Lite; see Appendix C for exact API IDs). All models are queried zero-shot at temperature 0 where supported, with a 16,384-token output cap (65,536 for Gemini); requests that exceed a model's context window or fail at the API are counted as unresolved, so all rates use the 205 tasks as the denominator. Each prompt provides the MPC-related problem statement, the relevant pre-patch source file(s), and instructions to emit complete modified files using a «<FILE: path»>...«<END»> delimiter format.

Test execution. Generated patches are applied to the repository at the base commit and executed using library-specific test runners—`pytest` with a 300-second per-test timeout for CrypTen, tf-encrypted, SecretFlow, and PySyft, and a custom `compile.py` + `emulate.sh` pipeline for MP-SPDZ. An instance is marked *functionally resolved* iff all F2P tests pass and all P2P tests continue to pass; functionally resolved patches then undergo the additional checks of Section 3.3, and a patch that passes all of them counts toward the *check-passing* resolution rate. All runs use an HPC cluster (RHEL 8) with per-instance isolation via separate git worktrees.

5.2 Functional Resolution Rates

As Figure 6 shows, even the strongest evaluated model resolves only 22.9% of the 205 tasks, and difficulty varies sharply across the five MPC frameworks—high-level PyTorch-style libraries are comparatively tractable while protocol-level systems remain almost entirely unresolved by every evaluated model. We unpack this aggregate picture by model and by library below.

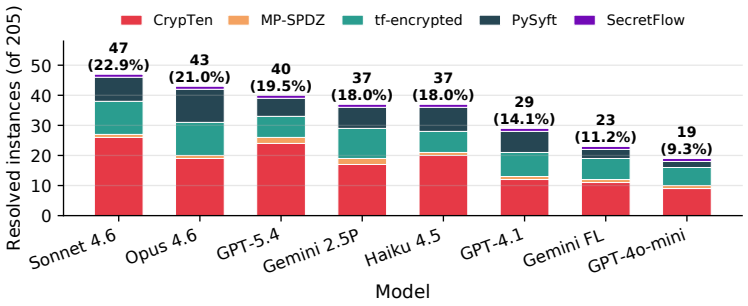

Figure 6: MPC-Patch-Bench model ranking and overall functional resolution rates across 205 task instances. Bars are split by library; Sonnet 4.6 has the highest functional resolution rate (22.9%).

Anthropic's Claude family leads the ranking—notably, the lighter Sonnet 4.6 (22.9%) outperforms the heavier Tier-1 models Opus 4.6 (21.0%) and GPT-5.4 (19.5%). The remaining models cluster from mid-teens (Gemini 2.5P / Haiku 4.5 at 18.0%, GPT-4.1 at 14.1%) down to 11.2% (Gemini FL) and 9.3% (GPT-4o-mini). The per-library breakdown is sharply bimodal: CrypTen is consistently the most tractable framework, likely because its PyTorch-aligned APIs overlap closely with LLM pretraining data, while MP-SPDZ at the opposite extreme remains nearly unsolved (top models resolve at most 2 of its 41 instances), reflecting the difficulty of its bespoke, C++-like scripting language and low-level arithmetic protocol logic. SecretFlow, tf-encrypted, and PySyft fall between.

These results indicate that MPC repair remains difficult for current models, with difficulty concentrated in protocol-level frameworks.

5.3 Effect of the Additional Checks

Figure 7 applies the additional checks of Section 3.3 to every functionally resolved patch. Across models, the checks reject from 2 of 19 (GPT-4o-mini, 11%) to 16 of 40 (GPT-5.4, 40%) functionally passing patches.

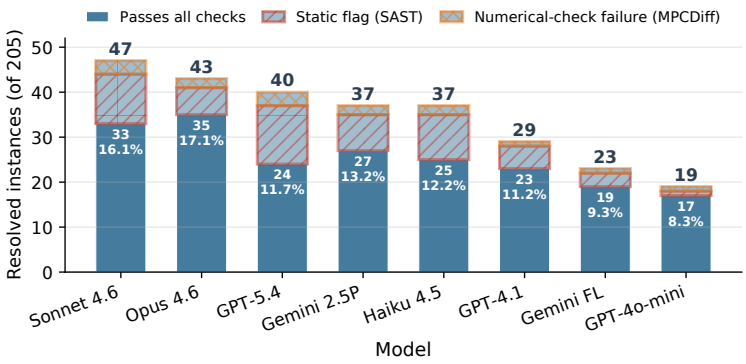

Figure 7: Functional resolution vs. check-passing resolution per model. Bar height and the number above each bar give functional resolutions; white labels give patches that pass all checks (count and share of 205 tasks). Rejected patches are split into static flags (SAST) and numerical-check failures (MPCDiff).

Sonnet 4.6 loses 14 of its 47 functional resolutions (30%; 11 static flags and 3 numerical-check failures), leaving 33 (16.1%), while Opus 4.6 loses 8 of its 43 and retains 35 (17.1%), which makes it the top-ranked model under the additional checks. GPT-5.4 loses the most, 16 of its 40 (13 static flags and 3 numerical-check failures). The margin between Opus 4.6 and Sonnet 4.6 is small (35 vs. 33 of 205 tasks), so we read this change as evidence that functional resolution and check-passing resolution are distinct signals, not as a definitive ordering of the two models. Because static flags mark potential violations rather than confirmed vulnerabilities, these counts indicate how many functionally passing patches warrant review, not how many are insecure.

6 Conclusion

We presented MPC-Patch-Bench, a repository-level benchmark of 205 executable repair tasks from five MPC frameworks, built through MPC-specific curation and expert-reviewed completion. Supplementing functional tests with MPC-specific static and dynamic checks shows that a patch can pass the repair tests yet fail the additional checks, and that evaluating both can change which model appears strongest. We hope the benchmark supports studying this gap.

Limitations and future work. Passing the additional checks is not a proof of protocol security: the dynamic checks cover a finite set of inputs, and static flags mark potential violations that require review rather than confirmed vulnerabilities. The change in the top-ranked model rests on a small margin (35 vs. 33 of 205 tasks). Because all tasks come from public repositories, contamination of pretraining data remains unresolved (Appendix D). The current release covers five MPC frameworks; extending it to additional cryptographic stacks (e.g., fully-homomorphic-encryption hybrids and zero-knowledge programming environments) is future work.

AI use statement

In this work, generative AI tools are part of the research methodology in two ways that require disclosure. First, benchmark construction uses LLM-based agents: the Domain-Specific Curation Agent (Section 3.2.1, Appendix E) classifies and filters candidate pull requests, and the synthesis

agent of the Human-AI Completion Engine (Section 3.2.2) drafts missing problem statements and Fail-to-Pass/Pass-to-Pass tests under the guidance of a human expert. Every synthesised instance was validated by a human expert and executed in the evaluation harness before being admitted to the benchmark. Second, the eight evaluated LLMs are the object of study (Section 5.1, Appendix C). Additionally, we used generative AI tools to assist in creating scientific figures, in revising the manuscript text for clarity, and in reformatting the manuscript and references to the ICLR template. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

Ethics statement

MPC-Patch-Bench is built exclusively from publicly available open-source repositories whose licenses permit research use, benchmarking, modification, and redistribution (Appendix B); no personal data is collected. The only human input is in-house expert annotation of cryptographic source code by the authors; no crowdsourcing or human-subjects research is involved. The benchmark and its checks are intended to help evaluate LLM-generated MPC code beyond functional tests: by exposing the gap between functional resolution and resolution under additional MPC-specific checks, they discourage premature trust in LLM-generated privacy-preserving code. Passing the checks does not certify a patch as secure. An MPC-aware code-repair benchmark could in principle accelerate the development of MPC implementations whose flaws facilitate downstream attacks; we consider this risk limited, because the released rules describe defensive checks rather than offensive capabilities.

Reproducibility statement

An anonymised snapshot of the 205 validated instances, the MPC Verifier, the evaluation harness, and the per-library Docker images is available for review at `https://anonymous.4open.science/r/MPC_bench-D496/`. Appendix C reports the exact API model identifiers, generation parameters, prompt template, per-library test runners, execution protocol, and computational resources. Appendix A reports per-stage retention statistics, and Appendix E formalises the curation-agent routing logic and thresholds. All evaluations use temperature 0 where supported, and the verifier is deterministic given a fixed patch and base commit.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and taxes: a privacy-preserving study using secure computation. *Proceedings on Privacy Enhancing Technologies*, 2016(3):117–135, 2016. doi: 10.1515/popets-2016-0019.

Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. MOTION – A framework for mixed-protocol multi-party computation. *ACM Transactions on Privacy and Security (TOPS)*, 25(2):1–35, 2022. doi: 10.1145/3490390.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL `https://arxiv.org/abs/2107.03374`.

Morten Dahl, Jason Mancuso, Yann Dupis, Ben Decoste, Morgan Giraud, Ian Livingstone, Justin Patriquin, and Gavin Uhma. Private machine learning in TensorFlow using secure computation. *arXiv preprint arXiv:1810.08130*, 2018. URL https://arxiv.org/abs/1810.08130. Privacy Preserving Machine Learning Workshop at NeurIPS 2018.

Xiaoning Dong, Peilin Xin, Jia Li, and Wei Xu. SPDZCoder: Combining expert knowledge with LLMs for generating privacy-computing code. *arXiv preprint arXiv:2501.00363*, 2024. URL https://arxiv.org/abs/2501.00363.

Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: General purpose compilers for secure multi-party computation. In *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1220–1237. IEEE, 2019. doi: 10.1109/SP.2019.00028.

Brian Hie, Hyunghoon Cho, and Bonnie Berger. Realizing private and practical pharmacological collaboration. *Science*, 362(6412):347–350, 2018. doi: 10.1126/science.aat4807.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 1575–1590, 2020. doi: 10.1145/3372297.3417872. URL https://eprint.iacr.org/2020/521.

Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. CrypTen: Secure multi-party computation meets machine learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021. URL https://papers.neurips.cc/paper/2021/hash/2754518221cfbc8d25c13a06a4cb8421-Abstract.html.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning (ICML)*, 2023.

Yichen Li, Dongwei Xiao, Zhibo Liu, Qi Pang, and Shuai Wang. Metamorphic testing of secure multi-party computation (MPC) compilers. *Proceedings of the ACM on Software Engineering*, 1(FSE):1216–1237, 2024. doi: 10.1145/3643781. URL https://dl.acm.org/doi/10.1145/3643781.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Jinyu Liu, Gang Tan, and Kiwan Maeng. CrypTorch: PyTorch-based auto-tuning compiler for machine learning with multi-party computation. *arXiv preprint arXiv:2511.19711*, 2025. URL https://arxiv.org/abs/2511.19711.

Junming Ma, Yancheng Zheng, Jun Feng, Derun Zhao, Haoqi Wu, Wenjing Fang, Jin Tan, Chaofan Yu, Benyu Zhang, and Lei Wang. SecretFlow-SPU: A performant and user-friendly framework for privacy-preserving machine learning. In *2023 USENIX Annual Technical Conference (USENIX ATC)*, pp. 17–33, 2023. URL https://www.usenix.org/conference/atc23/presentation/ma.

Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 19–38. IEEE, 2017. doi: 10.1109/SP.2017.12.

Qi Pang, Yuanyuan Yuan, and Shuai Wang. MPCDiff: Testing and repairing MPC-hardened deep learning models. In *31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024. doi: 10.14722/ndss.2024.23380.

Aseem Rastogi, Matthew A Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy (SP)*, pp. 655–670. IEEE, 2014. doi: 10.1109/SP.2014.48.

Wenqiang Ruan, Xin Lin, Ruisheng Zhou, Guopeng Lin, Shui Yu, and Weili Han. HawkEye: Statically and accurately profiling the communication cost of models in multi-party learning. In *34th USENIX Security Symposium (USENIX Security)*, pp. 2673–2691, 2025. URL https://www.usenix.org/conference/usenixsecurity25/presentation/ruan.

Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *Proceedings on Privacy Enhancing Technologies*, 2019(3):26–49, 2019. doi: 10.2478/popets-2019-0035.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025. URL https://arxiv.org/abs/2504.21798.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. CoderEval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 1–12, 2024. doi: 10.1145/3597503.3623316.

Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. URL https://eprint.iacr.org/2015/1153.

Wenting Zheng, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 724–738. IEEE, 2019. doi: 10.1109/SP.2019.00045.

Alexander Ziller, Andrew Trask, Antonio Lopardo, Benjamin Szymkow, Bobby Wagner, Emma Bluemke, Jean-Mickael Nounahon, Jonathan Passerat-Palmbach, Kritika Prakash, Nick Rose, Théo Ryffel, Zarreen Naowal Reza, and Georgios Kaissis. PySyft: A library for easy federated learning. In *Federated Learning Systems: Towards Next-Generation AI*, Studies in Computational Intelligence, pp. 111–139. Springer, 2021. doi: 10.1007/978-3-030-70604-3_5.

A SWE-BENCH-STYLE EXTRACTION ON MPC REPOSITORIES

To quantify how many tasks can be obtained without the completion stage (Section 3.2.2), we applied a SWE-bench-style extraction (Jimenez et al., 2024) to our corpus of 7,305 MPC pull requests.

This extraction requires each PR to be linked to an issue and to contain both the code change and developer-written tests from which Fail-to-Pass tests can be derived. On our corpus it retained **42** instances (Table 3).

Per-repository inspection suggests three main sources of attrition:

- **Non-Standard and Internal Workflows:** Repositories backed by corporate entities often utilize opaque internal tracking systems. For instance, *CrypTen* relies on Meta's internal workflow, and none of its 236 crawled PRs has both a linked issue and a test change.
- **Atypical Issue-PR Linking Practices:** Many MPC maintainers favor direct bug pushes over formalized issue-PR workflows. *MP-SPDZ* demonstrates a high merge rate but negligible issue referencing (less than 5%). Even *PySyft*, our largest data source, is severely bottlenecked by a mere 6.2% link rate.
- **Lack of Execution-Ready Tests:** Even when issues are properly linked to PRs, the linked PRs frequently lack the strict test patches required by the SWE-bench format. *SecretFlow* is one such case: a high volume of merged PRs but a comparatively small fraction shipping with associated developer-authored test patches.

Table 3: Statistics for how many candidate task instances were kept after the completion of a stage across the construction and validation procedures.

Repo	Total PRs Crawled	Post-Conversion	Post-Curation	Post-Validation (Final)
crypten	236	0	0	0
secretflow	658	22	2	0
MP-SPDZ	110	4	1	1
PySyft	5,840	277	159	30
tf-encrypted	461	40	22	11
Total	**7,305**	**343**	**184**	**42**

Relying on this extraction alone would yield a small benchmark concentrated in two repositories (PySyft and tf-encrypted contribute 41 of the 42 instances), which motivates the completion stage.

Total PRs Crawled The initial dataset containing all raw, merged pull requests (PRs) harvested from the target GitHub repositories via the developer API.

Post-Conversion The first filtering stage where PRs are retained only if they are explicitly linked to a GitHub issue and include modifications to test files, ensuring a baseline for verifiability.

Post-Curation A domain-specific refinement stage in section 3.2.2 where general infrastructure-related tasks (such as CI/CD pipelines, environment setup, or documentation) are filtered out to ensure the remaining instances are strictly focused on core Multi-Party Computation (MPC) logic.

Post-Validation (Final) The finalized benchmark suite consisting of tasks that passed the execution harness. This confirms that the repository can be successfully installed and that the provided patch resolves a verified "Fail-to-Pass" test case.

B Repository Selection and License Compliance

In addition to the semantic filtering criteria discussed in Section 2, we strictly ensure that all selected MPC repositories are governed by licenses that permit non-proprietary use, academic research, code modification, and redistribution.

The majority of the software licenses in our curated corpus are highly permissive (e.g., Apache License 2.0, MIT License). For repositories operating under custom or academic-specific licenses, we manually inspected the license terms to confirm they explicitly authorize the benchmarking and research use cases exercised in our work. The specific licenses associated with each core repository evaluated in MPC-Patch-Bench are fully detailed in Table 4.

Table 4: Licenses associated with each repository included in MPC-Patch-Bench. All licenses are permissive and allow for public, academic, and non-profit research use.

License	Repository / Description
Apache License 2.0	`OpenMined/PySyft`, `secretflow/secretflow`, `tf-encrypted/tf-encrypted`
MIT License	`facebookresearch/CrypTen`
Open-Source / Academic License	`data61/MP-SPDZ` *Note: Publicly available under an academic license that explicitly permits protocol benchmarking and research extensions.*

C REPRODUCIBILITY

To ensure full reproducibility of our evaluation results, we report the exact model identifiers, API configurations, and evaluation protocol used.

Model Identifiers and API Details. All evaluations were conducted between March and April 2026 via official vendor APIs. Table 5 lists the exact model IDs queried.

Table 5: Exact model identifiers used in all MPC-Patch-Bench evaluations.

Model (Display Name)	API Model ID	Vendor	API Date
Sonnet 4.6	`claude-sonnet-4-6`	Anthropic	Mar 2026
Opus 4.6	`claude-opus-4-6`	Anthropic	Mar 2026
Haiku 4.5	`claude-haiku-4-5-20251001`	Anthropic	Mar 2026
GPT-5.4	`gpt-5.4`	OpenAI	Mar 2026
GPT-4.1	`gpt-4.1`	OpenAI	Mar 2026
GPT-4o-mini	`gpt-4o-mini`	OpenAI	Mar 2026
Gemini 2.5 Pro	`gemini-2.5-pro`	Google	Apr 2026
Gemini FL	`gemini-2.5-flash-lite`	Google	Apr 2026

Generation Parameters. All models were queried with a temperature of 0 (greedy decoding) where supported. For OpenAI's GPT-5.x series, temperature is not configurable via the API and defaults to the model's internal setting. Maximum output length was set to 16,384 tokens (Anthropic and OpenAI) and 65,536 tokens (Gemini). No system prompts beyond the task-specific evaluation prompt were used. For OpenAI models encountering rate limits (HTTP 429), exponential backoff was applied (initial wait 30s, max 300s). Requests exceeding the model's context window were logged as `request_too_large`; these, like other API failures, are counted as unresolved within the 205-task denominator.

Evaluation Prompt Template. Each model received an identical structured prompt consisting of:

1. A MPC based problem statement (derived from the linked GitHub issue).
2. The relevant source file(s) from the repository at the pre-patch commit.
3. Instructions to output the complete modified file(s) using the delimiter format: «<FILE: path»>...«<END»>.

No few-shot examples or chain-of-thought elicitation were used; all evaluations are strictly zero-shot.

Execution and Validation Protocol. For each generated patch:

1. The patch is applied to the repository checked out at the base commit.
2. The Fail-to-Pass (F2P) and Pass-to-Pass (P2P) test suites are executed using the library-specific test runner:
   - **CrypTen, tf-encrypted, SecretFlow, PySyft:** `pytest` with a 300-second timeout per test.
   - **MP-SPDZ:** Custom compile-and-emulate pipeline (`python3 compile.py <test> && Scripts/emulate.sh <test>`).
3. An instance is marked *resolved* if and only if all F2P tests pass and all P2P tests continue to pass.
4. For the evaluation under additional checks (Section 5.3), functionally resolved patches additionally undergo the static (SAST) and dynamic (MPCDiff-style) checks.

Computational Resources. All test execution was performed on an HPC cluster running RHEL 8 (Linux 4.18) with per-instance isolation via separate `git worktrees`. CrypTen's secure-tensor runtime requires a CUDA-capable GPU, for which we used the cluster's NVIDIA V100 32GB /

H100 PCIe / H100 80GB HBM3 partitions; the other libraries (tf-encrypted, SecretFlow, PySyft, MP-SPDZ) ran CPU-only. Cumulative vendor-API spend across the eight evaluated LLMs on the 205-instance benchmark was on the order of a few hundred US dollars.

D Data Contamination

All instances are derived from public repositories, so upstream fixes may appear in the pretraining data of the evaluated models. We do not measure contamination, and we do not claim that MPC-Patch-Bench avoids it.

Some features of the construction may reduce, but do not rule out, the benefit of memorisation. 163 of the 205 instances use problem statements and tests produced during completion, which do not appear verbatim upstream, and each task is evaluated against the repository state at its base commit. Neither these features nor the low resolution rates establish the absence of memorisation: a rewritten problem statement leaves the upstream code change unchanged, and a model can recall part of a fix and still fail the tests. Likewise, rejections by the additional checks do not show that patches are novel rather than recalled. Contamination is therefore an unresolved limitation of this work.

Future versions could measure contamination directly, for example by comparing resolution on instances created before and after each model's training cutoff or by applying membership-inference tests to the gold patches. Because the curation pipeline requires only a git repository with commit history, it can also be re-run on newer commit ranges, or on private MPC codebases, to produce fresh instances.

E Domain-Specific Curation Agent: Algorithm Pseudocode

The autonomous routing logic of the Domain-Specific Curation Agent (Section 3.2.1) is formalised in Algorithm 1.

Algorithm 1 Domain-Specific Curation Agent

Require: Unfiltered Candidates $\mathcal{C}_{raw}$, AI Agent $\mathcal{A}_{agent}$, Fixed MPC-Specific Heuristic $\mathcal{H}_{mpc}$, Confidence thresholds τ_{high} and τ_{med} (currently we set $\tau_{high} = 0.9$ and $\tau_{med} = 0.75$).
Ensure: Potential MPC Tasks $\mathcal{C}_{potential}$, Review Pool $\mathcal{C}_{review}$

Initialize $\mathcal{C}_{potential} \leftarrow \emptyset$, $\mathcal{C}_{review} \leftarrow \emptyset$
for each candidate $c \in \mathcal{C}_{raw}$ **do**
  $diff,\ desc \leftarrow \text{ExtractData}(c)$
  $prompt \leftarrow \text{ConstructPrompt}(\mathcal{H}_{mpc}, diff, desc)$
  {Three-layer evaluation: 1.Primitive Check, 2.Protocol Logic, 3.Domain Specificity}
  $layer_outputs,\ conf \leftarrow \mathcal{A}_{agent}.\text{Evaluate}(prompt)$
  $c.\text{metadata} \leftarrow layer_outputs$ {Save analysis for completion-stage verification}
  {Autonomous Routing based on Confidence Score}
  if $conf \geq \tau_{high}$ **then**
    $\mathcal{C}_{potential} \leftarrow \mathcal{C}_{potential} \cup \{c\}$ {Automatically accepted}
  else if $\tau_{med} \leq conf < \tau_{high}$ **then**
    $\mathcal{C}_{review} \leftarrow \mathcal{C}_{review} \cup \{c\}$ {Flagged for human adjudication}
  else
    Discard c as Noise {Low Confidence or None}
  end if
end for
return $\mathcal{C}_{potential}, \mathcal{C}_{review}$

F Dataset Difficulty Analysis

A primary motivation for MPC-Patch-Bench is to provide a benchmark that is genuinely challenging to state-of-the-art LLMs. Figure 8 presents a multi-faceted analysis of task difficulty as measured by resolution outcomes across 205 instances.

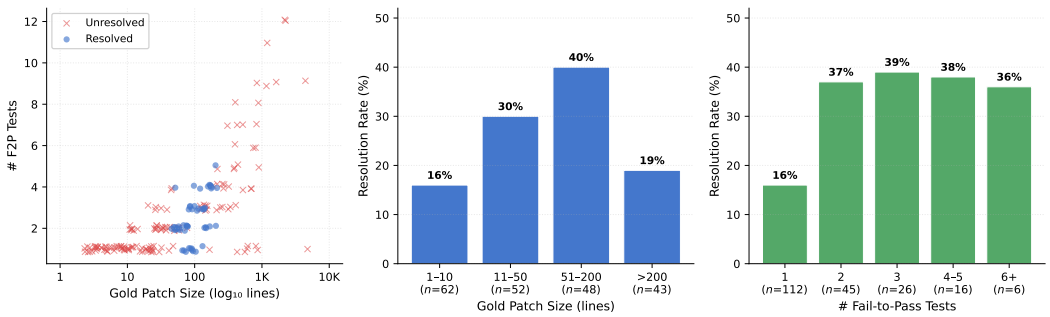

Figure 8: Difficulty analysis of MPC-Patch-Bench tasks. (Left) A scatter plot of Fail-to-Pass test count versus patch size for all instances, with resolved instances marked as circles (○) and unresolved as crosses (×). (Center) Instance-level resolution rate grouped by gold patch size. (Right) Resolution rate grouped by number of F2P tests. Both axes reveal significant variation in difficulty across the benchmark.

Tasks with gold patches in the 51–200 line range exhibit the highest resolution rate (40%, $n = 48$), suggesting that mid-complexity fixes—those large enough to require meaningful reasoning but small enough to fit within a model's effective context—represent the "sweet spot" where current LLMs demonstrate partial competence. In contrast, tasks requiring patches in excess of 200 lines drop to a resolution rate of only 19% ($n = 43$), and very short patches (1–10 lines) yield only 16% resolution ($n = 62$), the latter likely reflecting the need for highly precise, domain-critical one-liner edits that demand expert-level cryptographic understanding.

Analysing resolution by F2P test count reveals a complementary pattern: instances with 2 or more tests exhibit notably higher resolution rates (36–39%), while those with a single F2P test yield only 16% ($n = 112$). We hypothesise that a higher test count provides richer feedback signals and clearer task boundaries that assist models in solution generation.