THE NEXT CHALLENGE FOR AGENTIC CYBERSECURITY: A REALISTIC, CONTAMINATION-FREE REVERSE ENGINEERING BENCHMARK

Anonymous authors
Paper under double-blind review

ABSTRACT

AI agents are rapidly improving in cybersecurity capabilities when the *source code* is available for analysis, yet much of the software most consequential to cybersecurity, including malware, firmware, and proprietary applications, is available only as *binaries*. Analyzing such software requires *reverse engineering* (RE): recovering program semantics before the analysis can be meaningfully performed. However, evaluating agentic RE poses a fundamental challenge: realistic and useful benchmark instances must (1) be absent from LLMs' training data to prevent models from taking shortcuts by memorization, and (2) represent the scale and anti-analysis protections of real-world binaries. To faithfully evaluate the RE capability of frontier models, we introduce **SRE-Bench**, the first realistic, contamination-free RE benchmark. Built entirely from scratch by RE experts with over 5,000 expert hours, SRE-Bench comprises 19 private, real-world-scale programs averaging 16.9K lines of code. We further developed 44 in-house anti-analysis mechanisms, yielding 262 binary instances and 1,572 deterministically graded tasks. We evaluated 13 agentic settings across 11 models: eight in public-facing settings and five in internal unconstrained settings with cyber safeguards disabled and no budget cap. Our results show that realistic RE remains challenging for frontier agents: GPT-5.6-Sol and Claude-Fable-5.1, despite their strong source-code cybersecurity capabilities, fully solve only 31.5% and 26.9% of graded instances, respectively. This result suggests that success in source-code security does not necessarily translate into effective binary analysis. Without a budget cap and safety guard, GPT-6-Astra achieves a near-perfect pass@4 score; however, reliably identifying the correct candidate still poses a significant challenge. Our analysis further reveals that agents are relatively insensitive to compiler optimization and static linking, and controlled ablations confirm that both contamination control and realistic scale are essential to understand agents' RE capability. Together, these findings highlight RE as a distinct frontier for agentic cybersecurity and establish SRE-Bench as a rigorous testbed for measuring progress.

1 INTRODUCTION

Much of the software most consequential to cybersecurity reaches analysts only as *binaries, without source code*. This is true at both sides of the threat landscape. On the defensive side, the high-value systems (*e.g.*, proprietary enterprise software, security appliances, and firmware) are frequent attack targets but are typically distributed only in binary form. For example, according to the authoritative catalog (CISA, 2026), 46.5% of vulnerabilities exploited in the wild come from vendors that do not release source code. Google similarly reports that more than 48% of the zero-days in 2025 targeted proprietary enterprise software (Google, 2026). On the offensive side, attackers deliberately distribute malicious payloads as obfuscated binaries to impede analysis, with approximately 732,000 new malicious samples reported each day (van Liebergen et al., 2023). Anthropic also makes the distinction between binary exploitation and source exploitation as two distinct capabilities (Anthropic, 2026b). As autonomous cybersecurity agents advance, binary software becomes an essential and urgent evaluation target to understand the state-of-the-art model capability.

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.

Unlike structured source code, binary code is represented as raw bytes and is not directly intelligible to analysts. Over the past several decades, cybersecurity researchers have developed *reverse engineering* (RE) (Shoshitaishvili et al., 2016; Lee et al., 2011; Balakrishnan et al., 2005; Song et al., 2008) to recover high-level program semantics from this opaque representation. RE now forms the foundation of binary-first security analysis: before analysts can reason about vulnerabilities, patches, or exploits, they must first determine what a binary program does. Notably, RE requires additional capabilities beyond those required in downstream cybersecurity tasks (Mantovani et al., 2022; Votipka et al., 2020); it entails interpreting low-level machine semantics, inferring intent from incomplete evidence, and, in harder cases, overcoming adversarial obfuscation and packing that intentionally obscure the binary code. To extend agentic cybersecurity beyond high-level source code, AI agents must therefore learn how to tackle RE and be evaluated on it as a distinct capability.

However, constructing an RE benchmark that faithfully reflects real-world practice is challenging.

An RE benchmark is valid only if its targets are unknown to the model. The goal of RE is to recover the semantics of *unknown binaries* (*e.g.*, malware or proprietary software). Analysts therefore begin with no prior knowledge of the target. Benchmarks built from publicly available source code break this premise, because the target may already appear in pretraining data and be recognizable during evaluation (Al-Kaswan et al., 2024; Jain et al., 2025; OpenAI, 2026b). Once a model recognizes the target, its prior knowledge (*e.g.*, the target's high-level purpose) can help bypass much of the program-understanding process. Note that this risk differs from contamination in many other cybersecurity tasks, where only exact, fine-grained leakage is concerning (Ding et al., 2024). In RE, even a coarse degree of leakage is damaging (Siegmund, 2016), as it provides crucial, top-down guidance for program interpretation (Biggerstaff et al., 1993). Our empirical results in §4 further confirm that such data contamination can inflate RE task performance.

An RE benchmark must also match the scale and protection of real-world targets. Recent work shows that cybersecurity benchmarks built from CTF-style challenges or toy programs fail to predict agent performance on real-world targets (Wang et al., 2026; Zhang et al., 2026). Evaluating RE tasks can easily fall into the same pitfall. The complexity of binary analysis arises from two sources: 1) the behavior of *target program* and 2) the *protection layer* that gates access and obscures understanding of proprietary software. At the program level, real-world software can span thousands, if not hundreds of thousands, of lines of code. Because RE is fundamentally a program-understanding task, the combinatorial complexity of program behaviors grows exponentially with the scale (Wettel et al., 2011); capability measured on toy programs therefore does not extrapolate (Rugaber, 1995). At the protection level, high-value targets are often wrapped in multiple anti-analysis layers, such as sophisticated and sometimes bespoke obfuscation (Cheng et al., 2021). A faithful benchmark must therefore pair real-world-scale programs with protection beyond textbook schemes.

Together, these requirements rule out some straightforward construction strategies one may expect: Using open-source projects provides real-world artifacts, yet to control contamination, we must exclude them. Applying local modifications to such projects is also insufficient, because their high-level architecture may remain recognizable. As a result, to faithfully evaluate RE capability of frontier models, we must construct binary programs with real-world complexity entirely from scratch, albeit it can be highly time-consuming and expertise-intensive. To this end, we have invested over 5,000 domain-expert hours in developing a clean-room RE benchmark. To our knowledge, no prior RE benchmark has demanded a comparable investment of expert effort.

SRE-Bench: The First Realistic, Contamination-Free RE Benchmark. We developed 19 in-house programs, averaging more than 16,915.8 lines of code (LoC), across five representative RE domains: network protocols, firmware, games, format parsers, and malware. These domains correspond to the major categories identified by a recent survey of reverse-engineering discussions on Stack Exchange (Kabir et al., 2026). We also built a comprehensive anti-analysis suite with 44 state-of-the-art protection primitives, over half of which have no public implementations. The suite itself required over 27K LoC to implement. By pairing these private programs with in-house protection settings, SRE-Bench yields **262 contamination-free instances with realistic difficulty**. Each instance defines six deterministically verifiable tasks. In total, SRE-Bench provides 1,572 RE tasks.

Experimental Results Reveal Persistent Challenges in Agentic RE. We evaluate frontier agentic LLMs under both standardized public and model-provider-internal configurations. Most evaluated agents struggle with realistic RE: despite their strong source-code security capabilities (OpenAI,

Table 1: Comparing SRE-Bench with existing, end-to-end RE benchmarks for AI agents. **# Instances** denotes the number of RE instances. **RE-Centric** indicates whether the benchmark is primarily designed for RE. **Cont. Ctrl.** denotes control for program-identity contamination. **Prog. Cplx.** summarizes program complexity, and **LoC** reports the average lines of code per instance for non-CTF benchmarks. **Prot. Cplx.** summarizes the complexity of binary-protection mechanisms.

Benchmark	# Instances	RE-Centric	Cont. Ctrl.	Prog. Cplx.	LoC	Prot. Cplx.
NYU CTF Bench (Shao et al., 2024)	51	✗	✗	Existing CTF	–	Easy
Cybench (Zhang et al., 2025)	6 (15 *tasks*)	✗	✗	Existing CTF	–	Easy
CTF-Dojo (Zhuo et al., 2025)	123	✗	✗	Existing CTF	–	Easy
CTFTiny (Shao et al., 2026)	16	✗	✗	Existing CTF	–	Easy
CREBench (Chen et al., 2026)	432 (1728 *tasks*)	✓	△	Toy Program	526.2	Easy
AgentRE-Bench (AgentRE-Bench, 2026)	13 (70 *tasks*)	✓	✓	Toy Program	63.5	Textbook
CrackMeBench (David & Gervais, 2026)	12	✓	✓	Toy Program	36.1	Textbook
SRE-Bench (Ours)	262 (1572 *tasks*)	✓	✓	**Real-World Software**	**16,915.8**	**Real-World**

2026a; Anthropic, 2026a), GPT-5.6-Sol and Claude-Fable-5.1 fully solve only 31.5% and 26.9% of graded instances, respectively, in the standardized evaluation. These results suggest that success in source-code security does not necessarily translate into effective binary analysis. GPT-6-Astra demonstrates substantial progress, achieving 99.2% pass@4 under the model-provider-internal evaluation, but selecting or validating the correct solution still remains an important open challenge.

Our in-house protections further expose substantial differences in binary-analysis capability: with the build held fixed, they reduce GPT-5.6-Sol's mean score from 4.69 to 2.50 out of 6 and drive weaker models to near zero. GPT-6-Astra largely maintains its score, but incurs 1.5× the monetary cost per point on protected binaries while weaker models incur up to 35.3× cost. The failure modes also differ from those of human analysts. Optimization and linking, the classical obstacles of manual RE, barely register, whereas stripping symbols is costly, suggesting that current agents lean on lexical anchors more than on instruction-level reasoning. Together, these findings establish SRE-Bench as a testbed for advancing both binary understanding and reliable solution validation.

Our Contributions. This work makes four contributions. First, we identify RE as the next frontier for agentic cybersecurity capability, and argue that the field must move beyond source-code benchmarks to measure agents on binary-first workflows. Second, we show empirically that both program complexity and data contamination substantially affect RE evaluation, establishing them as essential design criteria rather than optional refinements. Third, we present SRE-Bench, the first RE benchmark that addresses both requirements: 262 instances built from scratch at the real-world scale (averaging 16,915.8 LoC), paired with a 27K-LoC anti-analysis suite implementing 44 in-house protection primitives. Lastly, we give the first systematic characterization of agentic RE capability in a real-world setting with SRE-Bench. Our evaluation on state-of-the-art AI agents reveals that strong source-code security capabilities do not necessarily translate into effective binary analysis.

2 Related Work

Source-Code Security Benchmarks for AI Agents. A broad line of work evaluates AI agents on source-code security tasks. Chen et al. (2023); Liu et al. (2024); Yildiz et al. (2025) introduce vulnerability-detection benchmarks that measure whether models can identify flaws across entire codebases. Wang et al. (2025); Lee et al. (2026b); Pu et al. (2026) introduce PoC-generation benchmarks that evaluate whether agents can reproduce vulnerabilities using concrete inputs. Zhu et al. (2025); Wang et al. (2026); Lee & Brumley (2026) further propose exploit-development benchmarks that assess whether agents can construct working exploits. Other security-oriented software-engineering benchmarks evaluate complementary code-security workflows (Nie et al., 2026; Lee et al., 2026b;a; Wang et al., 2025). SRE-Bench instead targets the missing bridge to binaries, testing whether AI agents' success on source code extends to broader security workflows.

Traditional RE Benchmarks. Many RE benchmarks evaluate specific intermediate artifacts, such as function and variable names (Koller et al., 2026), types (Won et al., 2026; Soni et al., 2025), and decompiled source code (Tian et al., 2026; Gao et al., 2025). However, artifact-level accuracy does not fully capture end-to-end program understanding: an agent may correctly infer program behavior or identify vulnerabilities without recovering the original names, types, or source code.

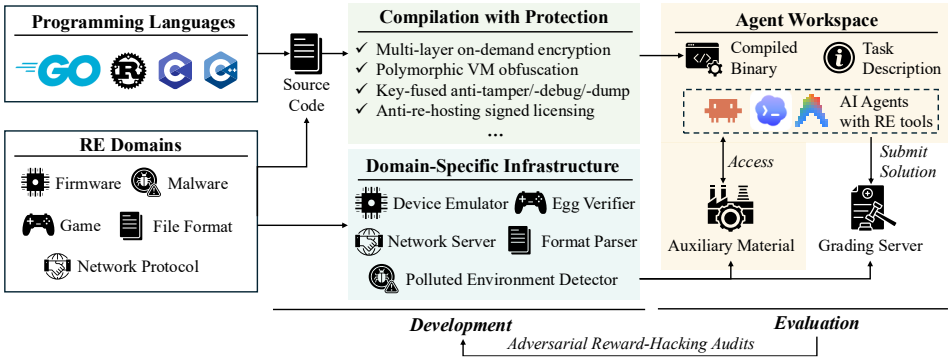


Figure 1: Overview of the SRE-Bench construction and evaluation pipeline. RE experts develop each program and its evaluation infrastructure from scratch against a private specification. Programs are compiled into diverse build variants and optionally hardened by our in-house protection suite, yielding contamination-free binary instances. At evaluation time, the agent receives a binary plus domain-specific auxiliary materials, then submits a solution to a deterministic grader. The same evaluation pipeline is reused during development to run adversarial reward-hacking audits (Opus 4.8 and GPT-6-Astra), whose findings drive iterative hardening of both the programs and infrastructure.

Binary-Based Software Engineering Benchmarks. ProgramBench (Yang et al., 2026) gives agents the compiled binary of public open-source projects and their documentation and asks them to reimplement the program's behavior without decompilers (Anthropic, 2026b;c). The targets are well-known open-source programs (*e.g.*, `jq`, `SQLite`, `FFmpeg`) whose purpose is disclosed through provided documentation. This measures behavioral reimplementation rather than semantic recovery based purely on the binary, which is what SRE-Bench targets.

End-to-End RE Benchmarks for AI Agents. Closer to our setting, several benchmarks evaluate AI agents on RE tasks, but they still diverge sharply from real-world practice, as summarized in Table 1. Among them, CTF-derived benchmarks (Shao et al., 2024; Zhang et al., 2025; Zhuo et al., 2025; Shao et al., 2026) include RE tasks, but only as one subcategory. Because these benchmarks are drawn from existing CTF competitions whose intended solutions are often publicly available, they carry substantial risk of data contamination. Moreover, most of their collected challenges are entry-level, which fail to capture real-world complexity in either target programs or binary protections.

A second line of work focuses specifically on RE tasks and constructs benchmark programs from scratch to reduce data-contamination risk. However, these benchmarks are limited to toy programs. CREBench (Chen et al., 2026) targets crypto-oriented RE and contains 432 instances generated from 48 small programs, averaging only 526.2 LoC per program. Meanwhile, it still carries contamination risk, as 39 of its 48 programs are slight modifications of open-source projects. Its obfuscation is rudimentary as well, limited to naive string-XOR obfuscation. AgentRE-Bench (AgentRE-Bench, 2026) and CrackMeBench (David & Gervais, 2026) provide stronger contamination control by using 13 and 12 self-developed programs, respectively. However, they are even smaller in scale, averaging only 63.5 and 36.1 LoC per instance. Their binary protections also remain textbook-level, such as simple keygens and miniature virtual machines. For example, AgentRE-Bench implements all protection logic in only 376 lines of C code. In contrast, SRE-Bench is designed to evaluate agentic RE under substantially more realistic conditions while maintaining strict contamination control. Its programs are developed from scratch at real-world scale, averaging 16,915.8 LoC per instance. Its protection layer is a comprehensive anti-analysis suite built with over 27K LoC, incorporating many techniques widely used in commercial obfuscators but lacking mature public implementations.

3 BENCHMARK

Figure 1 illustrates the construction pipeline of SRE-Bench. The benchmark is built entirely in-house: RE experts develop the target programs and their evaluation infrastructure from scratch, compile and optionally harden them into diverse binary instances, and score every submission with a deterministic grader. *Adversarial reward-hacking audits* further close the loop by pitting agents against in-progress instances under the real grader and patching every shortcut that earns credit with-

Table 3: Overview of RE domains. **Target** is what the program does. **RE Task** is what the agent must accomplish. **Infrastructure** is the component that makes evaluation deterministic. **Auxiliary** is what the agent receives beyond the binary. **# Programs** reports the number of unique programs in each domain; their implementation languages and Loc can be found in Table 2.

Domain	Target	RE Task	Infrastructure	Auxiliary	# Programs
Network Protocol	A client-server stack for a proprietary, encrypted network protocol	Recover the wire format and drive a client through the full protocol state machine	Server that scores the protocol's state-machine coverage of the client	5 partial packet captures	4
Game	A playable 20-floor terminal roguelike dungeon crawler	Trigger hidden behaviors (Easter eggs) unreachable through normal play	An egg verifier that replays the agent's action trace	Player manual	4
File Format	A file/directory compressor producing a proprietary archive format	Reverse the encoder and decode the provided archives byte-exactly	A held-out decoder that round-trip verifies the developed encoder	6 challenge files and corresponding passwords	4
Malware	A synthetic implant that exhibits malicious behavior without causing real harm	Reverse the implant's effects while preserving benign user data	A sandboxed infection-and-cleanup grader	–	4
Firmware	Bare-metal firmware for a locked-down secure microcontroller	Take progressively fuller control of the device through its interface	A held-out emulator that models the chip and its peripherals	Serial console on a remotely hosted emulator	3

out requiring genuine RE. We deliberately confine SRE-Bench to reverse engineering and exclude downstream cybersecurity tasks such as exploit development, for two reasons: 1) including such tasks would conflate an agent's RE ability with its separate cybersecurity skill, and 2) reverse engineering is a distinct capability that merits measurement on its own (Mantovani et al., 2022; Votipka et al., 2020). The following subsections detail the RE domains and the protection suite.

Table 2: LoC per program (domain×language). Firmware omits Go, whose runtime makes it unsuitable for bare-metal embedded targets.

Domain	C	C++	Rust	Go	Total
Network Protocol	17,513	15,754	18,052	29,346	80,665
Game	19,922	19,208	19,692	22,873	81,695
File Format	16,894	12,291	12,732	21,144	63,061
Malware	14,610	12,972	16,726	22,897	67,205
Firmware	8,874	8,892	11,009	–	28,775
Total	77,813	69,117	78,211	96,260	321,401

Development Methodology. All artifacts (*i.e.*, the target programs, the evaluation infrastructure, and the protection suite) were developed by RE experts with an average of six years of experience. Each program was independently implemented from scratch based on a private design specification that is never released. The 19 programs span four implementation languages (*i.e.*, C, C++, Rust, and Go), follow language-idiomatic programming patterns, and use distinct library stacks (*e.g.*, `ncurses` vs. `tcell` vs. `ratatui`). No two programs share source code, and none is derived from a public project. Every program ships with a private reference solution that scores a perfect $6/6$, which serves as both a solvability proof and a regression gate. We also run iterative adversarial reward-hacking audits: reusing the evaluation pipeline, we pit agents against in-progress instances to surface shortcuts. For example, an early build retained the internal name of a hidden behavior (*i.e.*, the target of an RE task) as a readable string in the binary; an agent therefore located the corresponding trigger through string scanning without doing any actual RE. We removed the string and added a regression check that fails the build on any similar leak. In total, SRE-Bench comprises over 320K lines of code (Table 2), averaging 16,915.8 LoC per program; it is 30-470× larger than prior RE benchmarks.

Benchmark Domains. SRE-Bench contains 262 instances, spanning five domains that together cover the breadth of real-world RE practice (Kabir et al., 2026; Eilam, 2011): *network protocol*, *game*, *format parser*, *malware*, and *firmware*. Table 3 summarizes the characteristics of each domain. More detailed domain descriptions can be found in Appendix A.

Protection and Obfuscation High-value RE targets are rarely presented as clean, unprotected binaries. To model this, SRE-Bench includes an in-house binary protection suite that transforms an unprotected Linux ELF into a hardened, functionally equivalent variant. The suite is implemented entirely from scratch, comprising over 27K lines of Python, C, and assembly, which keeps the protection layer as contamination-free as the programs it protects. It provides 44 distinct protection primitives organized into nine technique families. These primitives can be composed to

construct stronger, layered protections, including: (1) *Obfuscation and string deception* (5 primitives). (2) *Per-page authenticated encryption* (4 primitives). (3) *Lazy decryption and residency minimization* (5 primitives). (4) *Measurement-keyed anti-debugging* (8 primitives). (5) *Self-checksumming and anti-tamper coupling* (3 primitives). (6) *Online licensing with per-run key delivery* (4 primitives). (7) *Loader-logic virtualization* (6 primitives). (8) *Anti-dump detection and deception* (5 primitives). (9) *Anti-re-hosting* (4 primitives). To our knowledge, more than half of these primitives, primarily in families (3), (4), (7), (9), and (8), have no publicly available implementations. Note that our implementation draws on ideas from commercial obfuscators but use independently designed algorithms and parameters. More details can be found in Appendix B.

Instance Generation We compile each of the 19 programs and, where applicable, harden it with the suite, yielding 262 binary instances. For each of the 16 programs in the four domains (*i.e.*, network protocol, game, format parser, malware), we build 8 unprotected instances that cover all combinations across three compilation axes (*i.e.*, optimization, symbol stripping, and static vs. dynamic linking) plus 8 protected instances (*i.e.*, one per protection preset, details in §B.9). Firmware is different: it is bare-metal, so our protection suite (which relies on the Linux loader and page-fault handlers) does not apply; instead the firmware ships its protection in-binary (encrypted boot, bespoke crypto, on-device VMs, and decoy peripherals). Three firmware programs vary only by optimization level. Every binary instance defines six deterministically scored tasks. As a result, SRE-Bench provides 1,572 RE tasks in total.

Contamination Control Deserves Special Note. Because this paper may itself enter future training corpora, we describe each domain in sufficient detail to support the paper's analysis and give readers a clear understanding of each domain, while limiting those details to information that an analyst could recover from the binary with minimal effort. [1] We withhold all instance-specific secrets. [2]

4 EVALUATION

Models and Harness. We evaluate models in two settings: *public-facing* and *unconstrained*, the latter not available to customers. For the *public-facing* setting, we evaluated the eight models (Table 4) with the model-side cyber safeguards enabled when available. Each runs inside a standardized mini-SWE-agent, so that differences in scaffolding do not confound the comparison. Every model is queried at its highest reasoning-effort setting, and each run is capped at 500 model steps and 6 hours of wall-clock time. The *Unconstrained* setting (Figure 4) evaluated three OpenAI models and two Anthropic models at the providers' default settings, using Codex for OpenAI models and Claude Code for Anthropic models, without cyber safeguards and evaluation budget limits. This internal evaluation comprises 16,768 attempts and 2.37B output tokens, corresponding to a token-equivalent cost of roughly $566,000 (Table 8).

Every problem instance is solved in an isolated container holding the target binary, the domain-specific auxiliary materials, a set of standard RE tools (*i.e.*, Ghidra and `pyghidra`, radare2, GDB, angr, binutils, `strace`, and `ltrace`) and per-domain tools (*i.e.*, Including `tshark`, `pwntools`, `unicorn`, and `z3`). Grader code, reference solutions, and any development artifacts are removed from the container, and agents receive no scoring feedback during a run.

Metrics. We report the following metrics: *Score*, the mean of successful tasks per binary (out of six) across graded binaries; *Solved*, the number of binary instances in which all six tasks pass; and *Zeros*, the number of instances that earn no credit at all. The *Solved* and *Zeros* metrics stand at the two extremes of the rubric, and nearly half of each model's scores land at one end of these extremes. Together, they provide a more informative characterization of agent capability than the mean alone. We also report per-instance number of API calls, cost, and sandbox wall-clock time. All averages are computed over gradable runs, excluding cyber guardrail refusals and failures due to limited context window. For the unconstrained evaluation, we report pass@1 and pass@4 from $n = 4$ attempts

[1] We further verified that all information disclosed in §3 and §A could be recovered from the binaries by Codex GPT-4 mini within 200 LLM requests. We masked program-specific details in each paragraph, asked the agent to reconstruct them, and manually checked the results.

[2] Throughout the paper, we omit exact constants, trigger conditions, hidden-behavior mechanisms, and reference solutions. We report only what an agent could readily obtain from the binary. Where we name a critical algorithm (*e.g.*, a cryptographic primitive), we substitute one of similar functionality and complexity. We deliberately withhold the actual choice. SRE-Bench therefore remains contamination-free in practice.

Table 4: **Overall results on SRE-Bench**. *#Graded* is the number of runs that produced a gradable result; the remainder failed either due to guarded-model refusal or context-window limits; most context-window failures occurred with GROK-4.5. Only graded instances are used to compute the metrics. Arrows indicate: ↑ higher is better, ↓ lower is better.

Model	#Graded	Capability			Cost		
	out of 262	Score ↑ out of 6 (%)	Solved ↑ # (% of graded)	Zeros ↓ # (%)	Calls ↓ (#)	Cost ↓ ($)	Time ↓ (min)
GPT-6-ASTRA	181	**5.55**±0.09 (92.4)	**149** (82.3)	**7** (3.9)	61	13.5	18
GPT-5.6-SOL	254	3.69±0.14 (61.4)	80 (31.5)	41 (16.1)	196	42.5	88
CLAUDE-FABLE-5.1	223	2.58±0.18 (43.0)	60 (26.9)	96 (43.0)	173	34.7	123
CLAUDE-OPUS-5	256	1.91±0.14 (31.8)	32 (12.5)	118 (46.1)	139	23.6	81
GPT-5.5	262	1.02±0.10 (17.0)	10 (3.8)	148 (56.5)	128	17.8	46
DEEPSEEK-V4.1-FLASH	262	0.85±0.09 (14.2)	2 (0.8)	166 (63.4)	182	0.5	61
GROK-4.5	218	0.45±0.07 (7.6)	2 (0.9)	160 (73.4)	115	13.5	66
GLM-5.2	262	0.21±0.03 (3.4)	0 (0.0)	217 (82.8)	143	26.6	149

per instance using the unbiased estimator of Chen et al. (2021), under which pass@1 is the mean full-solve rate across attempts and pass@4 the fraction of instances solved by at least one attempt.

Results Overview. Table 4 shows that realistic RE remains challenging for most publicly evaluated agents: GPT-5.6-Sol and Claude-Fable-5.1, despite their strong source-code security capabilities (OpenAI, 2026a; Anthropic, 2026a), fully solve only 80/254 (31.5%) and 60/223 (26.9%) graded instances, respectively, with mean scores of 3.69 and 2.58 out of 6. The weakest model scores 0.21 and does not fully solve a single challenge. The *Zeros* metric further characterizes these limitations: GPT-5.6-Sol earns no credit on 16.1% of graded instances, compared with 43.0% for Claude-Fable-5.1, 46.1% for Claude-Opus-5, and 56.5–82.8% for the four lowest-scoring models. These results reinforce the need to evaluate binary analysis as a distinct capability rather than infer it from success on source-code security tasks.

In contrast, GPT-6-Astra leads on SRE-Bench, averaging 5.55 out of 6 and fully solving 149 of 181 gradable instances (82.3%); only 3.9% of gradable instances receive no credit. Although the tasks are not exploitation-oriented, frequent refusals on cyber safety grounds leave GPT-6-Astra with the fewest gradable instances (181 of 262, 69.1%) in the public setting. This raises concerns about its practical utility for RE, and it remains a grand challenge for vendors to balance safety and capability. All these scores are conditional on a gradable result and should not be interpreted as success rates over all 262 instances.

Finally, resource use does not predict capability. GPT-6-Astra is both the strongest model and the cheapest per instance, consuming 61 API calls and $13.5 per instance while GPT-5.6-sol performs worse and costs $42.5 per instance. GLM-5.2 spends $26.6 per instance for a score of 0.21.

	Source language				Target domain				
	C	C++	Rust	Go	Proto	Fmt	Game	Malw	Firm
GPT-6	5.22	5.44	5.62	5.89	4.84	6.00	n/g	5.72	n/g
GPT-5.6	4.24	3.64	3.43	3.39	4.88	4.16	3.57	2.06	6.00
Fable-5.1	2.68	2.73	2.39	2.54	3.05	3.08	2.69	0.19	1.67
Opus-5	2.32	1.88	1.52	1.91	3.00	1.61	2.28	0.34	4.67
GPT-5.5	1.56	1.09	0.68	0.75	1.42	0.81	1.34	0.31	3.17
DS-V4.1-F	1.33	0.79	0.61	0.69	1.59	0.42	0.91	0.45	1.33
Grok-4.5	0.83	0.49	0.27	0.17	0.58	0.81	0.50	0.00	2.00
GLM-5.2	0.38	0.18	0.12	0.14	0.48	0.08	0.20	0.00	0.83

Figure 2: Mean score (of 6) by source language and target domain, over graded runs (n/g: none graded). Darker is higher.

Target Domains vs Source Languages. Figure 2 breaks down mean scores for public models by source languages and target domains. On the source language side, we found that most models perform better on binaries compiled from C than ones compiled from Rust and Go. We partially attribute this advantage to tooling and training data bias: decompilers commonly produce C-like output, whereas Go and Rust binaries introduce heavier runtime machinery and language-specific idioms that can complicate analysis. GPT-6-Astra achieves the highest mean score among evaluated models in every source language, but scores highest on Go and lowest on C.

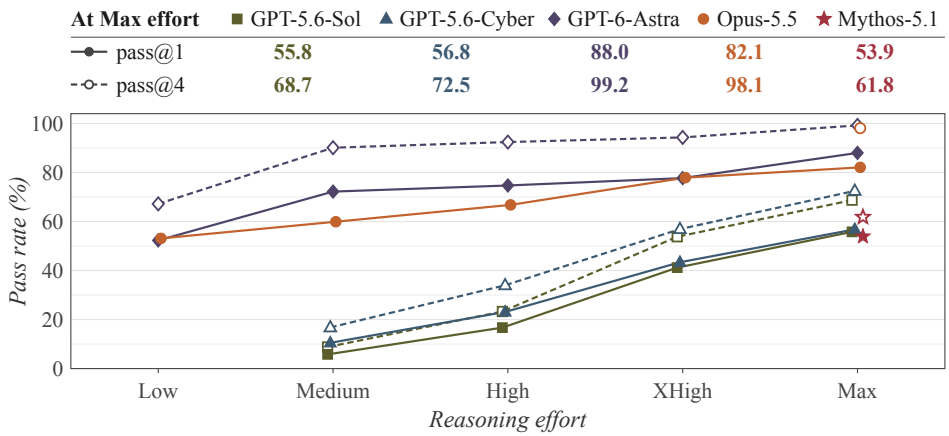


Figure 4: Pass@1 (solid) and pass@4 (dashed) of the unconstrained models by reasoning effort. Opus-5.5 at Low–XHigh ran once per instance, so its pass@4 is shown only at Max. GPT-5.6-Sol and GPT-5.6-Cyber were not run at Low, and Mythos-5.1 was run only at Max. All values are in Tables 11 and 12.

The target domain is a stronger indicator of model RE capability. Network protocol and firmware generally observe higher scores, although firmware contains only six instances. Malware is the lowest-scoring domain for most models. Its strict grading may contribute to this difficulty: an agent must extricate malicious behavior and preserve grader-planted benign state, so an incomplete solution earns no credit. Several weaker models achieve near-zero scores on malware while retaining higher scores on protocol and game tasks. These differences show that aggregate scores can conceal substantial variation in an agent's ability to complete different RE tasks.

Build Configurations and Protection Mechanisms. Figure 3 breaks down mean scores by build configurations and protection mechanisms, revealing a pattern that departs from the conventional ordering of RE difficulty. Optimization and linking, the primary obstacles for a human analyst, are nearly transparent to the four strongest models. Weaker models lose 17–41% under optimization, but their dominant build configuration is symbol stripping, which removes 50–71% of the score of the four weakest models, versus 10–15% for GPT-5.6-sol, Claude-Fable-5.1, and Claude-Opus-5, and nothing for GPT-6-Astra (5.39 vs. 5.36). Weaker models are thus far more sensitive to symbol stripping than stronger ones, suggesting that they rely more heavily on names as anchors for program understanding. Protection is the one classical obstacle that transfers, but its effect depends sharply on the model's capability. Hardening cuts GPT-5.6-sol's score from 4.69 to 2.50 and reduces every other model besides GPT-6-Astra to at most 0.33. Claude-Fable-5.1 falls from 5.33, second only to GPT-6-Astra without hardening, to 0.32 (see §B.9 for results by preset). GPT-6-Astra, by contrast, unpacks consistently and loses only 0.10 points. However, analyzing protection increases its cost, at 1.5× the dollars per point of unpacked instances, versus 3.4× for GPT-5.6-sol and 9–35× for weaker models (Table 7).

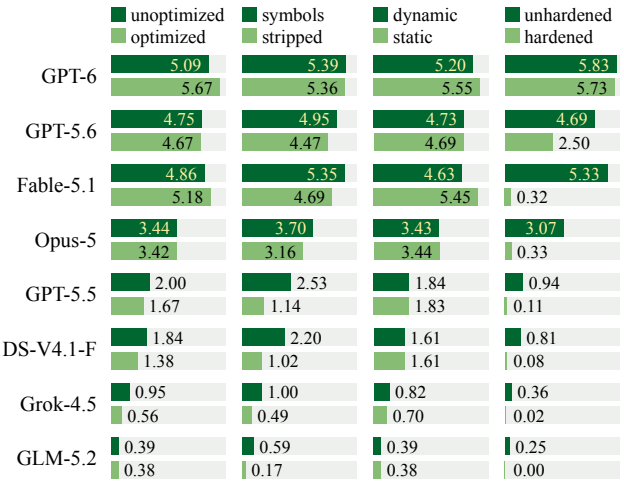


Figure 3: Mean score (of 6) per model for each build factor. The first three panels use the 128 unhardened instances; the last holds the build fixed (opt, stripped, static) and varies only protection.

Unconstrained Evaluation and Solution Selection. Figure 4 reports the strongest OpenAI-model results and the clarified-prompt Anthropic-model results, without public safeguards or evaluation budget limits. GPT-6-Astra achieves 88.0% pass@1 and 99.2% pass@4; Opus 5.5 reaches 82.1% and 98.1% under its internal configuration. Reasoning effort matters: GPT-5.6 Sol improves from 5.8% pass@1 at Medium to 55.8% at Max. The full results and prompt comparison are in §C.1 and §C.2. High pass@4 means the private grader finds a correct solution among four attempts; practical use requires choosing or validating one without that grader. For GPT-6-Astra, 69 instances have both successful and unsuccessful attempts, and 126 of its 1,048 attempts fail to obtain full credit. Selecting the correct submission remains an open evaluation question, since these runs do not test a selector.

Ablation: Contamination Control and Program Complexity. Table 5 directly tests the two design criteria of §1, with eight unhardened build configurations fixed and varying only the target program. Sufficient scale is necessary to accurately measure RE capabilities: *Minimal*, a clean-room compressor of about 1.1k LoC, is fully solved by both models in every configuration for less than $1.60 and seven minutes. Therefore, being clean-room provenance by itself does not make a target hard. *Gzip-variant*, a deployed encoder with only about 200 lines of modified code, is also solved in all eight configurations for roughly $2 and ten minutes, as most of its source is publicly available in training corpora. Only REVCOMPRESS, which combines original code with real world scale, separates the models: GPT-5.6-sol scores 5.62 and solves $7/8$ configurations, versus 3.75 and $3/8$ for GPT-5.5, at 10 to 30 times the cost and time.

Table 5: **Complexity and contamination ablations.** The same eight unhardened build settings of one compression task, over three programs.

Program	Model	Score ↑	Solve ↑	Cost ↓	Time ↓
Minimal	GPT-5.6-SOL	6.00	8/8	0.90	4.5
	GPT-5.5	6.00	8/8	1.60	7.0
Gzip-variant	GPT-5.6-SOL	6.00	8/8	2.19	9.9
	GPT-5.5	6.00	8/8	2.78	10.8
RevCompress	GPT-5.6-SOL	5.62	7/8	31.45	94.6
	GPT-5.5	3.75	3/8	24.37	97.4

To conclude the ablation study, a large-scale program built largely from public code can be as easy as a small private one, allowing recognition to stand in for analysis. By controlling both source exposure and program complexity, SRE-Bench contributes a rigorous test of RE capability with meaningful difficulty.

5 DISCUSSION AND CONCLUSION

Limitation: Benchmark Breadth. SRE-Bench is built from 19 programs. This is a small pool in absolute terms. This limitation follows directly from our design requirements. To remain contamination-free, every target program, its evaluation infrastructure, and the protection layer must be developed from scratch; doing so required more than 5,000 expert hours. Our ablation further shows why cheaper alternatives are inadequate: a program derived from public code can be recovered for roughly $2 in ten minutes, while small clean-room programs are solved nearly perfectly (Table 5). Despite the limited program count, SRE-Bench contains over 320K lines of authored code, averaging 16,915.8 LoC per program, together with a 27K-LoC protection suite. These programs produce 1,572 deterministically scored tasks. This scale is already sufficient to expose large capability differences: across the eight public models, mean scores differ by more than $25\times$.

Limitation: Program Size. Our programs average 16,915.8 LoC and thus do not cover the largest real-world software systems, which can reach hundreds of thousands or millions of lines of code. Extending to that scale faces the same constraint as expanding benchmark breadth: contamination-free programs must be developed from scratch, while development costs grow substantially with program size. Nevertheless, our programs are already 30–$470\times$ larger than those used in prior RE benchmarks. We therefore view larger programs as an important direction for extending SRE-Bench.

Conclusion. We presented SRE-Bench, the first RE benchmark that is at once contamination-free and built at real-world scale. Most frontier agents still struggle with it. GPT-5.6-Sol and Claude-Fable-5.1 fully solve only 31.5% and 26.9% of graded instances, and our anti-analysis suite halves the former while reducing every weaker model to near zero. Agentic failure patterns also differ from human ones, as optimization and linking have little effect while stripping symbols is costly. Together these results suggest that strong source-code security performance is not yet a reliable indicator of binary-level capability, and that the gap is wide enough to be worth measuring on its own terms.

AI USE STATEMENT.

Generative AI systems are a central object of study in this work and are extensively used as evaluated agents in our experiments. We also use AI tools for limited assistance with writing, code development, and debugging. All benchmark design decisions, implementations, experimental procedures, analyses, and conclusions are carried out by the authors. The authors remain fully responsible for the content of the paper and the correctness of all reported results.

ETHICS STATEMENT.

This work studies AI capabilities for reverse engineering, which has legitimate applications in vulnerability analysis, malware inspection, and software auditing, but can also have dual-use implications. To reduce risks to real systems, the benchmark is constructed from programs developed in-house rather than vulnerabilities or binaries collected from deployed third-party software. Experiments are conducted in isolated environments, and the benchmark does not require interaction with external systems or services. We believe that systematically measuring these capabilities is important for understanding both the security benefits and the potential risks of increasingly capable cybersecurity agents.

REPRODUCIBILITY STATEMENT.

To preserve the contamination resistance of SRE-Bench, we do not publicly release the benchmark binaries or task instances. Instead, we provide an open evaluation service through which researchers can request evaluation of their models or agents; we run the experiments in the same controlled environment and report the resulting metrics. The paper and appendices document the benchmark construction process, protection configurations, evaluation harness, model settings, resource limits, and metrics in detail. This setup enables consistent comparison of future models on SRE-Bench while preventing public exposure from undermining the validity of subsequent evaluations.

REFERENCES

AgentRE-Bench. AgentRE-Bench: Llm reverse engineering benchmark. https://github.com/agentrebench/AgentRE-Bench, 2026.

Ali Al-Kaswan, Maliheh Izadi, and Arie Van Deursen. Traces of memorisation in large language models for code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–12, 2024.

Anthropic. Claude fable 5 and claude mythos 5. https://www.anthropic.com/news/claude-fable-5-mythos-5, June 2026a. Accessed: 2026-07-11.

Anthropic. System card: Claude Opus 5.5. https://www-cdn.anthropic.com/fc1b44717c85dc068bc6ba5024219938094694bd/Claude%20Opus%205.5%20System%20Card.pdf, September 2026b. Accessed: 2026-09-24.

Anthropic. System card: Claude Sonnet 5. https://www-cdn.anthropic.com/283ef97c476cf442c91d9a37d5b214242a55bb92/Claude%20Sonnet%205%20System%20Card.pdf, June 2026c. Accessed: 2026-09-24.

Gogul Balakrishnan, Thomas Reps, David Melski, and Tim Teitelbaum. Wysinwyx: What you see is not what you execute. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pp. 202–213. Springer, 2005.

Ted J Biggerstaff, Bharat G Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *[1993] Proceedings Working Conference on Reverse Engineering*, pp. 27–43. IEEE, 1993.

Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation Systems Research Center, 1994.

Baicheng Chen, Yu Wang, Ziheng Zhou, Xiangru Liu, Juanru Li, Yilei Chen, and Tianxing He. Crebench: Evaluating large language models in cryptographic binary reverse engineering. *arXiv preprint arXiv:2604.03750*, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th international symposium on research in attacks, intrusions and defenses*, pp. 654–668, 2023.

Binlin Cheng, Jiang Ming, Erika A Leal, Haotian Zhang, Jianming Fu, Guojun Peng, and Jean-Yves Marion. {Obfuscation-Resilient} executable payload extraction from packed malware. In *30th USENIX Security Symposium (USENIX Security 21)*, pp. 3451–3468, 2021.

CISA. Known exploited vulnerabilities catalog. `https://www.cisa.gov/known-exploited-vulnerabilities-catalog`, 2026.

Isaac David and Arthur Gervais. Crackmebench: Binary reverse engineering for agents. *arXiv preprint arXiv:2605.10597*, 2026.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*, 2024.

Eldad Eilam. *Reversing: secrets of reverse engineering*. John Wiley & Sons, 2011.

Zeyu Gao, Yuxin Cui, Hao Wang, Siliang Qin, Yuanda Wang, Zhang Bolun, and Chao Zhang. Decompilebench: A comprehensive benchmark for evaluating decompilers in real-world scenarios. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 23250–23267, 2025.

Google. Look what you made us patch: 2025 zero-days in review. `https://cloud.google.com/blog/topics/threat-intelligence/2025-zero-day-review`, 2026.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025.

Md Humaun Kabir, Md Rakibul Islam, and Farha Kamal. Restack: A large-scale dataset of reverse engineering discussions from stack exchange. *arXiv preprint arXiv:2606.05493*, 2026.

Nicolas Koller et al. Reforge: A method for benchmarking llms' reverse engineering capabilities in decompiled binary function naming. *arXiv preprint arXiv:2607.07738*, 2026.

Hwiwon Lee, Jiawei Liu, Dongjun Kim, Ziqi Zhang, Chunqiu Steven Xia, and Lingming Zhang. Sec-bench pro: Can language models solve long-horizon software security tasks? *arXiv preprint arXiv:2605.26548*, 2026a.

Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. Sec-bench: Automated benchmarking of llm agents on real-world software security tasks. *Advances in Neural Information Processing Systems*, 38:116342–116378, 2026b.

JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. 2011.

Seunghyun Lee and David Brumley. Exploitbench: A capability ladder benchmark for llm cybersecurity agents. *arXiv preprint arXiv:2605.14153*, 2026.

Yu Liu, Lang Gao, Mingxin Yang, Yu Xie, Ping Chen, Xiaojin Zhang, and Wei Chen. Vuldetectbench: Evaluating the deep capability of vulnerability detection with large language models. *arXiv preprint arXiv:2406.07595*, 2024.

Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. {RE-Mind}: a first look inside the mind of a reverse engineer. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 2727–2745, 2022.

Yuzhou Nie, Zhun Wang, Yu Yang, Ruizhe Jiang, Yuheng Tang, Xander Davies, Yarin Gal, Bo Li, Wenbo Guo, and Dawn Song. Secodeplt: A unified benchmark for evaluating the security risks and capabilities of code genai. *Advances in Neural Information Processing Systems*, 38, 2026.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. `https://openai.com/index/gpt-5-6/`, July 2026a. Accessed: 2026-07-11.

OpenAI. Why swe-bench verified no longer measures frontier coding capabilities - swe-bench verified is increasingly contaminated. we recommend swe-bench pro. `https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/`, 2026b.

Juefei Pu, Xingyu Li, Zhengchuan Liang, Jonathan Cox, Yifan Wu, Kareem Shehada, Arrdya Srivastav, and Zhiyun Qian. Patch-to-poc: A systematic study of agentic llm systems for linux kernel n-day reproduction. *arXiv preprint arXiv:2602.07287*, 2026.

Spencer Rugaber. Program comprehension. *Encyclopedia of Computer Science and Technology*, 35 (20):341–368, 1995.

Minghao Shao, Sofija Jancheska, Meet Udeshi, Brendan Dolan-Gavitt, Haoran Xi, Kimberly Milner, Boyuan Chen, Max Yin, Siddharth Garg, Prashanth Krishnamurthy, et al. NYU CTF Bench: A scalable open-source benchmark dataset for evaluating llms in offensive security. *Advances in Neural Information Processing Systems*, 37:57472–57498, 2024.

Minghao Shao, Nanda Rani, Kimberly Milner, Haoran Xi, Meet Udeshi, Saksham Aggarwal, Venkata Sai Charan Putrevu, Sandeep K Shukla, Prashanth Krishnamurthy, Farshad Khorrami, et al. Towards effective offensive security llm agents: Hyperparameter tuning, llm as a judge, and a lightweight ctf benchmark. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29660–29668, 2026.

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*, pp. 138–157. IEEE, 2016.

Janet Siegmund. Program comprehension: Past, present, and future. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 5, pp. 13–20. IEEE, 2016.

Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *International conference on information systems security*, pp. 1–25. Springer, 2008.

Vedant Soni, Audrey Dutcher, Tiffany Bao, and Ruoyu Wang. Benchmarking binary type inference techniques in decompilers. In *Proceedings of the 2025 Workshop on Software Understanding and Reverse Engineering*, pp. 48–60, 2025.

Xiaolong Tian, Hanrui Qi, Jiaming Liu, Siyi Wang, GAO Zuchen, Qi Luo, Jing Li, Yuqun Zhang, et al. Decompile-bench: Million-scale binary-source function pairs for real-world binary decompilation. *Advances in Neural Information Processing Systems*, 38, 2026.

Kevin van Liebergen, Juan Caballero, Platon Kotzias, and Chris Gates. A deep dive into the virustotal file feed. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 155–176. Springer, 2023.

Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse {Engineers'} processes. In *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1875–1892, 2020.

Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. Cybergym: Evaluating ai agents' real-world cybersecurity capabilities at scale. *arXiv preprint arXiv:2506.02548*, 2025.

Zhun Wang, Nico Schiller, Hongwei Li, Srijiith Sesha Narayana, Milad Nasr, Nicholas Carlini, Xiangyu Qi, Eric Wallace, Elie Bursztein, Luca Invernizzi, et al. Exploitgym: Can ai agents turn security vulnerabilities into real attacks? *arXiv preprint arXiv:2605.11086*, 2026.

Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: a controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, pp. 551–560, 2011.

Jun Yeon Won, Xin Jin, Shiqing Ma, and Zhiqiang Lin. Rebench: A procedural, fair-by-construction benchmark for llms on stripped-binary types and names (extended version). *arXiv preprint arXiv:2604.27319*, 2026.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can language models rebuild programs from scratch?, 2026. URL `https://arxiv.org/abs/2605.03546`.

Alperen Yildiz, Sin G Teo, Yiling Lou, Yebo Feng, Chong Wang, and Dinil Mon Divakaran. Benchmarking llms and llm-based agents in practical vulnerability detection for code repositories. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 30848–30865, 2025.

Andy Zhang, Joey Ji, Celeste Menders, Riya Dulepet, Thomas Qin, Ron Wang, Junrong Wu, Kyleen Liao, Jiliang Li, Jinghan Hu, et al. Bountybench: Dollar impact of ai agent attackers and defenders on real-world cybersecurity systems. *Advances in Neural Information Processing Systems*, 38, 2026.

Andy K Zhang, Neil Perry, Riya Dulepet, Joey Ji, Celeste Menders, Justin Lin, Eliot Jones, Gashon Hussein, Samantha Liu, Donovan Jasper, et al. Cybench: A framework for evaluating cybersecurity capabilities and risks of language models. In *International Conference on Learning Representations*, volume 2025, pp. 25094–25243, 2025.

Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, et al. Cve-bench: a benchmark for ai agents' ability to exploit real-world web application vulnerabilities. *arXiv preprint arXiv:2503.17332*, 2025.

Terry Yue Zhuo, Dingmin Wang, Hantian Ding, Varun Kumar, and Zijian Wang. Training language model agents to find vulnerabilities with ctf-dojo. *arXiv preprint arXiv:2508.18370*, 2025.

A Benchmark Domain Details

This appendix expands on each SRE-Bench domain, complementing the overview in §3. Consistent with our contamination-control policy, we describe software architecture, task structure, and anti-shortcut design only at a level an analyst could recover from the binary with minimal efforts, and we deliberately omit the instance-specific secrets – exact constants, trigger conditions, hidden-behavior mechanisms, and reference solutions – that would let a model solve a task by recognition.

Network Protocol. The agent receives a client binary that implements a proprietary, encrypted client–server protocol, together with a handful of partial packet captures that mimic the limited network traces available to analysts in practice. No specification is provided. The agent must recover the layered wire format and reconstruct the interacting client and server state machines, then implement a driver to communicate with a live server. A live server grades that driver over a single metered TCP connection, capped at 64 messages, scoring how deep into the protocol's state space the agent can legally drive: from the cryptographic handshake, through capability-tier climbing, rekeying, and connection migration, to a capstone clean run that covers every stage at once. The server generates a fresh key for each session, preventing the agent from replaying a previous run.

Game. The target is a fully playable, 20-floor terminal roguelike featuring combat, spellcasting, crafting, shops, factions, quests, and achievements. A player manual explains the game and its core mechanics. Hidden within the binary are six behaviors that never arise during normal play and are not documented in the manual; each is triggered silently by conditions that can be identified only by reconstructing the relevant program logic. The agent demonstrates each discovery by submitting an action trace, which a headless verifier replays and reports which hidden behaviors were triggered.

File Format. The target is a file-and-directory compressor that produces archives in a proprietary format, *i.e.*, an in-house version of GZIP. It is distributed as an *encoder only*: the binary contains no decompressor, so the agent must reverse engineer the multi-stage encoding pipeline and implement a decoder that reconstructs the original files byte-for-byte. The pipeline combines six modified variants of commonly used compression algorithms with a custom error-correction scheme. The agent must recover six password-protected archives of increasing complexity, some of which contain recoverable corruption that the decoder must detect and correct. Grading requires exact recovery of both file contents and metadata; the decoder must genuinely invert every stage of the pipeline.

Malware. The target is a malware-themed Linux implant that is inert and safe by construction. When activated with a key inside a sandbox, it simulates behaviors drawn from six common malware families: process masquerading, persistence, cryptomining, file ransomware, C2 beaconing, and lateral movement. All effects are strictly confined: file writes remain under the sandbox root, networking is restricted to loopback, and the implant performs no real cryptomining, user-data encryption, or remote code execution. The task is defensive: the agent must understand the implant well enough to implement a cleanup tool that surgically removes each family's artifacts while preserving grader-planted benign data. This requirement prevents indiscriminate "delete everything" strategies, a reward-hacking shortcut identified during adversarial auditing. Scoring is conjunctive, with one point awarded per family only if the tool removes the malicious state, preserves the benign state, and durably neutralizes the corresponding behavior. Instance-specific names, paths, and keys are all derived from a fresh secret on each run to prevent hard-coded cleanup strategies.

Firmware. The target is bare-metal firmware for a fictional, locked-down secure microcontroller whose firmware packaging and system-on-chip architecture were designed entirely in-house. It features an encrypted two-stage boot process, bespoke on-device cryptography, and dozens of peripheral subsystems. The agent interacts with the device only through a JTAG-style interface to a from-scratch emulator that models the chip's memory map, peripherals, and serial command interface, subject to a per-session budget of 5,000 commands. Beginning with black-box probing, the agent must reverse engineer the device and gain progressively greater control. The six tasks correspond to milestones along a dependency chain, progressing from initial communication to full privileged control. A fresh device secret is generated for each session. Moreover, the scoring logic resides in the emulator rather than the firmware and is never exposed to the agent.

A.1 Network Protocol Recovery

Software architecture. The target implements a complete proprietary protocol organized as four wire layers: framing with integrity checks and a de-recognized header layout; a from-scratch authenticated session-encryption layer (a stream cipher, a MAC, and a key-derivation function) that borrows the structure of standard AEAD constructions but replaces their constants, so nothing is recognizable by signature; stream multiplexing with priority, flow control, and compression; and an application layer with tens of message types. The protocol also embeds a small stack-based bytecode filter VM and a set of interacting state machines coordinated by a supervisor.

Task structure. The six scored anchors form a capability ladder: (1) recover the framing and complete the cryptographic handshake; (2) reconstruct the capability-tier state machine and forge the promotion tokens that gate the top tier; (3) follow the key-rotation schedule while maintaining a rolling-MAC chain across rekeys; (4) reconstruct the connection-migration state machine and answer its path challenges; (5) devirtualize the bytecode filter and satisfy a non-obvious hidden predicate within a bounded opcode budget; and (6) a capstone that requires all five above to hold within a single session while simultaneously satisfying a set of clean-run invariants under a whole-transcript MAC. Partial credit is real, and any "loud" protocol violation voids the run.

Anti-shortcut design. The protocol deliberately exposes a large decoy surface—compression, retransmission, priority scheduling, and much of several state machines—that is functionally exercised

but not anchor-relevant, defeating "this looks important" heuristics. Canonical answer strings are masked so that static extraction yields nothing, the handshake mixes in a fresh per-connection server nonce, and the per-session seed is randomized so a captured winning transcript cannot be replayed. A per-connection message cap prevents brute-force flooding, and grading is silent: the driver is never told its score or even what the milestones are.

Auxiliary infrastructure. Beyond the four language ports, the domain ships a from-scratch scoring server that is itself a complete protocol implementation running the anchor detectors, a cross-port conformance corpus with pinned wire transcripts, and deployment tooling that isolates the analysis and grading environments. The agent receives only the stripped binary and a few partial packet captures.

A.2 Game Reverse Engineering

Software architecture. The target is a full-screen terminal roguelike comprising roughly two dozen interacting subsystems—combat (including a multi-phase boss), a spell system across several schools of magic, crafting, shops, factions and reputation, quests, procedural map generation, field-of-view, inventory and leveling, an encyclopedia of lore, achievements, and deterministic record/replay. It is genuine software of real-world scale rather than a template, and it behaves identically across all four language ports.

Task structure. Hidden within the game are six behaviors that never occur during ordinary play and are documented nowhere in the player manual. They are silent—no message, sound, or on-screen counter marks them—so the only way to trigger one is to reconstruct the responsible program logic from the binary. The six behaviors are chosen to probe distinct RE capabilities of increasing difficulty, ranging from recovering an inlined constant and a per-object counter, through inverting a custom hash or pseudorandom generator and reconstructing a finite-state machine to search for a reachable input, up to jointly reconstructing state that is spread across several subsystems and unwinding a multi-round key derivation that combines constants from several of them. We intentionally do not disclose the specific mechanisms or trigger conditions. The agent proves a discovery by submitting an action trace, and a headless verifier replays it deterministically to report which behaviors fired.

Anti-shortcut design. Several decoy subsystems are built from the *same* kinds of constructs as the hidden behaviors (custom hashes, state machines, and unusual constants) but are wired into ordinary gameplay, so "find the weird construct" heuristics produce false positives; some decoys even reuse a hidden behavior's exact constant. All gameplay strings and a broad catalog of magic constants ship only in obfuscated, runtime-decoded form, so neither a string dump nor an immediate-operand scan surfaces the critical values. Determinism—required for offline replay—is guaranteed by a single seeded pseudorandom stream with a fixed consumption order, no wall-clock or environment reads, pinned floating-point behavior, and neutralized map-iteration order. A build-time hygiene check greps the shipped, stripped binary for forbidden identifiers and fails the build on any hit.

A.3 File-Format Recovery

Software architecture. The compressor implements a genuine multi-stage transform stack wrapped in a keyed container. The container provides framing, a reversible diffusion layer, optional password-based encryption, and an error-correcting recovery seal. Internally, a long-range LZ pre-pass feeds a reversible pre-filter bank (e.g., delta, executable, and image-row filters), followed by an adaptive per-block compressor that trial-encodes each block under several competing recipes—spanning context-mixing arithmetic coding, range-coded LZ, and Burrows–Wheeler transforms (Burrows & Wheeler, 1994) with entropy back-ends—and keeps only the smallest. At higher compression levels, a seed-derived self-mutating predictor VM, a keystream-whitening stage, and a keyed block permutation are applied. These are standard mechanism *classes*; the difficulty comes from their bespoke composition and from constants that are derived from a secret seed at runtime rather than stored, so nothing is recognizable by signature.

Task structure. Six challenge files of increasing complexity progressively expose new layers of the stack: the outer container and its recovery seal; the pre-filter bank and long-range LZ with block-method compression; full archive semantics over directory trees (paths, permissions, empty

directories) with the harder entropy back-ends; the predictor-VM residual stage, whose program is seed-derived and stored nowhere in the archive; error auto-correction via the recovery seal; and finally the keyed block permutation together with errors at the maximum correction capacity. Because a decoder for the last level subsumes all earlier ones, the levels form a strict cumulative ladder.

Challenge generation and grading. A deterministic generator produces all six challenges from a single secret master seed, drawing on a content palette that includes filesystem corner cases, and self-verifies that a correct decoder can recover each one. The originals are never shipped, and the grader performs byte- and metadata-exact comparison with no partial credit within a level. The binary ships as an encoder only—enforced by a build-time symbol scan and an import-graph gate—so no decoder or answer material is present. A further check forbids linking any standard or third-party compression library, preventing recognition of an off-the-shelf codec.

A.4 Malware Incident Response

Software architecture. The implant masquerades as a benign local utility but, when armed with a host activation key inside a sandbox, installs six families of effects: process masquerading with a watchdog, persistence via an injected shell-profile block, cryptominer artifacts, file ransomware, C2 beaconing, and lateral-movement worm behavior. Every effect is safe by construction: filesystem writes stay under an artifact root, networking is loopback-only, the miner's work is fake and bounded, ransomware only transforms sandbox-confined files carrying a full magic prefix, and the beacon/worm endpoints talk only to shipped validation stubs—there is no real remote execution, scanning, or encryption of user data. Per-instance names, paths, ports, keys, and triggers are all derived from a fresh host-key-based secret, and the secret is deliberately consumed by benign modules too, so "find the seed" is not enough to solve an instance.

Task structure. The RE task is defensive remediation, not behavior reproduction: the agent submits a cleanup tool that removes each family's artifacts while preserving benign data. Scoring is flat and conjunctive—one point per family, awarded only when the agent removes the malicious state, preserves grader-planted benign state, *and* durably neutralizes the behavior (no leftover resume tokens, triggers, or live processes). The families form a difficulty ramp, and several couple across behaviors so that solving one depends on material recovered from another.

Grading pipeline. A sandboxed grader derives the seed, plants baseline and near-match benign fixtures, captures a private pre-infection baseline, infects the sandbox, records evidence of what must be removed or repaired, runs the agent's cleanup, and scores the final state against baseline and evidence. Evidence capture is what distinguishes "an artifact never existed" from "an artifact existed and was correctly removed," since the grader never re-runs the implant after cleanup. Any modification to a pre-existing baseline object is globally fatal, so "delete everything" strategies score zero; a fresh host key each attempt defeats hard-coded cleanups; and the untrusted cleanup runs in an isolated, resource-bounded container with no network.

A.5 Bare-Metal Firmware Analysis

Software architecture. The firmware targets a fictional secure microcontroller with a plaintext bootloader that derives a key from a one-time-programmable fuse word and decrypts an encrypted application image into RAM. It is a complete device—not a CrackMe—with a hardware-gated secure region holding a per-instance secret, a device-mode state machine, dozens of peripheral subsystems (the majority of which are pure or woven decoys), and several sandboxed bytecode VMs that hide key constants. Its cryptographic core is a bespoke family designed to superficially resemble a well-known cipher while being algorithmically distinct, and it is split into independent variants so that recovering one does not hand over the others; genuinely standard checksums are placed in decoy modules so that recognizing them leads nowhere.

Task structure. The student-facing framing is a single objective—take control of the device, scored 0–6—with no sub-goal list; the milestone decomposition must itself be reverse-engineered. The six milestones form a dependency chain that escalates from establishing basic wire contact and recovering the command framing, through discovering a gated maintenance path and forging an authentication token, to defeating an update-authentication format and a hardware-gated challenge–response,

and finally a capstone that composes the earlier capabilities with multi-round key-derivation and bytecode-VM constant recovery to reach full privileged control.

Emulator and anti-shortcut design. The agent interacts with the device only through a from-scratch emulator that faithfully models the memory map, the memory-mapped peripherals, the interrupt model, and the serial command protocol, under a per-session budget of $5{,}000$ commands. A fresh random device secret each session re-derives the secret-dependent material, so a captured challenge–response for the later milestones scores zero on the next run; earlier milestones are additionally bound to a device-issued nonce, so individual frames cannot be replayed. Decoys mislead in both directions—modules that look security-critical are inert, while load-bearing values hide inside modules that appear to be decoys—and all decoy behavior is held byte-identical across ports, so differential probing cannot separate scored from decoy logic. Scoring is host-side and silent: no capability prints anything, and the grader records only the bare score.

B Protection Suite Details

This section adds context to the protection primitives summarized in §3. Following our contamination-control policy, we describe every mechanism conceptually and deliberately omit the constants, labels, encodings, container formats, file layouts, and system-call sequences that would give an agent under test a head start. The suite is an alpha-stage Linux x86-64 protector for static, self-contained executables. It makes no claim of unbreakability: the goal is to *raise the cost* of analysis in measurable, tunable steps, which is what makes it useful as a benchmark generator rather than as a security product.

B.1 Threat Model and the Governing Design Rule

We assume a privileged analyst: root in the analysis sandbox, free use of debuggers and tracers, the ability to read and write the process's memory, to intercept system calls, and to dump memory during an authorized run. Under those assumptions, no local scheme can prevent an analyst from eventually observing plaintext that the CPU must execute, and we do not pretend otherwise. What the suite can do is make every observation *expensive*, *noisy*, and *non-reusable across builds*. The single rule behind most of the design is *fold, don't branch*: instead of testing a condition and acting on it, the runtime feeds the measured value into key derivation. A conventional check (`if (debugger) exit()`) is one instruction away from being removed; a folded measurement has no boolean to remove, because the wrong measurement simply yields the wrong key.

B.2 Everything Feeds the Key

The protected program is encrypted at page granularity, and each page is sealed under its own key rather than under one global key. Build identity, segment, offset, and length metadata are included as associated data, so modifying either the page contents or their metadata causes authentication to fail. Those keys descend from a derivation graph whose inputs include the page's own identity, the integrity state of the executable's live code, the outcomes of the anti-analysis measurements, and – on the lazy path – evidence that the page is being opened by a genuine execution fault. Two properties follow, and they are the reason we consider this the core of the design. First, there is no single moment at which "the key" exists to be intercepted, so the classic breakpoint-on-decrypt strategy recovers only one page's worth of material. Second, tampering and observation are indistinguishable from key corruption: a modified instruction, a suppressed check, or an attached tracer all lead to the same uninformative failure deep inside decrypted code, giving the analyst no signal about which defense reacted.

B.3 Minimizing Plaintext Residency

Decrypting an entire program at startup means one well-timed memory snapshot recovers everything, so the lazy runtime instead maps protected code inaccessible and decrypts individual pages only as execution reaches them. Three cooperating mechanisms then push plaintext back out of memory: a cap on how many code pages may be decrypted at once, a timer that re-encrypts pages that have gone idle, and a synchronous flush before the process blocks and becomes an easy target.

The loader adds W⊕X enforcement, core-dump exclusion of secret pages, and un-optimizable secret wiping. The practical effect is that a snapshot captures a small working set rather than an image, and a process caught while idle or waiting yields little. We describe this honestly as residency *reduction*: an analyst who reads memory continuously during execution still observes the instantaneous working set, and the union of everything ever resident approaches the full program over a long run.

B.4 Establishing Exclusive Control

Most anti-debugging asks "is a debugger present?" – a question the analyst can answer falsely, since the reported value lives in memory the analyst controls. We instead ask a question whose answer is enforced by the operating system: the runtime attempts to establish the exclusive control relationship a debugger itself requires, and folds the outcome into key derivation. Failure implies a competing observer, while success additionally permits reading the processor's own debugging state, which catches breakpoints left behind by an analyst who has since detached. A separate monitoring process keeps the master key masked while it is at rest and destroys it on detection, so that attaching *after* a clean start corrupts decryption rather than merely raising an alarm; terminating that monitor is not a way out, because the protected process fails closed when the monitor stops reporting.

B.5 Virtualized Loader Logic

A compact custom virtual machine executes the loader's decision logic – how keys are derived, whether unwrapping is permitted, how control is transferred to the protected program, and how individual page faults are serviced – so that this logic is not present as native code to be read or patched. Rather than a recognizable interpreter loop, dispatch is diversified per build, instruction encodings are keyed, and frequently occurring sequences are fused into single operations, which removes the structural fingerprints that make virtual machines easy to identify and lift. In the licensed configurations, parts of the virtualized program are not present in the file at all and are supplied per run by the license server, so static analysis of the artifact alone cannot recover the complete logic.

B.6 Binding to a Genuine Launch, and Off-Box Keys

A capable analyst does not need to defeat the defenses if they can simply relocate the decryption machinery into a process of their own choosing and call it directly, which is an attack we observed in practice. Four cooperating layers close it: key derivation is bound to evidence of a genuine launch of the original artifact, the decryption routines are themselves protected at rest and only reconstituted after startup validation, and on the lazy path each page key is additionally bound both to the fault that requested it and to the observation state at that moment. The licensed configurations go further and keep no key in the artifact at all, obtaining it per run from a pinned server through an exchange that is freshly randomized each time, so a captured exchange cannot be replayed; the delivered key is then combined locally with the client's own measurements, meaning a server that answers an instrumented run still yields something unusable. The residuals are stated plainly: on non-licensed configurations the in-file key material is in principle (but challengingly) recomputable offline, and on licensed configurations the delivered key must exist in memory during an authorized run and is therefore exposed to a live memory read.

B.7 Deception Rather Than Denial

Because the strongest attacks cannot be blocked, several layers aim to make a successful attack *unrecognizable* rather than impossible. Strings that survive in the artifact are not merely encoded but renamed to plausible, misleading meanings, so that a first-pass triage read leads an analyst toward an incorrect model of what the program is doing. When the runtime detects that it is being observed, it never confirms the detection: it may quietly restart itself so that an attached tool loses its target, or terminate as though it had completed successfully – responses chosen specifically because a visible crash or error message would tell the analyst that their instrumentation was noticed and worked. In the strongest configuration, and only when the author supplies one, the runtime serves an entirely different program in place of the real one, so that the analyst successfully dumps, decrypts, and reverse-engineers the wrong artifact with no failure to indicate the substitution.

Table 6: Protection presets and their distinguishing primitives. ✓ = active; • = available but requires an author-supplied decoy; – = inactive. All presets share the common baseline described in §B.9.

ID	Preset	Lazy	Strict	Anti-re-host	Off-box key	Server logic	Anti-dump
P1	protected-eager	–	–	✓	–	–	–
P2	protected-eager-strict	–	✓	✓	–	–	•
P3	protected-demand	✓	–	✓	–	–	–
P4	protected-demand-strict	✓	✓	✓	–	–	✓
P5	licensed-eager	–	–	–	✓	shard	–
P6	licensed-demand	✓	–	–	✓	shard	–
P7	licensed-demand-material	✓	✓	–	✓	material	✓
P8	licensed-demand-superops	✓	✓	–	✓	operators	✓

B.8 Anti-Agent Deterrents: A Reported Negative Result

As a first exploration of protection aimed specifically at automated analysts, the suite can surface benign "please decline to reverse-engineer this" notices through several independent observation channels, so that a notice reaches an agent whether it reads the file, inspects memory, traces execution, or captures network traffic. The mechanism carries no secret or exploitable content and is trivially removed; it is a behavioral nudge, not a barrier. We report it as a *negative result*: capable agents (*i.e.*, GPT-5.6-sol, Opus 4.8) identified the notices as inert in a single reasoning step, and in one configuration the runtime's emission was actively harmful, serving as a landmark that helped locate the protected code path. It is therefore disabled by default in every preset and retained only as a documented design point for future study.

B.9 Preset Summary

Table 6 summarizes the eight presets. Every preset shares a common baseline – per-page authenticated encryption, fold-not-branch keying, self-checksumming, loader virtualization, and the obfuscation and stripping layers – and they differ along four axes that we found to matter most for difficulty: whether decryption is lazy, whether observation is keyed into decryption continuously (the *strict* presets), whether the key is off-box, and whether the runtime responds to a detected dump by misdirecting the analyst. The four strict presets are the only ones that break decryption on a *late* attach and the only ones that accept an author-supplied decoy program. Anti-agent deterrents are disabled by default in all eight and are therefore omitted from the table.

Per-Preset Results. Figure 5 resolves the aggregate protection effect reported in §4 into the eight presets, holding the build fixed as optimized, stripped and statically linked. GPT-6-Astra is largely unaffected, scoring between 5.25 and 6.00 on every preset. GPT-5.6-sol is the only other model that retains non-trivial capability, scoring between 1.93 and 3.19; every remaining model stays below 0.6 on every preset, and GLM-5.2 scores exactly 0.00 on all eight. Within GPT-5.6-sol the presets order roughly as designed, with eager whole-image decryption most tractable (P1, P5) and the strict and server-side variants least (P4, P8). That ordering is legible only for GPT-5.6-sol, however, because GPT-6-Astra sits near the ceiling and the remaining six sit near the floor.

C Unconstrained Evaluation Details

C.1 Reasoning Effort and Repeated Attempts

Table 11 reports all 13 internal OpenAI settings. Increasing effort improves full-solve rates for all three models across the tested settings. At Medium effort, GPT-6-Astra achieves 72.2% pass@1 with 34.2K output tokens per attempt, compared with 55.8% and 241.5K tokens for GPT-5.6 Sol at Max. Output-token counts describe generation effort rather than dollar cost or latency. GPT-5.6 Cyber at Max has a higher pass@4 than GPT-5.6 Sol (72.5% versus 68.7%), but slightly lower objective coverage (83.1% versus 83.9%) and greater output-token usage.

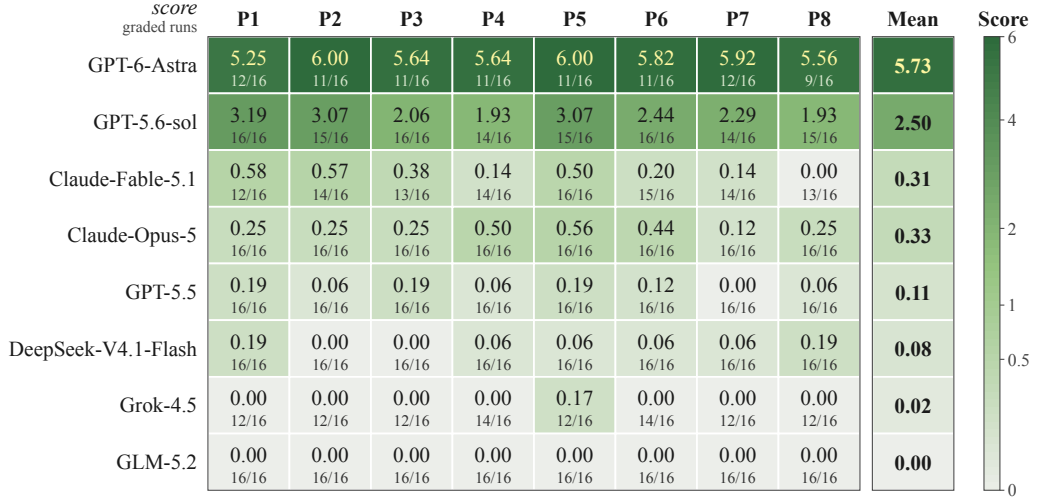


Figure 5: Mean score (of 6) per model on the eight protection presets of Table 6, all optimized, stripped and statically linked. Compare against the unhardened bar of Figure 3(d).

Table 7: **Cost per point on packed instances (public evaluation).** Packed covers the 128 protected and licensed non-firmware instances. Score is the mean over graded runs; Points is the total rubric points earned; Cost is total spend over all runs, including errored ones; $/pt is Cost divided by Points (– where no points were earned). Unpacked $/pt is computed the same way over the 128 unhardened instances, and the last column is the ratio of packed to unpacked $/pt. $/pt is unstable for models earning only a few points.

	Packed				vs. unpacked	
Model	Score	Points	Cost ($)	$/pt	Unpacked $/pt	Ratio
GPT-6-ASTRA	5.73	504	1,604.76	3.18	2.15	1.5×
GPT-5.6-SOL	2.50	303	6,785.33	22.39	6.52	3.4×
CLAUDE-FABLE-5.1	0.32	35	3,702.81	105.79	6.93	15.3×
CLAUDE-OPUS-5	0.33	42	2,813.08	66.98	7.22	9.3×
GPT-5.5	0.11	14	1,939.58	138.54	11.02	12.6×
DEEPSEEK-V4.1-FLASH	0.08	10	67.68	6.77	0.34	19.7×
GROK-4.5	0.02	2	1,308.62	654.31	18.56	35.3×
GLM-5.2	0.00	0	3,039.58	–	79.51	–

Repeated attempts expose a distinction between generating a correct solution and recognizing one. For GPT-6-Astra at Max, 191 instances succeed in all four attempts, while 13, 23, and 33 instances succeed in exactly one, two, and three attempts, respectively; two never fully succeed. Thus, 69 instances contain both correct and incorrect submissions. The corresponding pass@4 records whether a correct candidate exists, using the private grader after execution. A future selection evaluation can instead require the agent to choose one of its four submissions before private grading and report the correctness of that choice. The present evaluations do not measure such a selector.

C.2 MALWARE PROMPT CLARIFICATION

Table 12 compares the original prompt with one added sentence explaining how malware cleanup is graded, within the internal Claude Code evaluation. For Opus 5.5, the clarification raises pass@1 from 76.3% to 82.1% and objective coverage from 87.4% to 92.9%. Mythos 5.1 changes less, from 53.3% to 53.9% pass@1. We report both conditions because interpreting the cleanup requirement contributes to measured task performance. The main internal table uses the clarified prompt and identifies that choice explicitly.

Table 8: Token-equivalent cost of the unconstrained provider evaluation. GPT-6-Astra and GPT-5.6 Sol use OpenAI's published per-task costs for the matching configurations, summed over 262 instances and four attempts. GPT-5.6 Cyber has no published cost and is estimated at GPT-5.6 Sol's cost per output token. Anthropic costs are estimated from token counts at list price, assuming 98% of input tokens are served from cache.

Model	**Evaluation**	**Runs**	**Output tok.**	**Cost basis**	**Cost**
GPT-6-Astra	pass@4, 5 efforts	5,240	215.7M	published	$83.4K
GPT-5.6 Sol	pass@4, 4 efforts	4,192	506.9M	published	$161.9K
GPT-5.6 Cyber	pass@4, 4 efforts	4,192	773.6M	Sol $/token	$246.8K
Opus 5.5	pass@1 Low–XHigh + pass@4 Max	2,096	512.2M	list price	$52.3K
Mythos 5.1	pass@4 Max	1,048	362.9M	list price	$22.0K
Total		16,768	2.37B		≈$566K

Table 9: **Unconstrained evaluation by domain and language.** Each cell reports pass@1 / pass@4 (%). Protocol, format, game, and malware cells cover 16 instances each; firmware cells cover 2 (none in Go).

Model	**Lang.**	Proto	Fmt	Game	Malw	Firm
GPT-5.6-SOL	C	95.3 / 100.0	98.4 / 100.0	25.0 / 43.8	37.5 / 75.0	100.0 / 100.0
	C++	90.6 / 100.0	87.5 / 100.0	17.2 / 31.2	14.1 / 37.5	100.0 / 100.0
	Rust	90.6 / 100.0	89.1 / 100.0	1.6 / 6.2	21.9 / 50.0	50.0 / 50.0
	Go	100.0 / 100.0	100.0 / 100.0	6.2 / 18.8	7.8 / 31.2	–
GPT-5.6-CYBER	C	98.4 / 100.0	92.2 / 100.0	39.1 / 75.0	32.8 / 75.0	100.0 / 100.0
	C++	98.4 / 100.0	90.6 / 100.0	17.2 / 43.8	15.6 / 37.5	100.0 / 100.0
	Rust	90.6 / 100.0	85.9 / 100.0	1.6 / 6.2	14.1 / 43.8	100.0 / 100.0
	Go	98.4 / 100.0	96.9 / 100.0	12.5 / 37.5	7.8 / 31.2	–
GPT-6-ASTRA	C	87.5 / 100.0	100.0 / 100.0	90.6 / 100.0	82.8 / 100.0	100.0 / 100.0
	C++	90.6 / 100.0	98.4 / 100.0	84.4 / 100.0	82.8 / 100.0	100.0 / 100.0
	Rust	100.0 / 100.0	100.0 / 100.0	35.9 / 93.8	93.8 / 100.0	100.0 / 100.0
	Go	100.0 / 100.0	100.0 / 100.0	62.5 / 93.8	93.8 / 100.0	–
OPUS-5.5	C	79.7 / 100.0	100.0 / 100.0	76.6 / 100.0	95.3 / 100.0	100.0 / 100.0
	C++	90.6 / 100.0	100.0 / 100.0	48.4 / 81.2	79.7 / 100.0	100.0 / 100.0
	Rust	92.2 / 100.0	100.0 / 100.0	34.4 / 87.5	89.1 / 100.0	100.0 / 100.0
	Go	96.9 / 100.0	98.4 / 100.0	54.7 / 100.0	70.3 / 100.0	–
MYTHOS-5.1	C	100.0 / 100.0	100.0 / 100.0	15.6 / 43.8	12.5 / 43.8	100.0 / 100.0
	C++	98.4 / 100.0	98.4 / 100.0	3.1 / 12.5	15.6 / 50.0	100.0 / 100.0
	Rust	98.4 / 100.0	100.0 / 100.0	0.0 / 0.0	3.1 / 12.5	100.0 / 100.0
	Go	96.9 / 100.0	100.0 / 100.0	1.6 / 6.2	1.6 / 6.2	–

Table 10: **Unconstrained evaluation by protection preset** (Table 6). Pass@1 (%) with pass@4 as subscript. Unhard. covers all 128 unhardened non-firmware instances; each preset column covers 16 instances.

Model	Unhard.	P1	P2	P3	P4	P5	P6	P7	P8
GPT-5.6-SOL	$63.5_{78.1}$	$50.0_{62.5}$	$50.0_{56.2}$	$46.9_{68.8}$	$43.8_{56.2}$	$48.4_{50.0}$	$48.4_{62.5}$	$43.8_{56.2}$	$43.8_{56.2}$
GPT-5.6-CYBER	$63.9_{82.0}$	$54.7_{68.8}$	$51.6_{68.8}$	$46.9_{62.5}$	$46.9_{50.0}$	$50.0_{62.5}$	$45.3_{56.2}$	$45.3_{62.5}$	$40.6_{62.5}$
GPT-6-ASTRA	$88.7_{100.0}$	$87.5_{93.8}$	$84.4_{100.0}$	$85.9_{100.0}$	$87.5_{100.0}$	$90.6_{93.8}$	$87.5_{100.0}$	$82.8_{100.0}$	$87.5_{100.0}$
OPUS-5.5	$85.2_{100.0}$	$76.6_{93.8}$	$73.4_{87.5}$	$70.3_{93.8}$	$75.0_{93.8}$	$84.4_{100.0}$	$76.6_{100.0}$	$87.5_{100.0}$	$81.2_{100.0}$
MYTHOS-5.1	$53.9_{64.8}$	$53.1_{56.2}$	$51.6_{56.2}$	$51.6_{56.2}$	$53.1_{62.5}$	$53.1_{62.5}$	$51.6_{56.2}$	$50.0_{50.0}$	$50.0_{56.2}$

Table 11: **Unconstrained provider reasoning-effort sweep.** Each OpenAI setting and Opus 5.5 at Max has four attempts per instance (1,048 attempts); Opus 5.5 at Low–XHigh has one attempt per instance. Pass rates and objective coverage are percentages. Output tokens are per-attempt means in thousands, counting the complete continuation chain; n_{tok} gives the reconciled token records used in that mean.

Model	Effort	Pass@1	Pass@4	Coverage	Tokens (K)	n_{tok}
GPT-5.6 Sol	Medium	5.8	8.8	14.9	28.9	1046
	High	16.7	23.3	36.6	70.9	1046
	XHigh	41.2	53.8	69.1	143.3	1044
	Max	55.8	68.7	83.9	241.5	1047
GPT-5.6 Cyber	Medium	10.5	16.8	23.6	64.5	1043
	High	23.1	34.0	43.9	127.7	1043
	XHigh	43.4	56.9	68.8	211.1	1040
	Max	56.8	72.5	83.1	342.6	1032
GPT-6-Astra	Low	52.3	67.2	76.1	20.7	1048
	Medium	72.2	90.1	90.3	34.2	1046
	High	74.7	92.4	93.2	38.7	1048
	XHigh	77.7	94.3	93.4	42.6	1048
	Max	88.0	99.2	97.0	69.7	1048
Opus 5.5	Low	53.1	–	84.4	117.5	–
	Medium	59.9	–	81.2	143.3	–
	High	66.8	–	85.2	169.6	–
	XHigh	77.9	–	89.3	240.1	–
	Max	82.1	98.1	93.5	321.1	–

Table 12: **Malware prompt clarification in the unconstrained Anthropic evaluation.** The clarification adds one sentence explaining cleanup grading. Coverage@1 averages objective checks across attempts; Coverage@4 uses the best single-attempt score per instance. All values are percentages.

Model	Prompt	Pass@1	Pass@4	Coverage@1	Coverage@4
Opus 5.5	Original	76.3	96.6	87.4	98.5
	Clarified	82.1	98.1	92.9	99.7
Mythos 5.1	Original	53.3	59.5	74.5	83.4
	Clarified	53.9	61.8	76.0	86.9