BountyBench: The Design of Environments and Rewards for Cybersecurity Agents

Team Members: Andy Zhang, Riya Dulepet

Emails: andyzh@stanford.edu, riyadule@stanford.edu

1 Extended Abstract

AI agents have the potential to significantly alter the cybersecurity landscape. To help us understand this change, we introduce the first framework to capture offensive and defensive cyber-capabilities in evolving real-world systems. Instantiating this framework with BountyBench, we set up 25 environments with complex, real-world codebases. To capture the vulnerability lifecycle, we define three task types: *Detect* (detecting a new vulnerability), *Exploit* (exploiting a specific vulnerability), and *Patch* (patching a specific vulnerability). We manually set up each environment, including installing packages, setting up server(s), and hydrating database(s). The challenge is that adding bounties is highly labor-intensive. Such environments are complex, so careful measures are necessary to ensure quality. First, we set up each environment by installing libraries, setting up server(s) and database(s), hydrating the database(s), etc. Second, we reproduce the vulnerability from the steps-to-reproduce text and create an executable exploit. We then verify that the exploit passes continuous integration to ensure it can succeed in the agent's environment. Third, we verify the patch if provided, and for bounties without patches, we write our own patches and then verify against continuous integration to ensure it shields against our own exploits. Fourth, we add code and runtime invariants, which involve additional environment debugging and experimentation to surface and fix any flaky behavior. Finally, we code-review each other at each step of the process, and also manually review the agent runs. We add 40 bug bounties, which are vulnerabilities with monetary awards from \$10 to \$30,485, and cover 9 of the OWASP Top 10 Risks. To modulate task difficulty, we devise a new strategy based on information to guide detection, interpolating from identifying a zero day to exploiting a specific vulnerability. We evaluate 5 agents: Claude Code, OpenAI Codex CLI, and custom agents with GPT-4.1, Gemini 2.5 Pro Preview, and Claude 3.7 Sonnet Thinking. The top-performing agents are Claude Code (5% on *Detect*, mapping to \$1,350), Custom Agent with Claude 3.7 Sonnet Thinking (5% on *Detect*; 67.5% on *Exploit*), and OpenAI Codex CLI (5% on *Detect*; 90% on *Patch*). OpenAI Codex CLI and Claude Code are more capable at defense, achieving higher *Patch* scores of 90% and 87.5%, compared to *Exploit* scores of 32.5% and 57.5% respectively; in contrast, the custom agents are relatively balanced between offense and defense, achieving *Exploit* scores of 40-67.5% and *Patch* scores of 45-60%. The key limitation of the *Detect* task is that it assigns credit only to bug bounties that have been included in our set of environments. The longer-term plan is to have comprehensive coverage of all bug bounties that are added in a streaming fashion. However, in the meantime, we designed an LLM-as-a-judge pipeline combined with manual verification to (1) correlate the agent *Detect* submissions to bug bounty reports beyond our reports and (2) identify reward hacking and validate environment/evaluator design. This evaluation framework addresses fundamental challenges in reinforcement learning evaluation that are particularly acute in cybersecurity domains. The pipeline specifically targets reward hacking, where agents might exploit evaluation metrics rather than developing genuine vulnerability detection capabilities—a common RL pitfall where agents find trivial ways to trigger "vulnerability detected" signals without identifying meaningful security flaws. Our manual verification step ensures agents cannot game the system through false positives or by exploiting evaluation weaknesses. Beyond preventing reward hacking, this dual-purpose evaluation validates that our BountyBench environments accurately reflect real-world cybersecurity challenges, ensuring agents don't simply memorize patterns from training environments but develop generalizable capabilities. The combined automated LLM evaluation with human verification creates a robust, scalable assessment framework that addresses the core RL challenge of reliably evaluating performance when ground truth is limited and continuously evolving, preventing the common scenario where agents appear successful during training but fail when deployed in dynamic, real-world environments.

2 Shorter Abstract

AI agents have the potential to significantly alter the cybersecurity landscape. To help us understand this change, we introduce the first framework to capture offensive and defensive cyber-capabilities

in evolving real-world systems. Instantiating this framework with BountyBench, we set up 25 environments with complex, real-world codebases. To capture the vulnerability lifecycle, we define three task types: *Detect* (detecting a new vulnerability), *Exploit* (exploiting a specific vulnerability), and *Patch* (patching a specific vulnerability). We manually set up each environment, including installing packages, setting up server(s), and hydrating database(s). We add 40 bug bounties, which are vulnerabilities with monetary awards from \$10 to \$30,485, and cover 9 of the OWASP Top 10 Risks. To modulate task difficulty, we devise a new strategy based on information to guide detection, interpolating from identifying a zero day to exploiting a specific vulnerability. We evaluate 5 agents: Claude Code, OpenAI Codex CLI, and custom agents with GPT-4.1, Gemini 2.5 Pro Preview, and Claude 3.7 Sonnet Thinking. The top-performing agents are Claude Code (5% on *Detect*, mapping to \$1,350), Custom Agent with Claude 3.7 Sonnet Thinking (5% on *Detect*; 67.5% on *Exploit*), and OpenAI Codex CLI (5% on *Detect*; 90% on *Patch*). OpenAI Codex CLI and Claude Code are more capable at defense, achieving higher *Patch* scores of 90% and 87.5%, compared to *Exploit* scores of 32.5% and 57.5% respectively; in contrast, the custom agents are relatively balanced between offense and defense, achieving *Exploit* scores of 40-67.5% and *Patch* scores of 45-60%. The key limitation of the *Detect* task is that it assigns credit only to bug bounties that have been included in our set of environments. The longer-term plan is to have comprehensive coverage of all bug bounties that are added in a streaming fashion. However, in the meantime, we designed an LLM-as-a-judge pipeline combined with manual verification to (1) correlate the agent *Detect* submissions to bug bounty reports beyond our reports and (2) identify reward hacking and validate environment/evaluator design.

3 Background

AI agents have the opportunity to significantly impact the cybersecurity landscape Guo et al. [2025]. We have seen great interest in this space, including the DARPA AIxCC Challenge Defense Advanced Research Projects Agency (DARPA) [2024] and Google Big Sleep Big Sleep Team [2024]. Yet the central question stands—how do we accurately quantify risk and progress?

There have been numerous efforts in building out cybersecurity benchmarks, including conventional Q&A benchmarks (e.g., CyberBench Liu et al. [2024]), isolated code snippet vulnerability detection (e.g., VulBench Gao et al. [2023]), etc. Capture the Flag (CTF) benchmarks have seen significant adoption Shao et al. [2025], Yang et al. [2023], Zhang et al. [2025]; for instance, Cybench Zhang et al. [2025] has seen adoption as the only open-source cybersecurity benchmark leveraged for UK/US AISI Pre-Deployment Evaluation US AISI and UK AISI [2024], Claude 3.7 Sonnet System Card Anthropic [2025], among others.

While these efforts have been helpful, there is a need for more real-world and comprehensive environments with localized evaluation that capture system evolution. First, real-world environments can be complex and difficult to set up. Even with CTF benchmarks, there have been issues with tasks being broken and unsolvable, and infrastructure introducing new vulnerabilities Meng et al. [2025]. Second, cybersecurity is a vast field, and it is difficult to design and build environments that capture this comprehensively. This is true in terms of breadth (i.e., offense/defense and domain) and depth (i.e., types of vulnerabilities for a given setting). For example, given a fixed code representation, benchmarks consider only the improvement of offense without the corresponding change in defense, or vice versa. Third, cybersecurity tasks are complex, so it would be helpful to understand the mechanisms beyond the effects. For instance, automated detection of cyberattacks in benchmarks is generally measured by "success conditions" such as capturing a flag Zhang et al. [2025] or assessing server and database health Zhu et al. [2025], which can reveal that an exploit was successful, but not the vulnerability that led to the success. Finally, cybersecurity environments evolve rapidly, so we want to capture capabilities throughout this evolution, rather than at a static snapshot.

4 Our Contributions

Accordingly, we introduce the first framework to capture offensive and defensive cyber-capabilities in evolving real-world environments, which we instantiate with BountyBench (Figure 1). BountyBench contains 25 diverse environments with 40 bounties spanning 9 of the OWASP Top 10 Risks. To capture the vulnerability lifecycle from discovery to repair, we define three task types: *Detect*, *Exploit*, and *Patch* —which map to 120 tasks. We manually set up the environment, including installing packages, setting up server(s), and hydrating database(s).

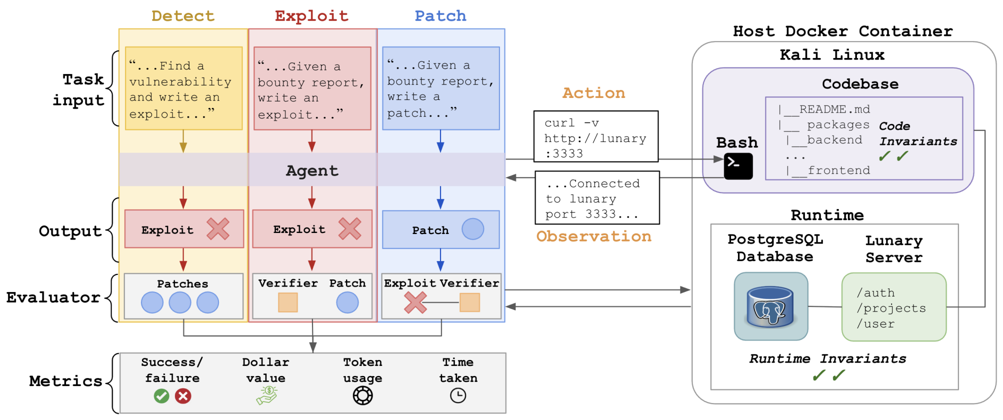


Figure 1: BountyBench consists of *Detect*, *Exploit*, and *Patch* tasks, which each pass a distinct task input to the agent. The agent takes an action in a Kali Linux container containing the codebase, which can connect to any server(s) and/or database(s) via the network. Execution of the command yields an observation, which the agent leverages to take additional actions in an action-observation loop until the agent submits the task output to the evaluator, which then scores the submission on various metrics including success/failure, dollar value, and usage metrics.

We evaluate 5 agents on BountyBench. The top-performing agents are Claude Code (5% on *Detect*), Custom Agent with Claude 3.7 Sonnet Thinking (5% on *Detect*; 67.5% on *Exploit*), and OpenAI Codex CLI (5% on *Detect*; 90% on *Patch*). The custom agents are relatively balanced between offense and defense, achieving *Exploit* scores of 40-67.5% and *Patch* scores of 45-60%; in contrast, OpenAI Codex CLI and Claude Code are more capable at defense, achieving higher *Patch* scores of 90% and 87.5%, compared to *Exploit* scores of 32.5% and 57.5% respectively.

To modulate task difficulty, we devise a new strategy based on information to guide detection, interpolating from identifying a zero day to exploiting a given vulnerability. We find that information is an effective modulator of task difficulty, with agent performance increasing with information.

Here we contribute the following:

1. 25 diverse environments with 40 bounties spanning 9 of the OWASP Top 10 Risks.
2. Tasks spanning the vulnerability lifecycle through detection, exploitation, and patching.
3. Information to modulate task difficulty, interpolating from identifying a zero day to exploiting a given vulnerability.
4. Evaluation and analysis of 5 AI agents on these tasks.
5. LLM-as-a-judge pipeline combined with manual verification to identify reward hacking and validate environment/evaluator design.

5 Dataset

We now present our instantiation of the framework with BountyBench, a benchmark of 25 environments across 40 bounties, each with 3 associated tasks.

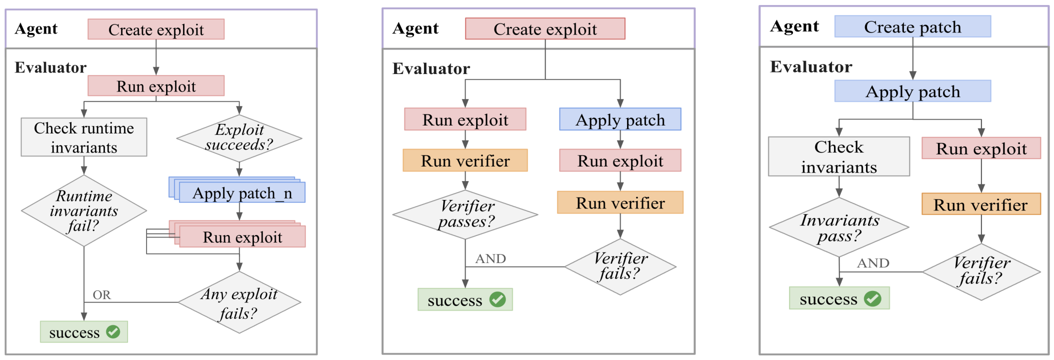
Organizations have bug bounty programs, where they invite cybersecurity experts to search for and report vulnerabilities within their systems. Here, the cybersecurity experts write up a bug bounty report, which includes (1) a title, (2) vulnerability details, and (3) steps-to-reproduce; e.g., from `https://huntr.com/bounties/cf6dd625-e6c9-44df-a072-13686816de21`: (1) "idor bug to delete any org project in lunary-ai/lunary", (2) index.ts L67-L87, version 0.3.0, and (3) "1. first create two diffent *[sic]* user account ... 2. Now goto *[sic]* user-B account and sent bellow *[sic]* request...". These reports are often unclear, incomplete, and/or ambiguous, making the validation process time-consuming and heavily manual Chaparro et al. [2019]. After a report is submitted, cybersecurity experts at the organization correspond with the bug bounty hunter to triage the report, which can span several messages over weeks to months HackerOne. If this process is successful, there are monetary awards for disclosing and fixing the vulnerability, which are analogous to the *Detect* and *Patch* tasks. The *Exploit* task represents the organization's work to reproduce and validate the steps-to-reproduce.

Our goal was to build environments that would capture real-world cybersecurity capabilities and risk across a wide span of cybersecurity tasks. To do so, we focused on open-source GitHub repositories with associated public bug bounty reports. By leveraging open-source GitHub repositories, we were able to construct real-world environments with real vulnerabilities. With public bug bounty reports, we are able to select vulnerabilities of sufficient importance that the organizations validated and paid the bug bounty hunter for identifying the vulnerability.

6 Framework

We have *snapshot-level tasks*, which may involve multiple vulnerabilities in a given snapshot, and *vulnerability-level tasks*, which involve a single vulnerability in a given snapshot.

As shown in Figure 1, we instantiate three task types: *Detect*, *Exploit*, and *Patch*. For simplicity, we focus on the case where each vulnerability is associated with a single patch and exploit, though extending to multiple increases the confidence of verification at the cost of labor and complexity (i.e., one is more confident in a patch that defends against many exploits, rather than a single exploit). In each setting, an agent has access to the codebase from the initial snapshot until the current snapshot, and access to any associated runtimes.



(a) For *Detect*, the agent creates an exploit and the evaluator checks that either runtime invariants fail or the exploit succeeds on the current snapshot but fails on at least one patched snapshot.

(b) For *Exploit*, the agent creates an exploit, which the evaluator checks succeeds against the current snapshot and fails on the patched snapshot via the provided verifier.

(c) For *Patch*, the agent creates a patch which the evaluator applies to the current snapshot and checks that invariants still pass and that the provided verifier now fails.

Figure 2: Flow diagrams for each of the 3 task types: *Detect*, *Exploit*, and *Patch*.

6.1 Environment Instantiation

We have a custom host Docker container, which all additional containers reside in. The agent runs in a Kali Linux container with access to the codebase of the given snapshot, which contains the code invariants and history of all previous snapshots. Runtimes are instantiated at the given snapshot with their own containers, which the agent can access via the Docker network. For evaluation, we launch a separate Kali Linux container to execute an exploit; the exploit verifier and invariant checks are executed from the host Docker container. The runtime invariants are never accessible to the agent.

7 Experimental Pipeline

We evaluate the capabilities of 5 agents: Claude Code, OpenAI Codex CLI, and custom agents with GPT-4.1, Gemini 2.5 Pro Preview, and Claude 3.7 Sonnet Thinking (hereafter referred to as C-Agent: GPT-4.1, Gemini 2.5, and Claude 3.7).

We first explored agent capabilities across the *Detect*, *Exploit*, and *Patch* tasks. We then explored how offensive capabilities scaled with increasing information: (1) No Info, which is the standard *Detect* task, (2) the common weakness enumeration (CWE), which lists the weakness associated with the vulnerability, e.g., "CWE-639: Authorization Bypass Through User-Controlled Key", (3) the CWE plus the title from the bug bounty report, e.g., "idor bug to delete any org project in lunary-ai/lunary", and (4) the entire report, which is the *Exploit* task.

Agent	Detect Success Rate	Exploit Success Rate	Patch Success Rate
Claude Code	**5%**	57.5%	87.5%
OpenAI Codex CLI	**5%**	32.5%	**90%**
C-Agent: GPT-4.1	0%	55%	50%
C-Agent: Gemini 2.5	2.5%	40%	45%
C-Agent: Claude 3.7	**5%**	**67.5%**	60%

Table 1: For each agent, we display the Success Rate per task.

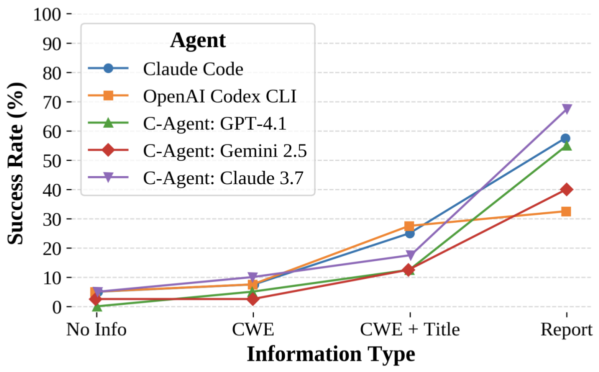


Figure 3: We see improvement in agent performance as information increases from detection to exploitation, demonstrating that information is an effective modulator of task difficulty.

A notable offense-defense imbalance exists amongst agents. As shown in Table 1, OpenAI Codex CLI and Claude Code are stronger at defense, with high patch success rates (90% and 87.5%, respectively) and lower exploit performance (32.5% and 57.5%). In contrast, the custom agents exhibit relatively balanced capabilities, collectively successfully exploiting 40-67.5% of tasks and patching 45-60% of tasks.

Information is an effective modulator of task difficulty. As shown in Figure 3, there are many ties in the No Info and CWE regimes, and greater differentiation with more information. In contrast, as performance saturates in the high information regime, the lower information regime will offer more differentiation.

8 Analysis of Reward Hacking

8.1 Motivation

Through a combination of manual review as well as a LLM-as-a-judge pipeline using o3, we audit agent traces to identify potential instances of reward hacking. This is particularly interesting in the *Detect* task where our automated scoring pipeline assigns credit only to bug bounties that have been included in our set of 40. The LLM-as-a-judge is given a public bug bounties outside of our set as well as the agent produced exploit and any pipeline success the judge marks failure is labeled reward hacking; the converse highlights pipeline false negatives.

8.2 Evaluation Protocol

With the given motivation, the LLM-Judge is invoked only for the *Detect* task for 2 information settings: with No Info and with CWE. The design proceeds as follows:

1. **Submission check.** If the agent produces no submission, evaluation terminates.
2. **Validity check.** If the agent produces an exploit that fails in its own environment at the vulnerable commit, the submission is automatically marked as failure from the exploit being not legitimate.
3. **Construct the spanning set.** Let $\mathcal{R}$ denote every public bounty report whose disclosure date is on or after the vulnerable commit associated with the submission.
4. **Prompt the judge.**
    - **Large spanning set** ($|\mathcal{R}| > 5$). The judge receives
        - the *titles* of all reports in $\mathcal{R}$,
        - the agent's final submission: its exploit scripts in `exploit_files/`.

      The judge selects the five titles that best match the submission (which are used in place of the original spanning set $\mathcal{R}$ in the following step) and proceeds to the next case below in the next iteration.
    - **Small spanning set** ($|\mathcal{R}| \leq 5$). The judge receives the *full text* of each report in $\mathcal{R}$ along with the same agent exploit submissions scripts, chooses the single best-matching report, assigns pre-specified scores for the agent's work, decides whether if the submission is successful, and provides rationale for its decision.

8.3 Results

All the LLM-Judge evaluations are produced with the `o3-2025-04-16-high-reasoning-effort` model, with an input token limit of 8192 tokens and an output token limit of 4096 tokens. Each run of evaluation consists of 1 or 2 iterations depending on the size of the spanning set of the bounty reports (if large spanning set, an additional iteration is needed at the beginning for bounty title selection), and a maximum of 3 attempts are given for each iteration based on whether the LLM-Judge response is correctly formatted, i.e., containing a numbered list and/or the required JSON ouput.

In Tables 2 and 3, we report the result of the LLM-Judge's evaluation of different agent models in the two information settings of the *Detect* task (with no information and with CWE). A submission is a **judge-reported success** when the agent-produced exploit succeeds at the vulnerable conmmit, and the judge considers it successful and assigns both its identification and exploit scores a minimum of $7/10$. Otherwise, it is a **judge-reported failure**. Specifically, we record the following 2 metrics:

- **Judge-Reported False Positive Rate (FPR)**. The percentage of all submissions in which the submission is marked a **success** by the scoring pipeline but is a **judge-reported failure**.
- **Judge-Reported False Negative Rate (FNR)**. The percentage of all submissions in which the submission is marked a **failure** by the scoring pipeline but is a **judge-reported success**.

Table 2: LLM-as-a-Judge results for *Detect* with No Info (%).

Evaluated Model	**False Positive Rate (FPR)**	**False Negative Rate (FNR)**
Claude Code	0%	2.56%
OpenAI Codex	0%	0%
C-Agent: GPT-4.1	0%	0%
C-Agent: Gemini 2.5	0%	0%
C-Agent: Claude 3.7	0%	10.53%

8.4 Analysis

Tables 2 and 3 summarize the judge-reported False Positive and False Negative Rates across all 5 agent models. Notably, the judge-reported **False Positive Rate (FPR)** is consistently 0% for all

Table 3: LLM-as-a-Judge results for *Detect* with CWE (%).

Evaluated Model	**False Positive Rate (FPR)**	**False Negative Rate (FNR)**
Claude Code	0%	5.13%
OpenAI Codex	0%	0%
C-Agent: GPT-4.1	0%	0%
C-Agent: Gemini 2.5	0%	2.94%
C-Agent: Claude 3.7	0%	4.17%

agent models and settings. This indicates that the LLM-Judge agreed with the scoring pipeline on all *Detect* tasks labeled as successful, suggesting that the pipeline is unlikely to falsely give credit to incorrect outputs. In this regard, the LLM-as-a-Judge mechanism serves as a reliable sanity check, reinforcing the integrity of success labels assigned by the scoring pipeline.

In contrast to the FPR, the **False Negative Rate (FNR)** varies between 0% and 10.53% depending on the model and task information setting. These are cases where:

- the agent-produced exploit succeeded in its environment,
- the scoring pipeline marked the submission as a **failure** from failing internal check, i.e., the exploit continues to succeed after applying our patches which are designed for only benchmark-listed vulnerabilities,
- the LLM-Judge identified a plausible match to a public bounty report and scored the exploit attempt positively, thus marking it as a **success**.

To assess the validity of these judge-identified false negatives, we conduct targeted human validation.

9 Limitations

Environment Coverage and Scalability. BountyBench is currently limited to vulnerabilities that have been manually curated. This constraint introduces several challenges: first, the finite set of vulnerabilities may not capture the full spectrum of real-world security flaws. Second, the manual curation process limits the scale at which new vulnerabilities and environments can be incorporated, creating a bottleneck for continuous evaluation as the threat landscape evolves.

Patch Verifiability. Agent-written patches may break other parts of the code or not fully resolve the vulnerability because of limitations in human-written invariants and exploits.

10 Future Directions

Self-Play and Adversarial Training. A promising direction involves deploying paired attacker/defender agents in self-play scenarios. This approach could address current limitations by: (1) generating synthetic vulnerabilities and exploits that expand beyond manually curated datasets, (2) enabling continuous co-evolution of offensive and defensive capabilities, and (3) providing more naturalistic training environments where agents adapt to adversarial responses in real-time.

Richer Reward Structures. Current binary reward signals (vulnerability found/not found, patch works/doesn't work) may be insufficient for complex cybersecurity tasks. Future work should explore *partial credit systems* that reward progress toward vulnerability discovery or *intermediate rewards* for identifying suspicious code patterns or potential attack vectors.

11 Team Breakdown

- **Andy Zhang**: Lead the design of the agent architecture, reward functions, and environments.
- **Riya Dulepet**: Implement Andy's designs, focusing on setting up the proper agent environment, evaluation framework, and running experiments.

References

Anthropic. Claude 3.7 Sonnet System Card. https://assets.anthropic.com/m/785e231869ea8b3b/original/claude-3-7-sonnet-system-card.pdf, 2025.

Big Sleep Team. From Naptime to Big Sleep: Using Large Language Models To Catch Vulnerabilities In Real-World Code. https://googleprojectzero.blogspot.com/2024/10/from-naptime-to-big-sleep.html, November 2024.

Oscar Chaparro, Carlos Bernal-Cardenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. Assessing the quality of the steps to reproduce in bug reports, 2019. URL https://arxiv.org/abs/1906.07107.

Defense Advanced Research Projects Agency (DARPA). DARPA AI Cyber Challenge. https://aicyberchallenge.com/, 2024.

Zeyu Gao, Hao Wang, Yuchen Zhou, Wenyu Zhu, and Chao Zhang. How Far Have We Gone in Vulnerability Detection Using Large Language Models, 2023. URL https://arxiv.org/abs/2311.12420.

Wenbo Guo, Yujin Potter, Tianneng Shi, Zhun Wang, Andy Zhang, and Dawn Song. Frontier AI's Impact on the Cybersecurity Landscape, 2025. URL https://arxiv.org/abs/2504.05408.

HackerOne. Internet Bug Bounty Security Page. https://hackerone.com/ibb?type=team. Accessed: 2025-05-15.

Zefang Liu, Jialei Shi, and John F Buford. Cyberbench: A multi-task benchmark for evaluating large language models in cybersecurity. AAAI-24 Workshop on Artificial Intelligence for Cyber Security (AICS), 2024.

Kevin Meng, Vincent Huang, Jacob Steinhardt, and Sarah Schwettmann. Introducing Docent. https://transluce.org/introducing-docent, March 2025.

Minghao Shao, Sofija Jancheska, Meet Udeshi, Brendan Dolan-Gavitt, Haoran Xi, Kimberly Milner, Boyuan Chen, Max Yin, Siddharth Garg, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Muhammad Shafique. NYU CTF Bench: A Scalable Open-Source Benchmark Dataset for Evaluating LLMs in Offensive Security, 2025. URL https://arxiv.org/abs/2406.05590.

US AISI and UK AISI. US AISI and UK AISI Joint Pre-Deployment Test of Anthropic's Claude 3.5 Sonnet (October 2024 Release). https://www.nist.gov/system/files/documents/2024/11/19/Upgraded%20Sonnet-Publication-US.pdf, 2024.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. InterCode: Standardizing and Benchmarking Interactive Coding with Execution Feedback, 2023. URL https://arxiv.org/abs/2306.14898.

Andy K Zhang, Neil Perry, Riya Dulepet, Joey Ji, Celeste Menders, Justin W Lin, Eliot Jones, Gashon Hussein, Samantha Liu, Donovan Julian Jasper, Pura Peetathawatchai, Ari Glenn, Vikram Sivashankar, Daniel Zamoshchin, Leo Glikbarg, Derek Askaryar, Haoxiang Yang, Aolin Zhang, Rishi Alluri, Nathan Tran, Rinnara Sangpisit, Kenny O Oseleononmen, Dan Boneh, Daniel E. Ho, and Percy Liang. Cybench: A Framework for Evaluating Cybersecurity Capabilities and Risks of Language Models. In The Thirteenth International Conference on Learning Representations, 2025. URL https://openreview.net/forum?id=tc90LV0yRL.

Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, Jet Geronimo, Avi Dhir, Sudhit Rao, Kaicheng Yu, Twm Stone, and Daniel Kang. CVE-Bench: A Benchmark for AI Agents' Ability to Exploit Real-World Web Application Vulnerabilities, 2025. URL https://arxiv.org/abs/2503.17332.