Large Language Models in Software Security Analysis

DYLAN WOLFF, National University of Singapore, Singapore
MARTIN MIRCHEV, National University of Singapore, Singapore
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

Cyber-security plays a pivotal role in safeguarding the integrity and reliability of critical systems — spanning healthcare, commerce, transportation, and power infrastructure. Yet software security teams are often critically under-funded and under-staffed given the breadth and complexity of our software infrastructure. Recently, Large Language Models (LLMs) have proven to be a transformative technology that is reshaping how we interact with software. In this paper, we explore how LLMs can be leveraged to holistically address longstanding challenges in cybersecurity. We start with a discussion on the structure of a Cyber Reasoning System that can both detect and repair vulnerabilities in software *autonomously*. We follow up by examining LLMs' strengths in aiding program analysis within such a system, finding that LLM-assisted analyses can accomplish several difficult analysis tasks such as extrapolating developer intent, filtering or augmenting the output of traditional analysis tools or even solving complex multilingual constraints. Lastly, we discuss the current challenges and limitations in constructing a composite system that can leverage these components. We hope that this article can provide insights into the evolving role of LLMs and inspiration in shaping the future of software security.

CCS Concepts: • **Security and privacy → Software and application security**; • **Software and its engineering → Software development techniques**.

Additional Key Words and Phrases: Cyber Reasoning System, Fuzz Testing, Automated Program Repair, Program Vulnerability

ACM Reference Format:
Dylan Wolff, Martin Mirchev, and Abhik Roychoudhury. 2024. Large Language Models in Software Security Analysis. In *Proceedings of CACM (Communications of the ACM).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software systems control numerous key aspects of society such as government agencies, medical services, utilities, national defense infrastructure and beyond. A core challenge in protecting these systems, however, is their diversity and scale. Modern software is composed of components written by different people, organizations, and increasingly generative AI tools. The resulting systems are composed of many programming styles, languages, deployment environments, and dependencies. This complexity can not only lead to additional bugs and vulnerabilities being introduced, but also increase the difficulty of analyzing and reasoning about these systems. Our software is often so large that it simply cannot be manually audited effectively, but it is simultaneously so diverse and convoluted that building automated analysis tools can also present serious practical challenges. As

Authors' Contact Information: Dylan Wolff, wolffd@comp.nus.edu.sg, National University of Singapore, Singapore; Martin Mirchev, mmirchev@comp.nus.edu.sg, National University of Singapore, Singapore; Abhik Roychoudhury, abhik@nus.edu.sg, National University of Singapore, Singapore.

Communications of the ACM, 2024, Research Article

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

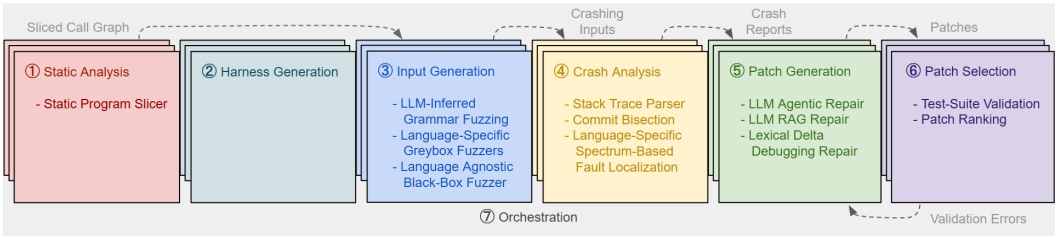

Fig. 1. A Cyber-Reasoning System Workflow; Example CRS tools listed with '-'s

a result, even the very tools developed to mitigate security issues in our software infrastructure are not exempt from the perils of growing complexity. This was made abundantly clear by the recent CrowdStrike incident[1], where the security system itself had a bug that caused worldwide outages across countless industrial sectors.

Many promising approaches have been proposed to improve aspects of software security, but have proven insufficient to address this issue in isolation. For example, fuzz testing techniques [20] –i.e. a biased random search over the domain of program inputs to detect vulnerabilities– have proven to be effective in real-world applications and at scale [24]. Yet the impact of the discovery of new vulnerabilities through fuzzing is greatly limited by the effectiveness of the engineers that must manually triage and fix them. Indeed, there is a dire shortage of cyber-security professionals, and a detected vulnerability may only be fixed in downstream, dependent software 90-150 days after it is reported [10]. Thus, to ensure that our software is truly secure, a holistic approach is needed that can automatically progress through all stages of the cyber-security pipeline. How can we discover *and* remediate vulnerabilities in complex software, prior to them being encountered in the field? What opportunities do recent advances in Large Language Models (LLMs) present for achieving this goal?

In this paper, we articulate in broad-brushes how LLMs can contribute to building so-called Cyber-Reasoning Systems (CRSs) which attempt to address these issues *autonomously*. We first outline the broad structure of a consolidated Cyber-Reasoning System (CRS) along with examples of the role that LLMs can play in various components of this system. Subsequently, we summarize the high-level technical challenges and opportunities in this emerging research area. Our article is partly inspired by the recently completed DARPA AI CyberChallenge[2] from which we take the term Cyber-Reasoning System, and partly also by the recent public interest in autonomous software engineering, where programs are fixed and improved automatically [11] using LLM agents such as AutoCodeRover [23, 29]. It is also informed by our research over the past decade in automated program repair and vulnerability discovery using variety of analysis techniques including search [5, 9], symbolic reasoning [21] and large language models [19, 29].

2 A Cyber Reasoning System

A *Cyber Reasoning System* (CRS) is a software system which can both detect and repair software vulnerabilities *autonomously* in a given System Under Test (SUT). Ideally, a CRS supports a wide range of real-world software systems, including those written in different programming languages and/or containing millions of lines of code. We note that substantial prior research has been conducted on various subgoals of a CRS [9, 18, 26]. Thus the core challenge of building a CRS lies in overall *flexibility*, *scale*, and the careful *combination* of disparate techniques into a coherent, effective system. Despite decades of research and industry interest, building such a CRS using only

[1]https://www.crowdstrike.com/falcon-content-update-remediation-and-guidance-hub/
[2]https://aicyberchallenge.com

traditional program analysis tools has proven to be impractical to the point of impossibility. The recent emergence of Large Language Models has made the realization of a CRS possible, via a broad new design space where traditional analysis techniques are replaced or augmented by LLMs.

A CRS Framework. To more clearly illustrate the challenges and opportunities presented by the construction of a CRS, we decompose it into seven sub-components, shown in Figure 1. A CRS might first use static analysis (①) to build a broad understanding of the SUT. This component could include general pre-analyses such as control-flow graph construction or more specific vulnerability scans to identify potentially buggy program locations. A CRS then needs to identify how to properly build and execute the SUT with new unit-tests or test inputs which could trigger a vulnerability (②). Once a CRS has a way to test the SUT, it (③) can search the space of possible testcases for those that may trigger a vulnerability via test generation. If a generated test triggers a vulnerability, the CRS can then analyze this test (④), along with other triggering and non-triggering tests to gather information about the vulnerability and determine if it has been previously discovered. This information can then be passed to patch generation tools (⑤), which attempt to find a repair for the given vulnerability. The CRS then selects (⑥) one or more patches that best fix the vulnerability without breaking the functionality of the program.

Figure 1 and the prior description presents a CRS as a mostly sequential pipeline. However, there may be other useful interactions between components; e.g. a crash analysis tool could incorporate information from a failed patch validation to refine its report (⑥ → ④). Additionally, each component may need to be scaled up or down dynamically: if a CRS finds many new vulnerabilities in a short period of time, test generation capacity could be reduced to shift additional resources towards crash analysis and repair of these vulnerabilities. As such, we identify orchestration (⑦) as an additional component of a CRS. The orchestrator determines resource allocation and facilitates communication between subcomponents. Any of these components, including the orchestrator itself, may leverage LLMs in lieu of or to complement existing program analysis tools.

The AIxCC Competition. The recent DARPA AI Cyber Challenge (AIxCC) Competition is the origin of the term Cyber-Reasoning System (CRS) and brought wide attention to this problem. The competition invited teams to build a CRS which could autonomously find and fix vulnerabilities in C and Java projects (including the Linux kernel), with the winner receiving a large cash prize. In this paper, we generalize the definition of CRS beyond the scope of the competition itself. For example, harness generation (②) was not part of the AIxCC competition, but is important for a CRS to be able to analyze real systems outside of a competition setting. Additionally, while the AIxCC competition was restricted to pre-trained models, a true CRS could include custom or fine-tuned models. on additional data. We also claim an ideal CRS should be able to analyze polyglot, heterogeneous software, including distributed systems, machine learning components, or embedded systems – which are were all outside the scope of the original competition.

An Example CRS. To make the concept of CRS more concrete, we present an open source cyber-reasoning system example in the style of the AIxCC competition: [https://github.com/DARPA-AIxCC/asc-crs-healing-touch]. The individual components of this CRS are outlined as bullet points in Figure 1. For input generation (③), it relies on an ensemble of fuzzers. As a baseline, it utilizes the existing greybox fuzzers LibFuzzer and Jazzer[3] for C and Java programs, respectively, in addition to a custom greybox fuzzer for the Linux kernel built with the LibAFL framework [8]. Additionally, it leverages LLMs to infer a grammar for the target program's inputs to allow a specialized fuzzer to generate well-formed test input. As fuzzers are a relatively mature and widely applied technology used in industry, we relied less on LLMs for this component. To analyze bugs found by our fuzzers

[3]https://llvm.org/docs/LibFuzzer.html and https://github.com/CodeIntelligenceTesting/jazzer

(④), the CRS parses the output from common vulnerability detection tools (called *sanitizers*[4]) and determine the bug introducing commit. For this component, we did not utilize LLMs at all, as the sanitizer output was given in a fixed format, but future work could leverage LLMs to be less sensitive to different crash reporting mechanisms. For Java programs, the CRS also uses the Joern[5] static analysis framework (①) to extract path constraints and then solve these constraints using an LLM, described in Section 3.2. Given a vulnerable input from a fuzzer, the CRS uses Spectrum-Based Fault Localization tools (SBFL) [26] to augment crash reports with suspicious lines before passing them on to be repaired. From a crash report, the CRS attempts to repair (⑤) the bug with three different repair tools. The first is a Retrieval Augmented Generation (RAG) [15] repair tool, which attempts to fix the bug using a single prompt to an LLM that includes similar vulnerability reports and patches. The second is an LLM agent AutoCodeRover [23, 29], which was adapted to work on C and Java programs. Lastly, it also includes a repair tool that does not utilize LLMs, instead using lexical delta-debugging[6] [28] on the differing lines between the current and pre-bug versions of the project. Once a patch is generated by any of the tools, it is selected for submission (⑥) if the validation tool Valkyrie determines that both the failing test input and the subject's functional test suite are free of crashes and errors. For orchestration (⑦), the CRS uses a custom Python framework which used a greedy static resource allocation across available hardware and passes data between components as shown by the dashed arrows in Figure 1. In the case of our example CRS, neither ⑥ nor ⑦ leverage LLMs, although such an integration is eminently feasible.

Example CRS Evaluation. While most of the subject programs from the AIxCC competition remain undisclosed, one –the `nginx` web-server– has been publicly released with 14 accompanying bugs from the semi-final. To assess the capabilities and potential of a CRS in more detail, we evaluated the example CRS described above on this benchmark in 20 trials of 4 hours each, with 64 allocated CPU cores and 16GB of allocated RAM. This emulates the setup for a single node of the three provided in the AIxCC semi-final competition. We also ran the CRS both with and without LLM-integrated components for bug discovery (③) and repair (⑤), respectively, to assess the impact of these components on the system as a whole. We used OpenAI GPT-4o (2024-08-06) as the sole LLM-backend of the CRS[7]. Overall, we saw that the example CRS is capable of finding *and* fixing vulnerabilities in an integrated workflow. This CRS finds more than two unique vulnerabilities on average per-trial ($\mu = 2.4$) and generates a plausible[8] patch for at least one of those discovered vulnerabilities in nearly half of all trials (9/20). In some instances, the plausible patches were also identical to the ground-truth patch given by the competition organizers. Additionally, in its best trial, the CRS found four unique vulnerabilities and generated plausible fixes for all four of these bugs within the time limit.

We also found that the components integrating LLMs were *crucial* to the success of the CRS. Removing the LLM inferred-grammar fuzzer from the CRS (③), the remaining fuzzers, including LibFuzzer, are unable to discover a single bug within a 4 hour time limit in *any* of 20 trials. To confirm these findings, we conducted an additional experiment with the state-of-the-art fuzzer AFL++, and saw that it also was unable to discover any bugs in any of 20 trials within the time-limit. We attribute the failure of conventional fuzzing to a lack of initial valid inputs provided in the competition; these are typically given to grey-box fuzzers to allow them to reach deeper program behaviors quickly. Similarly, when omitting the LLM-based repair tools (⑤), the remaining repair

[4]For example, ASAN detects memory safety bugs during program executions that might otherwise exit without errors
[5]https://github.com/joernio/joern
[6]https://security.googleblog.com/2024/06/hacking-for-defenders-approaches-to.html
[7]https://openai.com/index/hello-gpt-4o/
[8]Here plausible means the patched program no longer crashes on the PoV input and passes all competition-provided tests

tool was unable to generate a plausible patch for *any* of the found bugs, also across 20 trials. The delta-debugging repair tool struggles here because it is inherently sacrificing efficacy for generality: in order to be fully agnostic to the source language, it can only fix a small subset of regression bugs. We believe this difference highlights the importance of integrating LLM-assisted tooling into automated security workflows to create general and effective tools that can bridge gaps in existing analyses.

3 Opportunities for LLMs in Software Analysis

Each component of a CRS could leverage LLMs individually: static analysis (①) tools which analyze source code [1] could use LLMs to reduce the impact of syntactic noise; test harnesses (②) could be more robustly generated for different build systems and project structures by LLMs; fuzzers (③) could leverage LLMs for constraint solving; crash analysis and triage (④) using LLMs could respond dynamically to unforseen issues in production that cannot be accounted for *a priori*; patch generation (⑤) and selection (⑥) are already one of the most ubiquitous use-cases for LLMs; and lastly orchestration (⑦) is particularly well suited to agentic AI for making resource allocation and other planning decisions. There are many such design points, so we present three broad overarching opportunities for LLMs which generalize or span across multiple components of a CRS.

3.1 Augmenting Individual Analyses with Developer Intent

While Language Models can be powerful tools in their own right, prior work [29] and our experience in the AIxCC has shown us that LLMs can be most effective *in combination* with existing analyses.

Incomplete Analyses. Many algorithms in program analysis are *incomplete* in that they might report false positives, often because precise reasoning is algorithmically or practically impossible. A key opportunity for the application of language models lies in their ability to infer contextual intent, such as neighboring source code or comments, which can help filter or re-prioritize an analyzer's output. As an example, we used the SemGrep static analysis (①) tool to scan an open-source LLM orchestration project on Github[9] for hardcoded user credentials. The run of SemGrep generated a warning for the code shown in Figure 2b. This report is a false positive because the developer *intends* for users of the project to replace the `api_key` fields, as evidenced by the contents of the corresponding strings. Indeed, comments and other natural language artifacts often contain high-quality information, but this information has previously proven difficult to extract and leverage [13]. If we pass the report and source code context from Figure 2 to GPT-4o, it determines that the report can be ignored in 26 out of 30 trials.

```
rules:
 - id: hardcoded-openai-token
   message:
     - A hard-coded credential was detected. It is
       not recommended to store credentials in
      source-code...
   metadata:
     cwe:
      - "CWE-798: Use of Hard-coded Credentials"
     vulnerability_class:
      - "Leaking Secrets"
   severity: WARNING
   languages:
    - python
   pattern: OpenAI("...");
```

(a) Abbreviated SemGrep Rule for CWE-798

```
...
# Initialize OpenAI and Anthropic API clients
openai_client = OpenAI(api_key="YOUR API KEY")
anthropic_client = Anthropic(api_key="YOUR API KEY")
...
```

(b) Source Code for False-Positive of SemGrep Rule for CWE-798

Fig. 2. Example of An Unsound SemGrep Analysis

Unsound Analyses. Many other components of the CRS can be implemented with *unsound* program analyses. Such analyses can suffer from a large amount of *false negatives*. As with their incomplete counterparts, the usefulness of unsound analyses is often directly tied to the tightness

[9]https://github.com/Doriandarko/maestro/blob/main/maestro-gpt4o.py

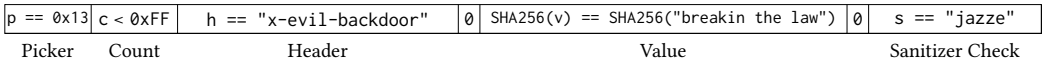

Fig. 3. Input Structure and Constraints for the Jenkins Exemplar Bug from the AIxCC Semi-final Competition

of this approximation; a fuzzer which misses many obvious bugs in the program is unlikely to be deployed in practice. Here, again, LLMs can leverage approximate developer intent to complement existing approaches and reduce false negatives. For example, an input grammar represents data that the program is *intended* to consume. Using an LLM, a CRS can quickly generate a grammar from the fuzzing harness. Producing such an input specification allows established techniques in grammar fuzzing [2] to rapidly explore the intended behavior of a program, as in our example CRS.

3.2 Generalizability and Flexibility of LLMs

Due to their immense training datasets, LLMs have been shown to generalize to a wide variety of technical tasks with little or no additional training [4].

Generalizability of Problem Domains. A CRS covers many disparate subproblems that previously required bespoke analysis tools to solve individually. LLMs lower the barrier to entry for writing these tools, allowing CRS builders to create components that are robust to different software versions, build tools, and environments while often remaining as capable as more fragile, complex analyses. One such analysis task is to demonstrate the Proof Of a Vulnerability (PoV) in the form of an input which triggers the bug (②). Figure 3 showcases one such PoV input for an exemplar bug from the Jenkins exemplar program for the AIxCC. In order to reach the buggy code, a CRS must craft a structured message with several fields exactly matched to particular byte-strings. The `Value` field, however, is hashed before it is compared to a desired byte-string, making it effectively *impossible* for typical constraint-solving approaches, such as symbolic execution, to solve analytically as this would amount to reversing a SHA256 hash. Yet, without additional training, a Language Model is able to generalize enough to attain the *effect* of a constraint solver without this limitation. To showcase this capability, we use a component of our example CRS (Section 2) – this tool gathers path constraints to a particular target location using static analysis and coalesces them into a prompt, see Figure 4. When querying the language model, out of 30 prompts, it synthesizes six correct and four nearly correct inputs. The nearly correct test inputs have only an additional null terminator, but are otherwise an exact match. We also ran the greybox fuzzer Jazzer for 24 hours, providing it with a "dictionary" file that contains all string constants present in the source code of the SUT, including the `Header` and `Value` strings needed for triggering the bug. In *none* of the 20 trials was the fuzzer able to find the vulnerability.

Prompting Template

```
You are a penetration testing engineer tasked with testing a Jenkins application.
Now, we notice there is one or more possible vulnerabilities of {{Vulnerability Class}}.
Note that we have a very different definition of this type of vulnerability. Here is the definition:
 {{Sanitizer Definition}}
Please ensure:
 {% for c in Path Constraints %} {{c}} \n {%endfor%}
Here are the dataflow slices of the program:
 {% for s in Dataflow Slices %} {{s}} \n {%endfor%}
This is a file that is related to the input data:
 {{Fuzz-Harness Source Code}}
Generate a Python program that dumps the expected data to the file 'input.bin'
```

Sample Templated Values

```
"Vulnerability Class": "OsCommandInjection",
"Sanitizer Definition": "for OSCommandInjection, APIs like `java.
      lang.ProcessBuilder' must contain a magic word named `jazze'
      rather than the actual command",
"Path Constraints": [
 "containsHeader(rqst.getHeaderNames(),\"x-evil-backdoor\")==true",
 "MessageDigest.isEqual(sha256, providedHash)) == true" ],
"Dataflow Slices": [
  "boolean isAllowed=jenkins().hasPermission(Jenkins.ADMINISTER);",
  "byte[] sha256 = DigestUtils.sha256(\"breakin the law\");",
  "String backdoorValue = request.getHeader(\"x-evil-backdoor\");",
  "byte[] providedHash = DigestUtils.sha256(backdoorValue);",  ...
```

Fig. 4. Prompting Template and Templated Values

Flexibility Across Languages. We manually translated the extracted constraints and harness in Figure 4 from Java to Python, resulting in Figure 5. We then passed these Python constraints to an LLM in the exact same prompting template with *no modifications*. In this scenario, the model was still able to solve the Python program constraints in 8/30 trials. This is remarkable in that the model did not require any changes to solve these constraints at the source level in *both* languages.

3.3 Opportunities for Agentic AI

AI Agents go beyond basic in-context learning by enabling LLMs to iteratively plan, reason, and take actions — such as invoking program analysis tools. As a result, they are often able to achieve complex, multi-step goals fully autonomously.

Agents can reasonably be used to instantiate any individual components of the CRS. Indeed, our example CRS uses the AutoCodeRover [23, 29] LLM agent to fix vulnerabilities (⑥). This agent uses code navigation and analysis tools to iteratively develop and refine a patch. AutoCodeRover has now been integrated in the widely used SonarQube static analysis tool to automatically fix detected security bugs, showing its practicality.

Templated Values Translated to Python

```
"Vulnerability Class": "OsCommandInjection",
"Sanitizer Definition":
 "for OSCommandInjection, APIs like `subprocess.Popen'
      must contain a magic word named `jazze' rather than
       the actual command",
 "Path Constraints": [
   "\"x-evil-backdoor\" in request.headers",
   cryptography.constant_time.bytes_eq(sha256,
      providedHash)",
],
"Dataflow Slices": [
 "isAllowed=jenkins().hasPermission(Jenkins.ADMINISTER)",
 "sha256=cryptography.hashes.Hash(crypto.hashes.SHA256())
      .update(\"breakin the law\").finalize()\")",
 "backdoorValue = request.headers[\"x-evil-backdoor\"]",
 "providedHash = cryptography.hashes.Hash(crypto.hashes.
      SHA256()).update(backdoorValue).finalize()\")",
      ...
]
```

Fig. 5. Python Constraints Passed to an LLM in the Template of Figure 4 for the *Translated* Jenkins Exemplar

Beyond individual analysis steps, the orchestration layer (Figure 1, ⑦) is particularly well-suited to AI agents; it involves numerous and repeated challenging planning decisions that would previously have been the purview of a human auditor. For example: If a vulnerability is found, should the CRS conduct additional static analysis of crash location or should it instead allocate those resources to further dynamic analysis of the vulnerable execution? What information from these analyses should be passed as context to the patch generator? As agents have begun to show significant progress in many individual tasks such as remediation [29], a more general and comprehensive deployment of LLM Agents *as the CRS itself* is promising area of future research.

4 Challenges for LLMs in CRSs

Unsoundness. A key strength of many static vulnerability scanners (①) is their soundness; they can sometimes *guarantee* a program to be free of particular vulnerabilities. An LLM cannot guarantee soundness alone [22], so we see determining how to recover these guarantees or increased assurance of correctness as a key challenge for LLM-based tools in a Cyber Reasoning System.

Incompleteness. As Language Models are statistical models at their core, they can also give *false positive answers* to analysis queries, amounting to an incomplete analysis. This may be mitigated in some situations where an efficient checking oracle exists [25] – for example, the input generation (③) task in Section 3.2 the LLM can be called iteratively until the oracle is passing i.e. the bug is triggered. Identifying these oracles will be an important area of research in CRSs.

Training and Evaluation Data. Language Models are highly dependent on the data used to train them. Because of their size, their corresponding training sets must also be extremely large to avoid over-parameterisation. Assembling large enough datasets for all components of the CRS workflow without introducing risk of *data leakage* [6] will be crucial to development and evaluation of CRSs.

5 Related Work

Vulnerability Repair. Code generation [3] and bug repairs [7] have been some of the applications of LLMs that can be utilized in a CRS. Early results indicate that LLM-based repair tools tend to outperform traditional repair tools on established datasets [27]. Similarly, LLMs have begun to be utilized for localizing the fault [12], with promising initial results. Agentic systems can automatically explore a project or invoke subtasks via repeated interactions with the LLMs [23, 29].

Vulnerability Discovery. Large Language models have also shown their capability in generating test harnesses [17] and unit tests [14]. Furthermore, they have been effective in fuzzing network protocols by providing the fuzzer a grammar for mutating the input messages [19]. In addition to dynamic analysis, LLMs are also beginning to see success in augmenting static analysis, particularly for bug discovery [16].

Benchmarks. SWEBench [11] s a widely used benchmark containing natural language description GitHub issues to be resolved by LLMs. These issues are mostly bug-fixes, but can span multiple files. Other benchmarks have even been developed to specifically evaluate LLM capabilities and risks [25]. To date, available benchmarks still only evaluate CRS sub-components, rather than a complete workflow; we hope that they can provide a starting point for a general CRS benchmark.

6 Threats to Validity

As CRSs are an emerging research area, our study was limited to the only dataset available for end-to-end CRS evaluation. As such, our results may not generalize beyond this dataset (c.f. Section 4). We did not evaluate the repair components in our CRS against traditional program repair research tools — a threat to the internal validity. However, we excluded these tools from the CRS due to their poor generality (c.f. Section 3.2), and similar comparisons have been published in prior work [27].

7 Perspectives

Innovative technologies often disrupt established sectors in unexpected ways with surprising speed. We believe that LLMs represent such a change, not just in how we write code, but also in how we reason about it. Automating this reasoning will be critical to keep up with our ever-growing, increasingly complex software infrastructure, and its security. While we can already see concrete benefits in augmenting or replacing existing software analysis with LLMs, such as fuzzing, symbolic execution or static analysis, many more possibilities remain. Broadly, we believe LLMs can complement gaps in existing analyses by grasping developer intent, generalizing to different programming constructs and environments, and providing agentic multi-step reasoning spanning numerous individual analyses. By focusing on these opportunities, researchers will be able to push the frontier of what is possible in an integrated Cyber-Reasoning System.

Acknowledgments

We thank Cristian Cadar (Imperial), Van-Thuan Pham (Melbourne), Lin Tan and Xiangyu Zhang (Purdue) and their respective research groups for their collaboration. This research is partially supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing <NRF-NCR25-Fuzz-0001>) and by a Singapore Ministry of Education (MoE) Tier 3 research grant (Automated Program Repair <MOE-MOET32021-0001>). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, MoE or Cyber Security Agency of Singapore.

References

[1] [n. d.]. https://semgrep.dev/

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for deep bugs with grammars.. In *NDSS*.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL] https://arxiv.org/abs/2108.07732

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* 45, 5 (2019), 489–506.

[6] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. 2022. Quantifying memorization across neural language models. In *The Eleventh International Conference on Learning Representations*.

[7] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.

[8] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1051–1065.

[9] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[10] Ponemon Institute. 2024. Costs and Consequences of Gaps in Vulnerability Response. https://www.servicenow.com/lpayr/ponemon-vulnerability-survey.html

[11] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. [n. d.]. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*.

[12] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 1424–1446.

[13] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. How good are the specs? a study of the bug-finding effectiveness of existing Java API specifications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 602–613.

[14] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-Trained Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 919–931.

[15] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[16] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499.

[17] Dongge Liu, Jonathan Metzman, Oliver Chang, and Google Open Source Security Team. [n. d.]. AI-Powered Fuzzing: Breaking the Bug Hunting Barrier. https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html

[18] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.

[19] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large Language Model guided Protocol Fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*.

[20] Barton P Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[21] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[22] Niklas Risse and Marcel Böhme. 2024. Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection. In *33rd USENIX Security Symposium (USENIX Security 24)*. 4247–4264.

[23] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2025. Specrover: Code intent extraction via llms. In *International Conference on Software Engineering (ICSE)*.
[24] Kostya Serebryany. 2017. {OSS-Fuzz}-Google's continuous fuzzing service for open source software. (2017).
[25] Shengye Wan, Cyrus Nikolaidis, Daniel Song, David Molnar, James Crnkovich, Jayson Grace, Manish Bhatt, Sahana Chennabasappa, Spencer Whitman, Stephanie Ding, Vlad Ionescu, Yue Li, and Joshua Saxe. 2024. CYBERSECEVAL 3: Advancing the Evaluation of Cybersecurity Risks and Capabilities in Large Language Models. arXiv:2408.01605 [cs.CR] https://arxiv.org/abs/2408.01605
[26] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.
[27] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 819–831.
[28] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on software engineering* 28, 2 (2002), 183–200.
[29] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *International Symposium on Software Testing and Analysis (ISSTA)*.