Carnegie Mellon University
Software Engineering Institute

Using LLMs to Automate Static-Analysis Adjudication and Rationales

Lori Flynn and Will Klieber

May 2024

This is a pre-publication version of the article that has been accepted for publication in the August 2024 edition of "CrossTalk: The Journal of Defense Software Engineering."

Abstract

Software vulnerabilities are a serious concern for the Department of Defense (DoD). Software analysts use static analysis as a standard method to evaluate the source code, but the volume of findings is often too large to review in their entirety, causing the DoD to accept unknown risk. Large Language Models (LLMs) are a new technology that show promising initial results for automation of alert adjudication and rationales. This has the potential to enable more secure code, better measure risk, support mission effectiveness, and reduce DoD costs. This article discusses our model for using LLMs to handle static analysis output, initial tooling we developed and our experimental results, related work by others, and additional work needed. Beyond static-analysis alert adjudication, similar techniques can be used to create LLM-based tools for other code analysis tasks.

Motivation for Improving Static Analysis on DoD Software

The Authorization To Operate (ATO) process assesses risks that software may introduce [1]. During Test & Evaluation (T&E) and Independent Verification & Validation (IV&V), software analysts evaluate source code for security weaknesses to measure risk and enable code improvement in preparation of ATO and fielding. Static analysis (SA) is widely used and is one of the best techniques available: it is much more practical than full formal verification, and it can catch vulnerabilities that can evade dynamic analysis. But static analysis still requires significant manual effort and is inherently difficult, time-consuming, and expensive (See Section "Static Analysis vs. Dynamic Analysis" for more information about these methods). Manual effort is required for each SA alert to adjudicate whether it is a true or false positive, since SA tools sometimes produce false positives. There are many types of code flaws identified in taxonomies such as the Common Weakness Enumeration (CWE), and SA tools produce alerts for many types. Human analysts must be able to analyze each kind to be able to adjudicate the alert, which requires great expertise.

The most common strategy to adjudicate alerts given finite time and resources is to prioritize potential vulnerabilities by a combination of likelihood (with static-analysis tools usually pre-filtering out unlikely ones in their default configurations) and severity (e.g., Security Technical Implementation Guide (STIG) Category 1 [2]) and then manually review only the top alerts. However, even code weaknesses in lower-severity Application Security and Development (ASD) STIG categories can also cause costly mission failure. Many types of code flaws can lead to vulnerabilities that common attack patterns use; e.g., the

Common Attack Pattern Enumeration and Classification (CAPEC) [3] describes an attack pattern [4] that takes advantage of a lower-category weakness [5]. As another example, the Ariane flight V88 rocket explosion (which resulted in a loss of more than $370 million) was caused by code flaws that static analysis tools can detect (integer overflow and improper exception handling) [6][7] but that often aren't put in STIG Category 1.

Latest LLMs as Breakthroughs for Automating Static Analysis Alert Adjudication

Large Language Models, such as GPT-4 (OpenAI's latest Generative Pre-trained Transformer (GPT)) [8], present a significant breakthrough, for two major reasons:

1. They produce a detailed explanation to support their final answer, in contrast to older machine learning (ML) techniques [9] which involve statistical algorithms that can learn from data and generalize to unseen data. These older ML techniques lacked interpretability and often pivoted on irrelevant details that merely correlated with vulnerabilities in their training data. The generated explanation can be double-checked by both humans and the LLM itself.
2. They can generate and use function summaries, function preconditions, and other intermediate results to enable LLM-based tools to adjudicate alerts whose adjudication requires analyzing multiple functions spread across the codebase.

Chan et al. use LLMs to detect over 250 vulnerability types in code being edited. They deployed their model as a VSCode extension with ~100K daily users, with a 90% reduction in the rate of vulnerabilities in developer code [10]. Fan et al. developed an intelligent agent that responds to queries by processing code and interactions with LLMs, SA tools, code retrieval tools, and web search tools to check intentions of code segments and detect bugs [11]. LLMAO (LLM fAult lOcalization) [12] is an LLM-based approach for localizing program defects at line level, outputting bug probabilities for each line of code. It localized more bugs in the same set of benchmark codebases than the previous best deep-learning fault localizer, and it doesn't require any additional training or test cases to handle unseen projects. It uses a bidirectional language model, allowing it to consider both preceding and following lines.

Two recent papers have attempted to quantify the benefits of applying LLMs to the problem of pruning false positives from static-analysis alerts [13][14]. Both have explored the benefits of designing a prompting algorithm, highlighting the importance of chain-of-thought, task decomposition, and progressive prompting strategies. One found that an LLM-enabled system demonstrated high precision and recall in a real-world scenario and even identified 13 previously unknown use-before-initialization (UBI) bugs in the Linux kernel [14]. While these studies provide useful templates for system design, they do not fully address the DoD's challenges because they both focused on the relatively narrow application of UBI bugs in the Linux kernel as a single case study.

Sherman [16] found that LLMs often perform poorly when asked to find all security issues in a snippet of code. We have found that LLMs do much better when asked to adjudicate a specific type of issue on a specific line. Li et al [13] also found that GPT-4 works well for this task.

Our Initial Results Using LLMs for SA Alert Adjudications

We developed a model of how an LLM-based tool could be used for SA alert adjudications, shown in Figure 1. The LLM-based tool ingests source code and SA alerts, then creates a query (a "prompt") to the

LLM for each alert. The LLM ideally outputs adjudicated true positives along with a trace, adjudicated false positives with a proof sketch, or it adjudicates as "uncertain."

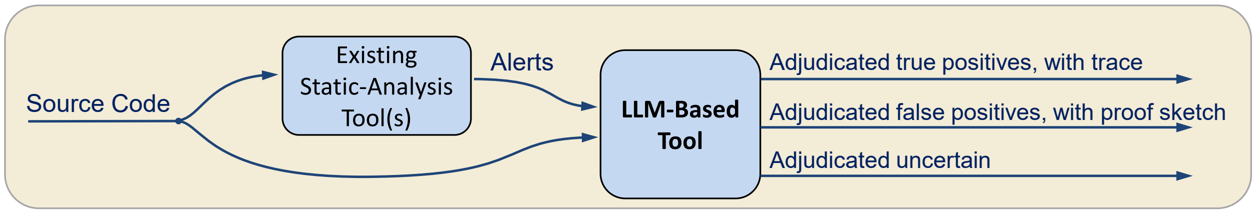
Source Code
Existing Static-Analysis Tool(s)
Alerts
LLM-Based Tool
Adjudicated true positives, with trace
Adjudicated false positives, with proof sketch
Adjudicated uncertain

Figure 1. *Using LLMs for SA Alert Adjudications.*

We developed partial automation to test this concept. A script inserts "// ALERT" to the code line that the SA alert identifies. The script creates an LLM prompt that includes the source code of the function that contains the alerted-about line, the type of code flaw to adjudicate (e.g., "integer overflow"), and additional data from the alert.

In this article, the GPT-4 links (meaning all the links that start with "https://chat.openai.com") go to webpages that show tests that we conducted. They show the exact input that we provided to GPT-4. Each page also shows the full text of the responses from GPT-4, which often includes extensive step-by-step analysis of the code and the possible code flaw. We provide summaries and encourage those interested in additional detail to look at the full interactions shown at those links.

We note that GPT-4 is reliable at correctly following instructions to produce JSON (JavaScript Object Notation) output in a specified schema, making it relatively easy to write a script to parse the output from a GPT-4 API (application programming interface) call. In the rare case that it fails to produce output in the correct format, we simply try again until it produces output in the correct format.

Strategies for Mitigating Context-Window Limits

LLMs have a limited context window, which means that an LLM can usually ingest a single function but not an entire codebase. Sometimes, the LLM can make an adjudication based only on the function that contains the flagged line of code, but in other cases, additional context is needed. To overcome the context-window limit, we must summarize the relevant parts of the codebase enough so that the LLM can digest it.

Some strategies for this have been documented in the literature:

- Use traditional static analysis to produce required information, as in [17].
- Use the LLM itself to generate the function summaries, as in [14].

We have also tested a couple other strategies:

1. As part of the prompt, direct the LLM to ask for needed information. Our tool will then supply it to the LLM. Example: https://chat.openai.com/share/b01b0394-55f2-49f7-8a11-bfda15362297

   Figure 2 and **Figure 3** show the start and end of the prompt, respectively.

I want you to adjudicate whether a static-analysis alert is correct or a false alarm. The alert message is "Null pointer passed to 1st parameter expecting 'nonnull'". If you need to know the behavior of other functions (e.g., whether the function aborts execution), please ask and I will provide their source code. The alerted line-of-code is marked in the below snippet with "/* ALERT */":

Figure 2: *Start of prompt directs LLM to ask for needed information (source code not displayed here for brevity)*

If you can determine whether the alert is correct or a false alarm, please indicate this determination and explain your reasoning, and at the end of your response, say either `{"answer": "true positive"}` or `{"answer": "false positive"}`. If you need the source code of other functions, please indicate which functions you need, using the format `{"needed_functions": ["func1", "func2", ...]}`, and I will provide their source code

Figure 3. *End of (same) prompt directs LLM to ask for other types of needed information.*

2. Use the LLM to generate preconditions for avoiding a bad state in a function with an alert, and then use the LLM to check whether the callers of the function satisfy the preconditions.
    a. Example of creating a precondition: https://chat.openai.com/share/cfeabe6f-5757-4c25-be82-f9569f8c9df2
    In this example, GPT-4 analyzes a function named “greet_user” that takes a string as an argument. GPT-4 is asked to adjudicate an alert about a buffer overflow. In its response, GPT-4 correctly determines that the buffer overflow can happen only if the length of the input string is too long. It returns a precondition for avoiding the buffer overflow:
    [{"precond": "strlen(username) <= 52", …}]
    b. Example of using a precondition: https://chat.openai.com/share/bbbf7df7-4fba-43b1-8f46-f09c4bd290cb
    In this example, GPT-4 analyzes a function that calls the “greet_user” function analyzed above. GPT-4 is given the precondition that it previously computed, and it is asked whether this precondition is satisfied. It correctly determines that the precondition can be violated.

Figure 4 shows an example of the strategy of creating and using preconditions.

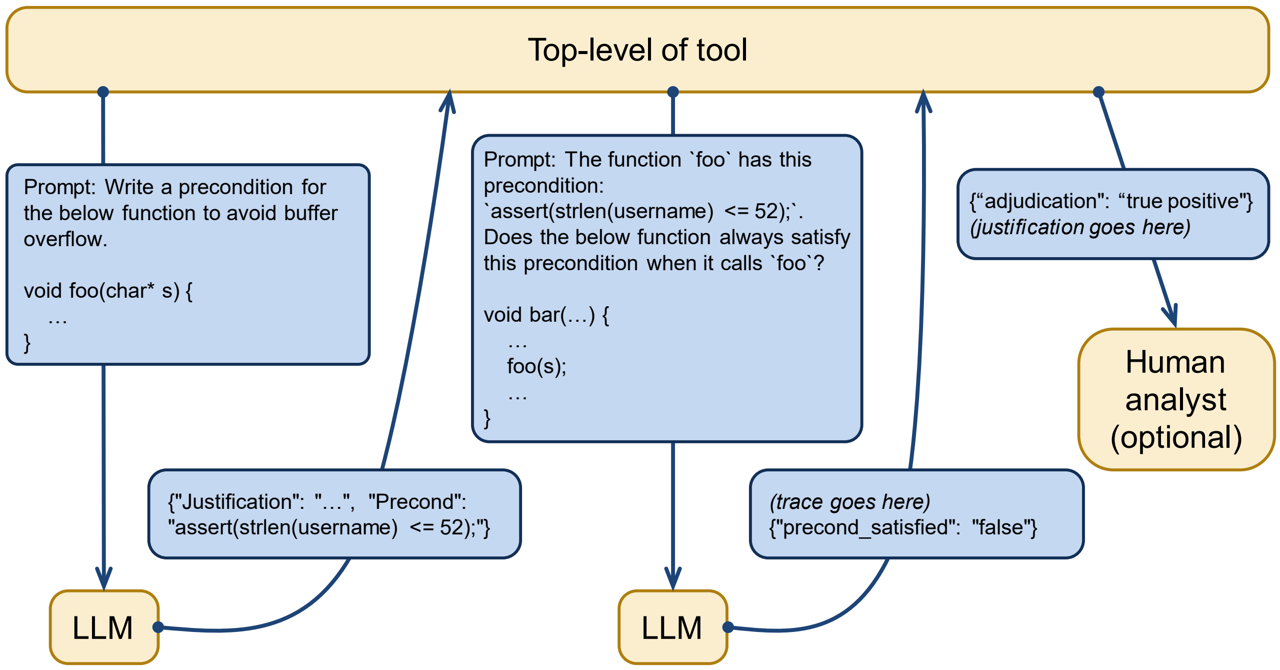
Top-level of tool
Prompt: Write a precondition for the below function to avoid buffer overflow.
void foo(char* s) {
…
}
{"Justification": "…", "Precond": "assert(strlen(username) <= 52);"}
LLM
Prompt: The function `foo` has this precondition: `assert(strlen(username) <= 52);`. Does the below function always satisfy this precondition when it calls `foo`?
void bar(…) {
…
foo(s);
…
}
(trace goes here)
{"precond_satisfied": "false"}
LLM
{"adjudication": "true positive"}
(justification goes here)
Human analyst (optional)

Figure 4. *Creating and Using Preconditions.*

Example: GPT-4 Adjudicating an Alert in the Linux Kernel

This example demonstrates GPT-4 successfully adjudicating an alert for vulnerability CVE-2022-41674 [15], about an integer-overflow leading to a buffer overflow in the Linux kernel: https://chat.openai.com/share/4ce0cdae-47b7-4648-9462-9e0a381ccc37

First, our script adds comments identifying two code locations that the alert specifies.

Next, we submit a prompt to GPT, which has a few sections:

a. The first part of the prompt is the following text:

   I want you to adjudicate whether a static-analysis alert is correct or a false alarm. The alert warns of a buffer overflow during `memcpy` on the line ending with "// ALERT-2" that happens if there is an integer overflow on the line ending with "// ALERT-1."

b. The middle part of the prompt consists of the source code of the alerted-about function.

c. The final part of the prompt is the following text:

   If you can determine whether the alert is correct or a false alarm, please indicate this determination and explain your reasoning, and at the end of your response, say either `{"answer": "true positive"}` or `{"answer": "false positive"}`. First identify whether integer overflow can happen. If it can't, then report the alert is a false positive. If it can happen, then examine whether it can lead to a buffer overflow.

Step-by-step, GPT-4 determines the following, concluding that the alert is a true positive:

1. An integer overflow can happen on the line `cpy_len = mbssid[1] + 2; // ALERT-1` if `mbssid[1]` is equal to 255, since cpy_len is an unsigned 8-bit integer.

2. GPT-4 analyzes the relation between the allocated size of the `new_ie` buffer (into which `pos` points) and the amount being copied into it. It determines that a large value of `mbssid[1]` should (and does) result in a small allocated buffer and should (but doesn't) result in a small amount copied into the buffer. Due to the integer overflow, a large amount is actually copied into the small buffer, overflowing the buffer.
3. It then provides its final answer at the end of its response, in the format requested by the prompt:
   "`{"answer": "true positive"}`."

We've summarized GPT's determinations, but it's important to note that GPT-4 more fully states the basis for its reasoning prior to the determination and that a human analyst can verify its reasoning.

If asked about the patched version, GPT-4 correctly identifies that the vulnerability is no longer present: https://chat.openai.com/share/7ee8e60b-1fed-4b67-b77b-7edd289fee90

Example: GPT-4 Double-Checking its Work

LLMs sometimes make mistakes. One way to mitigate is to run the LLM multiple times and check that the answers aren't inconsistent except on a small threshold percent of runs. Another mitigation is to ask the LLM to double-check its own work. This is an example of GPT-4 double-checking its work:

- GPT-4 mistakenly claims that a program has a use-after-free bug, but it corrects itself when asked to double-check the reported trace: https://chat.openai.com/share/78200ee7-37e7-413d-8b2d-fd9051c28106

  The first prompt provides the source code for the function and tells GPT-4 to:
  "Investigate whether there any use-after-free vulnerabilities in this C++ code, thinking step-by-step. Don't worry about any other problems with code; focus exclusively on use-after-free bugs. If you find a use-after-free bug, provide a trace that demonstrates memory being used after being freed."

  GPT-4 responds by defining use-after-free vulnerabilities: "A use-after-free vulnerability occurs when a program continues to use a pointer after it has been freed." Then, it inspects code for pointers and allocation and freeing of memory for the pointers it finds, and provides its analysis and then steps of a trace it thinks ends with a use-after-free, ending with this summary:
  "The vulnerability arises due to the fact that the `R` command can potentially free memory that's still being pointed to by entries in `sub_list`. To fix this, we should avoid using the reallocated memory from `getline()` for items that have been copied to the `sub_list` or ensure we update any relevant pointers in `sub_list` when renaming in `full_list`."

  The next prompt is: "OK, now please double-check that the trace is correct, stepping thru the code in detail."

  GPT-4 responds with a detailed step-by-step walkthrough, then ends with: "Upon a detailed step-by-step walkthrough, the trace provided previously appears to be incorrect. The use-after-free scenario doesn't occur because when an item from `full_list` is copied to `sub_list` using the 'C' command, a duplicate of the string is created with `strdup()`, ensuring that

`sub_list` entries are independent of any reallocations or frees that might occur with `full_list` entries.
Therefore, there is no use-after-free vulnerability with the described trace in the provided code."

- Deleting the `strdup` call from the above example introduces a real use-after-free bug, and GPT-4 correctly identifies this use-after-free bug and correctly reports that it is a real bug when asked to double-check the reported trace in detail: https://chat.openai.com/share/36b75aaa-d52a-480c-8694-f8f62e64fdcc
  GPT-4's detailed double-check ends with, "After a detailed step-through, the trace does hold. The use-after-free vulnerability indeed exists, and the trace correctly demonstrates it."

Example: GPT-4 Finding and Proving Loop Invariants

GPT-4 has some ability to find and prove loop invariants. Example:

- Example: https://chat.openai.com/share/88c782ff-c7b7-4d4c-8cb1-988df4a3f1a8

The full initial prompt is shown in **Figure 5**, including source code for the function. GPT-4 steps through the code, finds the loop invariant ("0 <= low <= high < n"), and then sketches a proof of the loop invariant. A second prompt, "Please analyze the Maintenance step in more detail," elicits additional detail from GPT-4 which supports its previous adjudication. That GPT-4 response ends with, "In both scenarios, whether we're updating `low` or `high`, we can see that our loop invariant `0 <= low <= high < n` is maintained. Thus, during any given iteration of the loop, the index `mid` always falls within the safe range `[0, n-1]`, ensuring there is no buffer overflow."

Investigate whether there is a buffer overflow in the below code, thinking step-by-step. If possible, generate a precondition that guarantees absence of a buffer overflow, and give a proof sketch demonstrating that the precondition guarantees absence of buffer overflow. If helpful, find and prove a loop invariant.
```
bool binary_search(int arr[], int n, int x) {
  int low = 0;
  int high = n - 1;

  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (arr[mid] == x) {
      return true;
    }
    if (arr[mid] < x) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  return false;
}
```

Figure 5. *Prompt for No-Buffer-Overflow Proof.*

LLM SA Alert Adjudication Can Also Be Used for Automated Code Repair and Shifting Security Left

LLM Adjudications can be Combined with Automated Code Repair

SA alert adjudication by LLMs can be used in combination with automated repair technology (either LLM-based or using an algorithmic approach [20][19]). This combination can be used to greatly speed up the process of hardening a system against cyberattacks.

Shifting Security Left

The need for improved static analysis effectiveness also applies to DevSecOps [21], which is a set of practices that combines code development and IT operations (DevOps) with security integrated throughout.

A continuous authorization to operate (cATO) formalizes and monitors specific technologies to reduce risk [23][24]. Such evaluation is done as part of ongoing DevSecOps, if there is a cATO. Integrated developer environments (IDEs) can do static analysis for some flaws while developers are coding [25], plus static analysis can be run as part of continuous integration testing.

Static Analysis vs. Dynamic Analysis

Dynamic analysis executes the code, running it in particular test environments with a set of test inputs. Two examples: Fuzz testing automatically injects inputs to try to reveal defects, monitoring for negative effects such as crashes and memory leaks, with black-box, grey-box, and white-box categories of fuzzers having different knowledge about the software [26]. Dynamic taint analysis inspects data sources and sinks during execution to identify data flows that should not happen, such as leaking sensitive data to a remote web address [27].

Static analysis analyzes code without executing it, using techniques that often include automated parsing of the code into a grammar and abstract syntax tree; creating a control flow graph; and analyzing data flow, control flow, and/or type flow to inspect for code flaws that could lead to security or functional problems [28][29].

Formal verification is a mathematical approach to check whether a software system meets formal specifications. Formal verification is a type of a static analysis, although the term "static analysis" usually connotes a less rigorous type of static analysis. There are several techniques used in formal verification, including model checking, theorem proving, abstract interpretation, and equivalence checking. Formal verification has been successfully used on small software systems (e.g., the seL4 microkernel), but it often is impractical for large software systems.

Previous Work with AI/ML for Static Analysis

To date, there has been a significant amount of research on using machine learning to aid in efficiently identifying source code flaws [30][31][32][33]. Researchers trained the ML using manually-adjudicated alerts ("labeled data") and features such as code cohesiveness metrics, lines of code per function and file, developer ID, and recency-of-code edits around that code location. Some work found that aggregating alerts from multiple SA tools for the same code location improved classifier precision [34], and other work developed a lexicon and adjudication rules to enable consistent adjudications to improve classifier training data [35]. Classifiers trained on labeled data from the same codebase generally perform better than those trained on data from different codebases (latter is called "cross-project prediction"), but techniques have been developed that improve cross-project prediction [36]. The high cost and time required for experts to manually adjudicate and create enough labeled data can be a barrier to creation of accurate static analysis classifiers. Flynn et al. made novel use of test suites [37][38] to create large datasets of labeled (true/false) SA alerts (augmenting ~7500 manually-adjudicated alerts on natural code). This resulted in high-precision ML classifiers for a larger set of CWE types than the natural dataset alone [39][40] and created a framework for use with multiple SA tools, ML classification, adaptive heuristics, labeled datasets, and test suites [41][42]. Gallagher et al. also used ML for finding code flaws but did that using LLVM intermediate representation instead of source [43] (LLVM is a set of compiler and toolchain technologies [44]). Flynn and Gallagher both found that artificial code and flaw injection can cause classifiers to use features not helpful with natural code.

As part of modern continuous integration (CI) code development, automatically cascading adjudications (manual adjudications true or false for alerts) to SA alerts for a later version of the codebase is important, but there are tradeoffs between fast `diff`-based cascading methods and higher-precision methods that may be too slow for practical CI use [45]. Classifier accuracy depends on the quality of the labeled data the classifier is created with, so incorrectly cascaded alert adjudications can be expected to produce lower accuracy in the resulting classifier [46].

LLMs for Education on Alert Adjudication, Coding Standards, and Flaw Taxonomies

Another potential benefit of using LLMs for static analysis adjudication involves education. Static analysis alert adjudication skills require understanding the coding standard for the programming language, understanding the code flaw taxonomy (such as CWE or CERT Coding Rules [47]), being able to grasp what is happening in the code, and understanding the static analysis tool's alert messages and checkers. A human analyst can try to follow the LLM's rationale, validating its claims by checking the code, language standard, and taxonomy item. Regardless of correctness of the LLM's rationale, the person can learn by following the reasoning and inspecting the source documents (code, standard, and taxonomy). A developer using an LLM to adjudicate SA alerts on their own code might learn to avoid inserting flawed code constructs, understanding why and how by following the LLM's rationale for a true positive adjudication. Early-stage research found that novice and expert programmers had significantly different needs for guidance, personalization, and integration of an LLM (e.g., with less confidence in their own capabilities, a novice may place too much trust in the LLM's output, while expert programmers reported more learning through use of LLMs). The research subjects lacked diversity (most highly educated and male), so further research is needed to determine how generalizable those findings are [48].

Future Directions

The 2024 Cyber Developmental Test and Evaluation policy and guidance specifies a responsibility to "Identify, assess, and document potential weaknesses that could affect technical, functional, and operational performance" [49]. Use of LLMs to support efficient and correct static analysis adjudication shows potential to improve performance on that mission-impacting responsibility.

Further work is needed to validate and widen the impact of previous results discussed in this article, by applying the methods to a wide range of DoD codebases and more types of code flaws and static analysis tools.

Looking to the future, LLMs may greatly help to enable formal verification of software, an area that has long been impractical for large codebases due to the amount of manual effort involved. Wu et al [50] report success in using LLMs for generating formal proofs on a benchmark of formal-verification problems [51], beating state-of-the-art formal-verification tools on a number of hard cases. Generating and proving loop invariants and function pre-/post-conditions is often a crucial and challenging part of formal verification, and as evidenced by our initial experimental results, LLMs show promise for helping with this task.

Acknowledgments

Carnegie Mellon University 2024. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific entity, product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute nor of Carnegie Mellon University - Software Engineering Institute by any such named or represented entity.
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.
CERT® and Carnegie Mellon® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM24-0303

References

1. Dukes, Curtis W. "Committee on National Security Systems (CNSS) Glossary, CNSSI No. 4009." *DoD Ft Meade* (2015).
2. Synopsis. “Coverity DISA ASD STIG report”, Synopsis webpage. https://sig-product-docs.synopsys.com/bundle/coverity-docs/page/reports/disa-stig/coverity_disa_stig.html. Accessed 5 March 2024.
3. MITRE, “Common Attack Pattern Enumeration and Classification (CAPEC)”, https://capec.mitre.org/.
4. MITRE, “CAPEC-124: Shared Resource Manipulation”, https://capec.mitre.org/data/definitions/124.html
5. MITRE, CWE-1331: Improper Isolation of Shared Resources in Network On Chip (NoC)”, https://cwe.mitre.org/data/definitions/1331.html
6. “Ariane flight V88." Wikipedia, The Free Encyclopedia, 8 Feb. 2024. https://en.wikipedia.org/w/index.php?title=Ariane_flight_V88&oldid=1204984683.
7. Lions, Jacques-Louis. "Flight 501 failure." *Report by the Inquiry Board* 190 (1996).
8. OpenAI, “Research: GPT-4”, OpenAI contributors, published 14 March 2023, https://openai.com/research/gpt-4
9. Flynn, Lori. "Challenges and Progress: Automating Static Analysis Alert Handling with Machine Learning." (2018). https://insights.sei.cmu.edu/documents/4174/2018_017_101_518025.pdf
10. Chan, Aaron, et al. "Transformer-based vulnerability detection in code at EditTime: Zero-shot, few-shot, or fine-tuning?." *arXiv preprint arXiv:2306.01754* (2023).
11. Fan, Gang, et al. "Static Code Analysis in the AI Era: An In-depth Exploration of the Concept, Function, and Potential of Intelligent Code Analysis Agents." *arXiv preprint arXiv:2310.08837* (2023).
12. Yang, Aidan ZH, et al. "Large language models for test-free fault localization." *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 2024.

13. Li, Haonan, et al. "Assisting static analysis with large language models: A chatgpt experiment." *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2023.
14. Li, Haonan, et al. "The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models." *arXiv preprint arXiv:2308.00245* (2023).
15. National Vulnerability Database contributors. "CVE-2022-41674 Detail", National Vulnerability Database (NVD), National Institute of Standards and Technology. https://nvd.nist.gov/vuln/detail/CVE-2022-41674 accessed 14 March 2024.
16. Sherman, Mark. "Should I Trust ChatGPT To Review My Program?", *InfoSec World,* (2023).
17. Ahmed, Toufique, et al. "Improving few-shot prompts with relevant static analysis products." *arXiv preprint arXiv:2304.06815* (2023).
18. SEI CERT Coding Rules Contributors, "INT32-C. Ensure that operations on signed integers do not result in overflow", SEI CERT Coding Standard for C, Software Engineering Institute of Carnegie Mellon University. https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow accessed 14 March 2024.
19. CWE Contributors. "CWE-190: Integer Overflow or Wraparound", MITRE Common Weakness Enumeration. https://cwe.mitre.org/data/definitions/190.html accessed 14 March 2024.
20. Monperrus, Martin. "The Living Review on Automated Program Repair", Technical Report HAL # hal-01956501, 2018. https://hal.science/hal-01956501/document/.
21. Nichols, W. R., Yasar, H., Antunes, L., Miller, C. L., & McCarthy, R. (2022). Automated data for DevSecOps programs. Acquisition Research Program.
22. Klieber, William, et al. "Automated code repair to ensure spatial memory safety." 2021 IEEE/ACM International Workshop on Automated Program Repair (APR). IEEE, 2021.
23. McKeown, David W. "MEMORANDUM FOR SENIOR PENTAGON LEADERSHIP DEFENSE AGENCY AND DOD FIELD ACTIVITY DIRECTORS." Subject: Continuous Authorization To Operate (cATO). DoD Senior Information Security Officer (SISO). 3 Feb. 2022, https://media.defense.gov/2022/Feb/03/2002932852/-1/-1/0/CONTINUOUS-AUTHORIZATION-TO-OPERATE.PDF
24. Lam, T., and N. Chaillan. "DoD Enterprise DevSecOps Reference Design." Department of Defense, Chief Information Officer Library, 12 Aug. 2019. https://dodcio.defense.gov/Portals/0/Documents/DoD%20Enterprise%20DevSecOps%20Reference%20Design%20v1.0_Public%20Release.pdf
25. Microsoft developers, Visual Studio Code, Microsoft. https://code.visualstudio.com accessed 15 March 2024.
26. Lemieux, Caroline, and Koushik Sen. "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage." *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 2018.
27. Newsome, James, and Dawn Xiaodong Song. "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software." *NDSS*. Vol. 5. 2005.
28. Harrold, Mary Jean, and Mary Lou Soffa. "Interprocedual data flow testing." *ACM SIGSOFT software engineering notes* 14.8 (1989): 158-167.
29. Horwitz, Susan, Thomas Reps, and David Binkley. "Interprocedural slicing using dependence graphs." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.1 (1990): 26-60.
30. Heckman, Sarah, and Laurie Williams. "A systematic literature review of actionable alert identification techniques for automated static code analysis." *Information and Software Technology* 53.4 (2011): 363-387.

31. Kremenek, Ted, et al. "Correlation exploitation in error ranking." *ACM SIGSOFT Software Engineering Notes* 29.6 (2004): 83-93.
32. Flynn, Lori, et al. "Static Analysis Classification Research FY16 20: for Software Assurance Community of Practice." (2020): 68.
33. Ruthruff, Joseph R., et al. "Predicting accurate and actionable static analysis warnings: an experimental approach." Proceedings of the 30th international conference on Software engineering. ACM, 2008.
34. Flynn, Lori. “Prioritizing Alerts from Static Analysis with Classification Models”, Software Engineering Institute, Research Review 2016, 1 Nov. 2016. https://insights.sei.cmu.edu/library/prioritizing-alerts-from-static-analysis-with-classification-models-2/. Accessed 5 March 2024.
35. Svoboda, David, Lori Flynn, and Will Snavely. "Static analysis alert audits: Lexicon & rules." *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 2016.
36. Wang, Song, Taiyue Liu, and Lin Tan. "Automatically learning semantic features for defect prediction." *Proceedings of the 38th International Conference on Software Engineering*. 2016.
37. Intelligence Advanced Research Projects Activity (IARPA) contributors. STONESOUP”, National Institute of Standards and Technology (NIST) Software Assurance Reference Dataset (SARD). https://samate.nist.gov/SARD/documentation#iarpa. Accessed 5 March 2024.
38. Black, Paul E. *Juliet 1.3 test suite: Changes from 1.2*. US Department of Commerce, National Institute of Standards and Technology, 2018. https://nvlpubs.nist.gov/nistpubs/TechnicalNotes/NIST.TN.1995.pdf accessed 5 March 2024.
39. Flynn, Lori, et al. "Prioritizing alerts from multiple static analysis tools, using classification models." *Proceedings of the 1st international workshop on software qualities and their dependencies*. 2018.
40. Flynn, Lori, William Snavely, and Zachary Kurtz. "Test suites as a source of training data for static analysis alert classifiers." *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 2021.
41. Flynn, Lori, et al. "SCAIFE API (OpenAPI v.3).” Software Engineering Institute site at GitHub, https://github.com/cmu-sei/SCAIFE-API/. Accessed 5 March 2024.
42. Flynn, Lori, et al. "SCALe Repository (scaife-scale branch)." Software Engineering Institute site at GitHub, https://github.com/cmu-sei/SCALe/tree/scaife-scale Accessed 5 March 2024.
43. Gallagher, Shannon K., et al. "LLVM intermediate representation for code weakness identification." (2022). https://apps.dtic.mil/sti/trecms/pdf/AD1178536.pdf accessed 5 March 2024.
44. “The LLVM Compiler Infrastructure”. https://llvm.org/ accessed 5 April 2024.
45. Guo, Xiuyuan, et al. "A Study of Static Warning Cascading Tools (Experience Paper)." *arXiv preprint arXiv:2305.02515* (2023).
46. Flynn, Lori. "Rapid Adjudication of Static Analysis Alerts During CI.", Research Review Presentation, Software Engineering Institute at Carnegie Mellon University, (2020): 35.
47. Carnegie Mellon University Software Engineering Institute CERT Division and community contributors. “SEI CERT Coding Standards”. https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards accessed 25 April 2024.
48. Chen, John, et al. "Learning Agent-based Modeling with LLM Companions: Experiences of Novices and Experts Using ChatGPT & NetLogo Chat." *arXiv preprint arXiv:2401.17163* (2024).
49. Standard, Sarah. “DoD Cyber Developmental Test and Evaluation Policy and Guidance Policy and Guidance”, DAU Webinar, 01 Feb. 2024. https://www.dau.edu/events/cyber-dte Accessed 04 March 2024.

50. Wu, Haoze, Clark Barrett, and Nina Narodytska. "Lemur: Integrating Large Language Models in Automated Program Verification." *arXiv preprint arXiv:2310.04870* (2023).
51. SV-COMP contributors. “SV-Benchmarks”, Competition on Software Verification (SV-COMP) benchmarks, International Competition on Software Verification. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks. Accessed 3 March 2024.