Proceedings of the 58th Hawaii International Conference on System Sciences | 2025

Using LLMs to Adjudicate Static-Analysis Alerts

Will Klieber
Software Engineering Institute, Carnegie Mellon Univ.
weklieber@sei.cmu.edu

Lori Flynn
Software Engineering Institute, Carnegie Mellon Univ.
lflynn@sei.cmu.edu

Abstract

Software analysts use static analysis as a standard method to evaluate the source code for potential vulnerabilities, but the volume of findings is often too large to review in their entirety, causing the users to accept unknown risk. Large Language Models (LLMs) are a new technology with promising initial results for automation of alert adjudication and rationales. This has the potential to enable more secure code, support mission effectiveness, and reduce support costs. This paper discusses techniques for using LLMs to handle static analysis output, initial tooling we developed, and our experimental results from tests using GPT-4 and Llama 3.

Keywords: LLM, Software, Cybersecurity.

1. Introduction

1.1. Motivation for Improving Static Analysis

In high-assurance areas (such as military systems, avionics, medical devices, etc.), software analysts evaluate source code for security weaknesses before deploying new software. 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. Manual effort is required for each SA alert to adjudicate whether it is a true or false positive, since most general flaw-finding SA tools produce many 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.

Software assurance teams typically prioritize potential vulnerabilities by a combination of likelihood and severity, and then they manually review only the top alerts. One might imagine that low-severity categories could be ignored entirely, but sometimes even code weaknesses categorized as lower-severity can cause costly failures. Many types of code flaws (even those categorized as lower severity) can lead to vulnerabilities that common attack patterns use. For example, the Common Attack Pattern Enumeration and Classification (CAPEC) (MITRE, 2023) describes an attack pattern (MITRE, 2020) that takes advantage of a lower-category weakness (MITRE, 2024). As another example, the Ariane flight V88 rocket explosion (which resulted in a loss calculated in 1997 as more than $370 million) was caused by code flaws that static analysis tools can detect (integer overflow and improper exception handling) (Lions, 1996) but are often not considered high priorities to identify and fix. In 2015, an integer overflow was discovered in the Boeing 787 Dreamliner that would cause loss of electrical power after 248 days of continuous power (Cheng, 2016).

1.2. 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)) (OpenAI, 2023), 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 (Flynn, 2018) 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.

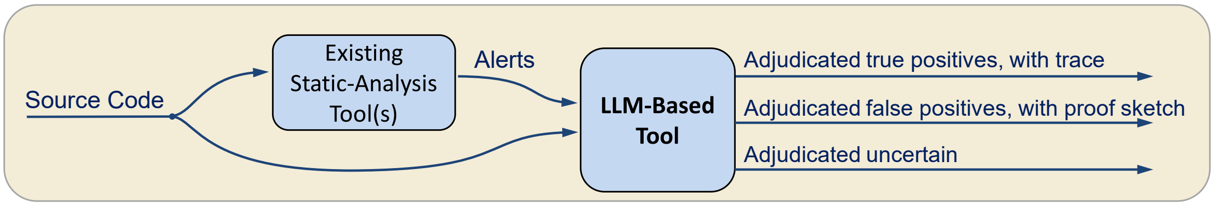

Figure 1. Using an LLM to adjudicate static-analysis alerts

Recent work shows that LLMs can perform well in detecting and localizing software defects. Chan et al. (2023) 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 developed code. Fan et al. (2023) 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. LLMAO (LLM fAult lOcalization) is an LLM-based approach for localizing program defects at line level, outputting bug probabilities for each line of code (Yang et al., 2024). 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 (Li et al., 2023a, 2023b). 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 (Li et al., 2023b). While these studies provide useful templates for system design, they do not fully address SA alert adjudication challenges because they both focused on the relatively narrow application of UBI bugs in the Linux kernel as a single case study.

Sherman (2023) 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. (2023a) also found that GPT-4 works well for this task.

2. Techniques for Using LLMs for Static-Analysis Alert Adjudications and Evaluation of these Techniques

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, and it 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".

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 the experiments discussed in this paper, we used GPT-4 and Llama 3 to test our queries. Although the web version of GPT-4 enables optional web search and Python execution, none of our experiments used these capabilities.

For the Llama experiments, we used the Llama 3.1-70B model publicly web-accessible from Meta (2024), in this paper called simply "Llama 3". This model has 70 billion parameters ("70B"). It should be noted that Llama 3.1-405B may perform better. We used the medium-size 70B model because it is the largest size that would run comfortably (with 1 byte per parameter) on hardware under $10k. (In contrast, the 405B model would require closer to $50k, again at 1 byte per parameter.) Some potential users require an LLM that they can run locally on-premises; they are unable to use commercial cloud-based LLM services due to the sensitivity of their data.

In this paper, the GPT-4 links (meaning all the links that start with "https://chatgpt.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 output in a specified schema, making it relatively easy to write a script to parse the output from a GPT-4 API 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.

Tables 1-2 (at the end of this paper) summarize the experiments (labeled "**Expt-*N***") discussed below.

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

In the simplest case, we can just provide the LLM with an alert and the function containing the flagged line of code.

Expt-01: We give an example of this case, where GPT-4 successfully adjudicates an alert for the vulnerability CVE-2022-41674 (which is about an integer-overflow leading to a buffer overflow in the Linux kernel): <https://chatgpt.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. 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."

The middle part of the prompt consists of the source code of the alerted-about function. 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 does not) 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 above but it is 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.

Expt-02: If asked about the patched version, GPT-4 correctly identifies that the vulnerability is no longer present: <https://chatgpt.com/share/7ee8e60b-1fed-4b67-b77b-7edd289fee90>.

2.2. 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 containing the flagged line of code. In other cases, it needs more context. 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 Ahmed et al. (2023).
- Use the LLM itself to generate the function summaries, as in Li et al. (2023b).

We have also tested two other strategies. One strategy is: As part of the prompt, direct the LLM to ask for needed information. In particular, we prompted the LLM to ask for the definition of a called function when it needs to know how the called function behaves. Our tool will then supply it to the LLM.

Figure 2 shows the start of the prompt.

> 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 for asking for missing functions

Figure 3 shows the end of the prompt.

> 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 prompt for asking for missing functions**

Expt-03: An example of this strategy is here: <https://chatgpt.com/share/b01b0394-55f2-49f7-8a11-bfda15362297>. In this example, to determine that a null-pointer alert is a false positive, the LLM needs the body of the user-defined function "out_of_memory" to determine that the function never returns.

A second strategy is to 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.

Expt-04: An example of creating a precondition is provided here: <https://chatgpt.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", …}]

Expt-05: Now we ask GPT-4 to use this precondition: <https://chatgpt.com/share/bbbf7df7-4fba-43b1-8f46-f09c4bd290cb>. In this example, GPT-4 analyzes a function that calls the "greet_user" function mentioned above. GPT-4 is given the precondition that it previously computed (in a separate conversation), 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.

2.3. GPT-4 Double-Checking its Work

LLMs sometimes make mistakes. One way to mitigate this is to run the LLM multiple times and check that the answers aren't inconsistent except on a small percentage of runs. Another mitigation is to ask the LLM to double-check its own work.

Expt-06: Example of GPT-4 double-checking its work: <https://chatgpt.com/share/78200ee7-37e7-413d-8b2d-fd9051c28106>. In this example, 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. The first prompt provides the source code for the function and instructs GPT-4 as follows:

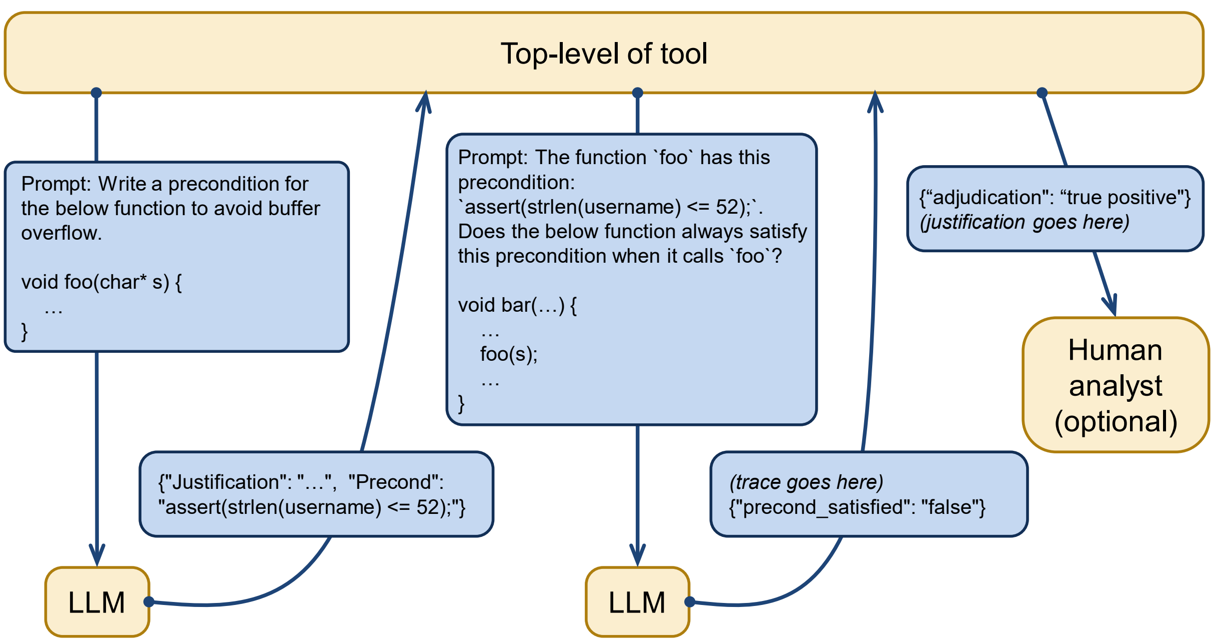

Figure 4. Creating and using preconditions.

> Investigate whether there are 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. Next, it provides an analysis, then lists steps of a trace it mistakenly 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.

Expt-07: Deleting the strdup call in the above example introduces a real use-after-free bug. We removed this call to strdup and separately (i.e., in a new conversation) asked GPT-4 to analyze this version of the code. GPT-4 correctly identifies the use-after-free bug and correctly reports that it is a real bug when asked to double-check the reported trace in detail: <https://chatgpt.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.”

2.4. GPT-4 Finding and Proving Loop Invariants

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

Expt-08: We tested it on a simple example of binary search: <https://chatgpt.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, specifying a “Maintenance step”, 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. This 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.”

2.5. Writing proof annotations in Frama-C

There are existing proof verification tools, such as Frama-C, that can check hand-written proofs of certain

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.

program properties. Writing such proofs is often laborious and requires significant expertise. Using an LLM to write such proofs would be a big win, especially since the LLM's output doesn't need to be trusted because it can be externally verified.

Expt-09: We asked GPT-4 to write a precondition for the following function sufficient to ensure absence of buffer overflow:

```
int rand_val_of_array(int* arr, int n) {
  int i = random() % n;
  return arr[i];
}
```

The conversation with GPT-4 is provided here: <https://chatgpt.com/share/56894914-aba0-4d5a-9f67-7ad3071314ad>. At first, GPT-4 produced an invalid precondition:

```
/*@ requires \valid(arr + (0..n-1));
    requires n > 0;
    ensures \result == arr[\old(i)];
 */
```

We then provided GPT-4 with the error message generated by Frama-C:

```
$ frama-c -wp temp25.c
[kernel] Parsing temp25.c (with preprocessing)
[kernel:annot-error] temp25.c:3: Warning:
  unbound logic variable i. Ignoring logic specification of function rand_val_of_array
[kernel] User Error: warning annot-error treated as fatal error.
[kernel] User Error: stopping on file "temp25.c" that has errors. Add '-kernel-msg-key pp'
  for preprocessing command.
[kernel] Frama-C aborted: invalid user input.
```

In response, GPT-4 generated a correct function precondition:

```
/*@ requires \valid(arr + (0 .. n - 1));
    requires n > 0;
    ensures \exists integer k; 0 <= k < n && \result == arr[k];
 */
```

This example suggests that LLMs have potential for writing program annotations in languages such as Frama-C, but the existing state-of-the-art (GPT-4) doesn't work very well for this task. Fine-tuning, with a mix of real-world data and synthetic data, might enable significantly better performance.

2.6. Detecting multiple bugs in FormAI_101087.c

Expt-10: Here, we ask GPT-4 to adjudicate an alert about an invalid pointer dereference in the file FormAI_101087.c from (Tihanyi, 2023). The dereference in question is "sock_addr->sin_addr". GPT-4 correctly identifies that sock_addr might be uninitialized at that point of the program. However, it provides an incomplete explanation and an incomplete repair, consisting of inserting the following check before the dereference:

```
if (!sock_addr) {
    fprintf(stderr, …);
    exit(EXIT_FAILURE);
}
```

The conversation with GPT-4 is provided here: <https://chatgpt.com/share/5f9fc88a-6fd1-4664-8e80-453955e8b8df>.

Expt-11: After applying GPT-4's suggested repair, we again asked it whether the expression "sock_addr->sin_addr" might have an invalid pointer dereference. This time, it finds the second flaw: The earlier call to freeifaddrs can free the memory that sock_addr points to. (In the likely lucky case, this memory hasn't been reused and the program works as expected, which is how the bug escaped testing.) <https://chatgpt.com/share/6a2a18bc-1f62-4714-a6d6-b716e131ab48>. Looking at the code at this link, one might notice that there is another problem with GPT-4's initial repair: sock_addr might be uninitialized at that point in the program.

Expt-12: We asked GPT-4 to identify any use-before-initialization errors in the function, and it correctly identified an instance of such an error: <https://chatgpt.com/share/30c52cf0-d64f-4e2a-8a00-e7d681c1e710>.

2.7. Llama 3

We ran Llama 3 on the examples in Table 1, using the same prompts as we used for GPT-4. The saved Meta AI links show the full responses.

Expt-01: Llama 3 incorrectly adjudicated the alerts. It identified that mbssid[1] is an unsigned 8-bit integer, and the maximum value it can hold is 255. However, then it incorrectly stated that no integer overflow happens after adding 2 to that value since that result "is within the range of size_t (which is typically an unsigned 32-bit or 64-bit integer type)". In fact, the relevant variable is an 8-bit integer (not a size_t), so integer overflow occurs. Based on that initial incorrect analysis, Llama 3 then incorrectly adjudicates the buffer-overflow alert as false.

Expt-02: Llama 3 correctly adjudicates the alert. However, its reasoning is faulty since it incorrectly states an unsigned 8-bit integer could not overflow from trying to hold the result of 255 + 2. In fact, it can. However, in the line identified in the prompt, that result instead is assigned to a variable cpy_len of the larger type (size_t) on the line identified as having a possible integer overflow. The cpy_len would not overflow, which results in the alerted-about buffer overflow adjudication being false. Again, the Llama 3 reasoning is incorrect however, as it extraneously states that it comes to its conclusion because: “The value of cpy_len does not directly affect the buffer size, and any potential overflow would only change the offset within the buffer, not the buffer size itself.” Though true, that should not lead to any conclusion about buffer overflow being possible.

Expt-03: Llama 3 makes the correct final adjudication “false positive”, just like GPT-4. Differently than GPT-4, Llama 3 does not ask for information about the behavior of the out_of_memory function, which GPT-4 specified was necessary to adjudicate the alert. Without information that that function calls abort(), Llama 3 does not have justification for its stated rationale “the out_of_memory function is called if the memory allocation fails, which would prevent the null pointer from being passed to memcpy”, which it uses for reasoning towards its adjudication.

Expt-04: Llama 3 identifies that a buffer overflow could happen, just like GPT-4. Llama 3’s function precondition specifies that strlen(username) should be less than 13, which is incorrect. (In contrast, GPT-4 correctly identifies that strlen(username) should be less than 52.)

Expt-05: Llama 3 incorrectly determines that the precondition was satisfied, based on an incorrect determination that greet_user will always receive a string of length 53 or less (actually it could receive a string of length 54 when the input doesn’t include a newline, as GPT-4 determines). Note that we used the correct precondition for greet_user in this prompt rather than the incorrect precondition that Llama generated in Expt-04.

Expt-06: Initially, Llama 3 comes to the same incorrect true-positive conclusion as GPT-4, and offers one similar recommendation for a fix and an alternative fix by using a different data structure to manage memory (GPT-4 didn’t suggest the latter option). After being prompted to double-check its trace, both LLMs correct their adjudication to determine there is no use-after-free vulnerability.

Expt-07: Llama 3 initially comes to the correct initial conclusion that there is a use-after-free vulnerability. After being prompted to double-check its work, only Llama 3 incorrectly concludes there is not a use-after-free vulnerability.

Expt-08: Llama 3 generates only one function precondition (n >= 0), which is the same as one of the two function preconditions that GPT-4 generates, but it misses the other precondition (that the input array should have at least size n). Both LLMs find the same loop invariant.

Expt-09: Llama 3 chooses to use ACSL (ANSI/ISO C Specification Language) to specify function preconditions, same as GPT-4. GPT-4 (but not Llama 3) provides detail on how to run Frama-C to verify the proof annotations. Llama 3’s initial attempt contains the “requires \valid(arr + (0 .. n-1))” annotation (like GPT-4) but lacks the “requires n > 0” annotation. (Note that modulo-by-zero, like division-by-zero, is undefined behavior in C.) Frama-C reports timeouts on verifying the division_by_zero and mem_access properties; when provided with this information, Llama fails to realize that it needs a “requires n > 0” annotation.

Expt-10: Llama 3 correctly identifies a potential invalid pointer dereference, same as GPT-4. Both LLMs provide code to fix it by checking if sock_addr is NULL, with slightly different error-handling: GPT-4 prints a descriptive error message to stderr then exits, while Llama 3 has a comment for the developer to add code to handle that or exit.

Expt-11: Llama 3 correctly identifies the potential invalid pointer dereference, same as GPT-4. They both recommend fixes that use inetop() instead of the inettoa() function. Only Llama 3 specifies that inettoa() is deprecated, but only the provided fix code from GPT-4 actually uses inetop().

Expt-12: Llama 3 correctly identifies the potential use-before-initialization error, same as GPT-4. Both LLMs provide the same code fix. Llama 3 provides a bit of additional unprompted-for secure coding advice and review related to mutex checking.

3. Related 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 (Flynn et al., 2020; Kremenek et al., 2004; Ruthruff et al., 2008; Flynn, 2016). 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 (Flynn, 2016), and other work developed a lexicon and adjudication rules to enable consistent adjudications to improve classifier training data (Svoboda et al., 2016). 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 (Wang et al., 2016). 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 (IARPA, 2018; Black, 2018) 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 (Flynn et al., 2018; Flynn et al., 2021) and created a framework for use with multiple SA tools, ML classification, adaptive heuristics, labeled datasets, and test suites (Flynn et al., 2021; Flynn et al., 2024). Gallagher et al. also used ML for finding code flaws but used LLVM intermediate representation instead of source (Gallagher et al., 2022). Flynn and Gallagher both found that artificial code and flaw injection can cause classifiers to use features not helpful with natural code.

4. Limitations and Future Directions

There are several notable limitations of our work. We have tested only two LLMs: GPT-4 and Llama 3. Furthermore, the version of Llama 3 that we tested was the medium size (70B) version rather than the large size (405B). We also investigated only a small number of different alert types (CWEs). We explored a few different types of prompts, but there are many possible variations that may give noticeably better or worse performance. Even for exactly the same prompt, different runs of the LLM can give different results (except when the LLM's temperature parameter is set to zero) due to different random seeds. Further work is needed to validate and quantitatively measure the techniques discussed in this paper. For example, we would want to methodically test the performance of the LLM on a large random sample of static-analysis results. This would ideally include running the LLM multiple times on each alert and conducting a statistical analysis to quantify the confidence in the results. To demonstrate greater applicability, we could apply the methods to a wide range of codebases and code flaws from multiple static analysis tools. If such experiments demonstrate that certain LLMs reliably adjudicate certain types of alerts, then the LLMs could be employed in practice to reduce the burden on analysts.

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. (2023) report success in using LLMs for generating formal proofs on a benchmark of formal-verification problems (SoSy Lab, 2023), 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.

5. 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.

6. References

Ahmed, T., Pai, K. S., Devanbu, P., & Barr, E. T. (2023). *Improving few-shot prompts with relevant static analysis products*. arXiv. arXiv:2304.06815 https://arxiv.org/abs/2304.06815v1

Black, P. E. (2018). Juliet 1.3 test suite: Changes from 1.2. US Department of Commerce, National Institute of Standards and Technology. https://nvlpubs.nist.gov/nistpubs/TechnicalNotes/NIST.TN.1995.pdf

Chan, A., Kharkar, A., Moghaddam, R. Z., Mohylevskyy, Y., Helyar, A., Kamal, E., et al. (2023). *Transformer-based vulnerability detection in code at EditTime:*

Zero-shot, few-shot, or fine-tuning? arXiv. arXiv:2306.01754

Cheng, X., Zhou, M., Song, X., Gu, M., & Sun, J. (2016, June). Automatic fix for C integer errors by precision improvement. In *2016 IEEE Computer Software and Applications Conference* (Vol. 1, pp. 2-11). IEEE.

Fan, G., Xie, X., Zheng, X., Liang, Y., & Peng, D. (2023). *Static code analysis in the AI era: An in-depth exploration of the concept, function, and potential of intelligent code analysis agents*. arXiv. arXiv:2310.08837

Flynn, L. (2018). *Challenges and progress: Automating static analysis alert handling with machine learning*. Carnegie Mellon University Software Engineering Institute. https://insights.sei.cmu.edu/documents/4174/2018_017_101_518025.pdf

Flynn, L., et al. (2016). Prioritizing alerts from static analysis with classification models. Software Engineering Institute, Research Review. https://insights.sei.cmu.edu/library/prioritizing-alerts-from-static-analysis-with-classification-models-2/

Flynn, L., et al. (2020). Static Analysis Classification Research FY16 20: for Software Assurance Community of Practice.

Flynn, L., et al. (2021). SCAIFE API (OpenAPI v.3). Software Engineering Institute site at GitHub. https://github.com/cmu-sei/SCAIFE-API/

Flynn, L., et al. (2024). SCALe Repository (scaife-scale branch). Software Engineering Institute site at GitHub. https://github.com/cmu-sei/SCALe/tree/scaife-scale

Flynn, L., Snavely, W., & Kurtz, Z. (2021). Test suites as a source of training data for static analysis alert classifiers. IEEE/ACM International Conference on Automation of Software Test (AST).

Gallagher, S. K., et al. (2022). LLVM intermediate representation for code weakness identification. https://apps.dtic.mil/sti/trecms/pdf/AD1178536.pdf

Intelligence Advanced Research Projects Activity (IARPA) contributors. (2018). STONESOUP. National Institute of Standards and Technology (NIST) Software Assurance Reference Dataset (SARD). https://samate.nist.gov/SARD/documentation#iarpa

Kremenek, T., et al. (2004). Correlation exploitation in error ranking. ACM SIGSOFT Software Engineering Notes, 29(6), 83-93.

Li, H., Hao, Y., Zhai, Y., & Qian, Z. (2023a). 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*. 2107–2111. https://doi.org/10.1145/3611643.3613078

Li, H., Hao, Y., Zhai, Y., & Qian, Z. (2023b). *The hitchhiker's guide to program analysis: A journey with large language models*. arXiv. arXiv:2308.00245

Lions, J.-L. & Inquiry Board 190. (1996). *Ariane 5: Flight 501 failure*. European Space Agency.

Meta. (2024). Meta AI chatbot. https://www.meta.ai/

MITRE. (2020). *CAPEC-124: Shared resource manipulation*. https://capec.mitre.org/data/definitions/124.html

MITRE. (2023). *Common Attack Pattern Enumerations and Classifications (CAPEC)*. https://capec.mitre.org

MITRE. (2024). *CWE-1331: Improper isolation of shared resources in network on chip (NoC)*. https://cwe.mitre.org/data/definitions/1331.html

OpenAI. (2023). GPT-4 Technical Report. arXiv preprint arXiv:2303.08774. https://arxiv.org/abs/2303.08774v1

OWASP. (2021). *OWASP Top Ten*. https://owasp.org/www-project-top-ten/

Ruthruff, J. R., et al. (2008). Predicting accurate and actionable static analysis warnings: an experimental approach. Proceedings of the 30th international conference on Software engineering. ACM.

Sherman, M. (2023). *Should I trust ChatGPT to review my program?* [Conference presentation]. InfoSec World 2023, Orlando, FL, United States.

SoSy Lab, SV-COMP contributors. (2023). *SV-Benchmarks* [Software]. International Competition on Software Verification. GitHub. Retrieved March 3, 2024, from https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

Svoboda, D., Flynn, L., & Snavely, W. (2016). Static analysis alert audits: Lexicon & rules. IEEE Cybersecurity Development (SecDev).

Synopsis. (2024). *Coverity DISA ASD STIG report*. Retrieved March 5, 2024, from https://sig-product-docs.synopsys.com/bundle/coverity-docs/page/reports/disa-stig/coverity_disa_stig.html

Tihanyi, N., Bisztray, T., Jain, R., Ferrag, M. A., Cordeiro, L. C., & Mavroeidis, V. (2023, December). The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification. *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*.

Wang, S., Liu, T., & Tan, L. (2016). Automatically learning semantic features for defect prediction. Proceedings of the 38th International Conference on Software Engineering.

Wu, H., Barrett, C., & Narodytska, N. (2023). *Lemur: Integrating large language models in automated program verification*. arXiv. arXiv:2310.04870

Yang, A. Z. H., Le Goues, C., Martins, R., & Hellendoorn, V. (2024). Large language models for test-free fault localization. *Proceedings of IEEE/ACM International Conference on Software Engineering*. Article No. 17. https://doi.org/10.1145/3597503.3623342

Table 1: URLs for GPT-4 and Llama 3 conversations

Expt #	Full URLs for GPT-4 conversations	Full URLs for Llama 3 conversations
Expt-01	https://chatgpt.com/share/4ce0cdae-47b7-4648-9462-9e0a381ccc37	https://meta.ai/s/nVjyNWofwQMedi1Q/
Expt-02	https://chatgpt.com/share/7ee8e60b-1fed-4b67-b77b-7edd289fee90	https://meta.ai/s/Dbyg7keJJu94ao3f/
Expt-03	https://chatgpt.com/share/b01b0394-55f2-49f7-8a11-bfda15362297	https://meta.ai/s/Ki1S1eCY657GceMF/
Expt-04	https://chatgpt.com/share/cfeabe6f-5757-4c25-be82-f9569f8c9df2	https://meta.ai/s/tMmxyRYA2N5bTV7X/
Expt-05	https://chatgpt.com/share/bbbf7df7-4fba-43b1-8f46-f09c4bd290cb	https://meta.ai/s/Zk1qGbKjz8GrRXBW/
Expt-06	https://chatgpt.com/share/78200ee7-37e7-413d-8b2d-fd9051c28106	https://meta.ai/s/5Ccw75y4pRyhzYHy/ https://meta.ai/s/L6bt2UYMWCUaYfRC/
Expt-07	https://chatgpt.com/share/36b75aaa-d52a-480c-8694-f8f62e64fdcc	https://meta.ai/s/jfsByVvTDekXMNTx/ https://meta.ai/s/qu11EiJ9trmfsQCR/
Expt-08	https://chatgpt.com/share/88c782ff-c7b7-4d4c-8cb1-988df4a3f1a8	https://meta.ai/s/ii8x7pTwjQWvJ2aj/
Expt-09	https://chatgpt.com/share/56894914-aba0-4d5a-9f67-7ad3071314ad	https://meta.ai/s/bisCooGPtkp1mFJD/ https://meta.ai/s/gPgp8pZwp9Fp1n65/
Expt-10	https://chatgpt.com/share/5f9fc88a-6fd1-4664-8e80-453955e8b8df	https://meta.ai/s/NFTpk2xdSJjr6ZNs/
Expt-11	https://chatgpt.com/share/6a2a18bc-1f62-4714-a6d6-b716e131ab48	https://meta.ai/s/VD1szypGbLKQF6ok/
Expt-12	https://chatgpt.com/share/30c52cf0-d64f-4e2a-8a00-e7d681c1e710	https://meta.ai/s/UZk23q1ywYQKNZCJ/

Table 2. Experiments with GPT-4 and Llama 3

Expt #	Paper section	Codebase	Lang	Correct adjud-ication	Code flaw type	GPT4: Initial result correct	GPT4: Second result correct	Llama: Initial result correct	Llama: Second result correct	Prompt type
Expt-01	2.1	Linux kernel	C	true positive	Integer Overflow, Buffer Overflow	yes	N/A	no	N/A	Step-by-step reasoning
Expt-02	2.1	Linux kernel	C	false positive	Integer Overflow, Buffer Overflow	yes	N/A	yes	N/A	Step-by-step reasoning
Expt-03	2.2	Zeek	C	false positive	Null Pointer	yes	N/A	yes	N/A	Ask for needed info
Expt-04	2.2	Toy example	C	N/A	Buffer Overflow	yes	N/A	no	N/A	Generate preconditions
Expt-05	2.2	Toy example	C	true positive	Buffer Overflow	yes	N/A	no	N/A	Check preconditions
Expt-06	2.3	Toy example	C++	false positive	Use-After-Free	no	yes	no	yes	Double-checking answer
Expt-07	2.3	Toy example	C++	true positive	Use-After-Free	yes	yes	yes	no	Double-checking answer
Expt-08	2.4	Toy example	C++	false positive	Buffer Overflow	yes	N/A	no	N/A	Find func precond and loop invariant
Expt-09	2.5	Toy example	C	N/A	Buffer Overflow	no	yes	no	no	Proof annotations in Frama-C
Expt-10	2.6	FormAI 101087	C	true positive	Invalid Pointer Dereference	yes	N/A	yes	N/A	Find and fix invalid pointer dereference
Expt-11	2.6	FormAI 101087	C	true positive	Invalid Pointer Dereference	yes	N/A	yes	N/A	Find and fix invalid pointer dereference
Expt-12	2.6	FormAI 101087	C	true positive	Use-Before-Initialization	yes	N/A	yes	N/A	Find and fix use-before-initialization

The column "Paper section" identifies the section of this paper that describes the experiment for GPT-4; the Llama version is described in Section 2.7. The column "Correct adjudication" indicates whether the static-analysis alert is actually a true positive or a false positive, as determined by manual inspection. The columns "Initial result correct" and "Second result correct" indicate whether the LLM produced a correct result in response to the first and second prompt in the conversation, respectively. Zeek is from https://github.com/zeek/zeek