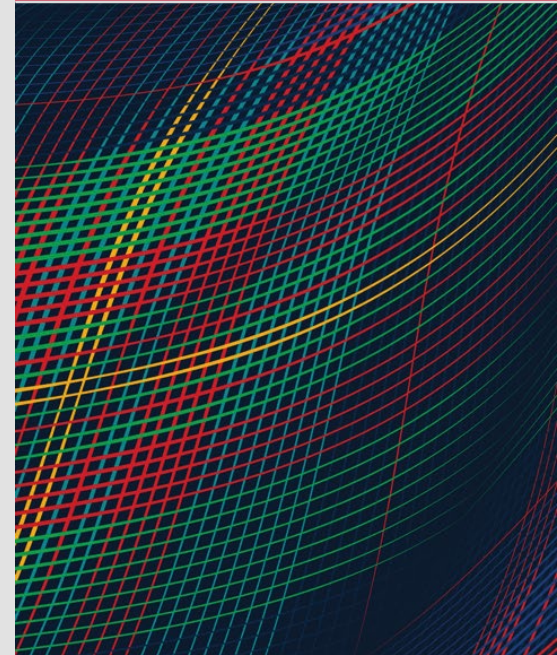


# Using LLMs to Adjudicate Static-Analysis Results

**AUGUST 2024**

Will Klieber and Lori Flynn

**Carnegie  
Mellon  
University**  
Software  
Engineering  
Institute



Copyright 2024 Carnegie Mellon University.

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.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR

RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[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.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at [permission@sei.cmu.edu](mailto:permission@sei.cmu.edu).

CERT® and Carnegie Mellon® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM24-0988

# Problem

- Software vulnerabilities are a serious concern for DoD.

## Software may have known or unknown vulnerabilities

Many known vulnerabilities in software, some in third-party packages

- **240,830** published software vulnerability records from **CVE**: <https://www.cve.org/>

CVE mission to “identify, define, and catalog publicly disclosed cybersecurity vulnerabilities

- The NIST National Vulnerability Database (**NVD**) includes CVEs and more

## Code flaws can lead to vulnerabilities


- Flaw taxonomies such as:


### Common Weakness Enumeration (CWE)


<https://cwe.mitre.org/>

CERT Coding Rules <https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards>



**New 2.0 APIs**

**CVSS v4.0 Support**

**New Parameters**

**Last 20 Scored Vulnerability IDs & Summaries**

**CVSS Severity**

<b>CVE-2023-33859</b> - IBM Security QRadar EDR 3.12 could disclose sensitive information due to an observable login response discrepancy. IBM X-Force ID: Z57697. <b>Published:</b> July 10, 2024, 12:15:02 PM -0400	V2.2 <b>High</b>
<b>CVE-2024-34443</b> - Improper Neutralization of Input During Web Page Generation (XSS or "Cross-site Scripting") vulnerability in ThemePunch OHG Slider Revolution allows Stored XSS. This issue affects Slider Revolution from v16 before 6.7.11. <b>Published:</b> June 15, 2024, 11:15:59 AM -0400	V2.1 <b>High</b>
<b>CVE-2024-40332</b> - idcms v1.35 was discovered to contain a Cross-Site Request Forgery (CSRF) vulnerability via /admin/moneyRecord_detail.php?mod=delRecord <b>Published:</b> July 10, 2024, 10:15:12 AM -0400	V2.2 <b>High</b>
<b>CVE-2024-40334</b> - idcms v1.35 was discovered to contain a Cross-Site Request Forgery (CSRF)	V2.1 <b>High</b>

**MEM30-C. Do not access freed memory**

Covered by Robert C. Stead, last modified by SEI on April 20, 2023

Evaluating a pointer—including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignment—into memory that has been deallocated by a memory management function is undefined behavior. Pointers to memory that has been deallocated are called **dangling pointers**. Accessing a dangling pointer can result in exploitable vulnerabilities.

According to the C Standard, using the value of a pointer that refers to space deallocated by a call to the `free()` or `realloc()` function is undefined behavior. (See undefined behavior [7].)

Reading a pointer to deallocated memory is undefined behavior because the pointer value is indeterminate and might be a trap representation. Fetching a trap representation might perform a hardware trap (but is not required to).

It is at the memory manager's discretion when to reallocate or recycle the freed memory. When memory is freed, all pointers into it become invalid, and its contents might either be returned to the operating system, making the freed space inaccessible, or remain intact and accessible. As a result, the data at the freed location can appear to be valid but change unexpectedly. Consequently, memory must not be written to or read from once it is freed.

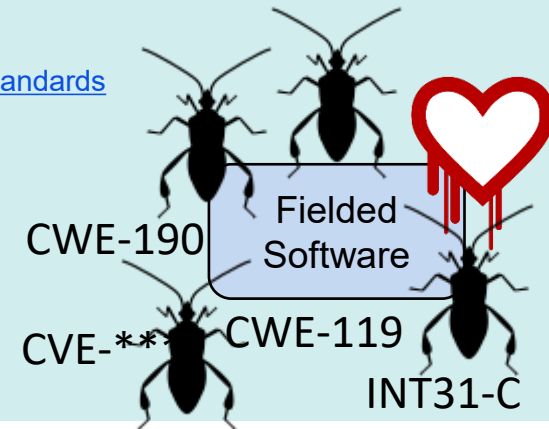
**Noncompliant Code Example**

This example from Brian Kernighan and Dennis Ritchie (Berrighan 1988) shows both the incorrect and correct techniques for freeing the memory associated with a linked list. In their (intentionally) incorrect example, `p` is freed before `p->next` is executed, so that `p->next` reads memory that has already been freed.

```
#include <stdlib.h>

struct node {
    int value;
    struct node *next;
};

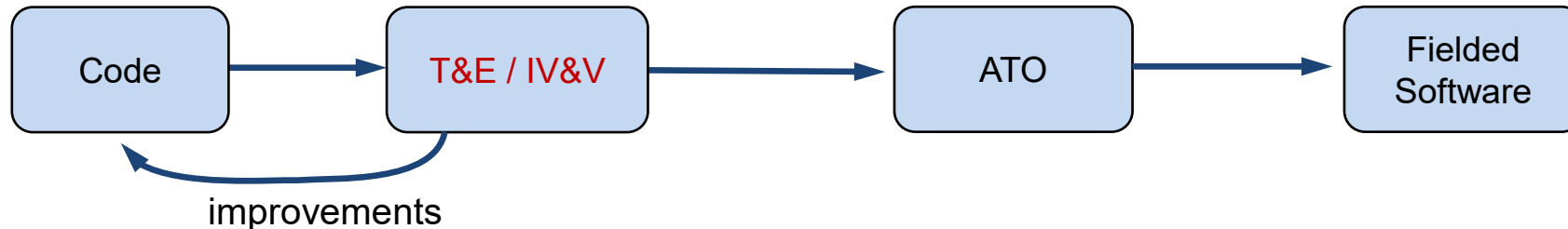
void free_list(struct node *head) {
    for (struct node *p = head; p != NULL; p = p->next) {
        free(p);
    }
}
```



# Problem

- Software vulnerabilities are a serious concern for DoD.
- Before software is fielded, the Authorization To Operate (ATO) process assesses risks that the software may introduce.
- 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**
  - Dynamic analysis
  - Unit tests
  - Integration tests
  - Performance tests
  - Model checking
  - (etc.)



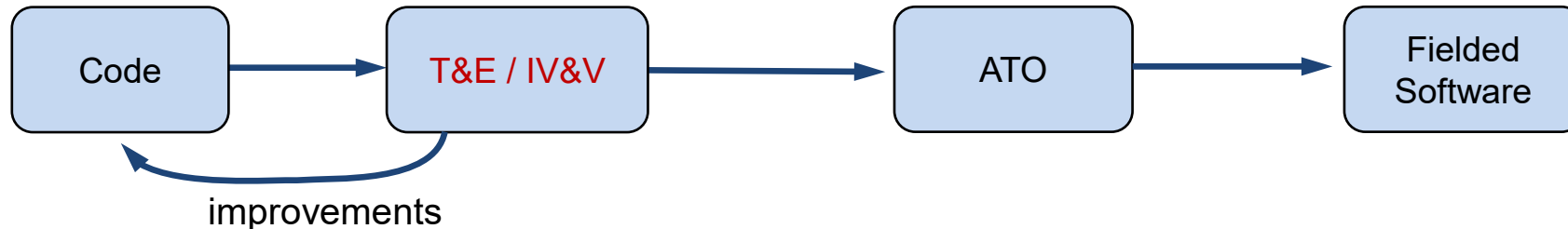
# Problem

- Software vulnerabilities are a serious concern for DoD.
- Before software is fielded, the Authorization To Operate (ATO) process assesses risks that the software may introduce.
- 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.
- Software analysts use static analysis as a standard method to evaluate the source code.

- **Static analysis**
- (additional T&E)

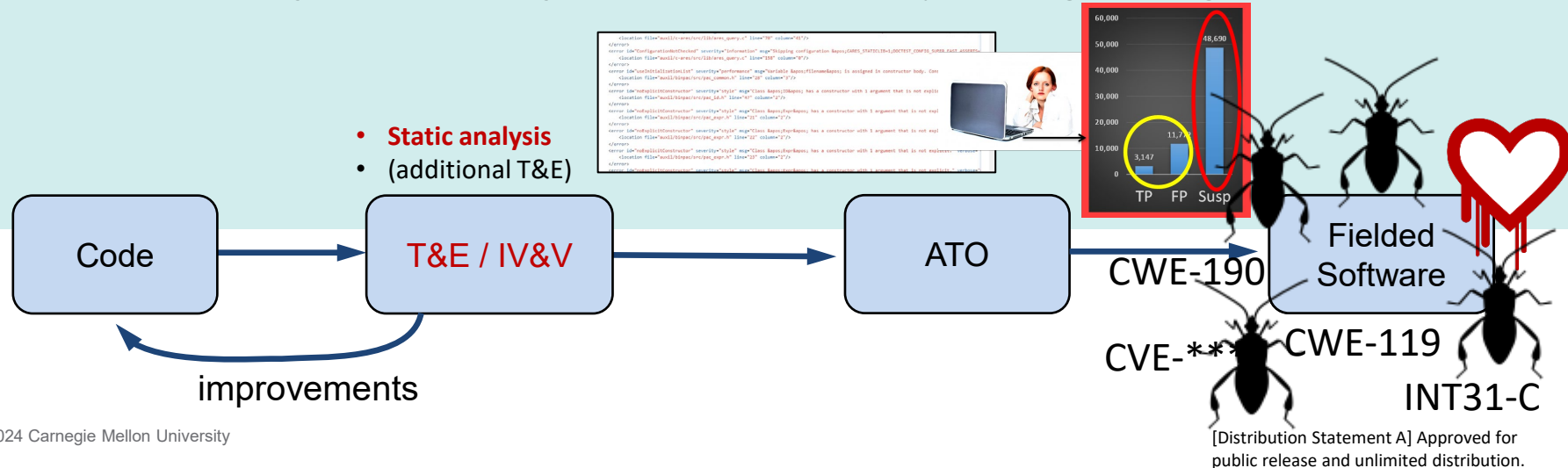
```
<location file="auxil/c-ares/src/lib/ares_query.c" line="70" column="41"/>
</error>
<error id="ConfigurationNotChecked" severity="information" msg="Skipping configuration &apos;CARES_STATICLIB=1;DOCTEST_CONFIG_SUPER_FAST_ASSERTS=
<location file="auxil/c-ares/src/lib/ares_query.c" line="158" column="0"/>
</error>
<error id="useInitializationList" severity="performance" msg="Variable &apos;filename&apos; is assigned in constructor body. Consider performing
<location file="auxil/binpac/src/pac_common.h" line="28" column="3"/>
</error>
<error id="noExplicitConstructor" severity="style" msg="Class &apos;ID&apos; has a constructor with 1 argument that is not explicit." verbose="Cl
<location file="auxil/binpac/src/pac_id.h" line="47" column="2"/>
</error>
<error id="noExplicitConstructor" severity="style" msg="Class &apos;Expr&apos; has a constructor with 1 argument that is not explicit." verbose="
<location file="auxil/binpac/src/pac_expr.h" line="21" column="2"/>
</error>
<error id="noExplicitConstructor" severity="style" msg="Class &apos;Expr&apos; has a constructor with 1 argument that is not explicit." verbose="
<location file="auxil/binpac/src/pac_expr.h" line="22" column="2"/>
</error>
<error id="noExplicitConstructor" severity="style" msg="Class &apos;Expr&apos; has a constructor with 1 argument that is not explicit." verbose="
<location file="auxil/binpac/src/pac_expr.h" line="23" column="2"/>
</error>
<error id="noExplicitConstructor" severity="style" msg="Class &apos;Func&apos; has a constructor with 1 argument that is not explicit." verbose="
```

SA alerts for  
potential flaws  
(CWE, CERT rules,  
etc.)



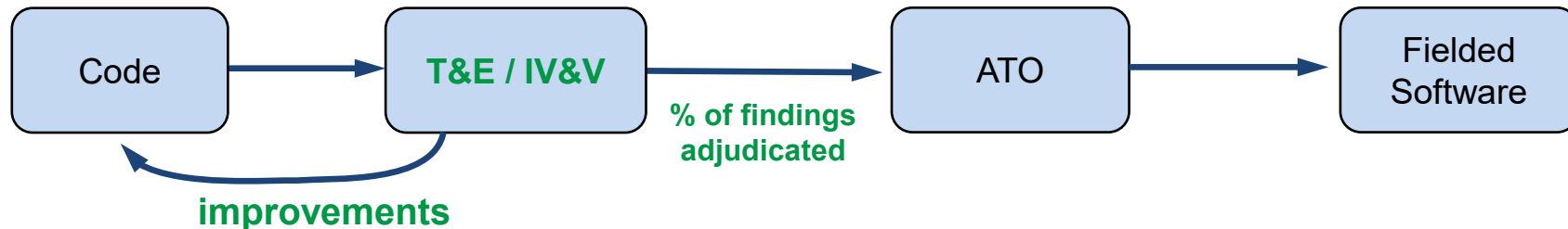
# Problem

- Software vulnerabilities are a serious concern for DoD.
- Before software is fielded, the Authorization To Operate (ATO) process assesses risks that the software may introduce.
- 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.
- Software analysts use static analysis as a standard method to evaluate the source code.
- Static analysis requires significant manual effort and is inherently difficult, time-consuming, and expensive.
- The volume of findings is often too large to review in their entirety, **causing the programs to accept unknown risk.**



# Problem and Our Tool to Address It

- Software vulnerabilities are a serious concern for DoD.
- Before software is fielded, the Authorization To Operate (ATO) process assesses risks that the software may introduce.
- 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.
- Software analysts use static analysis as a standard method to evaluate the source code.
- Static analysis requires significant manual effort and is inherently difficult, time-consuming, and expensive.
- The volume of findings is often too large to review in their entirety, causing the programs to accept unknown risk.
- We address the problem with **an LLM-based tool for use in T&E / IV&V**. Goal: tool will **automatically adjudicate** a large percentage of **alerts** with high accuracy, **enabling more complete adjudication, thereby reducing unknown risk** and enabling DoD to **remove vulnerabilities before the software is fielded**.



# State of the Art

Recent research shows that LLMs work well in code analysis tasks (although challenges remain, e.g., context window):

- Haonan Li et al [1, 2] found that GPT-4 works well for adjudicating use-before-initialization (UBI) alerts.
- Chan et al. [3] used LLMs to detect flaws in code being edited, reducing the rate of code flaws by 90%.
- Wu et al [4] report success in using LLMs for generating formal-verification proofs, beating state-of-the-art formal-verification tools on a number of hard cases.

How are things done today?

- Analysts typically manually adjudicate only the highest prioritized alerts (e.g., STIG Category 1).
- This leaves significant unknown risk to the software for the DoD program to accept.

[1] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian, "Assisting Static Analysis with Large Language Models: A ChatGPT Experiment" (FSE'23, accepted 2023-07-19).

[2] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian, "The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models" (Aug 2023).

[3] "Transformer-based Vulnerability Detection in Code at EditTime: Zero-shot, Few-shot, or Fine-tuning?" (May 2023).

[4] Haoze Wu, Clark Barrett, and Nina Narodytska. "Lemur: Integrating Large Language Models in Automated Program Verification." (2023).



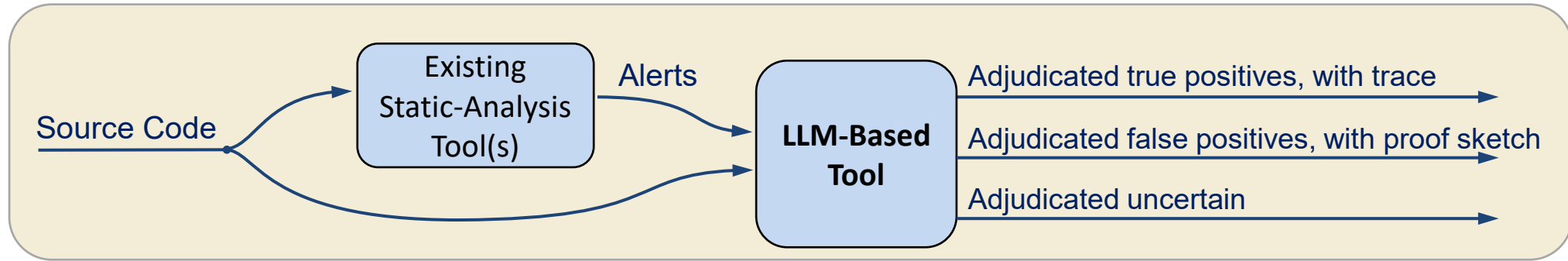
# LLMs are a significant breakthrough for code analysis

- Older ML techniques lack interpretability and often pivoted on irrelevant details that merely correlated with code flaws in their training data.
- LLMs produce a detailed explanation to support their final answer.
  - This can be manually double-checked.
  - GPT-4 has demonstrated an ability to correct its own mistakes when asked to double-check its work.
- An LLM-based tool can handle even alerts whose adjudication requires analyzing multiple functions spread across the codebase.
  - To do this, our tool will use the LLM to generate function summaries, function preconditions, and/or other intermediate results that can be combined to yield a final answer.

# Llama for on-prem

- In our preliminary work that we're presenting here, we've used GPT-4.
- GPT-4 is available only as web service and cannot handle CUI (IL4 / IL5) or classified code.
- A competitive alternative LLM is Llama 3, which is open-weight and can be downloaded and used on-premises. Available in 405B, 70B, and 8B sizes.
  - The larger sizes require high-RAM GPUs.

# Overview of our LLM-based tool



Simplest case:

- Tool creates a query (a “prompt”) to the LLM for each alert. The prompt includes:
  - (1) alert from static analyzer,
  - (2) source code of the relevant function, and
  - (3) instructions on what to do.
- LLM provides output, with its final answer in easily parsed JSON.

The next slide gives an example of this simple case.

# Example prompt: integer-overflow in Linux (CVE-2022-41674)

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."

...

```
static void cfg80211_update_notlisted_nontrans(...) {  
    ... /* omitted to fit on slide */  
    u8 cpy_len;  
    ... /* omitted to fit on slide */  
    pos += (mbssid - (trans_ssid + cpy_len));  
    /* copy the IEs after MBSSID */  
    cpy_len = mbssid[1] + 2; // ALERT-1  
    memcpy(pos, mbssid + cpy_len, ((ie + ielen) - (mbssid + cpy_len))); // ALERT-2  
    ... /* omitted to fit on slide */  
}  
...
```

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. Note: u8 here denotes an unsigned 8-bit integer type.

# Summary of GPT-4 output for prompt from previous slide

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 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"}``

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>`.

# Strategies for Mitigating Context-Window Limits

Due to the limited *context window* (maximum size of input plus output), an LLM can usually directly ingest a single function or a few functions at once, 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.

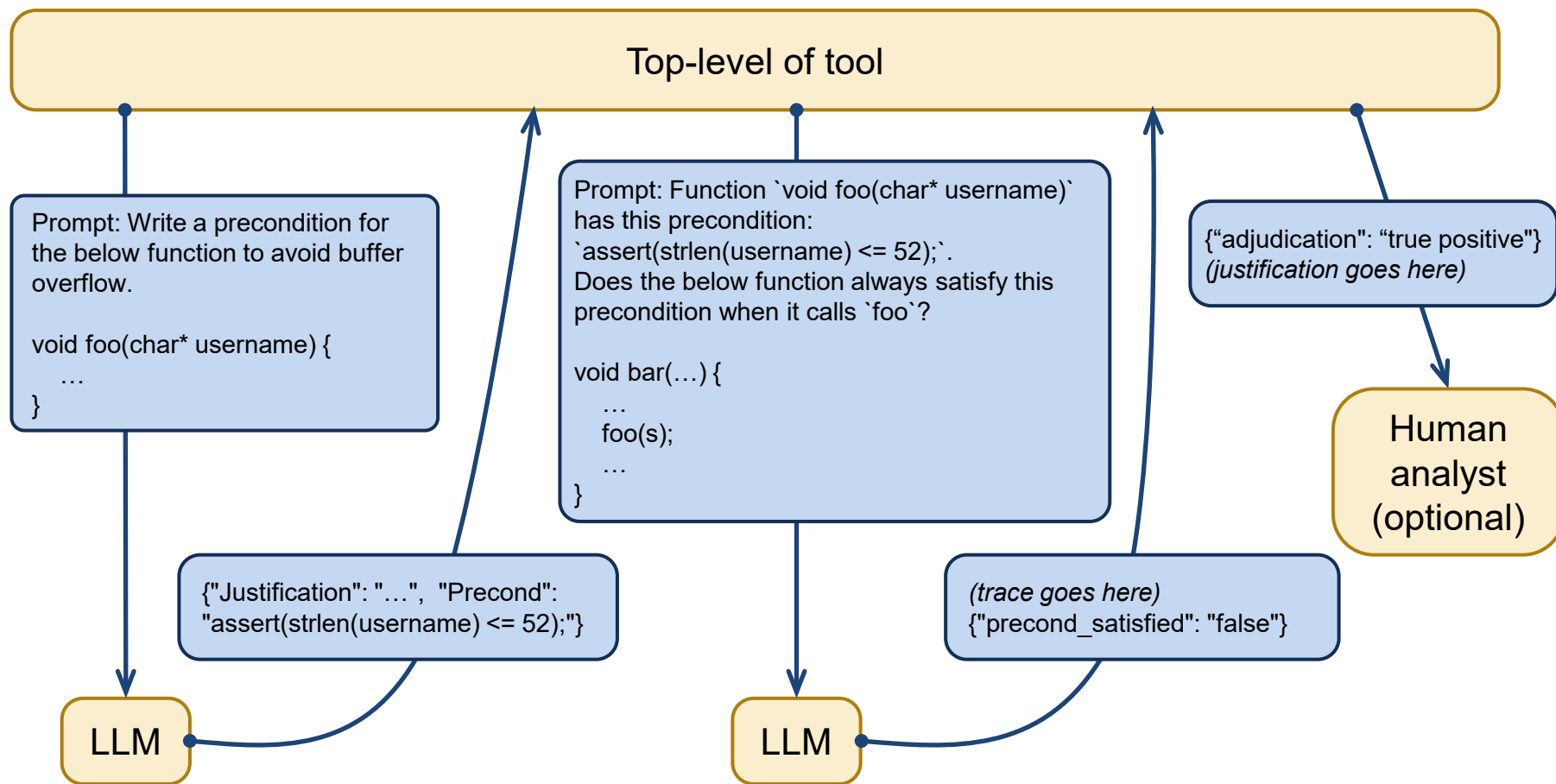
To overcome the context-window limit, we must summarize the relevant parts of the codebase enough so that the LLM can digest it. Some strategies for this have been documented in the literature:

- Use traditional static analysis to produce required information, as in Ahmed et al. (2023).
- Use the LLM itself to generate the function summaries, as in Li et al. (2023).

We have also tested a couple other strategies:

- As part of the prompt, direct the LLM to ask for needed information, such as type declarations and definitions of called functions. Our tool will then supply it to the LLM.
- 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.

# Creating and using preconditions



# LLM 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 possibly on a small threshold percent of runs).

Another mitigation is to ask the LLM to double-check its own work.

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 [1]:

```
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()
```

Deleting the strdup call from the above example introduces a real use-after-free bug, and GPT-4 correctly reports that it is a real bug when asked to double-check the reported trace in detail [2].

Why does this technique work?

[1] <https://chat.openai.com/share/78200ee7-37e7-413d-8b2d-fd9051c28106>

[2] <https://chat.openai.com/share/36b75aaa-d52a-480c-8694-f8f62e64fdcc>



# GPT-4 Finding and Proving Loop Invariants

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;
}
...
```

Precondition produced by GPT-4: “The input array arr should have at least size n, and n should be non-negative (i.e.,  $n \geq 0$ ).”

GPT-4 stepped through the code, found the loop invariant (“ $0 \leq \text{low} \leq \text{high} < n$ ”), sketched a proof of the loop invariant, and used the loop invariant to show that all array accesses are within bounds.

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

# Writing proof annotations in Frama-C [ 1 / 3 ]

Frama-C can check hand-written proofs of certain program properties.

We asked GPT-4 to write a precondition sufficient to ensure absence of a buffer overflow[1].

GPT-4 produced an **invalid** precondition:

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

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

[1] <https://chatgpt.com/share/56894914-aba0-4d5a-9f67-7ad3071314ad>

# Writing proof annotations in Frama-C [ 2 / 3 ]

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.
```

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

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];
*/
```

[1] <https://chatgpt.com/share/56894914-aba0-4d5a-9f67-7ad3071314ad>

# Writing proof annotations in Frama-C [ 3 / 3 ]

- LLMs have potential for writing program annotations in languages like Frama-C.
- State of the art (GPT-4) currently doesn't work well for that.
- Fine-tuning with real-world and synthetic data might improve performance.

# Detecting multiple bugs in FormAI\_101087.c (slide 1 of 2)

We asked [1] GPT-4 to adjudicate an alert about an invalid pointer deference `sock_addr->sin_addr` in the file FormAI\_101087.c from [2].

GPT-4 correctly identifies `sock_addr` may be uninitialized at that program point, but it provides an incomplete explanation and incomplete solution, only inserting the following check before the dereference:

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

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.

[1] <https://chatgpt.com/share/5f9fc88a-6fd1-4664-8e80-453955e8b8df>

[2] <https://core.ac.uk/download/pdf/581006296.pdf>

# Detecting multiple bugs in FormAI\_101087.c (slide 2 of 2)

There is another problem with GPT-4's initial repair, shown again below:

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

The problem is that `sock_addr` might be uninitialized at that point in the program.

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

To resolve the error, GPT-4 says to initialize `sock_addr` to `NULL` at the point of declaration:

```
struct sockaddr_in *sock_addr = NULL;
```

This way, the `if (!sock_addr)` check will reliably detect whether or not a suitable interface was found, avoiding undefined behavior.

# Automated Program Repair (APR) Eliminates Need to Adjudicate Some Alerts

**Developers:** auto-repair code (+review?)

**Security analysis:** provide repairs `diff`

- See SEI APR research project and Redemption tool<sup>1</sup>.
- More APR tools: MS Visual Studio C/C++ plugins, IntRepair, Clang repairs, Repairnator, AllRepair, Deepfix, and many others.

C/C++ APR tools/techniques exist for these (and more):

- |                                                                                      |                                                                 |
|--------------------------------------------------------------------------------------|-----------------------------------------------------------------|
| 1. CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer; | 6. CWE-457: Use of Uninitialized Variable                       |
| 2. CWE-125: Out-of-bounds Read                                                       | 7. CWE-20: Improper Input Validation                            |
| 3. CWE-476: NULL Pointer Dereference                                                 | 8. CWE-416: Use After Free                                      |
| 4. CWE-787: Out-of-bounds Write                                                      | 9. CWE-467: Use of sizeof() on a Pointer Type                   |
| 5. CWE-190: Integer Overflow or Wraparound                                           | 10. CWE-401: Missing Release of Memory after Effective Lifetime |

1. SEI APR research project and Redemption tool: <https://insights.sei.cmu.edu/library/redemption/>

# What type of code flaws are you most interested in?

## Help us determine which CWEs to prioritize in this work

Prioritize CWEs that APR tools don't yet fix?

Or, still need to adjudicate for offensive operations or other reasons?

C/C++ APR tools/techniques exist for these (and more):

1. CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer;
2. CWE-125: Out-of-bounds Read
3. CWE-476: NULL Pointer Dereference
4. CWE-787: Out-of-bounds Write
5. CWE-190: Integer Overflow or Wraparound
6. CWE-457: Use of Uninitialized Variable
7. CWE-20: Improper Input Validation
8. CWE-416: Use After Free
9. CWE-467: Use of sizeof() on a Pointer Type
10. CWE-401: Missing Release of Memory after Effective Lifetime



# We're looking for collaborators

**Contact: Will Klieber**  
weklieber@sei.cmu.org

## Our progress so far

- Developed some scripts
- Initial experiments with GPT
- CrossTalk Journal Article [1]

## Plans for further R&D (thru FY26)

- Collaborator testing and feedback
- On-prem LLM option
- Improved proofs correctness
- Enhanced prompt engineering
- More categories of code flaws

## Collaborate with us!

- Your org would test our tool on code you are working with, in your own environment.
- As an FFRDC, the SEI can rapidly iterate with DoD users based on their feedback.
- Existing work shows that LLMs perform reasonably well at adjudicating alerts and that techniques for mitigating their limitations have potential.
- **Your org could potentially better secure your code without spending more money, while contributing to advancing the research field.**

[1] <https://insights.sei.cmu.edu/library/using-llms-to-automate-static-analysis-adjudication-and-rationales/>