UDC 004.056.5:004.8:629.73
DOI: 10.15587/1729-4061.2026.352029

DETECTION OF VULNERABILITIES IN SOFTWARE FOR UNMANNED AERIAL VEHICLES BY USING LARGE LANGUAGE MODELS

Andrii Voiteskhovskyi
PhD Student*
ORCID: https://orcid.org/0009-0004-6009-9492

Iryna Stopochkina
Corresponding author
PhD, Associate Professor*
E-mail: i.stopochkina@kpi.ua
ORCID: https://orcid.org/0000-0002-0346-0390

Pu Sun
PhD Student**
ORCID: https://orcid.org/0009-0008-8105-3804

Junfei Xie
PhD**
ORCID: https://orcid.org/0000-0001-7406-3221

Mykola Ilin
PhD, Associate Professor*
ORCID: https://orcid.org/0000-0002-1065-6500

Oleksii Novikov
Doctor of Technical Sciences, Professor*
ORCID: https://orcid.org/0000-0001-5988-3352

*Department of Information Security
National Technical University of Ukraine
«Igor Sikorsky Kyiv Polytechnic Institute»
Peremohy ave., 37, Kyiv, Ukraine, 03056

**Department of Electrical and Computer Engineering
San Diego State University
Campanile Drive, 5500 (E-403B), San Diego, USA, CA 92182

This study considers binary software samples that operate and control unmanned aerial vehicles (UAVs). The task addressed is to detect vulnerabilities in UAV software given the absence of application source code.

An improved method for automated vulnerability detection has been proposed, as well as a corresponding algorithm, a universal instruction template, and an architectural model for automated vulnerability search involving the capabilities of large language models (LLMs). Compared to the fuzzing method, the proposed method provides an average increase in accuracy to 94.7% while reducing the analysis time by 4 times.

The method proposed for detecting UAV software vulnerabilities integrates binary analysis tools with the capabilities of logical inference and LLM pattern recognition. The corresponding algorithm for detecting UAV software vulnerabilities consists of processing stages, static analysis, logical inference using LLM, verification, correlation with known vulnerabilities, and reporting. The instruction template is independent of the features of the sample and tools and provides accurate logical conclusions. A new architectural communication model based on the Model-Context Protocol (MCP) provides universal interaction between LLM and decompilation tools.

A comparative analysis of the method's applications for different implementations of cloud LLMs was carried out. Key advantages include the generation of detailed vulnerability reports, decreasing analysis time from hours to minutes through automation, as well as reducing the qualification requirements for reverse engineers who perform the analysis. The proposed solutions enable proactive security assessment of UAV software, as well as automated vulnerability detection

Keywords: UAV firmware analysis, LLM-driven binary vulnerability analysis, MCP protocol, binary analysis context extension

Received 01.12.2025
Received in revised form 15.01.2026
Accepted date 05.02.2026
Published date 27.02.2026

How to Cite: *Voitsekhovskyi, A., Stopochkina, I., Sun, P., Xie, J., Ilin, M., Novikov, O. (2026). Detection of vulnerabilities in software for unmanned aerial vehicles by using large language models. Eastern-European Journal of Enterprise Technologies, 1 (2 (139)), 36–47. https://doi.org/10.15587/1729-4061.2026.352029*

1. Introduction

Unmanned aerial vehicles (UAVs) are currently widely used in civil, military, and commercial applications [1, 2]. The reliability of UAV networks depends significantly on the security of UAV software; an important task is to identify and eliminate vulnerabilities and undocumented functions that could potentially be used by attackers.

Commercial versions of UAVs have mostly cryptographically protected firmware, which complicates its analysis. However, there are open-source autopilot frameworks, such as PX4 [3] and ArduPilot [4]. These platforms provide researchers with full control over the UAV logic and software features. However, both secure and open-source UAV software demonstrate existing security vulnerabilities [5], which emphasizes the need to provide the robustness and security of any software.

When responding to incidents and assessing the functionality of UAVs, researchers typically obtain only binary images from the device under investigation. Focusing on binary files ensures that the methods remain applicable when the source code is not available and allows for the capture of assembler-specific behavior. Examples of such behavior include compiler-inserted code or obscured symbols that significantly impact performance. This makes it particularly important to devise vulnerability analysis methods specifically for binary files.

The number of vulnerabilities discovered continues to grow steadily in software across various domains, including the UAV domain. Existing vulnerabilities make it possible for malware to spread across UAV networks [6]. The prevalence of vulnerabilities in UAV systems is confirmed by existing Common Vulnerabilities and Exposures (CVE) records [7].

Classical approaches to binary analysis and vulnerability detection often require significant time and specialized expertise, making the process quite laborious, and sometimes impossible. A promising direction for solving these problems is the integration of reverse engineering tools with large language models (LLMs). Recent studies have demonstrated the potential of reverse engineering using LLM in general software analysis [8] and in the study of malicious code, in which LLM improved code understanding and generated effective indicators of compromise (IoC) [9]. However, the combination of LLM with arbitrary tools required for work through a communication architecture has not been carried out. This task could be solved by using the capabilities of the MCP protocol [10] and the corresponding architecture.

Given these considerations, research on improving and expanding methods for detecting vulnerabilities in binary samples of UAV software using LLMs is relevant.

2. Literature review and problem statement

Traditionally, software vulnerability identification is based on two key approaches: static and dynamic analysis, which are discussed in many papers. In particular, [11] considers the possibilities of combining static and dynamic analysis approaches. However, the scope of the study is web applications, which have their own specificity. UAV software has other features that require analysis of binary samples, due to the fact that the source code is not available.

It should be noted that, despite the prevalence of classical methods, a universal solution to the problem of threat detection has not yet been found, since each method has its own efficiency limitations. A promising option for overcoming these difficulties might be the use of tools with analytical capabilities, in particular large language models (LLMs).

Modern static analyzers, for example, [12], use control and data flow graphs to track information in the code. It has been shown that this makes it possible to detect typical error patterns, such as buffer overflows or memory leaks, without running the program. But the effectiveness of such tools is significantly reduced when working with obfuscated code or special protection mechanisms. The reason for this is the complexity of automatic interpretation of obfuscated code, which makes standard analysis ineffective. The difficulties are overcome by involving qualified experts and special tools for de-obfuscation, or using a virtual assistant based on LLM.

Dynamic analysis involves analysis by behavioral features of program execution, usually in a controlled environment. Dynamic analysis tools such as [13] effectively detect some errors in real time. However, they require constant monitoring by the researcher, which becomes a problem under conditions of shortage of time and qualified personnel. The solution in this case is the development of fully automated vulnerability detection technologies.

Work [14] considers fuzzing, one of the best known approaches to dynamic analysis, which is based on the generation of sets of randomized input data. Despite the advantages of the method, it also has disadvantages – time and incomplete coverage. Similarly, in [15], the work of a modern fuzzer for vulnerability analysis of drone firmware is demonstrated. The issue of complete code coverage is unresolved since dynamic analysis cannot examine all possible states of a complex program. The reason for this is the relatively low speed of the method. An optimization option in such cases might be the search for combined techniques that reduce the analysis time.

Another problem is the scale of the programs under study, since the analysis of large systems requires significant resources. In addition, anti-debugging mechanisms, which are the subject of study [16], reduce the performance of the analysis. The solution might be to devise new fast-acting approaches for automated vulnerability detection in UAV systems and rapid response to threats.

The integration of LLM into security processes offers significant advantages over conventional methods. In [17], the potential benefits of expanding the LLM context were considered; however, the architecture that would allow for automatic analysis of samples was not found. This is solved by designing interactions between a large language model and external tools.

In [18], it was shown that LLMs could identify vulnerable patterns even in modified code but the issue of applying them specifically to UAV software in the context of operational tasks was not raised.

In [19], the authors show that models often lack specific contextual data needed for binary analysis, which is also important for UAVs. However, the work focuses on the advantages of contextual taint analysis, without proposing solutions for the general case. Overcoming this shortcoming could be the combination of LLM with external specialized tools and exploring the prospects of application for a wide range of conditions, in particular in the field of UAV software.

In [20], the authors used context enrichment to detect weaknesses according to the Common Weakness Enumeration (CWE) classification. They showed that this increases the accuracy compared to existing methods. However, the issue of applying the method directly to binary code remained unresolved since the research focused on high-level languages. Other techniques are needed for reverse engineering and analysis of binary samples.

In study [21], a number of models (namely, Claude-3.5-Haiku (USA), GPT-4o-Mini (USA), DeepSeek-V3 (PRC), O3-Mini (USA), DeepSeek-R1 (PRC)) were evaluated under the standard and extended thinking modes. The experiments showed the superiority of models with logical inference capabilities (in particular, DeepSeek-R1). However, the work is mainly focused on a comparative analysis of LLM capabilities on general samples. Issues that were not covered in the work relate to methods with context expansion and UAV software. The solution is to conduct a study that would tackle the analysis of binary samples for unmanned systems, combined with the LLM architecture under context expansion conditions.

All this gives grounds to argue that it is advisable to conduct a study aimed at eliminating the contradiction between the need for prompt detection of vulnerabilities in binary samples of UAV software and the limitations of existing analysis methods. Available methods do not provide sufficient speed and require high qualifications of the expert to interpret the results. Accordingly, there is a need to devise new approaches by involving the intelligent capabilities of LLMs in combination with the analysis of binary UAV software. Architectural solutions to solve this problem should be designed based on the latest approaches to expanding the context using the "Model-Context" protocol [10].

The need to analyze binary samples arises among cybersecurity specialists, for example, when studying captured software samples used under combat conditions [2], or when responding to the spread of malicious software [6]. Accordingly, architectural and software solutions for finding vulnerabilities must be promising in terms of scaling from a single sample to an entire network of UAV hardware and software, taking into account tight time constraints and shortages of human resources.

3. The aim and objectives of the study

The aim of our study is to devise solutions using LLMs for automated vulnerability detection in UAV software, which would allow for improved accuracy, shortened analysis time, and reduced requirements for researcher's qualifications.

To achieve the goal, the following tasks were set:

– to design a communication architecture model based on the MCP protocol and propose its potential application as part of the architecture in the UAV swarm security system;

– to propose a structured template of instructions for binary analysis using LLMs for multiple use;

– to propose a method for vulnerability detection using LLMs, and its step-by-step implementation in the form of an algorithm;

– to conduct experiments on vulnerability detection in real and artificially fabricated samples;

– to compare different models for cloud-based LLM implementations in the context of the effectiveness of the proposed method and its potential application for mass analysis of samples in UAV networks.

4. Materials and methods

The object of our study is binary software samples for the operation and control of unmanned aerial vehicles (UAVs).

The principal hypothesis assumes that the integration of LLMs with binary analysis tools via the MCP protocol makes it possible to automate and accelerate the process of detecting vulnerabilities in UAV software, while maintaining high accuracy, compared to conventional methods, in particular, fuzzing.

The following assumptions were adopted:

– vulnerabilities in UAV software have statically defined patterns that can be detected by static analysis methods;

– the decompiled sample, despite the loss of variable names and other information, is sufficiently representative for analysis.

The following simplifications were accepted: only decompilation tools were used as a means of enriching the context, although this composition could be expanded for complex tasks.

The research methods included analysis, synthesis, and formalization to devise the method, instruction template, architecture model, and an algorithm. The experimental method was used to verify the performance of the proposed solutions and comparative analysis. The proposed method is based on the classic method of reverse engineering: static.

The initial data for the study are binary samples of UAV software that contain vulnerabilities. The study does not use any samples protected by the manufacturer using cryptography to prevent copyright and claims infringement. Instead, samples of open system firmware are used as binary samples for the experiments, in particular PX4 [3], ArduPilot [4]. To expand the initial data, vulnerable software implementations of a type known to researchers were artificially introduced into software that does not have undocumented vulnerabilities. The vulnerabilities that are detected can be attributed to the types listed in Table 1.

Table 1

Software vulnerabilities found in UAV control systems

Type of vulnerability	Relevance for UAV firmware	Applied domain
Heap buffer overflow	Very high	Video processing, telemetry, communication protocol libraries
Post-release use	High	Real-time event processing (e.g., route planning, sensor data processing)
Stack overflow	High	Typical for firmware with limited memory, where there is no ASLR (Address space layout randomization) mechanism or stacked «canaries»
Confusion of types	Medium	Object-oriented firmware code
Integer overflow or disappearance	High	Common when calculating coordinates, buffer sizes, velocities, etc.
Double release	Medium	In most RTOS (Real-time operating system) solutions, memory is not freed manually, but this is possible when using adapted functions malloc() or free()
String format vulnerabilities	Possible	Depends on the presence of functions of the printf() type with input data from the outside (for example, on UART, on Wi-Fi)
Errors per unit	High	Often implicitly occur in operations on arrays; may cause failures or potential use of exploits

Due to the fact that the experiments were performed on samples with known vulnerabilities, the possibility of objectively assessing the correctness of the conclusions based on the analysis algorithm was ensured.

Software solutions used.

The following software solutions were used to implement the method and implement the communication architecture model:

– LLM Claude Sonnet 4.0 [22] in interaction with MCP [10];

– IDA Pro Hex-Rays decompiler [12] and API (Application Programming Interface) [23].

Additionally, the following were used to conduct a comparative experiment:

– Binary Ninja decompiler [24];

– AFL++ fuzzer [25].

To implement a comparative analysis and cost-effectiveness of the method for cloud implementations of LLMs, the following were used:

– LLM Claude 4.1 Opus [26];

– Claude 4 Opus [22];

– DeepSeek-R1 [27];

– Amazon Nova-Pro [28];

– xAI grok-code-fast-1 [29];
– AWS Bedrock service in interaction with MCP.

Supporting materials. Samples, LLM instructions are provided in the GitHub repository [30].

5. Results of designing the architecture model and devising the vulnerability detection method

5. 1. Design of a communication architecture model based on the Model-Context protocol

To analyze binary files, reverse engineering engineers rely on auxiliary tools, in particular decompilers. To extend this capability to LLMs, the model must be able to interact seamlessly with such tools. Achieving this interaction requires a standardized communication mechanism. To fulfill this requirement, a communication architecture based on MCP is proposed, as shown in Fig. 1. MCP offers a unified interface for service integration and allows LLM to access external data and call analysis tools.

The basic components of the proposed model and the interaction between them are shown in Fig. 1.

The proposed architecture model assumes modularity and scalability. The model consists of the following main components:

1. A decompiler that disassembles and decompiles firmware binaries into high-level pseudocode. It provides cross-references, restores the control flow graph, performs type recovery, and has an API for interactions that are used to extract features for analysis.

2. An MCP server provides LLM access to tool endpoints (e.g., decompiler requests), normalizes responses, and provides a sandbox and access control. It consists of the following components:

– a decompiler plugin that, through the API of a specific decompiler (in this study, IDA Pro Hex-Rays), provides the set of tools that the MCP server supplies to the LLM client;

– an MCP bridge that provides a standardized interface for communicating with MCP clients and ensures protocol compliance, as well as constant communication with the IDA plugin.

3. LLM model: used for logical analysis of decompiler results, generalization of code behavior, ranking hypotheses about vulnerabilities and suggestions for their elimination. This model interacts with the following modules:

– MCP client, which manages hints, context collection and tool calls via MCP. It is compatible with different models and configurations;

– user context. The model performs logical analysis based on structured evidence. This evidence is stored in the user context provided via the standard MCP protocol (metadata, previous findings). The model then generates hypotheses and suggestions. All the model's knowledge about the code is obtained via MCP tools and is not generated directly.

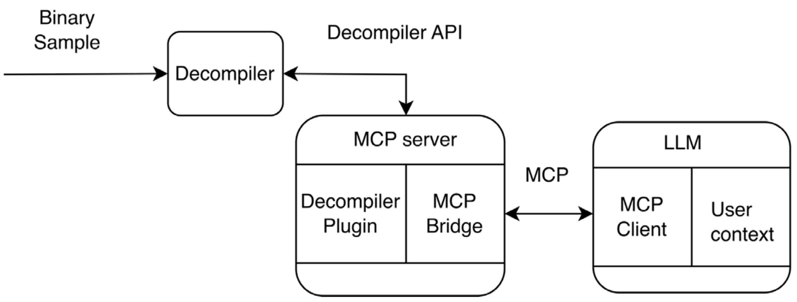


Fig. 1. Communication architecture model based on the Model-Context protocol

The communication between the MCP server and the MCP client using the MCP protocol allows for flexible replacement of the decompiler without changing the underlying logic of interaction with the LLM. The MCP server operates independently of the client and is implemented as a separate application. The MCP bridge operates asynchronously and remains active regardless of the state of the plugin, which increases the flexibility of the analysis process. Thus, the proposed architecture is quite flexible and can be applied in more global UAV safety management tasks.

In particular, in Fig. 2, the proposed model is suggested to be used as part of the architecture of the UAV security center. The architecture is aimed at counteracting the spread of malware and the exploitation of vulnerabilities in UAV networks. Similar to the search for vulnerabilities, the detection of the influence of malware and the vulnerable areas that it exploits is based on the same principles. Therefore, the architecture model proposed in the work can be applied to the tasks of building digital immunity systems as part of a centralized UAV swarm security system.

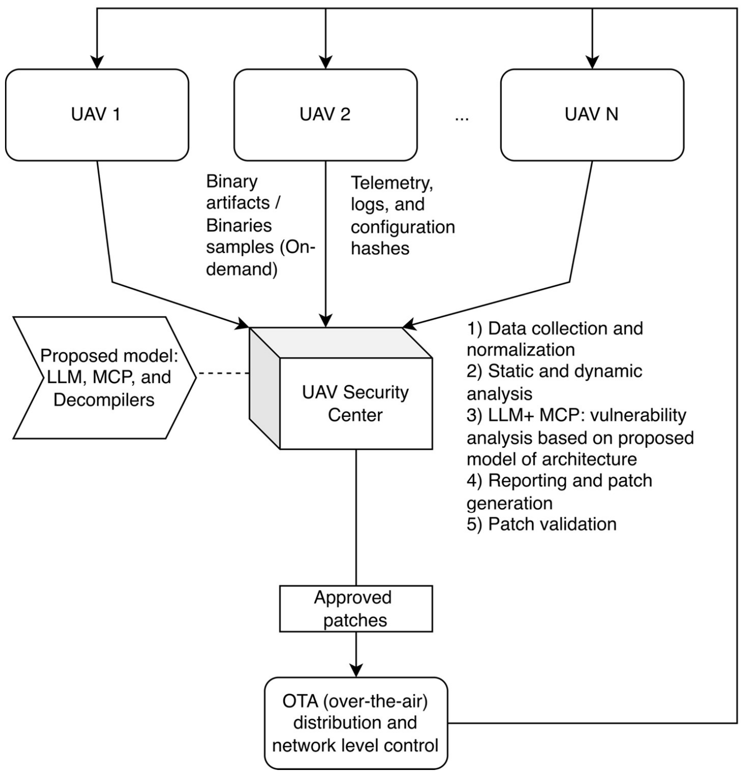

UAV 1
UAV 2
...
UAV N
Binary artifacts / Binaries samples (On-demand)
Telemetry, logs, and configuration hashes
Proposed model: LLM, MCP, and Decompilers
UAV Security Center
1) Data collection and normalization
2) Static and dynamic analysis
3) LLM+ MCP: vulnerability analysis based on proposed model of architecture
4) Reporting and patch generation
5) Patch validation
Approved patches
OTA (over-the-air) distribution and network level control

Fig. 2. Position of the proposed model in the architecture of a centralized security system for a swarm of unmanned aerial vehicles

Fig. 2 essentially represents the immune system of a UAV swarm, which is implemented in several stages (stages 1–5 in Fig. 2):

– monitoring and detection of malware, represented by points 1), 2): UAVs periodically transmit telemetry, logs, and hashes of critical configuration files to the center. In case of anomalies or hash mismatches, the center requests full binary file dumps for in-depth analysis;

– LLM-based analysis and solution generation 3), 4): the obtained samples undergo automated processing according to the proposed method. At this stage, the attack vector is detected, and a fix (patch) is generated. This is where the architecture model (Fig. 1) proposed in this work is used;

– an important stage, reflected in the diagram, is patch validation 5) – checking the correctness of the generated code before sending it. Approved security patches are distributed over secure communication channels to all swarm devices or selected nodes using Over-the-Air (OTA) technology. This allows for rapid network-wide remediation of a threat (in this case, a virus infection).

5. 2. Structured instructions template for binary analysis

Structured instructions for LLMs that should be used in the vulnerability detection process have been proposed. Since binary analysis requires precise logical reasoning, interaction with tools, and context awareness, unstructured instructions often lead to incomplete or ambiguous results. To address this problem, a structured instructions template has been devised that follows the following principles:

1. The specificity of the binary being analyzed should be provided, describing the architecture, operating system, compiler, and other significant details (e.g., size, entry point position, file type).

2. The prompt should be structured and contain the following parts:

a) a brief statement of the analysis goal;

b) specific steps that the LLM should perform;

c) constraints (e.g., do not use brute force);

d) expected output (e.g., a report).

3. Instructions for using tools should be clear (e.g., instructions for MCP tools).

4. Reporting should be mandatory and the required format should be specified (e.g. .md or other standard file format). The report should contain logical explanations supported by practical code samples.

5. The LLM should be instructed to avoid false positives and focus on detecting the specified types of vulnerabilities (Table 1).

6. The analysis should continue until complete code coverage is achieved, i.e., the report should not be generated until every function has been tested.

The instruction template provided below is specifically designed to instruct the LLM on how to perform binary analysis and ensures consistency and reproducibility. It contains the modules Task, Target Information, Instructions, Constraints, Result.

The Task module sets the task and goal, namely: to analyze a binary file using IDA Pro via the MCP server, in order to perform a static reverse engineering process and identify vulnerabilities or restore logic, depending on the mode.

The Target Information module provides data about the file under study:

– processor architecture;

– operating system;

– compiler;

– binary file type: elf/PE/Raw or other; application in the UAV network: library/firmware/console application/driver or other.

The Instructions module consists of instructions:

1. Start with a decompiled IDA Pro file.

2. If information is lacking, disassemble the functions and check the assembly code.

3. Use MCP tools to convert numbers, remove cross-references, call graphs and control flow, and help with type correction.

4. Rename all unknown functions and variables to descriptive names.

5. Add comments to explain control logic and unusual behavior.

6. If the task is specifically to analyze vulnerabilities:

– check all functions for memory corruption, type confusion, use after free, etc.;

– report only high-confidence issues;

– describe the exploitability of the vulnerability and the steps to reproduce it, as well as a potential fix.

The Constraints module consists of the following caveats for the model:

– do not use direct search;

– do not use external assumptions or statistical hypotheses;

– do not use manual conversion of number systems – always use tools;

– do not complete the work until all relevant functions have been analyzed.

The Result module contains the requirements for reporting, namely:

1. Save all findings and applied steps in the report.md file.

2. Use clear formatting of the report in .md format with the following sections:

1) overview;

2) feature analysis;

3) vulnerabilities (if any);

4) recommendations or Fixes (if any).

3. Notify the user about the final results only after the report is complete.

5. 3. Method for detecting vulnerabilities in unmanned aerial vehicle software using large language models

The proposed method combines deterministic static analysis algorithms and artificial intelligence inference mechanisms. The method is based on the following provisions:

1. The “black box” principle. The input data are binary files (samples) without the original high-level code, and in the general case – without symbolic information (“stripped binary”). This requires the ability to reconstruct high-level code solely based on knowledge of machine instructions.

2. The decomposition principle. The toolkit that makes it possible to build the structure of the system that provides context analysis modular is the MCP protocol. LLM, combined with external binary analysis tools (a decompiler is enough) by configuring the LLM interaction via the MCP protocol and the tool API. LLM does not depend on a specific version of the decompiler, and works with a structured context (control flow graph, data types, etc.).

3. Formalization of instructions. The analysis process is guided by an instruction template that controls the transformation of the source data into a semantic result (whether the vulnerability exists or not, why it exists, what danger it poses, how to eliminate it).

4. Hypothesis generation. A binary sample is fed to the decompiler input, an instruction template with filled fields is fed to the LLM input, which indicates information about the target obtained using the decompiler (section 5.2, module "Information about the target") and contextual analysis information from the decompiler. After logical conclusions according to the instructions, the LLM generates a report. In case of vulnerability detection, the LLM provides information about the location, causes of the vulnerability, ways to eliminate it, and PoC (proof of concept) – proving the possibility of using the vulnerability and other information that is crucial for generating UAV software patches.

5. Expert validation. The result must be analyzed by a human researcher to make final decisions based on this. If appropriate, decisions are made regarding the generation of fixes, recall of vulnerable software, or others.

The step-by-step work of the method is formalized in the form of an algorithm (Fig. 3).

Stage 1 should establish basic assumptions about the binary file for further analysis. Here, the file header and loader information are analyzed to determine the target architecture, format (ELF/PE), and the presence of special characters. If characters are removed, light recovery can be attempted (pattern-based stub naming, library signature matching, startup function heuristics).

Stage 2 is intended to build a structural model of the program and identify areas of code that appear to pose a potential risk.

Stage 3 serves to obtain reproducible crashing inputs, test cases, and crash signatures (location, sanitizer report). These data are evidence and starting points for further analysis.

Stage 4 is the central stage that provides findings using LLM. The vulnerabilities found are provided with type, location, paths, and, when possible, a minimal PoC (proof of exploit) or prerequisites for exploitation in the UAV system. These findings are stored with references to the code sections that were used.

Step 5 is required to determine whether the confirmed issue corresponds to a known vulnerability in order to speed up the patching and reporting.

Step 6 is intended to generate a report that presents the necessary results to software researchers.

In the proposed algorithm, M denotes metadata about the binary file B, such as the architecture, platform, and subsystem of the UAV. CVE_DB represents a public CVE database containing known binary vulnerabilities (e.g., from the MITRE/NVD databases). The output of the algorithm is a structured vulnerability report, denoted as R, which includes both newly discovered and CVE-matched vulnerabilities.

```
1. Preprocessing
Extract metadata $M$ from the binary file $B$: architecture, format (ELF/PE), stripped/unstripped
if $B$ is stripped then
    Attempt symbol recovery (e.g., via heuristics or pattern matching)
end if
2. Static Analysis
Select decompiler (IDA, Ghidra, Binary Ninja) based on platform
Decompile and analyze $B$ to construct CFG/DFG
Identify suspicious patterns:
  - Buffer overflows
  - Use-after-free
  - Integer overflows
  - Format string issues
if dangerous patterns found then
    Tag $B$ as "Memory-risk"
end if
3. Dynamic Fuzzing
Create sandbox environment (e.g., QEMU, ASAN builds)
Execute fuzzing using AFL++ or LibFuzzer
for each crash do
    if ASAN or valgrind confirms memory violation then
        Store crash signature
    end if
end for
4. LLM-Assisted Reverse Engineering
if $M$.requires_deep_inspection then
    Extract vulnerable-looking functions
    Generate LLM prompt with function code, metadata, task
    Query LLM for vulnerability analysis
    if LLM reports exploitable issue then
        Store vulnerability type, location, PoC
    end if
end if
5. CVE Correlation
for each confirmed vulnerability $V$ in $B$ do
    Compute or extract:
      - Function name
      - Code fingerprint (e.g., hash, opcode signature)
      - File/module name
      - Behavior trace (if crash)
    Search CVE_DB for match by:
      - Affected software component
      - Code fingerprint similarity
      - Vulnerability class (e.g., heap overflow in MAVLink parser)
    if match found in $CVE\_DB$ then
        Tag $V$ with CVE ID (e.g., CVE-2024-38952)
        Retrieve CVSS score and remediation data
    else
        Tag $V$ as "potential 0-day"
    end if
end for
6. Reporting and Remediation
Generate report $R$:
  - List of vulnerabilities
  - Mapped CVEs (if any)
  - Severity ratings (CVSS or inferred)
  - Exploitation conditions and PoC
  - Recommended fixes (code or architecture)
Export $R$ as:
  - Human-readable PDF/Markdown
  - JSON for integration with SIEM, issue trackers
return $R$
```

Fig. 3. Binary analysis and vulnerability assessment algorithm

5. 4. Experiments on real and artificial samples

5. 4. 1. Drivers

The method was tested on a number of samples belonging to different types. During the experiments, LLM Claude Sonnet 4.0 was used. One of these types is the "Living Off The Land" drivers (LOL Drivers). They pose a significant threat to UAV systems. This is especially true for UAV ground control stations, which usually run under Windows. These are Windows kernel-mode drivers that contain vulnerabilities or provide functionality that can be exploited by attackers. LOL Drivers are not malicious by design. However, if vulnerabilities are present, they can grant kernel-level privileges, bypass mandatory driver signing and other security mechanisms. Among other things, they can read or modify kernel memory and execute arbitrary code with elevated privileges.

In UAV systems, an adversary capable of loading a vulnerable driver at the ground control station could intercept telemetry, introduce unauthorized commands, compromise the autopilot, or disrupt mission control. Therefore, security systems should consider the risks associated with LOL Driver, especially in critical or defense-oriented UAV applications.

In this experiment, the proposed method was evaluated on the vulnerable driver BdAPIUtil.sys.

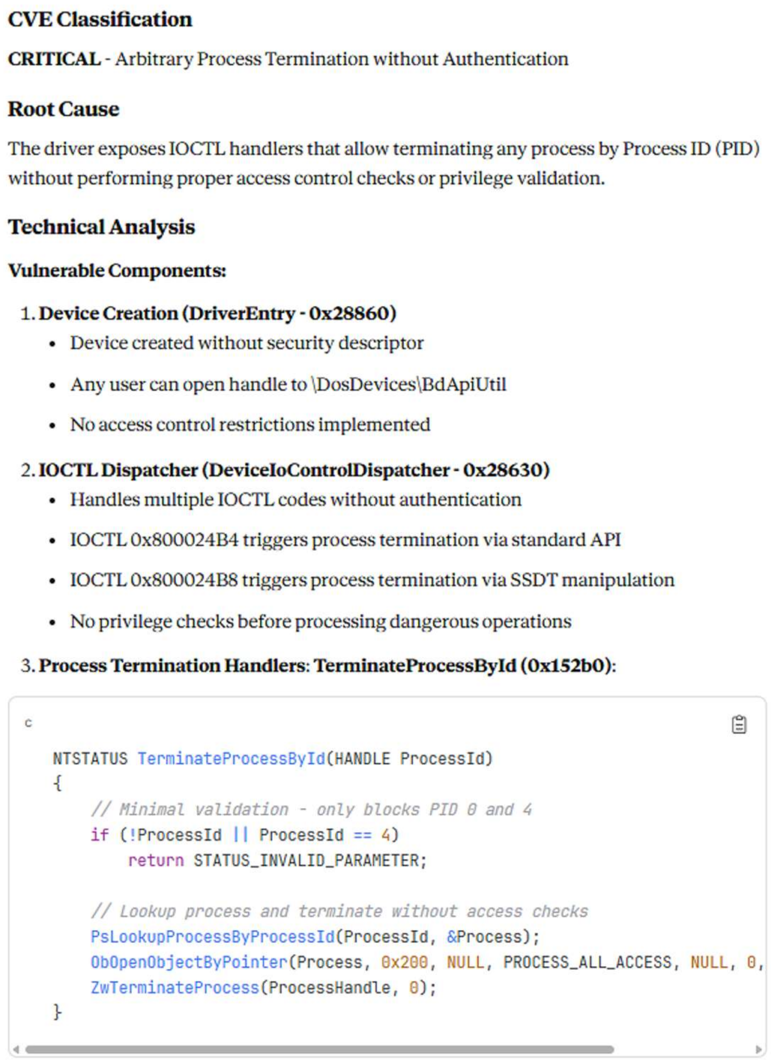
As shown in Fig. 4, it successfully identified functions that could be used for malicious purposes and provided recommendations for their potential undocumented use. In particular, the report generated by LLM contains:

1) a general description of the driver, including short metadata and unique identifiers (indicators of compromise, IOCs) such as hashes;

2) a detailed description and classification of the detected vulnerability.

The report states that the BdApiUtil.sys driver contains a critical privilege escalation vulnerability that allows any user with access to the device to terminate arbitrary processes on the system. The model was provided with a detailed technical analysis of the vulnerability, including proof-of-concept (PoC) code that demonstrates its exploitation. At the end of the report, the model provided recommendations for mitigating the identified vulnerability, including code-level fixes.

The vulnerabilities of the analyzed driver are publicly documented, and the model's conclusions fully coincide with those of cybersecurity experts, confirming its accuracy in analyzing complex binaries. The model was not specifically trained on this driver; it relied solely on its knowledge of the vulnerabilities and code patterns commonly associated with them to construct logical conclusions. Despite this, the vulnerability was successfully detected.



CVE Classification

CRITICAL - Arbitrary Process Termination without Authentication

Root Cause

The driver exposes IOCTL handlers that allow terminating any process by Process ID (PID) without performing proper access control checks or privilege validation.

Technical Analysis

Vulnerable Components:

1. **Device Creation (DriverEntry - 0x28860)**
   - Device created without security descriptor
   - Any user can open handle to \DosDevices\BdApiUtil
   - No access control restrictions implemented
2. **IOCTL Dispatcher (DeviceIoControlDispatcher - 0x28630)**
   - Handles multiple IOCTL codes without authentication
   - IOCTL 0x800024B4 triggers process termination via standard API
   - IOCTL 0x800024B8 triggers process termination via SSDT manipulation
   - No privilege checks before processing dangerous operations
3. **Process Termination Handlers: TerminateProcessById (0x152b0):**

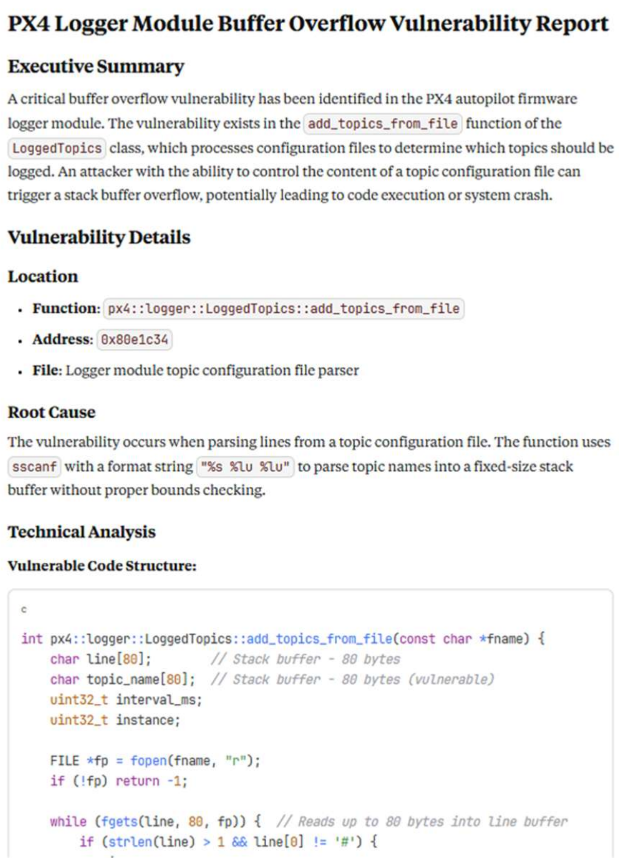
```c
NTSTATUS TerminateProcessById(HANDLE ProcessId)
{
    // Minimal validation - only blocks PID 0 and 4
    if (!ProcessId || ProcessId == 4)
        return STATUS_INVALID_PARAMETER;

    // Lookup process and terminate without access checks
    PsLookupProcessByProcessId(ProcessId, &Process);
    ObOpenObjectByPointer(Process, 0x200, NULL, PROCESS_ALL_ACCESS, NULL, 0,
    ZwTerminateProcess(ProcessHandle, 0);
}
```

Fig. 4. Part of the report generated by the large language model for the BdAPIUtil.sys driver

5. 4. 2. Firmware

This experiment evaluated the ability to detect known vulnerabilities in the PX4 logger module. The binary analyzed was px4_fmu-v5_default.elf, an ELF 32-bit LSB executable for ARM, corresponding to the PX4-Autopilot firmware version 1.14.3. A fragment of the generated vulnerability report is shown in Fig. 5.



PX4 Logger Module Buffer Overflow Vulnerability Report

Executive Summary

A critical buffer overflow vulnerability has been identified in the PX4 autopilot firmware logger module. The vulnerability exists in the `add_topics_from_file` function of the `LoggedTopics` class, which processes configuration files to determine which topics should be logged. An attacker with the ability to control the content of a topic configuration file can trigger a stack buffer overflow, potentially leading to code execution or system crash.

Vulnerability Details

Location

- **Function:** `px4::logger::LoggedTopics::add_topics_from_file`
- **Address:** `0x80e1c34`
- **File:** Logger module topic configuration file parser

Root Cause

The vulnerability occurs when parsing lines from a topic configuration file. The function uses `sscanf` with a format string `"%s %lu %lu"` to parse topic names into a fixed-size stack buffer without proper bounds checking.

Technical Analysis

Vulnerable Code Structure:

```c
int px4::logger::LoggedTopics::add_topics_from_file(const char *fname) {
    char line[80];        // Stack buffer - 80 bytes
    char topic_name[80];  // Stack buffer - 80 bytes (vulnerable)
    uint32_t interval_ms;
    uint32_t instance;

    FILE *fp = fopen(fname, "r");
    if (!fp) return -1;

    while (fgets(line, 80, fp)) {  // Reads up to 80 bytes into line buffer
        if (strlen(line) > 1 && line[0] != '#') {
```

Fig. 5. Vulnerability report in PX4 logger module

As a result, a vulnerability corresponding to the registered record was successfully identified [13]. The vulnerability was discovered offline, without any external intervention, prompts, or access to information sources.

5. 4. 3. Applications and libraries

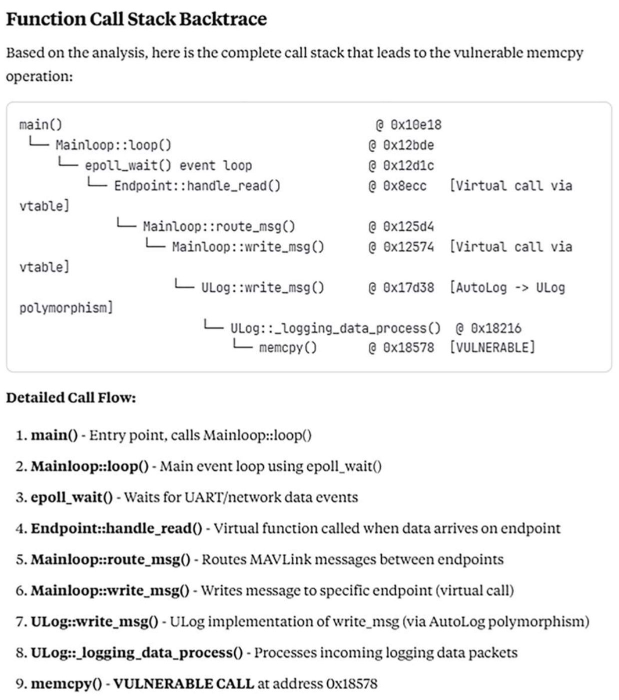
Console applications and libraries are widely used in UAV systems. The experiments described below were conducted to evaluate the effectiveness of the proposed approach in detecting vulnerabilities in these types of binaries.

Console applications could be vulnerable to a variety of vulnerabilities, including stack buffer overflows, use after free, double free, and format string vulnerabilities.

Representative binaries were selected to evaluate the capabilities of the model, each containing intentionally introduced vulnerabilities that reflect real-world issues found in UAV firmware and mission control components. Although these binaries have a relatively simple control flow, they serve as benchmarks for testing the LLM's ability to analyze low-level code structures. As an example, Fig. 6 shows a fragment of a report generated during buffer overflow analysis in the MAVLink Router application (mavLink-routerd).

LLM additionally provided an analysis of the vulnerability found, with an assessment of its severity, description and analysis of the reasons that led to its appearance.

Artificially developed applications. Additionally, experiments were performed on artificially developed examples. For the experiment described below, the vulnerability was intentionally introduced into the source code of an open source C++ application. C++ is a language often used in UAV systems. As shown in Fig. 7, the vulnerability was successfully identified, showing its effectiveness in detecting vulnerabilities in applications developed in the object-oriented programming style.



Function Call Stack Backtrace

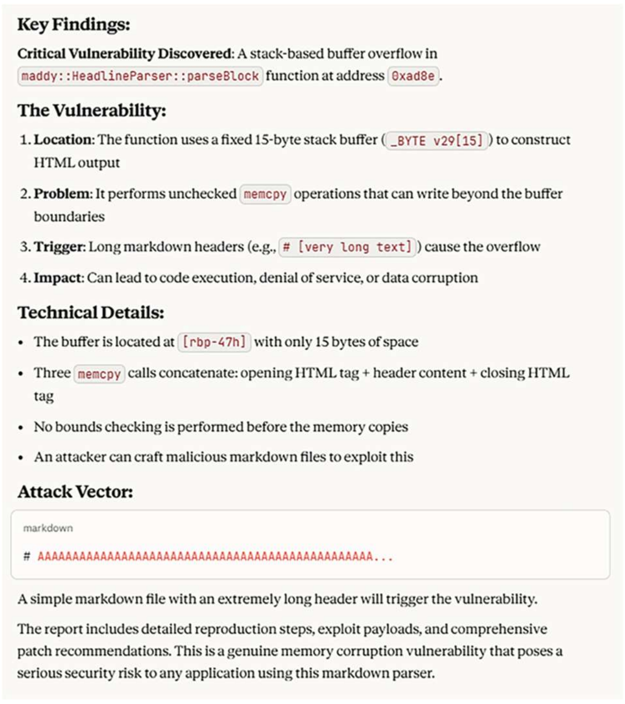
Based on the analysis, here is the complete call stack that leads to the vulnerable memcpy operation:

```
main()                                          @ 0x10e18
 └── Mainloop::loop()                          @ 0x12bde
     └── epoll_wait() event loop               @ 0x12d1c
         └── Endpoint::handle_read()           @ 0x8ecc   [Virtual call via
vtable]
             └── Mainloop::route_msg()         @ 0x125d4
                 └── Mainloop::write_msg()     @ 0x12574  [Virtual call via
vtable]
                     └── ULog::write_msg()     @ 0x17d38  [AutoLog -> ULog
polymorphism]
                         └── ULog::_logging_data_process()  @ 0x18216
                             └── memcpy()      @ 0x18578  [VULNERABLE]
```

Detailed Call Flow:

1. **main()** - Entry point, calls Mainloop::loop()
2. **Mainloop::loop()** - Main event loop using epoll_wait()
3. **epoll_wait()** - Waits for UART/network data events
4. **Endpoint::handle_read()** - Virtual function called when data arrives on endpoint
5. **Mainloop::route_msg()** - Routes MAVLink messages between endpoints
6. **Mainloop::write_msg()** - Writes message to specific endpoint (virtual call)
7. **ULog::write_msg()** - ULog implementation of write_msg (via AutoLog polymorphism)
8. **ULog::_logging_data_process()** - Processes incoming logging data packets
9. **memcpy()** - **VULNERABLE CALL** at address 0x18578

Fig. 6. A fragment of the vulnerability report in the mavlink-routerd application



Key Findings:

Critical Vulnerability Discovered: A stack-based buffer overflow in `maddy::HeadlineParser::parseBlock` function at address `0xad8e`.

The Vulnerability:

1. **Location:** The function uses a fixed 15-byte stack buffer (`_BYTE v29[15]`) to construct HTML output
2. **Problem:** It performs unchecked `memcpy` operations that can write beyond the buffer boundaries
3. **Trigger:** Long markdown headers (e.g., `# [very long text]`) cause the overflow
4. **Impact:** Can lead to code execution, denial of service, or data corruption

Technical Details:

- The buffer is located at `[rbp-47h]` with only 15 bytes of space
- Three `memcpy` calls concatenate: opening HTML tag + header content + closing HTML tag
- No bounds checking is performed before the memory copies
- An attacker can craft malicious markdown files to exploit this

Attack Vector:

```markdown
# AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
```

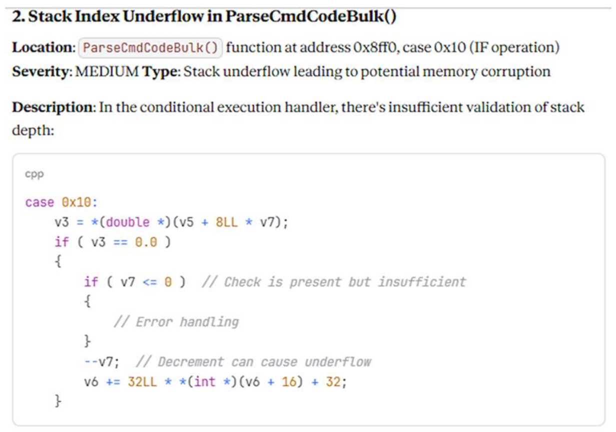
A simple markdown file with an extremely long header will trigger the vulnerability.

The report includes detailed reproduction steps, exploit payloads, and comprehensive patch recommendations. This is a genuine memory corruption vulnerability that poses a serious security risk to any application using this markdown parser.

Fig. 7. Fragment of a report on a stack-based buffer overflow vulnerability in an artificially developed application

In UAV applications, libraries are often used to perform a wide range of mathematical calculations. The open source muparser library was used in this experiment.

LLM identified three potential vulnerabilities, two of which were considered false positives, likely due to the complexity of the code base and the limitations inherent in the IDA decompilation process. However, the real flaw was found in the mu::ParserBase::ParseCmdCodeBulk() function (Fig. 8).


2. Stack Index Underflow in ParseCmdCodeBulk()

Location: `ParseCmdCodeBulk()` function at address 0x8ff0, case 0x10 (IF operation)

Severity: MEDIUM **Type:** Stack underflow leading to potential memory corruption

Description: In the conditional execution handler, there's insufficient validation of stack depth:

```cpp
case 0x10:
    v3 = *(double *)(v5 + 8LL * v7);
    if ( v3 == 0.0 )
    {
        if ( v7 <= 0 )  // Check is present but insufficient
        {
            // Error handling
        }
        --v7;  // Decrement can cause underflow
        v6 += 32LL * *(int *)(v6 + 16) + 32;
    }
```

Fig. 8. Vulnerability report in the muparser library

This vulnerability leads to an integer overflow, which can be exploited to achieve arbitrary code execution, which is a critical issue for UAV systems. Such libraries can be used in onboard computing modules or telemetry analysis routines. The vulnerability found corresponds to the documented vulnerability OSV-2020-1349.

5. 4. 4. Experimental evaluation of the method with specific tools and modes

To further verify the effectiveness of the proposed method, we shall compare its performance with an alternative analysis method, namely fuzzing using AFL++ with extended mutation strategies. Like the proposed static method, fuzzing is able to work with any, including cleaned binary files. It is popular and effective, and, like the proposed method, is under equal uncertainty conditions, which ensures the correctness of the comparison.

We shall also compare the proposed method in the cases of integration of Binary Ninja with LLM Claude Sonnet 4.0 via MCP, and integration of IDA Pro Hex-Rays with LLM Claude Sonnet 4.0, evaluated under different LLM operating modes.

A total of 21 binary samples were analyzed in this experiment.

The comparison results are given in Table 2.

Table 2

Comparative analysis results

Method	True-positive, %	False positives, %	Average time, min
Fuzzing using AFL++	93.3	0.0	40.4
Binary Ninja and Claude Sonnet 4.0	66.7	0.3	4.2
IDA Pro Hex-Rays and Claude Sonnet 4.0 (standard mode)	80.0	0.26	3.5
IDA Pro Hex-Rays and Claude Sonnet 4.0 (advanced thinking mode)	94.7	0.28	9.7

The result is considered a "true positive" when the method correctly identifies the vulnerability.

For the combination of IDA Pro with LLM Claude Sonnet 4.0 under a standard mode, the proposed method has shown its effectiveness, including in cases with firmware written in C or C++. The decompiler successfully performed the reconstruction of object-oriented code patterns and minimized the noise in the pseudocode representation. In contrast, Binary Ninja paired with LLM Claude Sonnet 4.0 showed lower detection accuracy, as evidenced by an increase in the number of false positives and missed vulnerabilities, which limits its reliability in environments where accuracy is crucial.

Among all the variants, the proposed method with LLM under an extended thinking mode achieved the highest detection rates, compared to fuzzing, which is considered one of the most effective analysis methods.

5. 5. Comparison of different options for cloud-based implementations of large language models

Given the growing diversity of LLM models and the widespread use of the cloud approach, it is important to evaluate how the proposed method works with different LLM implementations. This experiment evaluates the efficiency and cost-effectiveness of the proposed approach when applied to several cloud-based LLMs, including Claude 4.1 Opus, Claude 4 Opus, DeepSeek-R1, Amazon Nova-Pro, and xAI grok-code-fast-1.

To support multiple LLMs accessible via AWS Bedrock, an adaptation of the MCP-based communication architecture was made: model calls were routed through cline as an orchestration layer, and Bedrock provided access to the models. The rest of the workflow – instruction template, tool usage order, MCP bridge on the decompiler side, and structured reporting – was performed according to Algorithm 1 (Fig. 3) without any changes.

Each model received the same analysis plan and input data. For each run, the following was recorded:

a) tokens uploaded into the model (tokens up);

b) tokens generated by the model (tokens down);

c) provider-side prompt-cache tokens (tokens cache);

d) total wall-clock time;

e) two binary outcomes – whether a Markdown report was generated (md_file) and whether the target vulnerability was localized/confirmed (find);

f) two conservative token-based cost proxies: Proxy A, calculated as max(0, up+down−cache) , and Proxy B, calculated as (up + down + 0.1 ×cache).

The results are summarized in Table 3, in which k denotes 10^3 tokens, and throughput (tokens/s) is calculated as the ratio of tokens (tokens down) to the total execution time.

As can be seen, four out of five models successfully completed the entire analysis pipeline (md_file = T and find = T); while DeepSeek-R1 failed to generate a report or identify a vulnerability.

Among the successful runs, grok-fast-1 achieved the lowest latency (134 s), under caching-friendly pricing, and near-zero Proxy A due to its granular instruction caching. Claude 4.1 Opus balanced performance with moderate latency (359 s) and competitive token volume, while Claude 4 Opus had a ~10% higher token cost and ~40% longer execution time compared to Cloud 4.1 Opus. Amazon Nova-Pro completed the analysis but required significantly more input context (3.1 million tokens), resulting in the highest cost across both indirect price estimates.

Table 3

Results of evaluating different models

Model	Up, k	Down, k	Cache, k	Time, s	md_file, boolean	Find, boolean	Throughput, tokens/s	Proxy A, k	Proxy B, k
Claude 4.1 Opus	480.3	5.1	41.0	359	T	T	14.2	444.4	489.5
Claude 4 Opus	528.8	5.1	43.9	503	T	T	10.1	490.0	538.3
DeepSeek-R1	417.3	12.1	60.8	154	F	F	78.6	368.6	435.5
Amazon Nova-Pro	3100.0	2.7	224.0	410	T	T	6.6	2878.7	3125.1
xAI grok-fast-1	173.5	5.9	3100.0	134	T	T	44.0	0.0	489.4

6. Discussion of characteristics of the proposed software model for the network of unmanned aerial vehicles

Our results from experiments illustrated in Fig. 4–8 are attributed to the ability of LLMs to accurately identify vulnerabilities present in the binary sample, when combined with the context provided by the decompiler. The results given in Table 2 indicate that not only the use of LLM affects the result but also the properties of the decompiler itself and its ability to accurately process binary samples. The mode of operation of the LLM is also important. Significant differences between the characteristics of the models (Table 3) are explained by the different policies of cloud service providers and different potential purposes of cloud large language models. This indicates the need for a well-founded choice of model for tasks of mass analysis of samples.

The devised communication architecture model differs in the use of the MCP protocol, LLM capabilities, and decompiler. Unlike existing solutions [17–20], which use monolithic frameworks, the proposed modular system is flexible. The results of works [8, 9] were supplemented by the design of an MCP-oriented architecture of interactions between LLM and reverse engineering tools. It connects LLM with decompilers (for example, IDA Pro) via the MCP protocol, which makes it possible to call tools, exchange data, and easily replace decompilers with others. The proposed architecture model (Fig. 1), unlike existing ones focused on long-term analysis, allows its use for operational analysis and security solution search tasks (Fig. 2). This is relevant for quick security fixes when malware spreads in UAV networks under operational conditions, as shown in Fig. 2. However, unlike the solution proposed in [15], this architecture model and method outperform dynamic analysis tools in terms of speed.

The proposed structured instruction template for binary analysis embeds a binary context and provides step-by-step construction of logical conclusions of LLM for reverse engineering, which gives an advantage over conventional approaches [11]. Unlike the instruction template in work [19], focused on the taint analysis method, the proposed template is universal. The presence of such a template makes it possible to reduce the requirements for the technical training of the specialist performing the analysis; however, it makes it possible to obtain positive results even in the requirements of studying complex samples (for example, drivers, libraries, firmware of UAVs, as shown by experiments (Fig. 4–8)).

The method illustrated by the algorithm in Fig. 3 offers a formal analysis pipeline that integrates the tools used in [11, 12, 14, 15], and, unlike existing works, enriches their capabilities by applying the capabilities of the logical justification of LLMs. These include static and dynamic analysis, LLM inference process and CVE correlation, report generation containing code references, proof of concept (PoC) (Fig. 4–8). Software mitigation recommendations in the form of reports are focused on different types of UAV software.

Experimental verification confirmed the feasibility and competitiveness of the proposed method compared to the well-known fuzzing method [14]. Fuzzing remains a powerful method; however, its main drawback is low computational efficiency (Table 2). When applied to full-scale UAV firmware, fuzzing can require from several hours to days due to the size and complexity of the firmware logic. Also, fuzzing only provides a fault trace and, as a rule, does not provide an idea of the origin, location, or criticality of the identified vulnerabilities, which requires further manual reverse engineering. In contrast, the proposed method is distinguished by the generation of vulnerability hypotheses that can be used by a specialist conducting the research. Experiments on 21 binary files, including firmware, libraries, drivers and specially created samples, demonstrate the ability of the proposed approach to autonomously identify vulnerabilities. In particular, experiments were conducted with the vulnerabilities of PX4/ArduPilot, muparser, and BdApiUtil.sys and other binary samples used in UAV systems. Our results are consistent with the findings related to known vulnerabilities. The proposed approach, while maintaining high accuracy (TP = 94.7%), is on average 4 times faster than fuzzing (Table 2), and less demanding on the qualifications of the researcher.

An experimental comparison of the work of the proposed approach for cloud implementations of LLM was carried out. Analysis revealed the effectiveness of the proposed method on various cloud implementations of LLM integrated via the MCP protocol. Unlike study [21], the proposed method was used in the experiments to establish its suitability for security analysis in UAV systems. As a result of the comparison (Table 3), the xAI grok-fast-1 model showed the best time (134 s) and the highest throughput (44 t/s), which indicates its use in scenarios of blocking a rapidly spreading malware, at a favorable cost factor. The Amazon Nova-Pro model turned out to be economically inefficient for this task (due to the consumption of input tokens and high cost), so it is difficult to recommend it for mass analysis of UAV software samples. The Claude 4.1 Opus model is slower than Grok but, as a rule, has deeper reasoning capabilities for complex vulnerabilities, and by other factors is also acceptable for centralized solution of swarm security tasks.

The results of the study could be used by UAV software developers for automated security audits of third-party developers, as well as for checking their own binary samples. The proposed architectural solutions and method could be applied in the components of the automated patch generation system in UAV swarm security systems, provided that the software samples being tested are compact. It has potential for use in cyber incident response centers for the operational analysis of suspicious binary files extracted from captured UAV samples or UAV control stations in order to analyze them for signs of malware activity. The expected effects of use are quick and labor-intensive obtaining of information about the presence of a vulnerability, its causes, and ways to fix it.

A limitation of the proposed approach is the potential stochasticity in the results of experiments, due to the non-determinism of the nature of the response generation algorithm inherent in LLMs. However, with clear instructions, a large language model allows us to detect the very fact of the presence of vulnerability with a high degree of stability.

The disadvantage of our study is that a relatively limited number of samples were used as the initial data for the experiments. This is primarily due to the fact that only open source UAV software was used in order not to violate the manufacturer's requirements, since most of the firmware available in the public domain is cryptographically protected. The next requirement was the use of software with declared vulnerabilities for confident validation of responses, which also narrowed the range of samples. It should be noted that the number of input data was partially increased by introducing artificial samples with known vulnerabilities developed for the experiment.

This study in the future may involve a thorough analysis of the practical implementation of the proposed approach within the framework of digital immunity systems, as shown in Fig. 2. The results of our study provide opportunities for further advancements in security systems for UAV networks, as well as hot fixes of vulnerable UAV software components. Future research may focus on building and evaluating the performance of such systems in real-world interaction with devices.

7. Conclusions

1. A model of the architecture of LLM interaction with binary analysis tools has been proposed, as part of the architecture of UAV network security systems. Unlike existing monolithic solutions, the model is based on the adaptation of the MCP protocol to the tasks of reverse engineering of binary samples of UAV software. This makes it possible to ensure the speed and quality of analysis of binary samples of UAV software for existing vulnerabilities. Additionally, the need for the level of technical training of a specialist who performs the analysis of binary vulnerabilities is reduced.

2. A technique for forming contextual constraints in the form of an instruction template has been devised. The proposed structure and content of the instruction template provides a reduction in hallucinations and incorrect logical conclusions of LLMs when interpreting contextual data obtained from the decompiler, as well as provides multiple use of the template for various tasks of analyzing binary samples of UAV software, in contrast to partial cases of prompts. This is ensured by the presence of variable template fields, into which information is substituted for the current sample, other instructions do not need to be changed since they are universal.

3. A method for automated vulnerability detection in UAV software has been proposed by integrating static analysis tools with LLM logical inference. Unlike existing solutions, the algorithm of actions according to the method works as a conveyor based on the proposed communication architecture, using a template. This allows for a full cycle of the vulnerability detection process with minimal human expert intervention. Compared to conventional highly effective methods (fuzzing), the proposed method provides an average increase in accuracy of up to 94.7% while reducing the analysis time by 4 times, under the expanded thinking mode. Unlike existing ones, the proposed approach provides automated reconstruction of the logic of stripped binary files ("stripped") and minimal human expert intervention.

4. Experimental verification of the method was carried out on software found in UAV systems, in particular firmware, drivers, libraries, and accompanying applications, represented by binary samples. Experiments have demonstrated the efficiency of the proposed method in finding vulnerabilities on real and synthetic binary samples of UAV system software with a total of 21 samples. We have illustrated the ability of LLM to correctly and in an acceptable time draw conclusions about the presence, causes of vulnerability and provide recommendations for elimination, which is a promising characteristic in the composition of automated UAV security systems. This is evidenced by the indicators of 94.7% true-positive activations and 0.28% false-positives with an average analysis time of 9.7 minutes.

5. A comparative analysis of the practical application of the method using different versions of LLM revealed the prospects of the xAI grok-fast-1 and Claude 4.1 Opus versions for mass sample analysis tasks, in particular in the tasks of the UAV network security center. xAI grok-fast-1 achieved 134 s delay, with a favorable conservative cost estimate, due to caching, Cloud 4.1 Opus showed 359 s. delay and a moderately conservative cost estimate, compared to other models. Analysis using cloud versions of LLMs is well suited for automated static analysis of UAV firmware during development, especially in CI/CD pipelines. However, in the future, it remains possible to use it for real-time analysis, in the tasks of the UAV network security centers and digital immunity systems of unmanned networks.

Conflicts of interest

The authors declare that they have no conflicts of interest in relation to the current study, including financial, personal, authorship, or any other, that could affect the study, as well as the results reported in this paper.

Funding

The work was supported by the National Science Foundation (NSF) under grant CAREER-2048266 and the National Academy of Sciences of the United States.

Data availability

The manuscript has associated data in the GitHub repository [30].

Use of artificial intelligence

The experiments used the capabilities of large language models, according to the method described in the paper. The reports provided by LLMs, shown in Fig. 4–8, are built in combination with the instructions and architecture proposed by the authors of the paper, according to the proposed method, and are part of the current study. Validation of the model responses was carried out by the authors as experts in the subject area, on known vulnerability samples.

Acknowledgments

The authors express their gratitude to Yurii Prykhodko, a student at the Institute of Physics and Technology, for his assistance in setting up practical experiments.

Authors› contributions

Andrii Voitsekhovskyi: Software, Methodology, Investigation, Data curation; **Iryna Stopochkina:** Conceptualization, Methodology, Writing – original draft; **Mykola Ilin:** Resources, Validation; **Oleksii Novikov:** Conceptualization, Supervision; **Junfei Xie:** Writing – review and editing, Methodology; **Pu Sun:** Resources, Investigation.

References

1. Sivakumar, M., Tyj, N. M. (2021). A Literature Survey of Unmanned Aerial Vehicle Usage for Civil Applications. Journal of Aerospace Technology and Management, 13. https://doi.org/10.1590/jatm.v13.1233
2. Stopochkina, I., Novikov, O., Voitsekhovskyi, A., Ilin, M., Ovcharuk, M. (2025). Simulation of UAV networks on the battlefield, taking into account cyber- physical influences that affect availability. Theoretical and Applied Cybersecurity, 6 (2). https://doi.org/10.20535/tacs.2664-29132024.2.318182
3. PX4. Available at: https://px4.io/
4. What is ArduPilot? Available at: https://ardupilot.org/
5. CVE-2023-47625 Detail. National Vulnerability Database. Available at: https://nvd.nist.gov/vuln/detail/CVE-2023-47625/
6. Tyshchenko, A., Stopochkina, I. (2025). Design of a simulation tool for planning UAV mission success under combat constraints. Eastern-European Journal of Enterprise Technologies, 5 (9 (137)), 14–26. https://doi.org/10.15587/1729-4061.2025.340918
7. Vulnerabilities (Dronecode). Dronecode CVEs and Security Vulnerabilities – OpenCVE. Available at: https://app.opencve.io/cve/?vendor=dronecode
8. Siala, H. A., Lano, K. (2025). Towards Using LLMs in the Reverse Engineering of Software Systems to Object Constraint Language. 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 1–6. https://doi.org/10.1109/saner64311.2025.00096
9. Williamson, A. Q., Beauparlant, M. (2024). Malware Reverse Engineering with Large Language Model for Superior Code Comprehensibility and IoC Recommendations. https://doi.org/10.21203/rs.3.rs-4471373/v1
10. Contributing to MCP. Model Context Protocol a Series. Available at: https://modelcontextprotocol.io/community/contributing
11. Silva, C. E., Campos, J. C. (2013). Combining static and dynamic analysis for the reverse engineering of web applications. Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, 107–112. https://doi.org/10.1145/2494603.2480324
12. IDA Pro. hex-rays. Available at: https://hex-rays.com/ida-pro
13. Valgrind. Available at: https://valgrind.org/
14. Chen, C., Cui, B., Ma, J., Wu, R., Guo, J., Liu, W. (2018). A systematic review of fuzzing techniques. Computers & Security, 75, 118–137. https://doi.org/10.1016/j.cose.2018.02.002
15. Kim, Y., Cho, K., Kim, S. (2024). Challenges in Drone Firmware Analyses of Drone Firmware and Its Solutions. *arXiv*. https://doi.org/10.48550/arXiv.2312.16818
16. Zhang, B. (2021). Research Summary of Anti-debugging Technology. Journal of Physics: Conference Series, 1744 (4), 042186. https://doi.org/10.1088/1742-6596/1744/4/042186
17. Zhou, X., Zhang, T., Lo, D. (2024). Large Language Model for Vulnerability Detection: Emerging Results and Future Directions. *arXiv*. https://doi.org/10.48550/arXiv.2401.15468
18. Li, H., Hao, Y., Zhai, Y., Qian, Z. (2024). Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. Proceedings of the ACM on Programming Languages, 8 (OOPSLA1), 474–499. https://doi.org/10.1145/3649828
19. Liu, P., Sun, C., Zheng, Y., Feng, X., Qin, C., Wang, Y. et al. (2023). Harnessing the Power of LLM to Support Binary Taint Analysis. arXiv. https://doi.org/10.48550/arXiv.2310.08275
20. Li, Y., Li, X., Wu, H., Xu, M., Zhang, Y., Cheng, X. et al. (2025). Everything You Wanted to Know About LLM-based Vulnerability Detection But Were Afraid to Ask. *arXiv*. https://doi.org/10.48550/arXiv.2504.13474
21. Qin, W., Suo, L., Li, L., Yang, F. (2025). Advancing Software Vulnerability Detection with Reasoning LLMs: DeepSeek-R1′s Performance and Insights. Applied Sciences, 15 (12), 6651. https://doi.org/10.3390/app15126651
22. Introducing Claude 4. Anthropic. Available at: https://www.anthropic.com/news/claude-4
23. IDAPython API Reference. Available at: https://python.docs.hex-rays.com/
24. Binary Ninja. Available at: https://binary.ninja/
25. AFL++ Overview. Available at: https://aflplus.plus/
26. Claude Opus 4.1 (2025). Anthropic. Available at: https://www.anthropic.com/news/claude-opus-4-1
27. DeepSeek-R1 Release. DeepSeek. Available at: https://api-docs.deepseek.com/news/news250120/
28. Amazon Nova - generative AI foundational models. Available at: https://aws.amazon.com/ru/nova/
29. Grok Code Fast 1. x.ai. Available at: https://x.ai/news/grok-code-fast-1
30. Impress-U-IS-KPI. Available at: https://github.com/Impress-U-IS-KPI/data_processing