Automated LLM Exploit Generation FrameWork (ALEGF)

J. Santos and J. Guerreiro

Software development frequently suffer from programming defects, mistakes and design flaws introducing critical security vulnerabilities. These weaknesses can be exploited causing significant operational disruption and organizational losses. Identifying and exploiting such vulnerabilities is a complex task that requires binary or source code detailed analysis, operating systems deep knowledge, file system formats and processor architectures, as well as expertise in multiple techniques. Traditional Automatic Exploit Generation (AEG) approaches rely on symbolic execution, fuzzing and heuristic-based methods. In this paper, an automated framework is presented and designed to evaluate Large Language Models (LLMs) to detect vulnerabilities and generate exploits without human intervention. The framework operates within a controlled, sandboxed and fully reproducible environment to prevent real-word security risks while enabling systematic experimentation. The objective is to assess whether LLMs can serve as auxiliary tools for software security testing and vulnerability analysis. The results provide insights into the feasibility, limitations and potential LLM-driven automated exploitation, yet not competitive with AEG tools state-of-the-art.

1. Introduction

Software incremental parts being developed with tight deadlines can inadvertently introduce software vulnerabilities when joint together if security is not integrated into the process.

Not only agile processes can increase vulnerabilities risks, also software architecture, design and programming flaws can contribute to software vulnerabilities. Software vulnerabilities are, nowadays, one of the major threats to computer systems, therefore identifying code vulnerabilities with techniques like auditing, fuzzing applications, threat modeling and penetration testing becomes critical in software development to mitigate vulnerabilities and decrease the number of attacks. Commonly, a vulnerability is triggered by a specific input and a software crash may occur and be exploited, consequently starting an attack using exploit models that deal with distinct scenarios .

Attackers traditionally use manual exploits for assessment, but they are time consuming and depend on large experience in security knowledge to analyze and create a working and efficient exploit for the found vulnerability. The primary objective of this work is to evaluate the capability of LLMs to analyze C-language source code for vulnerabilities and autonomously generate a exploit payloads in a fully human-free setting. To achieve this objective, four specific goals were defined:

- **Vulnerability Analysis:** Access whether LLMs can examine C source code and accurately identify security vulnerabilities;

- **Automated Exploit Generation:** Determine whether LLMs are able to synthesize candidate exploits for controlled test programs without human intervention;
- **Sandboxed Validation:** Implement safe and isolated environments to validate the generated exploits and measure their effectiveness while ensuring that all experimentation remains contained and non-destructive;
- **Evaluation of Reliability and Limitations:** Quantify the success rate, reproducibility and constraints of fully automated LLM-driven exploit generation, providing insights into current capabilities and potential areas for improvements.

The main contributions of this paper include: (i) an empirical evaluation on LLMs ability to autonomously analyze vulnerable C programs and construct exploit payloads, (ii) a structured evaluation framework for assessing LLM-driven AEG and (iii) a discussion of the strengths and limitations of using LLMs as automated tools for vulnerability discovery and exploit synthesis.

The remainder of this article is organized as follows: in Section 2 Related Work is presented. The Code Vulnerability Analysis and Automated Exploit Generation Framework using LLMs is addressed in Section 3, tests and results are presented in Section 4 and Section 5 concludes the article and describes the future work.

2. Related Work

A wide range of research has focused on detecting software vulnerabilities methods and generating automated exploits. Program analysis techniques generally fall into two categories: static analysis and dynamic analysis. Static analysis examines source code without executing the program, evaluating data flow, control flow and lexical structures to identify potential weaknesses. Dynamic analysis, in contrast, involves executing the program and observe runtime behavior, detect memory corruption and uncover execution errors using automated debugging tools . Several studies have shown that combining dynamic analysis with visualization techniques enhances structural and semantic understanding, supporting design flaws detection and complex vulnerability patterns . Fuzzing is another widely adopted technique for vulnerability detection. By injecting randomized inputs into a program, fuzzing exposes crashes, exceptions and undefined runtime behaviors such as memory leaks . Avgerinos *et al.* introduced the first end-to-end system capable of detecting vulnerabilities automatically and generating exploits. Their method combined static and dynamic analysis with preconditioned symbolic execution to identify program constraints and produce concrete exploit strings. In their 14 open-source programs evaluation, the system generated 16 control-flow hijacking exploits, including two previously unknown vulnerabilities. Xu *et al.* proposed an automated exploit generation tool, using symbolic execution, to detect buffer overflow vulnerabilities in binary programs. The authors approach generates exploit able to bypass operating

system protection by solving specific vulnerabilities constraints. Results demonstrated significant improvements compared to the Avgerinos *et al.* proposed system. Wang *et al.* developed an approach for automatic polymorphic exploit generation based on trampoline instructions to hijack control flow. The authors framework generalizes constraints, detects user-controllable memory regions, identifies hijack points and generates polymorphic exploit payloads capable of triggering program crashes. Dixit *et al.* introduced AngErza, a framemork built on the *angr* analysis platform allowing developers to test program behavior using automatically generated exploits. AngErza first identifies buffer overflows and estimates the overflow length, if none is found, the system searches for format string vulnerabilities. Using the detected constraints, AngErza generates payloads capable to exercise and validate the discovered vulnerabilities. Liu *et al.* proposed AEG-E, a symbolic-execution based framework that extracts control-flow graphs from binary programs while mitigating path explosion. A configurable exploit model enables the complex exploit synthesis for triggering program crashes and generating crash-based exploit payloads. The authors report that AEG-E framework achieves exploit generation 2913x faster than the REX framework proposed by Shoshitaishvili *et al.* . Jin *et al.* introduced ExGen, a cross-platform exploit generation system for smart contract vulnerabilities on blockchain platforms. By generating symbolic attack contracts and executing multi-transaction exploit sequences, ExGen identifies constraint violations within private key chain environments. The system achieved exploit 89.9% success rates on Ethereum and 96.9% on EOS, demonstrating its effectiveness in decentralized environments. More recently, Fedorchenko *et al.* presented a compiled exploit-code model for cybersecurity analysis in industrial systems. The authors method correlates vulnerability metrics, system calls and event dependencies to support automated exploit detection and classification, particularly for previously unknown exploit behavior.

On LLMs, Vaswani et al, introduced the transformer architecture that revolutionized natural language processing, eliminating recurring and convolutions needs, relying on self-attention mechanisms and consequently being more efficient in parallelization and better handling of long-range dependencies in text, founding modern LLMs.

Inaccurate data in media platforms presents, nowadays, as an obstacle to Natural Language Processing (NLP) search techniques and non-credible information may be shown, thus, high performance transformer models have been applied, like Hugging Face Transformers that have been pre-trained, enabling high-level understanding and generation of human language, augmenting the detection of fake news (). J. Devlin *et al*, developed a pre-training of Deep Bidirectional Transformers for Language Understanding model named BERT, optimizing the Hugging Face Transformers for language understanding tasks by exploring the pre-training, providing a useful reference point to detect fake news.

LLMs are machine learning models designed for natural language processing trained to learn over large amounts of text, capable of using pre-trained transformers that can be fine-tuned into specific tasks ().

LangChain is a framework to create language model-driven applications as a solution for NLP and document analysis, chatbots and code analysis, providing tools to manage prompts, handle chains of actions and interface using diverse data sources ().

LangChain uses StreamLit, a web application framework, used as an easy navigation interface. Langchain combines NLP algorithms and LLM, providing an extremely sophisticated tool to search, process and extract information from documents like PDF file formats (). A highly customized open-source framework to build end-to-end question-answering systems and enabling LLM integration with search and retrieval systems is Haystack. Haystack enables efficient searches, indexing and information retrieval from large datasets. Integrating Haystack facilitates the development and simplifies with efficient accuracy the data extraction ().

The following section (3) presents the proposed LLM-based analysis and automated exploit generation framework ALEGF, extending prior work by evaluating the human-free exploit generation feasibility using modern AI capabilities.

3. Automated LLM Exploit Generation Framework (ALEGF)

ALEG Framework

Vulnerability and exploit reports (stored in database) Models $\leftarrow$ {`Gemma3`, `Mistral`, `LLaMa3.1`, `Qwen2.5`}

Initialize central database for provenance, analyses, artifacts and results

Step 1: Code Ingestion Module Insert src into framework Prepare ingestion payloads for each LLM Write payloads to database

Step 2: LLM Integration Module Instantiate local client interface for `model` Configure adapter (prompt formatting, rate limiting, retries)

Step 3a: Analysis (Analysis & Exploit Module) Send ingestion payload to `model` Receive analysis_report Store analysis_report in database (tagged by model) Aggregate analyses Save unified analysis report Create derived payloads Persist derived payloads to database

Step 3b: Exploit Generation (Analysis & Exploit Module) Send $\langle src, derived\_payload \rangle$ to `model` Receive candidate_exploit Store candidate_exploit in database

Step 4: Validation) Submit candidate_exploit Receive validation_result $\in$ {`true`, `false`} and diagnostics Store validation_result and diagnostics in database

Step 5: Reporting and Visualization Module Generate unified report (analyses, exploits, outcomes) Compute metrics (success rate, reproducibility, timing,

confidence) Persist final reports and metrics in database return Final reports and evaluation metrics

ALEGF was designed to integrate multiple LLMs to achieve three core objectives:

- Perform static analysis of C source Code using each integrated LLM;
- Automatically generate candidate exploits based on the previously produced analyses;
- Validate each generated exploit inside a controlled, containerized execution environment (Docker).

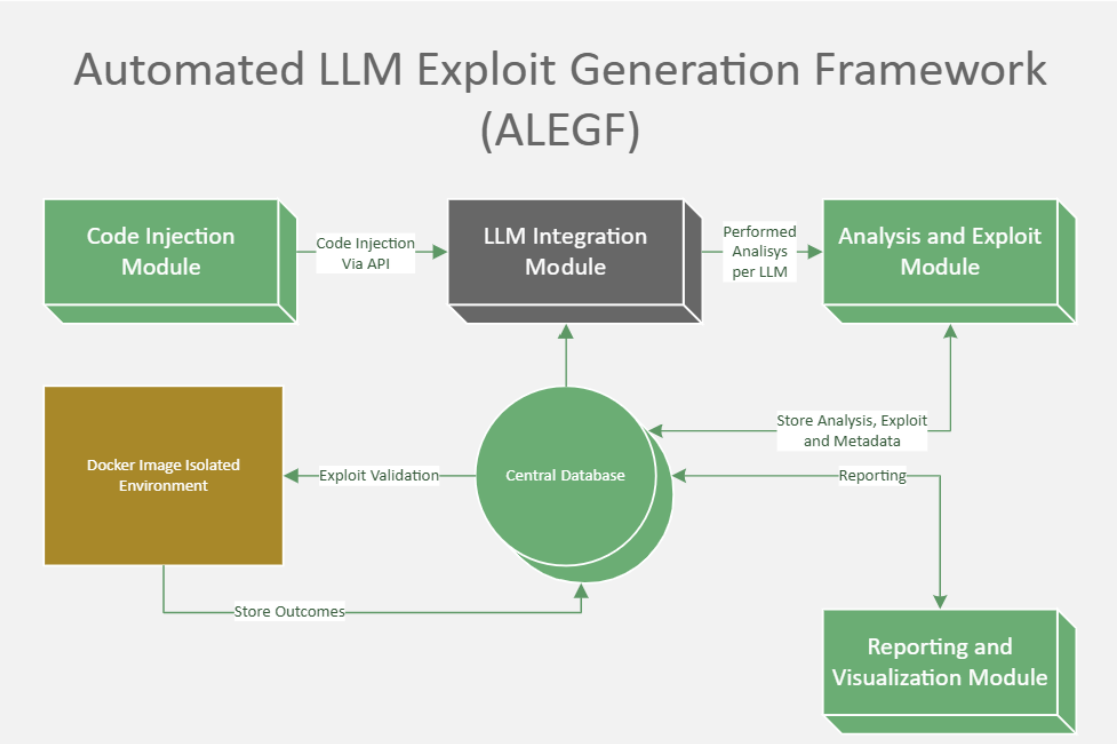

Figure 1: ALEG Framework

ALEGF was implemented primarily in Python leveraging its extensive Application Programming Interface (API) ecosystem for process orchestration, model interaction and data management. A multi-model pipeline was built using LangChain to provide a uniform interaction interface across heterogeneous LLM APIs, streamline prompt and output handling, supporting chain-of-thought style orchestration when appropriate. Eachmodel is accessed through a dedicated API adapter within the LLM Integration Module.

The workflow (Figure 1) proceeds as follows:

- A binary or source code program is supplied to ALEGF through the Code Ingestion Module which normalizes the input and forwards it to each LLM for independent static analysis.

- Each LLM produces an analysis describing potential vulnerabilities, hypotheses regarding vulnerability classes (e.g. buffer overflow, off-by-one, etc.) and diagnostic commentary. All analyses are stored in a central database alongside metadata identifying the producing model.
- The Stored analyses are subsequently redistributed via API to the LLMs via API for automated exploit proposal generation. Each generated candidate exploit is logged in the database and queued for validation.
- For each candidate exploit, ALEGF instantiates an isolated environment using Docker. Once the container is ready, the candidate exploit is executed within the sandbox and all outcomes are collected and stored.

Algorithm 1 Automated LLM Exploit Generation Framework (ALGEF)

- **Ensure:** Vulnerability and exploit reports (stored in database)
- 1: Models ← {Gemma3, Mistral, LLaMa3.1, Qwen2.5}
- 2: Initialize central database for provenance, analyses, artifacts and results
- 3: **Step 1: Code Ingestion Module**
- 4: Insert src into framework
- 5: Prepare ingestion payloads for each LLM
- 6: Write payloads to database
- 7: **Step 2: LLM Integration Module**
- 8: **for** model ∈ Models **do**
- 9: Instantiate local client interface for model
- 10: Configure adapter (prompt formatting, rate limiting, retries)
- 11: **end for**
- 12: **Step 3a: Analysis (Analysis & Exploit Module)**
- 13: **for** model ∈ Models **do**
- 14: Send ingestion payload to model
- 15: Receive analysis report
- 16: Store analysis report in database (tagged by model)
- 17: **end for**
- 18: Aggregate analyses
- 19: Save unified analysis report
- 20: Create derived payloads
- 21: Persist derived payloads to database
- 22: **Step 3b: Exploit Generation (Analysis & Exploit Module)**
- 23: **for** derived payload ∈ DerivedPayloads **do**
- 24: **for** model ∈ Models **do**
- 25: Send ⟨src, derived payload⟩ to model
- 26: Receive candidate exploit
- 27: Store candidate exploit in database
- 28: **end for**
- 29: **end for**

- 30: **Step 4: Validation**)
- 31: **for** candidate exploit ∈ Database **do**
- 32: Submit candidate exploit
- 33: Receive validation result ∈ {true, false} and diagnostics
- 34: Store validation result and diagnostics in database
- 35: **end for**
- 36: **Step 5: Reporting and Visualization Module**
- 37: **for** validation report ∈ Database **do**
- 38: Generate unified report (analyses, exploits, outcomes)
- 39: Compute metrics (success rate, reproducibility, timing, confidence)
- 40: Persist final reports and metrics in database
- 41: **end for**
- 42: return Final reports and evaluation metrics

The ALEGF implementation is structured into modular components with clear separation of concerns:

- **Code Ingestion Module:** Receives input programs forwards them for analysis;
- **LLM Integration Moduyle:** Provides a unified API interface for communication with multiple LLMs. It manages request formatting, rate-limiting, retries and standardized response parsing;
- **Analysis and Exploit Module:** Coordinates static-analysis requests, triggers exploit proposal generation, persists produced artifacts and manages validation queues. It also implements collaborative analysis mechanisms that aggregate prior results to enhance exploit quality.
- **Validation Module:** Manages container life-cycles, loads instrumented execution images, submits candidate exploits for testing and collects validation results;
- **Reporting and Visualization Module:** Generates the comparative tables, evaluation dashboards and detailed reports linking analyses, exploit attempts, validation outcomes and mitigation recommendations.

Algorithm 1 summarizes the ALEGF workflow:

- **Step 1:** In Code Ingestion Module, source code is ingested, normalized and stored in the database along with prepared LLM prompts;
- **Step 2:** For each LLM, the client interface is instantiated and prompts are configured;
- **Step 3:** Static-analysis prompts are sent to each LLM and the resulting analyses are stored. All analyses are then aggregated into a unified report. Derived exploit-oriented prompts are generated and stored. Each LLM receives theses prompts and returns a candidate exploit, which is stored;

- **Step 4:** Each candidate exploit is executed in a isolated container and validation outcomes are collected;
- **Step 5:** Validation outputs are consolidated into a final unified report, metrics are computed and results are stored in the database.

3.1. Prompt Engineering

Extensive iterative prompt-engineering experiments were performed to ensure that LLMs produced consistent, interpretable and technically meaningful outputs. The final analysis prompt (Figure 2) avoids over-specification while providing sufficient structure for models to reason about program behavior. This iterative refinement process proved essential for obtaining reproducible results in fully automated exploit-generation scenarios.

An additional instruction - "This analysis has a maximum of 10 words" - significantly improved clarity by encouraging concise vulnerability identification, thereby enhancing the subsequent exploit generation phase.

The final exploit generation prompt is shown in Figure 3.

```
You are an expert in code security. Analyze the following C code for
   potential vulnerabilities.

Code:
{code}
```

Figure 2: Analysis Prompt

```
You are a security researcher.
Given the vulnerability analyses from various models:
{all_analysis}

------------
Create a python exploit using the pwntools module for the c code {code}.
RESPOND WITH THE EXPLOIT (no explanation).
Here is the template you should follow:
### IMPORTS ###
from pwn import *
### CONFIGURATIONS ###
binary = './vuln'
p = process(binary)
### YOUR CODE HERE ###

### VALIDATION ###
print(p.recvall())
```

Figure 3: Exploit Prompt

3.2. Collaborative Analysis

ALEGF incorporates a collaborative analysis mechanism that aggregates vulnerability analyses produced different LLMs. This additional contextual layers enables models to leverage complementary insights, for example, one model's static observations may guide another model's exploit generation reasoning. This cross-model reinforcement aims to increase the generated exploit quality and plausibility.

3.3. Security and Ethical Safeguards

Throughout development, emphasis was placed on ensuring ethical and operational safety. All experiments were executed exclusively in synthetic, intentionally vulnerable programs within controlled sandbox environments.

Only fully local LLMs were used - gemma3, llama3.1, qwen2.5 and mistral - reflecting the strongest on-premise capable models available in 2025. Local deployment ensures reproducibility, preserves data privacy and aligns with institutional security requirements, avoiding reliance on external commercial APIs.

ALEGF demonstrates a modular, auditable pipeline that integrates multiple LLMs for static analysis and automated exploit generation, supported by systematic reporting and validation. The framework balances automation with strict safeguards and establishes a foundation for empirical evolution of LLM-driven exploit generation on vulnerable C programs. The following section presents the results demonstrating the capabilities and limitations of LLM-based vulnerability analysis and exploit generation.

4. Tests and Results

Several intentionally vulnerable C programs were used to emulate common vulnerability patterns typically encountered in introductory security training. These programs were selected to ensure full isolation and deterministic behavior within a sandboxed, fully local testing environment, allowing the evolution to focus exclusively on the LLMs cognitive and analytical capabilities.

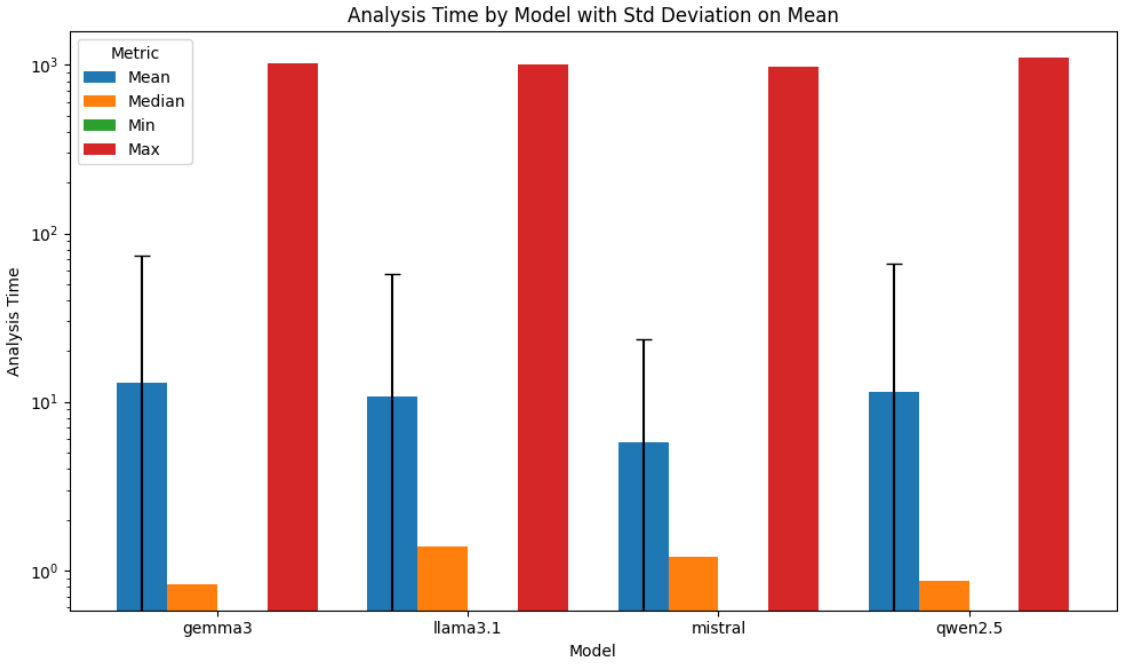

Figure 4: Analysis Prompt

Figure 4 presents the overall execution time for each model. Mistral demonstrates the fastest and most consistent runtimes with both mean and median values remaining stable across iterations. Qwen2.5 occasionally produces fast results but exhibits significant outliers, leading to a large gap between the mean and median. Gemma3 and Llama3.1 show moderate performance with noticeable variability. The large discrepancies between mean and median values indicate the presence of rare extreme runtimes, witch disproportionately affect the mean without representing typical model behavior.

Model performance was further evaluated across four prompt architectures:

- **10 Words Analysis:** A concise analysis prompt intended to force succinct vulnerability descriptions; 100 and 25 runs were executed per program;
- **No Analysis:** Designed to test whether LLMs can generate exploit without access to previous analyses; Evaluated over 100 runs;
- **Own Analysis:** Each model received its prior analysis output to support exploit generation; evaluated with 100 and 25 runs;
- **Collaborative Analysis:** Aggregates analyses from models to provide enriched context for exploit generation; 25 runs were executed.

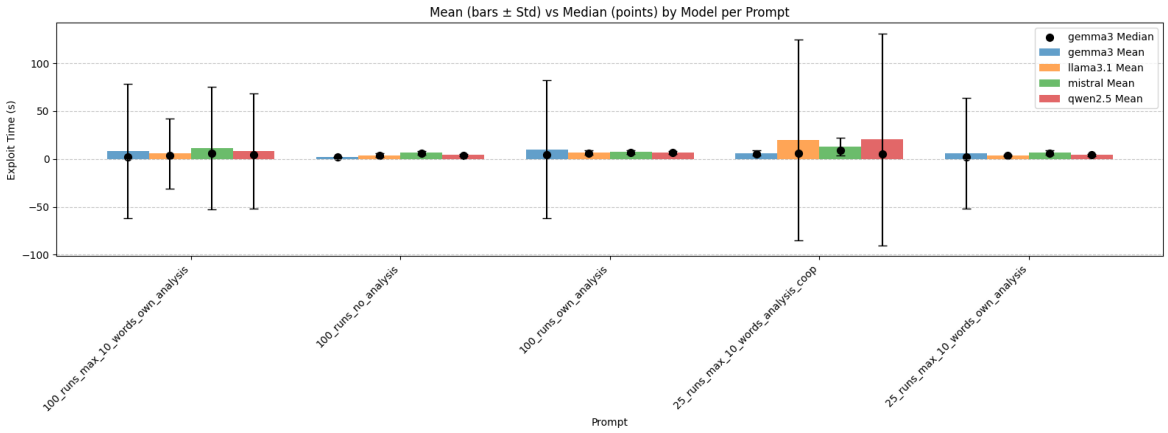

Figure 5: Prompt Engineering Analysis

Figure 5 summarizes analysis latency under different prompts. Prompts relying on "own analysis" consistently produce higher mean and maximum times due to additional processing overhead. Simpler prompts reduce median and mean runtimes, particularly for Mistral. Qwen2.5 remains highly sensitive to prompt variations, occasionally producing extreme runtimes even for simple prompts. Mistral and Llama3.1 are less affected, show stable behavior regardless of the prompt complexity. Across models, median runtimes more accurately reflect typical performance than mean times, due to the presence of extreme outliers. Models with high standard deviations (Qwen2.5 and Gemma3) display unpredictable behavior, whereas models with low variance (Mistral) demonstrate consistent and reliable performance. Outliers sensitivity is an important consideration for real-time or runtime-constrained applications. Overall, Mistral provides the most stable analysis times, whereas Qwen2.5 is the fastest only when outliers are ignored. Gemma3 and Llama3.1 occupy a middle ground, balancing performance with moderate variability.

Figure 6. shows the overall exploit-generation times.

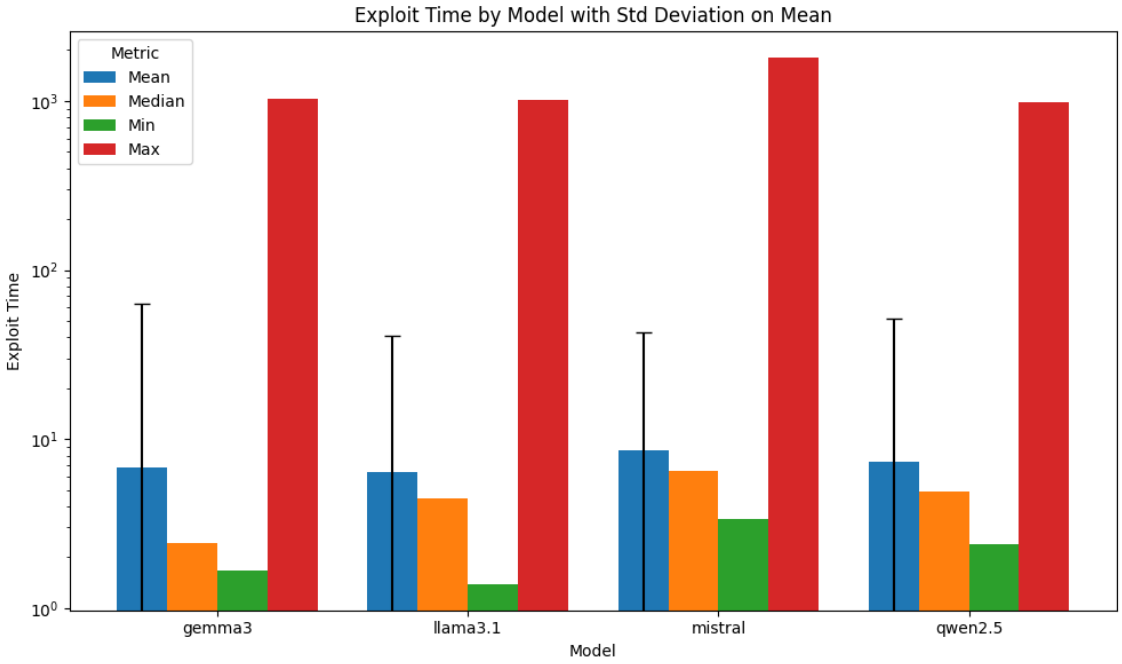

Figure 6: Overall Exploit Time

Mistral exhibits higher mean and median exploit times compared to other models, remaining consistent. Qwen2.5 demonstrate moderate average times but with high variability. Gemma3 and Llama3.1 achieve similar average exploit generation times, with occasional extreme results. Large gaps between mean and median again indicate sporadic long-duration runs, particularly when internal reasoning loops are triggered.

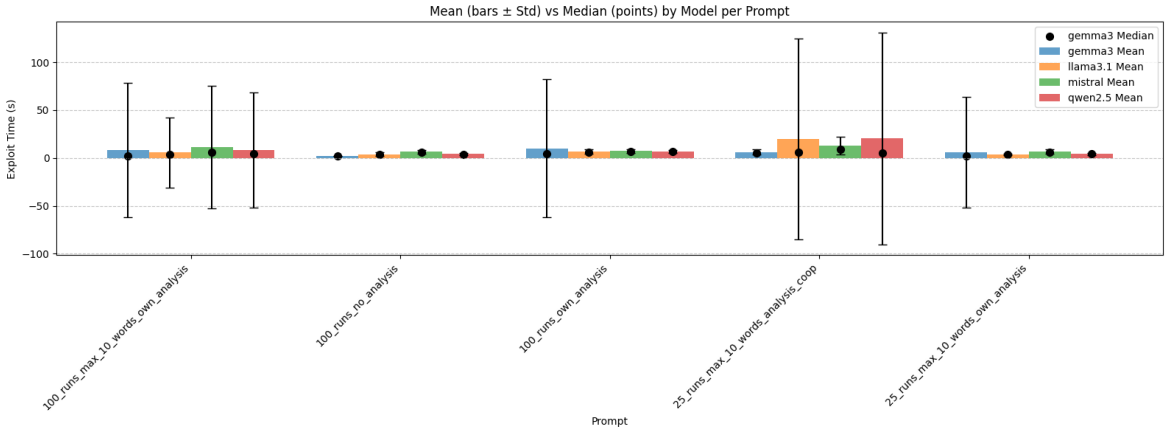

Figure 7: Overall Exploit Time

Figure 7 presents exploit-runtime distribution per prompt and model. Prompts involving "own analysis" result in significantly higher runtimes due to additional processing. Simpler prompts result in lower runtimes, especially for Mistral and Llama3.1. Qwen2.5 remains highly sensitive to prompt structure, reinforcing the need for careful prompt design in future work.

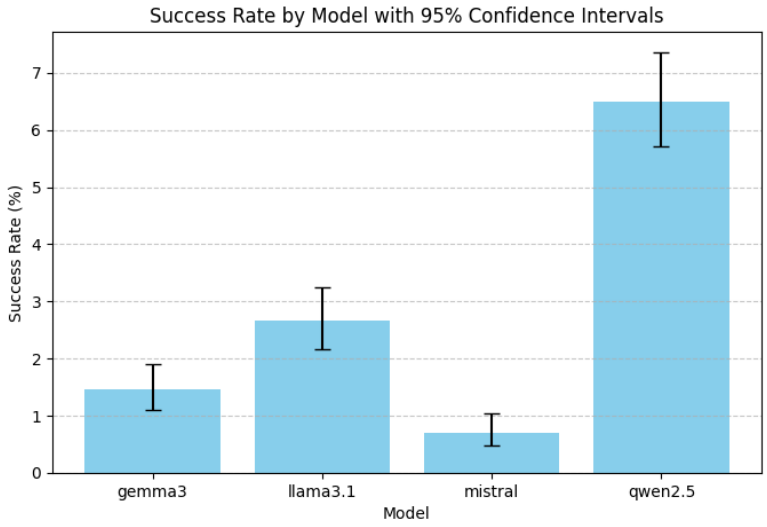

Figure 8: Overall Success Rate

Figure 8 summarizes the overall explot success rates. Results show substantial differences in model reliability. Success rates remain generally low across all models. The analysis integrates 95% Wilson confidence intervals (WCI) to quantify statistical uncertainty. Narrow intervals indicate consistent performance, while wide intervals, common in low success and small sample scenarios, reflect high variability. These intervals contextualize the results and help distinguish meaningful differences from stochastic variation. Qwen2.5 achieves the highest overall success rate, but also shoes moderate statistical uncertainty. Mistral achieves the lowest success rate but with narrow confidence intervals, indicating consistently low performance rather than random fluctuations. Gemma3 and Llama3.1 demonstrate similarly low success rates, with narrow WCIs implying stability rather than inconsistency. Even a small number of successful runs significantly impacts mean success rates and WCIs appropriately capture this effect.

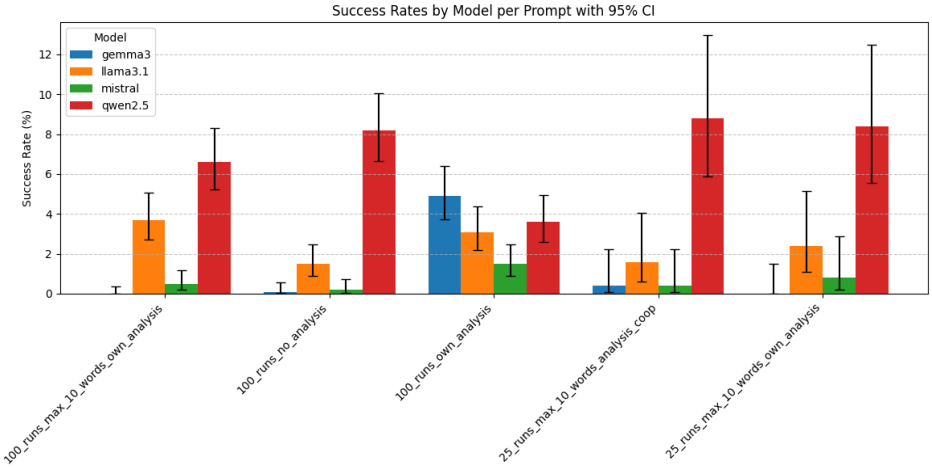

Figure 9: Prompt Success Rate

Figure 9 presents success rates per prompt. Gemma3, Mistral and Llama3.1 exhibit near-zero success using "own analysis" prompts. Qwen2.5 consistently outperforms the other models across prompt types, however, its confidence intervals widen for smaller sample sizes, indicating decreased reliability for rare success events. Mistral and Gemma3 are largely insensitive to prompt structure, showing consistently low success. Llama3.1 demonstrates intermediate behavior, achieving occasional but limited success. Collaborative analysis improves performance for Qwen2.5 but not for the remaining models.

5. Conclusions and Future Work

This work presented, to the best of our knowledge, the first systematic evaluation of LLM-driven automated exploit generation. The study examined analysis time, exploit-generation time and exploit success rates using statistical measures including mean, median, standard deviation and Wilson Confidence Intervals. The results characterize model performance, sensitivity to prompts and reliability across multiple evaluation settings. Although success rates remain low, the findings indicate that LLM performance can improve with systematic prompt engineering, greater input diversity and model-specific optimization.

Future work may explore dynamic outlier detection, automated prompt-tuning strategies and enhanced input pre-processing to increase success rates while reducing analysis and exploit-generation overhead.

This work is supported by NOVA LINCS ref. UIDB/04516/2020 (https://doi.org/10.54499/UIDB/04516/2020) and ref. UIDP/ 04516/2020 (https://doi.org/10.54499/UIDP/04516/2020) with the financial support of FCT.IP.

This work was also supported by Instituto Superior de Engenharia (ISE) from Universidade do Algarve.