UCLouvain

École polytechnique de Louvain

Detection of Command and Control Frameworks through LLM

Using Prompt Engineering

Author: **Hadrien Allegaert**
Supervisors: **Axel Legay, Kim Mens**
Readers: **Charles Pêcheur, Christophe Crochet, Charles-Henry Bertrand Van Ouytsel**
Academic year 2024-2025
Master [120] in Computer Science

Abstract

Detecting modern Command and Control (C2) frameworks is increasingly challenging, as attackers adopt encryption, obfuscation, and living-off-the-land techniques to evade traditional detection. Signature- and rule-based systems like `Snort` or `Suricata` often struggle to keep up, requiring constant updates and manual tuning to remain effective.

This Master's thesis explores whether Large Language Models (LLMs) can offer a more adaptive alternative. Specifically, it investigates the use of general-purpose LLMs to detect and classify C2 activity based on network traffic and C2-related documentation—without relying on predefined signatures or retraining.

The proposed method provides the LLM with structured network traces and C2 documentation, and asks it to determine whether the trace indicates C2 behavior—and if so, which framework is involved. The model also produces a natural language explanation of its reasoning. Detection and classification performance are evaluated using standard metrics such as Precision, Recall, and F1-score. For the reasoning component, semantic quality is assessed using BERTScore [77].

Results show strong detection performance, with an F1-score exceeding 80% on a dataset of 537 test samples. Classification yielded a `macro` F1-score around 60%, with a `weighted` precision above 70%.

This study demonstrates that off-the-shelf LLMs, when guided with prompt engineering, can effectively support the detection and attribution of C2 framework activity—without the need for model retraining or handcrafted rules.

Acknowledgments

I would like to express my deepest gratitude to everyone who supported me throughout this research project and the writing of this thesis.

First and foremost, I am indebted to Dr. Axel Legay, who proposed the topic and believed in my ability to see it through. His guidance and trust were instrumental in shaping my work. I am saddened by his departure from UCLouvain, and I deeply regret the circumstances that led to it. I wish to express my support for him and his contributions to our community. I would also like to express my regret regarding the departure of Dr. Benoît Duhoux, a former member of Axel's team.

I owe an enormous debt of gratitude to Sir Christophe Crochet, a Ph.D. (not so student) to be, and former member of Axel's team, for his unwavering kindness, patience, and guidance during the past year. Without his steady technical and moral advice, this thesis simply would not exist.

Special thanks go to Prof. Kim Mens, who kindly agreed to take over as promoter in the final semester. His willingness to step in at a critical moment ensured that my research could be completed.

I would also like to thank Prof. Charles Pêcheur and Dr. Charles-Henry Bertrand Van Ouytsel, for accepting to read my master's thesis and be part of the jury for the defense.

Finally, I want to thank my friend Guillaume Jadin, with whom I've been taking courses since the first year at UCLouvain. Warning: coding alongside him can give you a headache! His fanatical commitment to leaving every line of code spotless and maintainable really keeps you on your toes.

To everyone, named or unnamed, who encouraged me, offered feedback, or listened when I felt overwhelmed by the writing process, your support has meant more than words can say.

Thank you.

Glossary

APT Advanced Persistent Threat: *Prolonged and targeted cyberattack in which an intruder gains unauthorized access to a network and remains undetected for an extended period, aiming to steal data or monitor activity..*

C2 Command and Control Framework: *Usually used by hackers, these frameworks are designed to keep a backdoor between a compromised device and a controlled server..*

CDN Content Delivery Network: *Geographically distributed network of proxy servers and data centers that deliver web content to users based on their location, improving performance and reliability..*

CERT Computer Emergency Response Team: *Expert group tasked with receiving, reviewing, and responding to computer security incident reports and coordinating cybersecurity efforts..*

CNN Convolutional Neural Network: *Class of deep neural networks designed to process data with grid-like topology (e.g., images), using convolutional layers to automatically and adaptively learn spatial hierarchies of features..*

CRF Conditional Random Field: *Probabilistic framework for structured prediction used in machine learning to label and segment sequence data, such as part-of-speech tagging or named entity recognition..*

DNN Deep Neural Network: *Artificial neural network with multiple hidden layers between the input and output, capable of learning hierarchical representations and modeling complex patterns in data..*

DNS Domain Name System: *Hierarchical and decentralized naming system that translates human-readable domain names (e.g., example.com) into IP addresses, enabling users to locate and access resources on the internet..*

GET GET Method: *HTTP request method used to retrieve data from a specified resource; parameters are passed in the URL, and the server returns the requested content in the response body..*

GPT Generative Pre-trained Transformer: *A transformer-based large language model architecture developed by OpenAI, pre-trained on massive text corpora and capable of generating coherent and contextually relevant human-like text..*

HMM Hidden Markov Model: *Statistical model in which the system being modeled is assumed to be a Markov process with unobserved (hidden) states, often used for sequence analysis in speech recognition and bioinformatics..*

HTTP HyperText Transfer Protocol: *Application-layer protocol used for transmitting hypermedia documents such as HTML, forming the foundation of data communication on the World Wide Web..*

HTTPS HyperText Transfer Protocol Secure: *Extension of HTTP that uses Transport Layer Security (TLS) to encrypt communication between client and server, ensuring confidentiality and data integrity..*

IoCs Indicators of Compromise: *Pieces of forensic data—such as file hashes, IP addresses, URLs, or domain names—that signal a potential or active security breach or malicious activity on a network or system..*

JSONL JSON Lines: *File format where each line is a valid JSON object, commonly used for storing and streaming structured data to facilitate incremental parsing and processing..*

LLM Large Language Model: *Artificial Intelligence model fine-tuned to process natural language tasks (Natural Language Processing)..*

LSTM Long Short-Term Memory: *Specialized RNN architecture with gating mechanisms (input, forget, and output gates) designed to capture long-range dependencies and mitigate the vanishing gradient problem..*

MMLU Massive Multitask Language Understanding: *Benchmark suite for evaluating language models across a wide range of subjects and tasks to assess multitask and general knowledge capabilities..*

MoE Mixture of Experts: *Neural network architecture that combines multiple specialized subnetworks ("experts") with a gating mechanism, allowing the model to dynamically select which experts to activate for each input..*

mTLS Mutual Transport Layer Security: *Authentication protocol in which both client and server authenticate each other's certificates during the TLS handshake, providing bidirectional trust and encryption..*

NLP Natural Language Processing: *Field of artificial intelligence focused on the interaction between computers and human language, enabling machines to understand, interpret, and generate natural language text or speech..*

PCAP Packet Capture: *The process and file format for recording network packets transmitted and received over a network, used for traffic analysis, troubleshooting, and security forensics..*

PE Prompt Engineering: *The process of designing and refining prompts to elicit effective and accurate responses from large language models..*

POST POST Method: *HTTP request method used to submit data to a server to create or update a resource; parameters are included in the request body, and the server processes them accordingly..*

RAT Remote Access Trojan: *Type of malware that provides an attacker with unauthorized remote control over a victim's computer, often used for espionage, data theft, or further network compromise..*

RLHF Reinforcement Learning from Human Feedback: *Technique for aligning model behavior by training a policy or reward model based on human-provided feedback, improving performance and safety in large language models..*

RNN Recurrent Neural Network: *Type of neural network in which connections between units form directed cycles, allowing the network to maintain a memory of previous inputs for sequence modeling tasks..*

SOC Security Operations Center: *Centralized unit within an organization responsible for monitoring, detecting, analyzing, and responding to cybersecurity incidents and threats..*

TCP Transmission Control Protocol: *Core transport-layer protocol in the Internet protocol suite that provides reliable, ordered, and error-checked delivery of a stream of bytes between applications..*

TLS Transport Layer Security: *Cryptographic protocol that provides secure communication over a computer network by encrypting data and ensuring endpoint authentication and integrity..*

Contents

List of Figures

List of Tables

1 Introduction

Detecting contemporary C2 frameworks is increasingly difficult, because attackers tunnel traffic through encrypted protocols and imitate legitimate web or DNS services. Signature rules, dynamic analysis, and firewall policies often fall behind actively maintained frameworks such as `Sliver` [12] and `Cobalt Strike` [69]. The 2023 Volt Typhoon intrusion into U.S. critical infrastructure, leveraging living-off-the-land tools and compromised SOHO routers, underscores the need for adaptable detection [27].

Similar limitations appear in financially motivated campaigns. `FIN7`'s sustained operations, which combine spear-phishing with `Cobalt Strike` beacons to attack retail and hospital networks, demonstrated how well-maintained C2 tooling can evade rule-based defenses for months at a time [25].

Traditional C2 detection rests on three pillars: signature engines such as `Snort` and `Suricata` [60, 47], anomaly models that flag statistical outliers in flow data [6], and sandbox or dynamic-analysis systems that execute suspicious binaries in isolation [23]. Each pillar has well-documented weaknesses.
Signature rules degrade quickly when operators mutate payloads, encrypt channels, or tunnel traffic through benign-looking protocols, forcing SOC teams into a continual rule-maintenance cycle.
Anomaly detectors reduce rule drift but can trigger false-positive rates above 90% on enterprise traces, flooding analysts with benign alerts [40]. Sandboxes excel at catching novel binaries yet miss *living-off-the-land* techniques that rely solely on native commands.
Together, these gaps create a growing response lag: Microsoft reports a median 24-day delay between public rule release and enterprise deployment [44].
Bridging this lag requires a detector able to reason over behavior and documenta-

tion rather than fragile byte patterns.

Large Language Models offer that flexibility. Pre-trained transformers such as GPT-4o, Llama 3, and DeepSeek can ingest heterogeneous text (configuration files, log, packet traces) and draw semantic links without task-specific retraining. In malware research, Moussaileb et al. [45] show that prompt-engineered LLMs rival custom CNNs for static PE analysis, while Li et al. [34] demonstrate similar gains in bug triage. These successes suggest that an LLM supplied with C2 documentation plus network traces could classify unseen traffic by reasoning over protocol patterns, command syntax, and contextual cues that rule engines ignore. Crucially, this capability is delivered "off the shelf"; the cost is measured in prompt tokens, not months of data collection and model retraining.

This thesis asks: *Can a general-purpose LLM, guided only by prompt engineering, detect and attribute C2 activity in HTTP traffic?* Our goal is two-fold. First, we aim to determine whether the model can distinguish benign from malicious traces with high recall. Second, we test its ability to attribute each malicious trace to a specific framework while providing a human-readable explanation of its decision process.

To evaluate this question, we introduce two lightweight yet realistic C2 frameworks, `Slimper` and `Mimic`, that emulate the beaconing and data-exfiltration patterns of popular tools but remain safe for academic experimentation. Both communicate solely over HTTP.

We generate JSON-formatted traces (Safe, Slimper, Mimic) and craft concise documentation files describing each framework's (startup sequence, URI schema, command set). These inputs are concatenated with analyst-style instructions and submitted to GPT-4o via the OpenAI Assistants API. The model is prompted to declare the trace `SAFE` or `INFECTED`, name the responsible framework if infected, and justify its choice step by step. We score detection and classification with Precision, Recall, and F1; and assess explanation quality using BERTScore [77].

The LLM demonstrated an 82% F1 score in the binary detection task, as determined by an analysis of a corpus of 537 test samples. For the purpose of classification, macro-averaged F1 reaches 61%, with a weighted precision of 72%. BERTScore indicates strong semantic alignment (81%) between model explanations and ground-truth justifications, demonstrating that the LLM not only identifies malicious traces but also provides a correct reasoning for its detection.

This proof of concept makes three contributions.

1. It demonstrates the feasibility of using an off-the-shelf LLM, guided only by prompt engineering, to detect and attribute HTTP-based C2 traffic without handcrafted signatures.

2. It releases a reproducible benchmark set, synthetic Slimper and Mimic traces plus related documentation, that future studies can reuse or extend.

3. It provides an empirical assessment of the approach's strengths and limitations based on detection, classification, and explanation metrics.

The remaining chapters of this thesis are organized as follows: Chapter 2 provides a comprehensive overview of large language models and command-and-control frameworks. As outlined in Chapter 3, the experimental methodology includes the generation of datasets, the design of prompts, and the establishment of evaluation metrics. Chapter 4 reports the results of the detection, classification, and reasoning part. The final chapter provides a detailed discussion of the implications, limitations, and directions for future research in this field.

2 Theoretical Background

This chapter provides the foundational knowledge this master's thesis is based on. It is organized into five main sections.

The first section of this chapter describes the first component of this master's thesis: C2 Frameworks. It will describes their purposes, modern usage and the network architecture of the three-partite framework. The brief two last sub-sections give an overview of real-world examples of C2 Framework and the current detection methods used in Security Operation Center (SOC).

Then with the first component fully understood, the chapter will give a broad historical context about Natural Language Processing (NLP) and how it evolves into modern Large Language Models (LLM), described in the third section. This part of the chapter may seem highly technical, as this master's thesis never intended to develop a LLM from scratch, nor fine-tuning an existing open-source model, such as Llama [68], but it provides an overview of how works a LLM inside the black-box.

Section 2.4 provides the key concept to understand on which bases the prompt engineering relies to: the `context`. Directly followed by the section describing how works prompt engineering and the optimization techniques often used.

The last three sections provide a comparative overview of the most known model currently available, followed by the limitations and challenges coming with the use of a LLM, and finally concluding with a discussion about the contemporary usage of LLM in cyber security.

2.1 Command and Control (C2) Frameworks

A Command-and-Control (C2) frameworks are advanced software platforms that allows an attacker to remotely control and manage compromised systems from a central hub, often automating post-exploitation tasks such as privilege escalation, persistence, and data exfiltration [55].
Less formally, C2 Frameworks are software, usually used by hackers to keep an open backdoor between a compromised device and the C2 server, or red teams to assess vulnerabilities in the security of enterprises.

C2 frameworks have evolved from the IRC-based botnets of the 1990s. Today's C2 platforms are highly sophisticated, often disguising malicious traffic as legitimate network behavior and evading detection. Because these frameworks are frequently updated to stay ahead of defensive measures, organizations must invest substantial resources—often millions of dollars—into security teams tasked with analyzing every new version of the most prominent C2 tools to maintain an effective response posture.

These frameworks are usually composed of three distinct parts: the **implant**, a tiny malware similar to a RAT installed on a comprised, the **server**, in charge of relying communication, commands, and keeping traces of the active implants, and the **operator interface**.

2.1.1 Architecture and Components

The architecture of a C2 frameworks is a three layers distributed software shown in Figure 2.1. The first layer is the interface used by the operator to control the compromised devices. The second layer is the C2 server, in charge of keeping a trace of the connected victims and relay the communication between the interface and them. The last part is a tiny malware called implant. These implants blend their traffic by using common protocol as HTTP(S) [4, 3] or DNS [1, 2]. Modern C2 framework also mimic the expected behavior of a server for these protocol. For example if the implant run over HTTP, then the server will mimic a common HTTP server hosting a website.

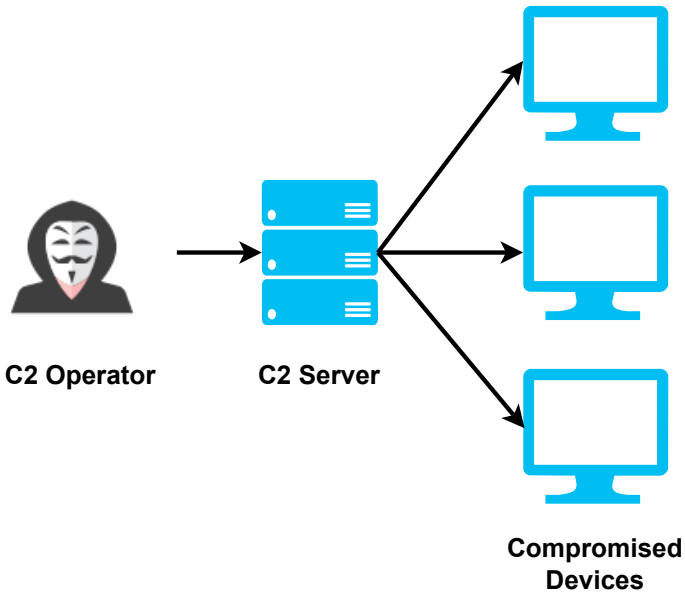


Figure 2.1: Command and Control Framework Network

C2 implants are tiny malware similar to Remote Access Trojan (RAT), signaling frequently its availability using predefined communication protocol and stealth behavior at its generation. These implants are able to execute commands (usually for privilege escalation, or data exfiltration) and blend their communication into regular network traffic, often by mimicking user activities on the web.

C2 Server are the core of a C2 framework. This central component is in charge of the management of the implants fleet, for so they need to keep of trace of available RAT. C2 servers are also communication relay between the implants and the operator interface.

The Operator interface component provides the operator of the C2 with a, usually, command-line terminal. This terminal can issues commands to specific implant(s) through the server. These commands can serve different purposes, such as managing the implants set, remotely control comprised device(s), exfiltrate data, and so on). Some C2 frameworks allows multiple operator interface to connect to the same server endpoint, leading to industrialized hacking group having control of a compromised devices fleet.

2.1.2 Examples of Real-World C2 Frameworks

Command and Control (C2) frameworks are not just hypothetical constructs, they are actively used and evolving software ecosystems. These actors offer modularity, scalability, and obfuscation strategies to a range of operators, including penetration testers and state-sponsored threat actors. This section gives a review of several of the most known real-world C2 frameworks. The focus of this review are: first, to highlight the architectures of these frameworks; and second, to discuss their usage in real campaigns and their relevance to detection efforts.

Sliver is an open-source C2 framework developed by *BishopFox*, designed as a modern alternative to traditional post-exploitation tool-kits like `Metasploit` [58]. It supports encrypted communications via MTLS [5, 15], HTTP(S), and DNS, making its traffic blend into normal communications [12]. `Sliver`'s implants are compiled for various architectures and operating systems, and are capable of command execution, screenshot capture, file transfer, and shell spawning. Sliver has been observed in 2022 in intrusions attributed to criminals and Advanced Persistent Threats (APT) actors. For instance, Microsoft CERT reported its use in campaigns targeting defense and government infrastructure in U.S. [27].

Mythic is another open-source C2 framework that offers a web-based interface, multi-language payload support, and collaborative operation [67]. It was explicitly designed for red team engagements and includes support for *Python*, *Go*, and *C#* agents. `Mythic` allows users to script new functionality and customize operational workflows through a plugin architecture. In 2023, `Recorded Future` reported on a campaign using Mythic to target U.S. education sector entities through compromised remote management services [20].

Cobalt Strike is a commercially licensed adversary simulation tool [69] that has been widely adopted by threat actors due to its stealth features and mature ecosystem. It provides built-in support for key-logging, privilege escalation, and evasion tactics. Unlike Sliver or Mythic, Cobalt Strike's infrastructure is closed-source, making reverse engineering and signature generation more difficult.
Its use is well-documented in a variety of advanced persistent threat (APT) campaigns. It featured heavily in post-exploitation stages of the `SolarWinds` compromise, as well as in attacks attributed to `FIN7` [25] and `Trickbot` [26] hackers group.

The Mirai framework is often miscategorized as a C2 framework due to its utilization of centralized infrastructure for command distribution. Mirai is a botnet malware family. Its architecture includes a C2 server, but its implants (bots) are limited in functionality. They primarily launch DDoS attacks or propagate to new IoT devices. They do not support full-featured command modules or interactive session control [7]. Mirai lacks the operational complexity found in systems like Silver or Mythic. It uses plaintext protocols, hardcoded credentials, and static behaviors. While effective at scale, these characteristics reduce its relevance as a case study for stealthy or evasive frameworks. Understanding Mirai's control mechanisms can provide useful context when differentiating between traditional malware and purpose-built C2 tools.

While powerful and widely adopted, real-world C2 frameworks such as `Cobalt Strike`, `Sliver`, and `Mythic` introduce significant complexity due to their modular designs and obfuscation techniques. Additionally, their use in actual attacks makes them ethically and legally sensitive to deploy for research purposes.
To overcome these challenges and better isolate the capabilities of Large Language Models in a controlled setting, this thesis introduces simplified Command and Control frameworks, developed specifically for experimentation. The subject of Section 3.1 is the description of these mock-ups.

2.1.3 Current C2 Frameworks Detection Methods

Modern security operations rely on three complementary classes of C2-detection techniques: signature engines, anomaly (or machine-learning) models, and behavioral or protocol analyses.

Signature-based intrusion detection are tools such as `Snort` and `Suricata` compare packet payloads or flow metadata against handcrafted rules written by analysts or shared through threat-intelligence feeds [60, 47]. When an attack reuses a known indicator (an HTTP URI, a fingerprint, a command string) signature matching offers near–zero false positives and minimal runtime cost. It is weak against obfuscation (such as minor change in URI encoding, header order, or encryption), renders the rules ineffective.

Anomaly and machine-learning detectors are statistical and ML systems, which build a baseline of *normal* traffic (using features such as byte frequency, packet inter-arrival time, or flow duration) and flag deviations marked as suspicious [6]. Because they do not depend on explicit IoCss, anomaly detectors can uncover previously unseen C2 variants.

However, real networks are noisy: legitimate software updates, Content Delivery Networks (CDN) bursts, or DNS tunnels can look anomalous. Studies on enterprise traces reveals that the false-positive rate exceeds 90 % in unsupervised settings, resulting in an overload of analysts with benign alerts [40]. Maintaining accuracy, therefore, necessitates continuous retraining and feature tuning.

Symbolic execution–based toolchains (e.g., SEMA using angr-powered analyses [39, 65, 11]) have also been proposed to dissect malware and C2 behaviors in a more precise, path-sensitive manner.

Behavioral and protocol analysis are the third approach. It profiles higher-level behaviors (beacon periodicity, command sequences, domain-generation algorithms or inspects protocol semantics). Frameworks such as `Cobalt Strike` counter these heuristics by introducing random jitter, padding payload sizes, and mimicking legitimate HTTP(S) or DNS servers. As encryption becomes widespread, it serves to further obfuscate the content of payloads, forcing detectors to rely on side-channel features that adversaries can manipulate.

Each technique contributes value, yet all suffer from manual upkeep or high noise when adversaries mutate their tooling. These limitations motivate the investigation in Chapter 3: *Can a large language model, guided only by documentation and network traces, detect and classify C2 activity without fine-tuning or continual retraining?*

2.2 Natural Language Processing (NLP)

NLP is the field of computer science and linguistics that focuses on enabling machines to understand, interpret, and generate human language. A fundamental objective of NLP is to capture semantics—the meaning behind words, phrases, and sentences—to support tasks such as translation, summarization, classification, or question answering. Early computational approaches to semantics were based on handcrafted rules, lexicons, and syntactic parsing. While these symbolic methods were precise in specific domains, they lacked scalability and robustness in real-world contexts [42].

With the advent of statistical learning in the early 2000s, NLP shifted toward probabilistic models such as Hidden Markov Models (HMMs) [56] and Conditional Random Fields (CRFs) [32], which offered enhanced flexibility in tasks like part-of-speech tagging and named entity recognition [28]. However, these approaches were

limited by the manual creation of features and their inability to capture richer, context-dependent language phenomena.

The introduction of deep learning marked a major turning point. Feedforward Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs) have demonstrated superior performance in numerous benchmarks by directly learning features from data [33]. CNNs have been particularly effective in extracting local features from word embeddings, rendering them suitable for sentence-level classification tasks [30]. These architectures encountered challenges in handling long-range dependencies and sequential reasoning [8].

This development precipitated the adoption of Recurrent Neural Networks (RNNs), particularly Long Short-Term Memory (LSTM) variants, which exhibited enhanced context retention across temporal domains. However, LSTMs have limitations in modeling complex dependencies over extensive text. The need for a mechanism scalable and based on attention led to the development of the transformer architecture, which now underpins most state-of-the-art NLP models [71]. The subsequent section will elucidate how this architectural innovation enabled the rise of LLMs.

2.3 Large Language Models (LLMs)

Large Language Models. (LLMs) are transformer-based neural networks scaled to billions of parameters and pretrained on massive text dataset. The Transformer architecture, introduced by Vaswani et al. [71], makes this scalability possible by capturing long-range dependencies through self-attention.

As LLM size and training data grow, these models not only improve raw performance but also acquire *emergent abilities* (few-shot reasoning, chain-of-thought prompting) that smaller models cannot replicate.
In our proof of concept, we rely on inherent reasoning of GPT-4o, without any additional tuning, to detect and classify C2 activity from HTTP traces and documentation.

The following subsections first briefly explain Transformer mechanism in the Section 2.3.1, and then outline the three-stage training pipeline: pretraining on unlabeled text, supervised instruction tuning, and alignment via human feedback in the Section 2.3.2. Finally, the Section 2.3.3 describes how emergent behaviors develop as model size increases.

2.3.1 Transformer Architecture

RNNs struggled with long-range dependencies and offered limited parallelism, motivating the search for a more scalable architecture. In 2017, Vaswani et al. introduced the *Transformer* model [71], which now underpins nearly all modern LLMs such as GPT, DeepSeek, and BERT [57, 17].

A Transformer block (Figure 2.2) consists of two sub-layers: a multi-head self-attention mechanism followed by a position-wise feed-forward network. Before entering these sub-layers, raw input text is split into tokens, each mapped to a learned embedding vector. A sinusoidal positional encoding is then added, enabling the model to distinguish token order without relying on recurrence [71]. Later work introduced *relative* positional encoding, which help the model reason about token distances (inside the text) more directly [64].

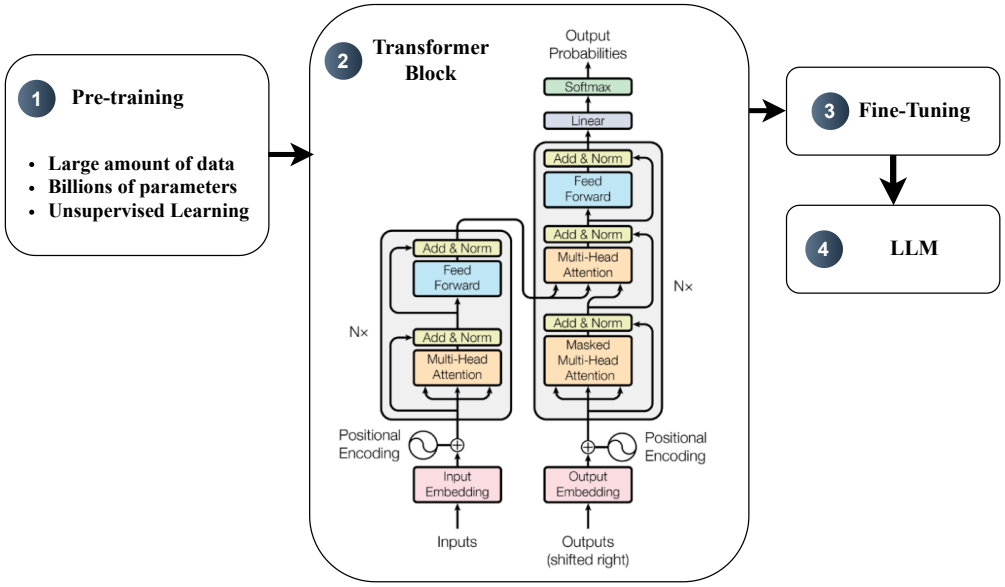


Figure 2.2: Insight of a transformer block inside a LLM training pipeline [71].

Within the multi-head attention sub-layer, each token's embedding is projected into query, key, and value vectors. Multiple attention *heads* compute scaled dot-product scores in parallel, allowing the model to focus on different parts of the sequence simultaneously. The resulting *attention* outputs are concatenated, linearly projected, and combined with the original embeddings via a residual connection, followed by a normalization layer.

Stacking these blocks multiple times enables the model to capture both local and global context dependencies at varying levels of semantical abstraction [71]. Training transforms this static architecture into a powerful reasoning engine. In auto-regressive LLMs (e.g., GPT), each Transformer block is trained to predict the next token given all previous tokens, using vast amounts of text data. In masked-language variants (e.g., BERT), the model learns to reconstruct masked tokens from context. Through this process, Transformers acquire rich statistical knowledge of syntax, semantics, and even factual information [17, 57].

Critically, the Transformer's self-attention mechanism scales to long sequences far more efficiently than RNNs. Whereas RNNs must process one token at a time, Transformers can attend to all tokens in parallel, limited only by the context window size. This property makes them ideal for tasks—such as analyzing HTTP traces and C2 documentation, where long-range interactions (e.g., a command appearing hundreds of tokens after its corresponding beacon) carry significant meaning.

2.3.2 Training Stages: Pretraining, Tuning, and Alignment

Large Language Models (LLMs) acquire their capabilities not just from architecture but from a three-stage training pipeline: pretraining, supervised fine-tuning, and human-feedback alignment (RLHF).

During the *pre-training* phase, the model ingests massive unlabeled dataset (books, websites, code) and learns to predict tokens, either the next token in auto-regressive (e.g., GPT) or masked tokens (e.g., BERT) [57]. This step builds broad linguistic competence and captures statistical patterns across diverse text sources.
Next, *supervised fine-tuning* uses annotated examples to teach the model task-specific behaviors and better align with user expectations. Instruction-tuned variants such as `InstructGPT` outperform vanilla LLMs on downstream tasks (summarization, Q&A, code generation) because they've seen real *prompt-and-response* pairs during training [52].
Finally, *RLHF (Alignment)* refines the model's outputs by training a reward model on human rankings of candidate responses [16]. Reinforcement learning then steers the base model toward replies that are helpful, harmless, and factually accurate.
Although this thesis does not train or fine-tune an LLM from scratch, these three stages, especially instruction tuning and RLHF, explain why off-the-shelf LLMs (like GPT-4o) can follow complex prompts and avoid malicious outputs. In our C2 detection experiments, we rely entirely on these pretrained behaviors to interpret HTTP traces and C2 documentation without additional model updates.

2.3.3 Emergent Abilities

As transformers scale in size, data, and compute, they often develop capabilities not present in smaller models [74, 21]. These *emergent abilities* include few-shot learning, in-context reasoning, and multi-step logical inference. For example, large models can generalize from only a handful of examples provided in a prompt, maintaining coherence over extended inputs and reliably following complex instructions [62, 14].

Although the precise mechanisms remain an active research area, the practical impact is clear: emergent abilities allow LLMs to tackle tasks for which they were never explicitly trained. In our work, we leverage this behavior because the model is not further fine-tuned for Command-and-Control detection. Instead, its ability to interpret context and produce reasoning chains "on the fly" is crucial.

These properties justify using state-of-the-art models such as GPT-4o. As scale increases, so do robustness, adaptability, and cross-domain generalization—qualities that matter in cybersecurity, where inputs can be noisy, ambiguous, or adversarial. While emergent abilities do not ensure flawless performance on every task, they provide a strong foundation for prompt-based methods in domains requiring semantic flexibility and complex reasoning without additional training.

2.4 Context in LLMs

The notion of context is central to the operational logic of LLMs. In the domain of NLP, context is defined as the sequence of tokens (i.e., words, symbols, or characters) that occur before or after a specific input element. This sequence is utilized by the model to infer aspects such as meaning, intention, or structural information present in the input data [42, 28]. In the case of LLMs, the provision of context enables the model to perform coherent and logically consistent reasoning over large sequences of information.

In practical terms, a context window denotes the maximum number of tokens the model can consider during inference in its global context. Transformer-based architectures such as GPT, BERT, or LLaMA process input by mapping each token into a high-dimensional embedding space, then applying self-attention mechanisms across the entire sequence within the context window [71, 36]. This mechanism enables models to reach a global context of the text, which is essential for downstream tasks such as question answering, and classification.

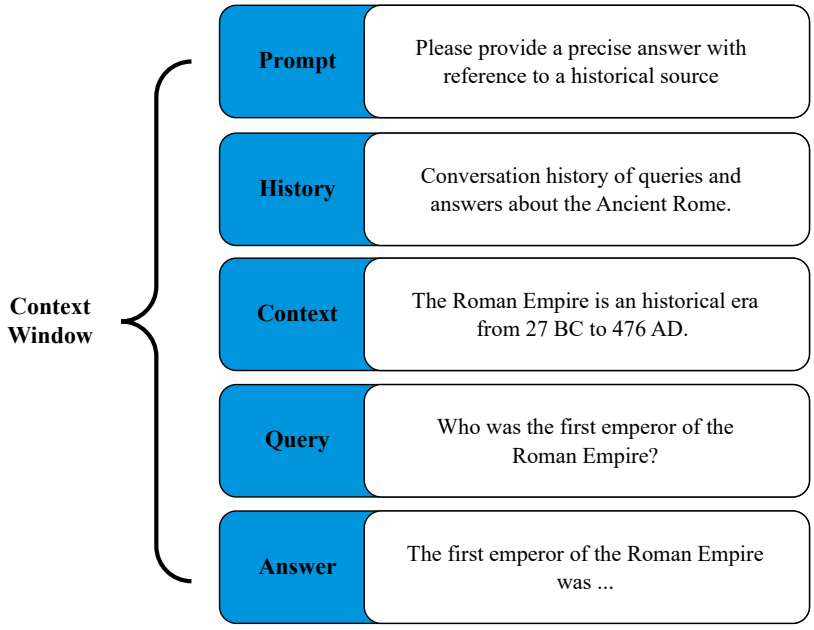


Figure 2.3: Context Window components and query process [41]

Figure 2.3 shows how a LLM's context window fills with static background text, previous turns of conversation, and the current user query. All three are concatenated up to the model's token limit.
Once that limit is reached, older tokens (usually from the conversation history) are dropped so the model always sees the most recent history alongside the original context. This ensures each response is based on both the fixed background and the latest dialogue within the window's capacity.

The size of the context window imposes a practical limitation on the amount of information that can be stored in. The capacity to maintain long-range dependencies enables LLMs to analyze documents, code bases, and multi-turn conversations without losing coherence. This feature has been particularly leveraged in domains such as malware analysis [45], bug detection [34], and medical diagnostics [53].

It is important to note that context enables LLMs to demonstrate in-context learning, whereby the model generalizes from examples presented directly in the input, without requiring gradient updates or fine-tuning [14, 38].
This capability forms the foundation for the prompting strategies examined in this thesis. The context window is populated with documentation, examples,

and instructions, thereby simulating an analyst's knowledge. By optimizing the structuring and delimitation of context, it becomes possible to leverage LLMs to accurately infer behavior over complex sequences, such as HTTP network traces, even without task-specific training.

Prompt engineering builds on this context dependency by structuring the input to guide the model's output. Because a LLM's responses depend entirely on the tokens in its context window, the way we present information (task instructions, illustrative examples, or decision criteria) directly shapes its performance. By embedding these elements within the prompt, we can align the model to specific objectives without changing its underlying parameters. In effect, prompt engineering turns LLMs into flexible tools capable of zero-shot or few-shot learning. The next section shows how these strategies can mimic analyst reasoning for specialized tasks like C2 framework detection.

2.5 Prompt Engineering

Prompt engineering is a widely adopted technique for modulating a pre-trained LLM's behavior by crafting its input. At its core, a prompt is simply a set of instructions inserted into the model's context. Those instructions, combined with the user query, guide the generation process according to established best practices (e.g., OpenAI's prompt engineering guide & the prompting guide [51, 63]). Historically, the notion of "prompting" first appeared in Radford et al. (2018)[57]

Sahoo et al. [61] present a thorough taxonomy of prompt engineering methods. In this thesis, we focus on a subset (specifically, zero-shot, few-shot, and chain-of-thought prompting) since these directly support our C2 detection tasks. Other methods (e.g., self-refinement) lie outside our scope but are described in more detail by Sahoo et al. and in OpenAI's guide [51].

Only techniques used in this master's thesis will be explained in the following subsection. If readers want to learn more about prompt engineering techniques, it is recommended to read the two resources given in the above paragraph. The two resources presented above will serve as a basis to write the following explanation of the techniques used. Specialized papers have been added for readers who need further description of a prompt engineering technique.

2.5.1 Prompt Engineering Techniques

As LLMs are inherently sensitive to the structure and semantics of their input, modifying the formulation of a prompt can significantly affect the outcome, particularly for structured reasoning or classification tasks. In the context of cybersecurity, prompt engineering becomes crucial to elicit consistent and explainable behaviors from the model across varied threat scenarios.

Several techniques have emerged to enhance LLM outputs through prompt manipulation:

Role prompting or setting the model's perspective ("You are a cybersecurity analyst...") guides the tone and scope of the output. This method can anchor the model in a specific knowledge domain or persona, as employed throughout this thesis.

Zero-Shot Prompting presents the task without examples, relying entirely on the model's pretraining to infer the intended behavior. While this is the most concise method, it often struggles with under-specified or domain-specific queries [72].

Few-Shots Prompting introduces one or more exemplars of the task, enabling the model to align better with the expected format and logic. This approach is widely used to reduce hallucinations and improve response structure, especially when the task requires formatting consistency [29, 68].
The example below shows how few shots helps in sentiment classification (prompting guide [63]).

Prompt:

```
This is awesome! // Negative
This is bad! // Positive
Wow that movie was rad! // Positive
What a horrible show! //
```

Output:

```
Negative
```

Chain-of-Thought Prompting (CoT) explicitly instructs the model to reason step by step, often improving logical consistency and multi-step inference . In tasks involving protocol analysis or threat classification, CoT can help ensure decisions are justified in a reasoning manner [75].

Standard Prompting

Model Input:

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11.

Q: The cafetaria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model output:

A: The answer is 27.

Chain-of-Thought Prompting

Model Input:

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. $5 + 6 = 11$. The answer is 11.

Q: The cafetaria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model output:

A: The cafetaria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have $3 + 6 = 9$. The answer is 9.

Long Context Prompting refers to the technique of feeding LLMs large, structured contexts, such as full documentation, logs, or concatenated instructions. This is particularly effective for tasks involving semantic alignment, cross-reference resolution, or multi-source reasoning. The ability to synthesize from hundreds of tokens of domain-specific information enables richer inference but requires careful prompt curation to avoid dilution of relevant signals [29, 10]. This will be the man technique used in the tests, stacking the instructions with the documentation files into the long context prompt.

2.6 Model Comparison: GPT, Llama, DeepSeek

This section compares three LLMs (GPT-4o, Llama 3.3 70B, and DeepSeek-V3) across their core properties, pricing, and benchmark performance. We start with the Section 2.6.1 by reviewing GPT's evolution, then discuss Meta's open-source Llama family in Section 2.6.2, and finally introduce DeepSeek's approach in the Section 2.6.3.
Section 2.6.4 draws on Vipin Vashisth's *DeepSeek-V3 vs GPT-4o vs Llama 3.3 70B* study to justify our choice of GPT-4o for this thesis, highlighting trade-offs in context length, modality support, and cost.

2.6.1 GPT

OpenAI is an AI research and deployment company. They provide GPT, a pre-trained LLM. The GPT model has been trained on millions of parameters. It is designed to answer users' queries by using the context and basic knowledge it has been trained on.

GPT-3 has over 175 billion parameters. GPT-3.5 has almost 200 billion more parameters. It can absorb more complex linguistic subtleties and produce replies that are more logical and contextually accurate because of the larger model size. GPT-4 significantly increased in size thanks to its astonishing 1.5 trillion parameter count. GPT-4's extensive parameter set allowed it to pick up even finer linguistic details, producing more cogent and context-sensitive replies [19].
Year to year, from the public release of GPT-3, OpenAI gains importance in the field of text generation specialized AI. Today it is the world leader in terms of public LLM providers.

Since the most recent and more efficient model of GPT proposed by OpenAI is GPT-4o, this master's thesis will not explore the other models of OpenAI. It is a necessary restriction deduced from the limitation of the API described in the section 2.8.
Until February 2023, no concurrent faced OpenAI. It changed with the release of the Llama 3.1 model by Meta, and more recently DeepSeek.

2.6.2 Llama

Founded in December 2015 by the Meta group, known to own Facebook and WhatsApp, Meta AI is the division in charge of the implementation of artificial intelligence solutions. It is the artificial intelligence laboratory of the Meta Group. In February 2023, the Meta AI division released Llama 3.1. The Llama model is

also used as a feature in Facebook and WhatsApp in some countries. Almost two years later, in December 2024, they released the Llama 3.3 model.

The main advantage of the Llama 3.3 model is its availability to download. As you can self-host the model, its usage becomes free. It would require a large amount of computational power. This amount of power is not accessible to everyone. And it could be a bad idea to run it on a lower configuration.
On the other hand, Meta AI, as OpenAI, proposes an API for developers desiring to customize their Llama assistant. Obviously, this comes with prices and limitations. The request rate limits, for example, set to 2 queries per second, which seems enough for tests, but it comes with a second limitation of 1200 requests (queries and retrievals) per hour. This is annoying when running a large bunch of tests.

Since January 2025, a new competitor entered the market: DeepSeek AI Company. This Chinese version of the models described above is cheaper than its main competitor: OpenAI. Making it a nice alternative for prompt engineers desiring to customize a pre-trained LLM model.

2.6.3 DeepSeek

Founded in July 2023, DeepSeek AI Company is the main competitor of U.S. LLMs. Like its predecessor from Meta AI, DeepSeek is an open-weight LLM. Open-weight AI models are characterized by the release of their pre-trained weights while keeping the training data, algorithms, and detailed architecture proprietary. This method of open software allows less freedom than open-source software. [31]

The most advanced model of DeepSeek is the DeepSeek-R1 model.
In this comparison the model used is the DeepSeek-V3, the predecessor of the R1 model. DeepSeek-R1 has been released in January 2025, which means at this time no comparison paper has been written about the DeepSeek-R1 model. To compare correctly the new DeepSeek model, it should be needed to investigate the changes made by the model upgrade. As the tests made on this paper, for most of them, have been made before the release of DeepSeek-R1.

The Mixture-of-Experts method, employed by DeepSeek, requires splitting the machine-learning model into separate sub-networks. Each separated sub-network acts like a specialized expert. They join their efforts to perform the tasks [9].

2.6.4 Models Comparison

A comprehensive comparison considers not only pricing and token limits but also benchmark accuracy, inference speed, and resource requirements. Vipin Vashisth's analysis on Analytics Vidhya [70] evaluates GPT-4o, Llama 3.3 70B, and DeepSeek-V3 across multiple dimensions:

Benchmark Accuracy Table 2.1 reports each model's performance on standard benchmarks. On `Massive Multitask Language Understanding` (`MMLU`) and `MMLU-Pro`, which test zero-shot and few-shot behaviors over 57+ subjects.
`GPT-4o` achieves 88.7 % and 74.68 % respectively, while `Llama 3.3` scores 88.5 % and 75.9 %, and `DeepSeek-V3` matches `Llama 3.3` on both.
In `Human Evaluation`, which evaluates code generation and problem-solving, `GPT-4o` attains 90.2 %, compared to 88.4 % for `Llama 3.3` and 82.6 % for `DeepSeek-V3`. These results suggest an advantage for `GPT-4o` in language understanding and coding tasks.

Benchmark	Description	GPT-4o	Llama 3.3	DeepSeek
MMLU	Massive Multitask Language Understanding- Test knowledge across 57 subjects including maths, history, law and more.	88.7%	88.5%	88.5%
MMLU-Pro	A more robust MMLU benchmark with more complex reasoning focused questions and reduced prompt sensitivity.	74.68%	75.9%	75.9%
MMMU	Massive Multitask Multimodal Understanding: Text understanding across text, audio,images and videos.	69.1%	N/A	N/A
HumanEval	Evaluates code generation and problem solving capabilities	90.2%	88.4%	82.6%

Table 2.1: Benchmark scores for the LLMs evaluated in this thesis[70]

Inference Speed and Throughput In *Vashisth*'s tests, `GPT-4o` under API access consistently returns responses faster than `Llama 3.3 70B` running on equivalent GPU hardware, despite having a larger parameter count. `DeepSeek-V3`'s `Mixture-of-Experts` (`MoE`) routing introduces additional latency (each input must be dispatched to the appropriate expert sub-networks) which can slow inference compared to a dense model. For our detection pipeline, lower latency and higher throughput make `GPT-4o` preferable.

Memory Footprint and Hardware Requirements

- `Llama 3.3 70B` can be self-hosted but requires over 200 GB of GPU memory (`VRAM`) to load all 70 billion parameters at full precision. This makes deployment on commodity hardware challenging.
- `DeepSeek-V3`, with 671 billion parameters but only 37 billion activated per token (via its `MoE` design), demands complex memory management for expert weights and gating. Practical use typically requires multi-GPU setups with specialized high-bandwidth.
- `GPT-4o` is only available via `OpenAI`'s API, shifting all hardware and maintenance overhead to `OpenAI`'s infrastructure. While it avoids local GPU constraints, the trade-off is a higher per-token cost.

Cost Considerations Table 2.2 summarizes pricing for each model. `GPT-4o` charges $2.50 per million input tokens and $10 per million output tokens, making it roughly 35× more expensive than `DeepSeek-V3` and 25× more expensive than `Llama 3.3` for output. In practice, however, `GPT-4o`'s managed infrastructure and optimized inference often justify its premium price, especially under tight development timelines.

Overall Trade-Offs Despite `GPT-4o`'s higher cost, its superior benchmark accuracy, lower latency, and zero-maintenance API access offer significant advantages for rapid prototyping and reliable performance. `Llama 3.3 70B` provides a cost-effective open-source alternative, at the expense of substantial hardware requirements. `DeepSeek-V3` sits between these extremes: excellent *on-paper* parameter scale via `MoE`, but higher inference latency and more complex deployment. For this master's thesis—where time, reproducibility, and consistent performance are paramount—GPT-4o emerges as the most balanced choice.

	GPT-4o	Llama 3.3 70B	DeepSeek-V3
Open-Source State	Proprietary	Open-Source	Open Weights
Context Window Size	128,000 Tokens	128,000 Tokens	128,000 Tokens
Max Output Tokens	16,400 Tokens	2048 Tokens	8,000 Tokens
Supported Input	Multi Modal	Text	Text
Price /M Input Tokens	$2.5	$0.23	$0.14
Price /M Output Tokens	$10	$0.4	$0.28

Table 2.2: Properties comparison: GPT-4o, Llama 3.3 70B & DeepSeek-V3.[70]

2.7 Use of LLMs in Cybersecurity

LLMs are progressively finding their place within the cybersecurity domain. Their capacity to process natural language, extract meaningful patterns from unstructured text, and generalize across domains renders them particularly well-suited for tasks that traditionally require human expertise.

Recent research has identified several emerging applications. For instance, Li et al. explored the integration of LLMs into static analysis pipelines, showing that LLMs can assist in identifying potential vulnerabilities by interpreting code and commenting patterns [34] . This approach complements to traditional program analysis by incorporating semantic reasoning that would otherwise require manual effort.

In addition to source code analysis, LLMs have also been employed for processing and understanding security documentation, vulnerability reports, and threat intelligence feeds.
This capability facilitates the detection of malware [45] and phishing [79]. Recent work suggests that LLMs can detect meaningful associations between, such as suspicious HTTP patterns or command-line behavior, with established threat indicators. This capability is particularly useful in threat detection and hunting scenarios.

A primary benefit of these models is their capacity to reduce the workload of analysts by automating tasks such as summarization, triage, or hypothesis generation. This redistribution of effort enables human experts to concentrate on more sophisticated counter-measure strategies while relying on the LLM as a reasoning engine [34].

In the context of this thesis, the use of an LLM is motivated by its ability to combine insights from HTTP communication traces with the semantic content of technical documentation. This hybrid approach aims to detect C2 frameworks whose behavior may not be readily apparent in raw traffic but is delineated in accessible documentation. These models offer promising avenues for detecting stealthy threats, especially those relying on legitimate protocols or *living-off-the-land* techniques, as exemplified by campaigns like `Volt Typhoon` [27].

2.8 Limitations and Challenges

While LLMs offer promising capabilities for interpreting data sources like C2 documentation, several limitations constrain their effective deployment in cybersecurity detection workflows.

First, LLMs are highly sensitive to prompt formulation. Minor changes in the structure, tone, or sequencing of a prompt can lead to substantial differences in classification outcomes [61, 63]. For C2 detection, which often hinges on subtle distinctions in behavior, this sensitivity may cause unreliable or unstable predictions unless extensive prompt tuning is performed [52].

Second, LLMs are bound by context window limitations. Current frontier models (e.g., `GPT-4-turbo`) support up to 128k tokens, but token capacity still imposes trade-offs between details and coverage [78, 29]. Long traces or verbose documentation may exceed this limit, forcing truncation and potentially omitting critical IoCss.

Third, there is a lack of native protocol awareness. LLMs process serialized text rather than structured network events, and cannot inherently parse low-level semantics such as TCP flags, packet intervals, or TLS handshake patterns [45, 18]. Their interpretation is limited to the textual representation of the trace, which can obscure important temporal or behavioral signals common in C2 analysis.

Fourth, file format restrictions in current LLM tools (e.g., `OpenAI`'s Assistants API) limit supported inputs to text-based formats such as .txt, .json, or .md [50]. Binary packet captures (.pcap) or protocol-specific files (.pcapng, .etl) must be pre-serialized or transformed into text, adding pre-processing complexity and potential losses.

Fifth, cost and scalability remain critical concerns. Commercial LLM usage incurs non-trivial financial cost, particularly when analyzing large datasets or engaging in prompt-intensive workflows with repeated iterations [48, 49]. This cost

barrier contrasts with the *near-zero* marginal cost of using signature-based engines like `Snort` or `Suricata` [60].

Sixth, output nondeterminism poses challenges for reproducibility. Even at temperature 0, identical queries can sometimes yield slight variations due to architectural randomness or latency-sensitive behaviors in deployment environments [50, 73]. In high-assurance domains like network forensics, this unpredictability can compromise auditability and confidence.

Finally, LLMs lack task-specific memory or long-term coherence unless contextually reinforced. In contrast to rule-based IDS systems, which encode attacker behaviors directly into detection logic, LLMs must infer everything dynamically during inference time [13]. This dependency makes them especially vulnerable to ambiguous or poorly structured documentation, which is often the case for lesser-known or obfuscated C2 frameworks [53].
These limitations must be considered carefully when deploying LLMs in detection pipelines. While they excel at integrating diverse data modalities and interpreting unstructured descriptions, their application in security remains limited by operational, technical, and economic trade-offs.

3 Methodology

The central question guiding this study is **whether a pre-trained Large Language Model, without additional fine-tuning, can effectively detect and classify Command and Control frameworks by analyzing HTTP traffic and associated technical documentation**.
It brings up three sub-research questions:

1. Can a large language model, guided only by natural language documentation, detect the presence of a Command and Control (C2) framework in a network trace?

2. If an infection is detected, can the LLM correctly attribute it to the responsible C2 framework among multiple known possibilities?

3. What are the limitations of this LLM-based detection approach, particularly in terms of trace incompleteness, context window size, and reliance on textual documentation?

To investigate these questions, a practical experimental setup was developed and iteratively refined. The following section outlines the components of this setup, including the system architecture, input formats, and design constraints.
It is important to note, all code, prompt templates, and synthetic data used in this study are released under an open-source license to ensure full reproducibility and allow other researchers to build on our work (Annexe A).
First, we introduce the two simplified C2 frameworks (*Slimper* and *Mimic*) used in this proof of concept. Next, Section 3.2.1 presents our prompt templates and documentation files. With those static inputs explained, Section 3.3 describes the experimental setup, its components, and the evaluation metrics. Finally, the chapter concludes with a brief discussion of why we chose a `Temperature` of 0.4.

3.1 Simplified C2 Frameworks: Slimper and Mimic

To facilitate the process of this proof of concept, the choice to self-develop and to use simplified Command and Control frameworks has been made. Two versions of these simplified C2 frameworks been developed. The simplest one is Slimper. Slimper server is a basic NodeJS HTTP server. It sends commands through the HTTP text-body field and doesn't try to hide more than that.
Slimper's implant is a simple python script which send HTTP request repeatedly to the Slimper's server.
Slimper is able to extract data by using the command *ls*, that obviously list the content working directory. Another available command is *kill*. When the implant receive a *kill order*, it answers to the server and then shut itself down.

The second simplified C2 framework and the most interesting one, is Mimic. Mimic server serves a simple flask website, taking the form a blog or a fan made wiki.
This server is able to hide command in a specific field in the HTML page sent in response to the implant request. The implant receiving the webpage search if some of the command field are present in the webpage, if it find one, the command is executed and a POST request, with a URI accepting this HTTP method, is prepared to extract the data. If no command are detected the next request will be a GET request. The selection of the next URI is made using the HTML page received and also a dictionaries loaded in the implant. These dictionaries contains a mapping of the URI and the accepted method of each one. It helps the choice made by the implant. Only the URIs present in the HTML page and accepting the method used by the next request participate in the random selection. This method, simulating an user browsing on the website, clicking on the available link, ensure the stealthiness of the Mimic framework.

Slimper has been made to assess the feasibility of this proof of concept, explaining why it is so simple. If the LLMs had failed the test on Slimper this proof of concept must stated that the method do not work. But since the results of Slimper detection were pretty good, achieving a high F1-score, Mimic has been made to continue the proof of concept and pushing harder the test on the LLM detection capability.

3.2 Prompts Design

3.2.1 Prompt Technique Used

The LLM utilized in this proof of concept was guided through its reasoning process by a handcrafted prompt, ensuring a structured and reproducible trajectory. The prompt is designed to emulate the actions of a cybersecurity analyst and asks the LLM to examine a network trace provided in `JSON` format. A comparison of the trace is warranted against documentation for known C2 frameworks.

The prompt incorporates logic to prioritize strong behavioral fingerprints. For instance, if a given framework exclusively initiates with a GET / request, this particular fingerprint is regarded as definitive, unless contradicted by other evidence. The LLM is also required to repeat the analysis several times and base its decision on a majority vote. This approach is designed to mitigate randomness and promote stability.
The final prompt used is structured as follows:

Role: The LLM is asked to assume the role of a cybersecurity analyst, specifically an expert in C2 frameworks. This given role ensures that the LLM prior context is filled with the appropriate basic knowledge in both subjects: cybersecurity and C2 frameworks.

Context-Constraints: The LLM is asked to read the documentation files several times before it starts the network trace analyze. Due to the chunk transformation and its overlapping, some part of a documentation file can be in the chunked with the beginning of the next file. The constraint to read several times each documentation file, in our case three times, ensures a certain homogeneity in the LLM context of these essential information. The analyze step is also required multiple times. Despite it delaying the answer, ordering to multiply the analyze gave better results compared to not constrained analyze. It is also constrained to use every network feature (such as IPs addresses, network protocol, ports used, and so on) to achieve the analysis. To build these constraints, different prompts and documentation files have been tested during my internship at `La Defense`. The full reports are in the Annex F and Annex G.

Tasks: The LLM is tasked to mark each provided network trace as `SAFE` before analyzing. After performing its analysis, if the network trace is marked as infected, the classification should be performed several times, and a majority vote should validate the final classification.

Output Format: The output template presented below in the Listing 1, is the only format of answer accepted in the tests. If the LLM generated an answer that doesn't match the output template, this response is directly rejected as it was a hallucination. The example below regroups the two output templates used. To simplify the LLM's task, one template is specialized for safe network traces, and the other one for infected network traces. The LLM is in charge of fulfilling this template using the information gathered during the analyze step.

````
```md
# The network trace seems safe / contains infection IoCs.
1. SAFE / Slimper / Mimic
2. None / {list of detected commands}
3. {Explanation}
```
````

Listing 1: Answer template provided in embedded markdown into the prompt

While the structure of this prompt is effective in establishing useful constraints, the overall effectiveness of the approach is contingent upon the LLM's capacity to align its understanding with the documentation and network trace features. As will be demonstrated in the subsequent sections, this alignment is imperfect, particularly in the context of distinguishing between similar frameworks or addressing noisy or ambiguous traces.

3.2.2 Simplified C2 Frameworks Documentation Files

The efficacy of the classification process is dependent not only on the network trace itself but also on the quality and integration of the Command and Control (C2) documentation.

To facilitate the LLM development towards effective pattern recognition and attribution, a set of handcrafted documentation files was generated for each simplified framework. The subjects in this study were identified as `Slimper` and `Mimic`.

These documentation files serve as the exclusive repository of knowledge from which the `GPT-based` assistant can derive insights into the behaviors, protocols, and anticipated interactions of the C2 frameworks under evaluation.

Each documentation file describes key characteristics, such as the framework's communication method (e.g., HTTP GET/POST), URI patterns, typical commands, and specific fingerprints, like startup behavior or command encoding. This documentation was organized into sections (e.g., *Implant*, *Server* Behavior, *Commands*, and so on) to emulate the kind of documentation frequently encountered in red team tools, in a simpler way. The LLM has not been pre-trained on these artificial frameworks, ensuring that any knowledge it exhibits during classification must stem from the documentation itself.

The `OpenAI`'s `GPT` assistant is provided with the documentation files at the beginning of each session using a vector store architecture. This mechanism guarantees that the assistant has access to both Slimper and Mimic documentation during the inference process. Given that the prompt explicitly requires the assistant to compare network traces to the behavior defined in these documents, this setup serves as the foundation for knowledge alignment between trace evidence and C2 characteristics.

The documentation files for the Slimper and Mimic, which are utilized in this thesis, can be found in Annex B and Annex B.

3.3 Experimental Setup

Figure 3.1 outlines the structure of the testing pipeline, which is composed of three main components: *OpenAI Requests*, *OpenAI Retrieves*, and *Scoring*. Each component contains several sub-modules detailed in the upcoming subsections.

The first component, *OpenAI Requests*, is responsible for preparing and sending prompts to the OpenAI API. This phase begins with the generation of synthetic network traces and their corresponding JSON files, which simulate both infected and safe traffic. This process includes data augmentation and prompt-based thread creation, each elaborated in sections 3.3.1, 3.3.2, and 3.3.3.

The second component, *OpenAI Retrieves*, manages the retrieval of the assistant's responses using thread IDs. Once the answer is fetched, it is split into two main parts: the classification (e.g., labels such as "SAFE" or a framework name like "Slimper", one of the two simplified C2 framework developed in this master's thesis) and the explanatory section, which justifies the classification part. This parsing facilitates a more targeted evaluation. The details are described in section 3.3.4.

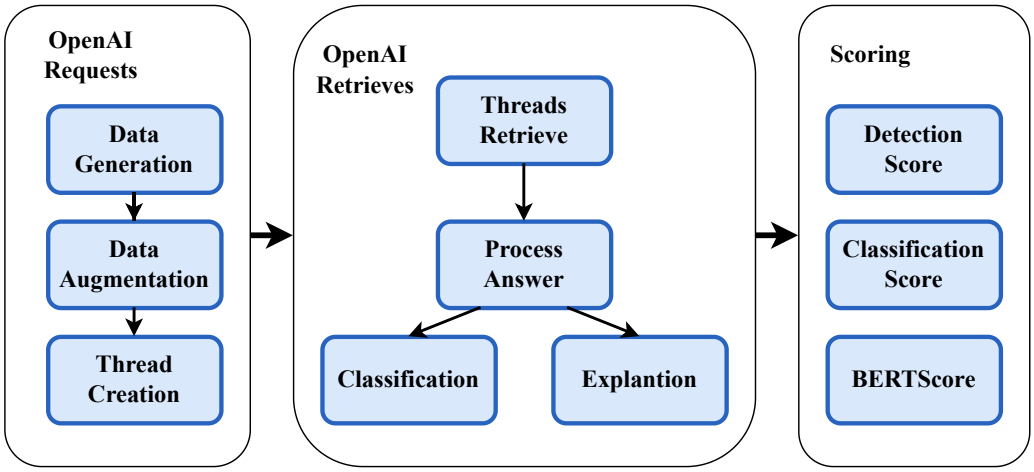


Figure 3.1: Testing Pipeline. Three main components and their sub-components.

The final component, *Scoring*, implements the evaluation of the assistant's performance. This includes three distinct metrics. First, the detection score assesses whether the model correctly identifies traces as infected or safe, irrespective of the attributed framework. For instance, if the LLM classifies an infected trace as "Mimic" when it was actually "Slimper", it still counts as a correct detection.
The second metric is the classification score, which is more stringent. It only considers a response correct if both the detection and framework attribution match the ground truth. This helps isolate the LLM's capacity to distinguish between different C2 frameworks. Both of these metrics are described in the section 3.3.5

Before assessing the assistant's capabilities, a representative dataset of network traces must be constructed. Since existing labeled datasets for Command and Control (C2) traffic are limited or proprietary, this study relies on synthetically generated data. The following section outlines the process used to simulate various network traces in a reproducible and controlled environment.

3.3.1 Data Generation

The data generation is the process of creating data using computer software and algorithms [22]. In the case of this thesis, and due to the experimental virtual machine network setup, the process of data creation was done manually.
The data used are packet capture (PCAP) files, sniffed from the infected devices, using WireShark.

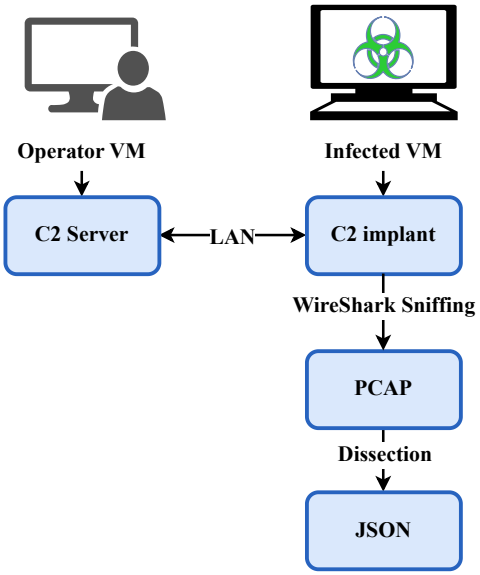


Figure 3.2: Data generation process

These PCAP file can be dissected into different file format. For instance, those tested in the section 3.4 have been dissected into plain text (TXT) and JavaScript Object Notation (JSON) formats. The results shown that the JSON format give tiny better results than the plain text format.

To generate these PCAP an experimental network environment has been setup. The Operator device, which also execute the C2 server, runs over a Kali Linux [46] virtual machine. The potentially infected devices run over a Windows Dev [43]. virtual machine. These machines has been connected through a local network.

To ensure our experimental setup that simulate the main limitation of a real one, there is a need to be aware the C2 implant will not escape the virtual network. To ensure this the virtual machines should not be connected to the world-wide web during the data generation. Instead, the devices have been connected to a local network, ensuring the network encapsulation of the implant. These precautions are mandatory in case of working with a real-world C2 framework. These frameworks often propagate them-self using malware as a bridge between the victims. *See Mirai: not a real C2 Framework but use a C2 server with malware for propagation and launch DDoS.*

This limitation leads to a time consuming process to generate noisy data. The problem when data are non-noisy is the LLM might take in account false correlation between the network trace and the documentation file. Situation which can lead to a false classification. To ensure this problem will not occur during the test process the data augmentation introduces random noise into the file. The following section delves into these details.

Also as the data were generated manually, the amount of data was too low to have sufficient results. To overpass this second problem the data augmentation process create new data introducing slight modifications in the network trace without changing any results or labeling.

3.3.2 Data Augmentation

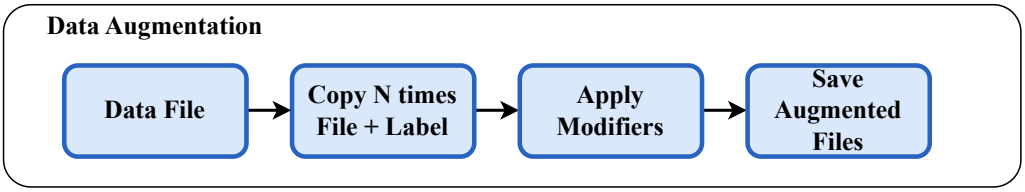


Figure 3.3: Data Augmentation process

Figure 3.3 shows the process of data augmentation. Data augmentation involves creating new versions of a data sample by introducing tiny variations within it. To achieve this goal, a modular pipeline has been designed, allowing developers to introduce new variations by writing their own *modifier*.

The initial step in the process of correct augmentation of files is the preservation of a record of their associated label. Subsequent to the selection of the data sample and its label, the label and the data sample are copied multiple times. At this stage, the labels employed adhere to the format illustrated in Listing 2. The application of a variation to a data sample copy depends on the objective of maintaining the integrity of the original network trace, or creating an augmented variation.

```
{
    "path": "FileName"
    "class" : 0
},
{
    "path": "FileName"
    "class" : 1,
    "C2": "Slimper"
}
```

Listing 2: JSON labels for network trace. Safe example above, infected below.

Modifiers are Python 3 scripts passed as arguments to the data-augmentation pipeline. Each modifier operates directly on the pandas DataFrame extracted from our JSONL files (which include packet labels). For example, the simplest modifier—called the *head trimmer*—selects a random number of packets (between 3 and 20) at the beginning of the trace and removes them. In the experiments reported in Chapter 4, we used three modifiers:

- **Head Trimmer:** Deletes a random 3–20 packets from the start of the trace.
- **Tail Trimmer:** Deletes a random 3–20 packets from the end of the trace.
- **IP Changer:** Extracts all unique IP addresses in the trace, assigns each one a new random IP, and replaces them throughout the copied trace.

The last step of the data augmentation pipeline is a simple save of the augmented files and labels. The trace of the modification applied is kept in the filename.

With the augmented dataset ready for evaluation, the next step involves submitting the prompt-based queries to the OpenAI API in order to assess the model's behavior under different variations of the same input. This process aims to determine whether the model maintains consistent classification across modified versions of the original trace, and to what extent its robustness can be empirically verified.

3.3.3 OpenAI Thread Creation

As explained in the introduction of the Chapter 3, the method used is the Long Context Prompting.
This technique uses the potentially high volume of tokens a LLM can take as context input. The prompt described in the section 3.2.1 is used alongside documentation files described in the section 3.2.2. These files are provided in the sole context to be used to analyze the dissected PCAP file sent with the query. As the final prompt used is exhaustive in terms of explanation given to the LLM, the query, sent alongside the dissected PCAP file, may be as simple as: *Analyze this dissected packet capture file.*

OpenAI API provides an easy interface allowing programmers to automatize queries in two main categories: threads and batches. A `thread` is a specific *conversation* with the AI assistant. This assistant has an empty context before reading its prompt. Using threads allows LLM to generate answers that are not influenced by precedent analysis.
The other category, batches, can be seen as a set of `threads`, sent at the same time to the API together. The main differences between these two categories are batches can process more queries in the same time slot than using single `threads`, and `threads` are built to generate the answer *as fast as possible* where batches can take up to 24 hours to answer queries.

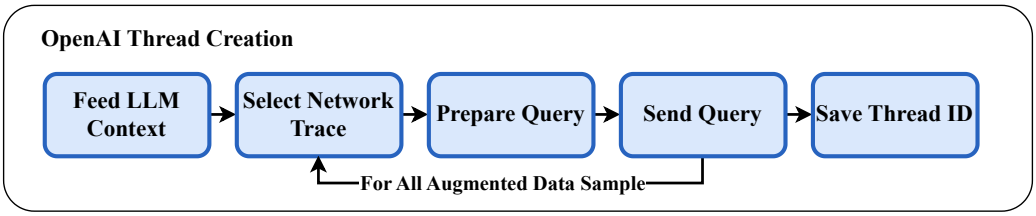


Figure 3.4: OpenAI thread creation sub-components

LLM is fed with the prompt instructions and the documentation files of the C2 frameworks expected to be detected. The label is kept secret from the LLM to avoid influencing the results. All the elements provided to the LLM at this step are pre-fed through the web interface of `OpenAI`'s API. The documentation files are stored in a *file vector* attached to the `GPT-based` assistant, accessible for all `threads` attached to the same assistant.

The Main Loop contains three sub-components. The first one is only in charge of iterating through the augmented dataset, selecting each network trace to send within the `OpenAI`'s query. The second sub-component will create the `thread` and prepare the *Message Object* used in the query. Finally, the query is sent through the `OpenAI`'s API.

Thread ID logs is in charge of formatting and saving the different metadata resulting from the creation of threads. To keep a trace of which thread relies on which label, a field containing the filename of the augmented network trace sent in the query is added in the logs.

3.3.4 OpenAI Thread Retrieval

This component is the simplest in the entire testing pipeline. Its role is to retrieve the assistant's response based on the *thread_id* list generated by the previous component.

A single loop iterates through the list of thread IDs. For each ID, the script queries the OpenAI API to retrieve the raw assistant response, including metadata and the conversation history. The output is then pre-processed to extract the assistant's textual content, which is split into two parts: the classification decision and the explanatory section.
These are then saved, alongside the original label and any applied modifier, in a JSONL file prepared for scoring.

```
{
    thread_id: <thread_id>,
    base_file: <path of the original file>,
    modifier: <none, ip, trim_head, trim_tail>,
    class: <0 for safe, 1 for infected>,
    expected_c2: <Safe, Mimic, Slimper>,
    predicted_c2: <Safe, Mimic, Slimper>,
    explanation: <Explanatory text>
}
```

Listing 3: Processed answer to save in JSONL.

3.3.5 Evaluation Metrics

Precision, Recall and F1-score

Precision, Recall, and F1-score are standard evaluation metrics in the field of machine learning, particularly in the context of binary classification tasks. Their popularity stems from their ability to provide informative feedback on classification quality using only the basic outcomes of a prediction task: true positives, false positives, true negatives, and false negatives.
In this work, the detection task is framed as a binary classification problem, where each input is classified as either infected (positive) or safe (negative). This makes it a well-suited context for the application of these metrics.

True Positive (TP) / True Negative (TN) are correct binary classifications. In our case, it means that the LLM correctly detected a C2 framework infection during the analyze phase for the true positives. Correctly classified network trace as safe, for the true negatives.

False Positive (FP) / False Negative (FN) are errors in binary classification. In our case, `false positive` means a network trace labeled as *safe* has been marked as *infected*. The `false negative` means that a network trace primarily labeled as *infected* has been marked as *safe* in the answer of the LLM.

Precision quantifies the proportion of predicted positive cases that are actually positive. It answers the question: *Of all the samples the model marked as infected, how many truly are?*

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall measures the proportion of actual positive cases that were correctly identified. It focuses on the model's ability to detect all relevant instances:

$$\text{Recall} = \frac{TP}{TP + FN}$$

F1-score is the harmonic mean of Precision and Recall. It provides a balanced measure that takes both false positives and false negatives into account, especially useful in imbalanced datasets:

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

These metrics will be used in Section 4.1 and 4.2 to evaluate the detection and classification ability of the LLM assistant. In case of classification, `Precision`, `Recall`, and `F1-score` cannot be directly used, but *Scikit-Learn* [66] provides an easy way to compute *weighted* and *macro* metrics, that provide support for non-binary classification.

Macro-averaging computes `Precision`, `Recall`, and `F1-score` for each class independently and then takes the unweighted mean across all classes. By treating each class equally, *macro-averaging* ensures that smaller or less frequent classes contribute just as much as larger ones. This is particularly helpful when we want to measure performance on underrepresented C2 families (or "normal" traces) without letting majority classes dominate the results.

Weighted-averaging also computes per-class `Precision`, `Recall`, and `F1-score`, but then takes a weighted mean, where each class's score is multiplied by its support (i.e., the number of true samples in that class) before averaging. As a result, classes with more examples have a greater impact on the final metric. *Weighted-averaging* is useful when overall accuracy should reflect the actual class distribution in the dataset—so if one C2 framework appears far more often, its performance will influence the aggregated score more heavily.

While these metrics are appropriate for evaluating binary and non-binary classification outputs, they are not suited for tasks involving textual comparison. Traditional metrics such as Precision, Recall, and F1-score are not designed to capture semantic similarity or contextual accuracy between two sequences of text. To overcome this limitation, the use of BERTScore, a metric based on contextual embeddings, offers a more meaningful assessment of semantic alignment.

BERTScore

The final metric used in this thesis is BERTScore [77], which is specifically suited to evaluate free-text outputs like the explanations generated by the assistant. While detection and classification scores are sufficient to assess whether the assistant selects the correct label (e.g., SAFE or Slimper), they entirely ignore the reasoning chain presented in the explanation field. Since the explanation is part of the mandatory template and is expected to reflect a correct, logically sound justification, it must also be evaluated.

Traditional metrics like BLEU [54] or ROUGE [35] rely on lexical overlap and n-gram matching, which are poorly adapted to cases where the same idea may be expressed differently. In this project, the reference explanations were written by GPT-4, selected and sometimes edited manually to ensure correctness. However, because they remain in natural language, slight variations in sentence structure, synonym usage, or ordering can occur. BERTScore provides a solution to this.

BERTScore uses contextual embeddings produced by a BERT model [17] to convert each token into a dense vector that captures semantic and syntactic meaning based on the surrounding words. Both the generated and reference texts are passed through the same BERT encoder. For every token in the candidate text, it finds the closest matching token in the reference, based on cosine similarity in the embedding space. This token-wise similarity is aggregated into a final score, representing how well the generated text semantically aligns with the reference.

This vector-based comparison avoids penalizing the assistant for using synonyms or paraphrased expressions, as long as the core message is preserved. In the context of this thesis, this is crucial. The LLM may explain an infection using slightly different words than those found in the reference, even when its analysis is correct. BERTScore enables scoring that is sensitive to meaning rather than surface form.

It is also important to note that no other automated method in the current literature offers this kind of fine-grained semantic alignment evaluation for short explanatory texts. In this regard, BERTScore stands as the sole feasible metric to evaluate the quality of textual reasoning produced by the assistant in a scalable way.

3.4 LLM Parameters

In LLMs, the `Temperature` parameter modulates the randomness of generated outputs. Lower `Temperature` values lead to more deterministic and consistent responses, which is particularly useful for tasks requiring repeatability. Conversely, higher `Temperature`s promote creativity and variability but can result in less reliable or less structured outputs.

To identify an appropriate setting for the experiments, five `Temperature` values were tested: 0.2, 0.3, 0.4, 0.5, and 0.6. These were evaluated using standard classification metrics, Precision, Recall, and F1-score, based on the model's ability to correctly detect infected traces. The results are reported in Table 3.1.

Although the differences between configurations are relatively narrow, cer-

Temperature	Precision	Recall	F1-score
0.2	0.72	0.86	0.78
0.3	0.66	0.98	0.79
0.4	0.68	0.95	0.79
0.5	0.67	0.98	0.80
0.6	0.67	0.92	0.78

Table 3.1: Detection metrics by `Temperature` value.

tain trade-offs are evident. The setting at 0.2 achieved a higher precision but showed a notable drop in recall, making it less suitable for a detection-focused task. `Temperature` 0.3 yielded high recall but with lower precision, suggesting a tendency to over-classify traces as infected.

The 0.5 setting delivered the highest F1-score, yet slightly less consistent results were observed across different runs. `Temperature` 0.4, while marginally behind in F1-score, was ultimately selected due to its favorable balance between precision and recall and its more stable behavior during preliminary tests, made during my internship (See Annex F and Annex G).

Given the cost and time constraints associated with OpenAI's API, this Master's Thesis did not explore finer-grained values or larger repetitions. However, based on the available evidence, `Temperature` 0.4 was considered a reasonable and practical default for the remainder of the evaluation. For this reason, 0.4 was selected as the `Temperature` used for the final tests.

4 Results

This chapter presents the results obtained from 537 tests, each executed within a separate thread to ensure isolation and reproducibility. Among these, 70 hallucinated[1] outputs were identified and excluded from the final evaluation dataset.

The chapter is organized into three core sections. The first section focuses on detection performance. To simplify the analysis, the two infected classes, *Slimper* and *Mimic*, were merged into a single *infected* class. This allowed for binary classification against the safe class in both the predictions and the ground truth.

The second section addresses classification outcomes. Since standard metrics like Precision, Recall, and F1-Score are primarily designed for binary tasks, this section relies on *macro* and *weighted* averages, as provided by Scikit-learn, to evaluate performance across the three classes. These averaging methods are particularly suitable for handling the class imbalance present in this dataset.

The third section explores the quality of the assistant's reasoning through the BERTScore, which measures the semantic alignment between the model's generated explanations and manually selected reference texts. This analysis provides insight into whether correct classifications were also supported by sound justifications.

Finally, the chapter concludes with a synthesis and comparison of the results across all evaluated dimensions.

[1] In this case the exclusion was made if the generated answer didn't match the template format

4.1 Detection Outcomes

Table 4.1 shows the high precision and F1-score, confirming that the LLM is effective at identifying infected network traces when it is confident. The 0.93 precision value indicates that when the model predicts an infection, it is correct most of the time, which is essential to reduce false positives in operational settings.

Precision	**Recall**	**F1-Score**
0.93	0.72	0.81

Table 4.1: Detection Precision, Recall, and F1-score. Merged *Mimic* and *Slimper* into an *infected* class.

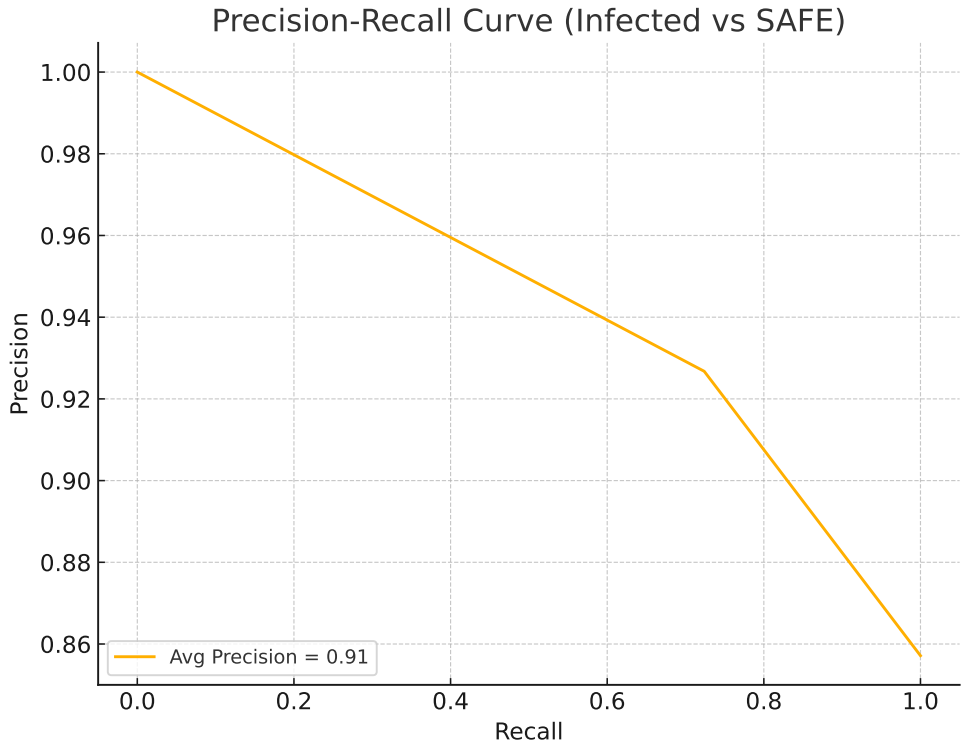


Figure 4.1: Detection Precision-Recall curve. Merged *Mimic* and *Slimper* into an *infected* class.

The F1-score of 0.81 shows a strong trade-off between Precision and Recall, suggesting that the LLM does not sacrifice too much sensitivity for accuracy. Although

the recall is slightly lower (0.72), it still reflects a solid ability to identify most infected traces, considering that no fine-tuning or specific model adaptation was performed. These metrics validate the hypothesis that a well-structured prompt and documentation-aware setup allow LLMs to perform semantic-aware binary classification tasks on network traces.

Figure 4.1 illustrates the losses in Precision when the Recall increases. The precision, staying above 0.8 when Recall is equal to one indicate a strong detection potential. It can be seen as *The model is well calibrated for binary detection of infected network trace*. The figure also doesn't show evidence of overfitting[2], since usually overfitting induces a big drop after a certain threshold between precision and recall, which is not the case here. The drop is limited around 10% of drop acceleration when Recall is over 0.75, which is already a big score for non fine-tuned LLMs.

In security-sensitive contexts such as SOC pipelines, a high precision value translates to fewer false alarms, ensuring analysts' time is focused on genuine threats. This makes the model behavior more operationally viable despite the moderate recall. The recall may be capped in part due to conservative behavior encoded in the prompt, which favors false negatives over false positives to avoid over classification. This behavior, while safe, might limit the model's discovery capabilities in ambiguous cases.

While the model performs strongly in binary detection, its capacity to distinguish between specific frameworks, Slimper versus Mimic, is the focus of the next section.

4.2 Classification Outcomes

The classification task is more complex to score than binary detection, as it involves three possible labels: *SAFE*, *Mimic*, and *Slimper*. In this context, classical Precision, Recall, and F1-score metrics are not sufficient to highlight the model's performance per class. Table 4.2 reports both *Macro* and *Weighted* averages, which are more appropriate for evaluating performance on unbalanced datasets.

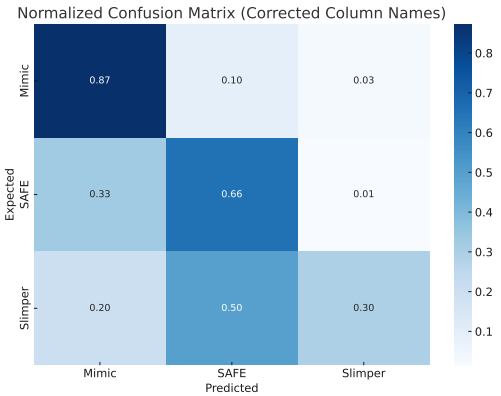
While the detection task yielded strong results, the classification task shows moderate success. The F1-score, arguably the most meaningful metric for classification, hovers around 60%. This result is better than random guessing (approximately 33%), but still insufficient to claim strong classification capabilities.

[2]Not a real overfitting as in AI training, but an overfitting in the score itself giving the unbalanced dataset

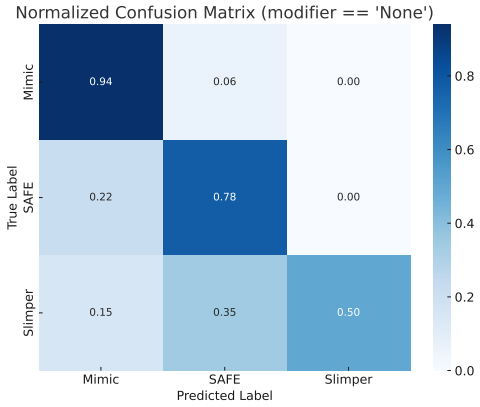
Class	Precision	Recall	F1-Score
Macro AVG	0.64	0.61	0.62
Weighted AVG	0.74	0.62	0.61

Table 4.2: Classification Precision, Recall, and F1-score in *Macro* and *Weighted* modes.

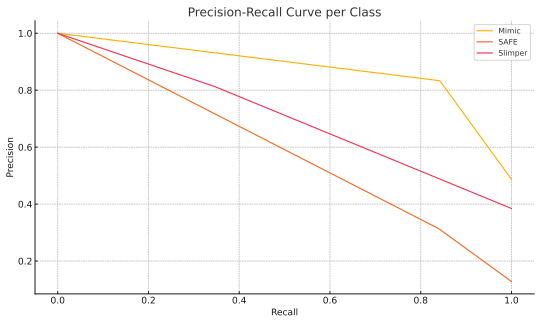
To better understand these scores, a confusion matrix is presented in Figure 4.2a. This matrix shows a high success rate for the *Mimic* class, with 87% correct predictions, and an acceptable 66% for *SAFE*. However, the model struggles with the *Slimper* class, reaching only 30%—lower than a random baseline. This weakness significantly impacts the global classification score.



(a) Confusion matrix of the classification results. Full augmented dataset.



(b) Confusion Matrix of predictions on original network traces only (no augmentation).



(c) Per-class Precision-Recall curve. Each class shows a different balance between recall and precision.

Several factors could explain this misclassifications. One possible cause is the nature of the modifiers used during data augmentation, especially those that trim key portions of the trace. Another possibility is the simplicity of the Slimper framework, which may not provide enough distinctive features for reliable classification. To validate this, Figure 4.2b presents the confusion matrix restricted to unmodified network traces. This subset removes the influence of augmentations. Here, the *Slimper* score increases to 50%, while both *Mimic* and *SAFE* also improve. This suggests that aggressive augmentation techniques like *trim_head* or *trim_tail* may remove critical fingerprints, thus harming classification accuracy.

Additionally, a per-class Precision-Recall (PR) curve is shown in Figure 4.2c. This visualization helps quantify the classification trade-offs and variability across the three classes.

Figure 4.2c displays the Precision-Recall (PR) curves for each of the three classes—*Mimic*, *Slimper*, and *SAFE*. The *Mimic* class shows the most robust and stable curve, maintaining high precision even as recall increases, which confirms its reliable detection observed in the confusion matrix. *SAFE* also demonstrates a good balance between precision and recall, though with slightly more fluctuation. However, the *Slimper* curve highlights the challenges faced in its classification: precision drops significantly as recall increases, suggesting the model often misclassify *Slimper* samples or detects them with uncertainty. This instability reinforces the hypothesis that Slimper lacks distinctive behavioral features, or that data augmentation adversely affected its fingerprinting.

These results highlight that while the LLM shows promising ability in detecting infection, attributing it to a specific Command and Control framework remains more challenging, particularly when semantic cues are degraded by augmentation. A final summary and comparison across the different evaluation axes will be discussed in the following section.

4.3 BERTScore Outcomes

To further evaluate the quality of the LLM's textual explanations, especially under augmentation, the BERTScore metric is used. Unlike traditional classification metrics such as Precision, Recall, and F1-score, BERTScore offers a semantics-aware approach to text evaluation [77]. It leverages contextual embeddings from large language models like BERT [17] or RoBERTa [37] to compare predicted and reference sentences at the token level, capturing deeper semantic alignment rather than surface-level similarity.

Because BERTScore's raw values tend to overestimate similarity, especially in structurally similar texts, the configuration `rescale_with_baseline=True` is applied. This option normalizes scores against baseline mismatched sentence pairs, yielding more conservative and interpretable results [59, 77].

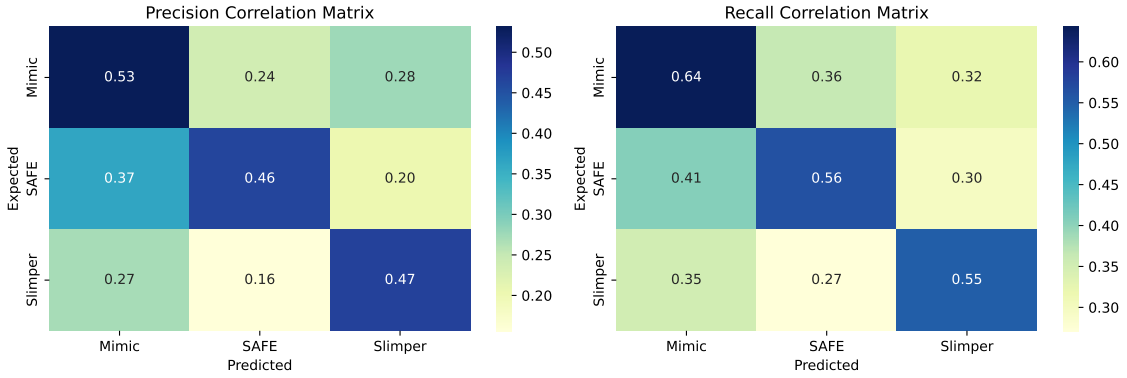
While some scores may appear low at first glance, they remain interpretable under this normalization. For reference:

- Scores below 20% typically indicate no meaningful semantic alignment.
- Scores between 20% and 40% suggest weak or noisy semantic similarity.
- Scores from 40% to 60% reflect partial or moderate alignment.
- Scores from 60% to 80% demonstrate strong alignment without excessive lexical overlap.
- Scores above 80% may indicate 'overfitting' or near-identical language.

This metric is particularly suitable here, as the reference texts were generated by GPT and manually selected. Since no human-authored gold standard was available, BERTScore provides a viable automated approach to assess explanation quality through semantic proximity.

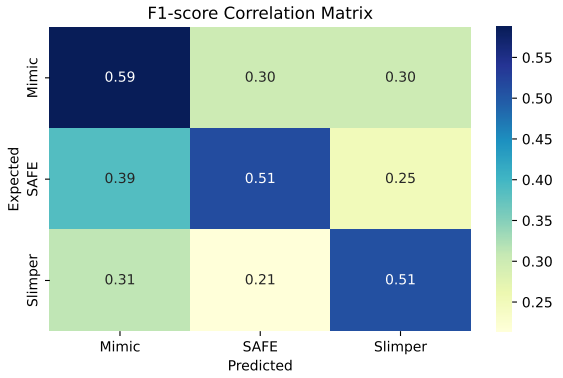
As illustrated in Figure 4.3, semantic alignment exhibits maximum intensity along the diagonal, indicating a precise match between the predicted and expected Command and Control (C2) labels. This finding lends further credence to the notion that accurate classifications are often accompanied by coherent explanations.

These results underscore the correlation between the efficacy of classification and the relevance of explanations. High BERTScore values along the diagonal serve to validate the model's capacity to produce a semantically consistent explanation



(a) Confusion Matrix of the BERTScore's Precision per classes

(b) Confusion Matrix of the BERTScore's Recall per classes



(c) Confusion Matrix of the BERTScore's F1-Score per classes

Figure 4.3: BERTScore correlation matrices across Precision, Recall, and F1 for each expected/predicted C2 pair. Diagonal values represent explanations aligned with the correct classification.

when it makes a correct attribution. By contrast, low off-diagonal values indicate that misclassifications are accompanied by justifications that diverge semantically from the expected rationale. This distinction underscores the value of BERTScore not only as a language similarity metric, but also as a proxy for explanation faithfulness in model interpretability.

It is important to note that explanations related to Slimper receive high scores only when Slimper is correctly identified. Conversely, misclassifications result in lower alignment scores. This indicates that, compared to Mimic, Slimper appears to rely more on explicit, surface-level patterns than on distinctive semantic structures,

which may limit the richness of generated justifications and reduce their alignment when misclassified. This discrepancy may be indicative of a latent model bias that favors distinguishable behavior patterns during attribution.

The evaluation results demonstrate that, while the LLM achieves strong infection detection performance, its ability to precisely classify the underlying C2 framework is more nuanced. This phenomenon is particularly evident in cases where semantic patterns are compromised or become ambiguous. However, the explanation analysis, as quantified by BERTScore, supports the hypothesis that correct predictions are often accompanied by coherent reasoning. These findings affirm that while classification fidelity varies, the semantic quality of justifications, when correct classification, is high, reinforcing the potential of LLMs in interpretable C2 detection systems. The subsequent chapter thus concludes this Master's Thesis and discusses future directions to improve robustness, attribution, and interpretation in AI-driven network analysis.

5 Conclusion and Future Work

This Master's Thesis explored the use of Large Language Models, specifically OpenAI's GPT-4o, to analyze network traces and detect infections caused by Command and Control (C2) frameworks. Through a structured evaluation pipeline, combining handcrafted prompts, documentation-based guidance, and semantic scoring methods, this proof of concept demonstrated that LLMs can offer meaningful insights when applied to network traces. While the approach shows promise, particularly in detecting infections and generating relevant explanations, it also reveals limitations in precise classification and consistency, especially under realistic conditions.

The following sections will summarize key contributions, evaluate their relevance compared to more traditional systems, and outline future research directions that could strengthen the capabilities of LLM-based detection systems.

5.1 Discussion

Summary of the findings This study evaluated whether an *off-the-shelf* LLM (`GPT-4o`), guided solely by prompt engineering and framework documentation, can detect and attribute HTTP-based C2 activity without any model fine-tuning. The key findings are as follows:

(1) Prompt and Temperature Tuning. Preliminary experiments showed that a Temperature of 0.4 offered the best balance between Precision and Recall, yielding stable binary detection performance without excessive randomness. Lower temperatures (e.g., 0.2) increased Precision at the cost of Recall, while higher values (e.g., 0.5) gave marginally higher F1 but less consistency.

(2) Binary Detection Performance. By merging Slimper and Mimic into a single "infected" class, the LLM achieved a Precision of 0.93, Recall of 0.72, and an F1-score of 0.81 on a test set of 537 traces. These results demonstrate that, even without task-specific training, the model can reliably flag most infected traces while maintaining a low false-positive rate. The Precision–Recall curve confirmed that precision remains above 0.80 even at high recall values, indicating strong calibration for the binary detection task.

(3) Multi-Class Classification Results. When distinguishing among SAFE, Slimper, and Mimic, performance declined. The macro-averaged F1-score was 0.62, with a weighted F1-score of 0.61 and a weighted Precision of 0.74. The confusion matrix revealed that Mimic was correctly identified 87 % of the time, SAFE 66 %, but Slimper only 30 %. This disparity suggests that `Slimper`'s simpler behavior, combined with aggressive data augmentation (e.g., `trim_head`), removed critical fingerprints, making `Slimper` less distinguishable.

(4) Explanation Quality (BERTScore). Using BERTScore to compare the LLM's generated justifications against reference explanations, we observed a strong diagonal in the correlation matrices: when the model's classification was correct, its rationale aligned semantically with the expected explanation at an average of 0.81. Off-diagonal values were significantly lower, indicating that correct labels were paired with coherent reasoning. This confirms that, for accurately classified samples, the LLM not only detects infection but also produces valid explanations.

(5) Impact of Data Augmentation. Aggressive augmentation techniques that removed or shortened key portions of the trace (e.g., `trim_head`, `trim_tail`) degraded classification accuracy, particularly for Slimper. When restricting evaluation to unmodified traces, Slimper's accuracy rose from 30% to 50%, and overall classification improved for all classes. This highlights that preserving behavioral fingerprints in the prompt context is critical.

In summary, this proof of concept confirms that `GPT-4o`, used with long-context prompts that embed documentation plus `JSON`-formatted traces, can detect `HTTP`-based C2 infections with an F1-score above 0.80. Classification among specific frameworks is feasible but more error-prone (especially for Slimper under heavy augmentation) yielding a macro F1 around 0.62. Importantly, high BERTScore values ($\tilde{0}.81$) for correct attributions underscore that the LLM's reasoning is semantically sound, validating its potential for explainable C2 detection.

Comparison with Existing Methods Although this proof-of-concept dataset is modest (537 HTTP traces across two toy frameworks) and entirely synthetic, the preliminary results allow us to compare the LLM pipeline to the three classical C2-detection pillars with some caution. Signature engines, such as Snort and Suricata, are renowned for their precision. Studies report "near-zero false positives" when the indicator is current. However, they also require constant rule maintenance. Microsoft notes a median 24-day lag between a rule's publication and deployment in enterprise networks. During this time, new beacons can slip through. Unsupervised anomaly detectors bridge that gap, but they cause analyst fatigue because their default configurations can generate more than 90 % false positives on real enterprise traffic.

In that light, the LLM achieved a precision of 0.93 and an F1 score of 0.81 for binary infection detection, which is already in the same low-noise range as signature rules, yet it was obtained without handcrafted indicators. Additionally, it achieved a macro-F1 score of 0.62 for attributing the trace to Slimper or Mimic, a task that most current engines do not even attempt. These results are far from conclusive. They come from a single model, one protocol, and no fine-tuning. However, they hint at a middle ground where precision approaches that of rule-based systems, coverage exceeds that of static signatures, and maintenance is limited to prompt updates rather than rule engineering or model retraining.

Limitations Several limitations of this work should be acknowledged. First, our evaluation used highly controlled, synthetic data: the `Mimic` and `Slimper` frameworks were simplified *proof-of-concept* C2 tools without encryption, polymorphism, or advanced evasion techniques. As a result, our findings may overstate performance on *real-world* malware, which often exhibits more variability and noise. We did not test on actual network captures or *high-noise* environments, so generalization is limited. Second, we used an *off-the-shelf* LLM (e.g. `GPT-4o`) in a zero-shot fashion without any security-specific fine-tuning. As others have noted, general LLMs trained on broad web data have low proficiency with domain jargon and can hallucinate . This choice simplified our setup but means the model's knowledge of C2 concepts likely comes from generic textual similarities rather than deep understanding. Third, the scope of frameworks tested was narrow (two prototypes). We cannot be sure the same prompt-based method scales to many frameworks or to more subtle distinctions. Finally, our evaluation lacked extensive quantitative metrics. We observed correct identifications qualitatively, but did not systematically measure accuracy, recall, etc., which leaves open the question of statistical reliability. Each of these factors suggests caution in interpreting the results as an endorsement of LLMs for all C2 detection.

Practical Implications From the perspective of practitioners in the field, the present study underscores novel prospects and limitations concerning the implementation of LLMs in the field of cybersecurity. In practice, an LLM-based detector could be used as an interactive analyst tool. To illustrate, a security operator could input captured log excerpts or example commands into the model and inquire about its resemblance to a specific framework. In response, the LLM would generate a narrative explanation, as observed in our experimental studies, that could expedite threat triage. This could be especially useful for novel or custom C2 tools lacking existing signatures. The significance of expeditious design became evident as well: we ascertained that meticulously articulated queries yielded more precise detections, thereby emphasizing that prompt engineering itself constitutes a pivotal skill for cybersecurity utilization. In addition, extant literature indicates that diminutive, optimized LLMs (employing techniques such as PEFT [76] or LoRA [24]) can attain commendable accuracy with negligible expense. In practice, this suggests that organizations may opt to deploy lightweight, potentially open-source LLMs locally, rather than relying exclusively on cloud services.

Concurrently, our findings underscore the necessity of exercising discernment when interpreting LLM outputs. It has been previously established that Large Language Models are susceptible to hallucinations and manipulation, which has led to concerns regarding their reliability. A miss-classification (the erroneous categorization of benign activity as C2 or vice versa) could result in the inefficient allocation of resources or the compromise of security measures. Consequently, the implementation of an LLM should serve as a complementary enhancement to existing detection systems, rather than a substitute.
From a societal and ethical standpoint, the implementation of automated LLM analysis has the potential to enhance the efficiency of defenders. However, it concomitantly introduces novel challenges. In the event that attackers are able to acquire knowledge of the LLM heuristics, there is a possibility that they will formulate communications designed to deceive the model. This phenomenon is analogous to the concept of adversarial attacks in the context of machine learning. Moreover, reliance on black-box AI introduces transparency issues and potential bias in threat attribution. These factors must be taken into consideration when evaluating the real-world adoption of LLM-based tools.

5.2 Future Works

This proof-of-concept is not yet mature enough for production deployment. Its main limitation is a deliberate focus on HTTP traffic: real-world C2 frameworks routinely tunnel over other encrypted protocols or even bespoke channels, and they can disguise themselves by mimicking local traffic patterns.

The choice to concentrate on HTTP stemmed from the time constraints of this thesis. That narrow scope, however, leaves ample room for future work to extend the assistant to a wider range of protocols and communication schemes.

Encrypted traffic analysis also stands outside the present study. When complete packet captures and session keys are available, though, tools such as Wireshark can decrypt the flows and make them accessible to the pipeline.

If further experiments confirm the promise of large language models in this setting, an obvious next step would be to fine-tune a model dedicated to network-trace analysis—one capable not only of detecting compromise but of attributing it to specific C2 frameworks even in the presence of obfuscation or noise.

Embedding such an LLM module in a SOC workflow could lighten analyst workload, accelerate triage, and automatically surface new signatures and behavioral indicators. Taken together, these prospects outline a research path toward more automated, resilient, and intelligent defenses built on LLMs.

Bibliography

[1] Domain Names—Concepts and Facilities. RFC 1034, 1987.

[2] Domain Names—Implementation and Specification. RFC 1035, 1987.

[3] HTTP Over TLS. RFC 2818, 2000.

[4] Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, 2014.

[5] The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, 2018.

[6] Mahmood Ahmed, Azween Mahmood, and Jiankun Hu. A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60:19–31, 2016.

[7] Manos Antonakakis, Tim April, Michael Bailey, Matthew Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chad Lever, Phillip Porras, Stefan Savage, Vern Paxson, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis`, 2017. [Online; accessed 25 May 2025].

[8] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[9] Dave Bergmann. What is mixture of experts? https://www.ibm.com/think/topics/mixture-of-experts. [Online; accessed 21 February 2025].

[10] Leonardo Berti, Flavio Giorgi, and Gjergji Kasneci. Emergent abilities in large language models: A survey, 2025.

[11] Charles-Henry Bertrand Van Ouytsel, Christophe Crochet, Khanh Huu The Dam, and Axel Legay. Tool paper-sema: symbolic execution toolchain for malware analysis. In *International Conference on Risks and Security of Internet and Systems*, pages 62–68. Springer, 2022.

[12] BishopFox. Sliver c2 framework. `https://sliver.sh`. [Online; accessed 23 Decembre 2024].

[13] Rishi Bommasani, Jack Hudson, Ehsan Adeli, and et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[14] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[15] Brian Campbell, John Bradley, Nat Sakimura, and Torsten Lodderstedt. OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens. RFC 8705, 2020.

[16] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[18] Daniel A Eisenberg, David L Alderson, Maksim Kitsak, Alexander Ganin, and Igor Linkov. Network foundation for command and control (c2) systems: literature review. *IEEE Access*, 6:68782–68794, 2018.

[19] Chude Emmanuel. Gpt-3.5 and gpt-4 comparison. `https://medium.com/@chudeemmanuel3/gpt-3-5-and-gpt-4-comparison-47d837de2226`. [Online; accessed 18 February 2025].

[20] Insikt Group (Recorded Future). Threat analysis: Abuse of mythic c2 in u.s. education sector. `https://www.recordedfuture.com/`

mythic-c2-abused-in-targeted-education-sector-attacks, 2023. [Online; accessed 25 May 2025].

[21] Deep Ganguli, Ankesh Bais, Samuel Sennhauser, Shun Xie, Joshua Achiam, Rebecca Roelofs, Fleur Hou, Pedro Castro, Feng Li, Zhihao Xu, et al. Predictability and surprise in large generative models. *arXiv preprint arXiv:2202.07152*, 2022.

[22] Mahmoud Ghorbel. The concept of data generation. https://www.marktechpost.com/2023/02/27/the-concept-of-data-generation/, 2023.

[23] Claudio Guarnieri and Jurriaan Pels. Cuckoo sandbox: Automated malware analysis. In *Black Hat USA*, Las Vegas, NV, 2012.

[24] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[25] Mandiant Threat Intelligence. Fin7: Revisited. https://www.mandiant.com/resources/blog/fin7-revisited, 2022. [Online; accessed 25 May 2025].

[26] Microsoft Security Intelligence. Trickbot disruption: Working with partners to take action. https://www.microsoft.com/en-us/security/blog/2020/10/12/trickbot-disruption/, 2020. [Online; accessed 25 May 2025].

[27] Microsoft Threat Intelligence. Volt Typhoon targets US critical infrastructure with living-off-the-land techniques. https://www.microsoft.com/en-us/security/security-insider/emerging-threats/volt-typhoon-targets-us-critical-infrastructure-with-living-off-the-land-techn
2023. [Online; Accessed 20 May 2024].

[28] Daniel Jurafsky and James H Martin. Speech and language processing. *Prentice Hall*, 2014. 3rd edition (draft), Chapter 1-3.

[29] Jared Kaplan, Sam McCandlish, Tom Henighan, et al. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[30] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, 2014.

[31] Aruna Kolluru. Exploring the world of open source and open weights ai. https://medium.com/@aruna.kolluru/exploring-the-world-of-open-source-and-open-weights-ai-aa09707b69fc. [Online; accessed 21 February 2025].

[32] John Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML)*, pages 282–289. Morgan Kaufmann, 2001.

[33] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[34] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):474–499, 2024.

[35] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. *ACL Workshop*, 2004.

[36] Zihan Lin, Xu Tan, and et al. A survey of transformers. *arXiv preprint arXiv:2106.04554*, 2021.

[37] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[38] Sheng Lu, Irina Bigoulaeva, Rachneet Sachdeva, Harish Tayyar Madabushi, and Iryna Gurevych. Are emergent abilities in large language models just in-context learning?, 2024.

[39] Serena Lucca, Christophe Crochet, Charles-Henry Bertrand Van Ouytsel, and Axel Legay. On exploiting symbolic execution to improve the analysis of rat samples with angr. In *International Symposium on Foundations and Practice of Security*, pages 339–354. Springer, 2023.

[40] Amjad Mahmood, Muhammad Faisal, and A. Roshandel. Reducing false positives in anomaly-based intrusion detection: A survey and taxonomy. *IEEE Access*, 9:97134–97158, 2021.

[41] Aleksei Makin. Ontology-driven knowledge management systems enhanced by large language models. 09 2024.

[42] Christopher D Manning and Hinrich Schütze. Foundations of statistical natural language processing. *MIT Press*, 1999.

[43] Microsoft Corporation. Windows dev environment virtual machines. `https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/`, 2025. [Online; accessed 29 May 2025].

[44] Microsoft Threat Intelligence. Microsoft digital defense report 2023. https://www.microsoft.com/content/dam/microsoft/final/en-us/microsoft-brand/documents/MDDR_FINAL_2023_1004.pdf, 2023. See section "Operationalizing intelligence," which cites a median 24-day delay between rule publication and enterprise deployment. Accessed 31 May 2025.

[45] Achraf Moussaileb, Abdullah Alshamrani, and Aziz Mohaisen. Malware detection using large language models: Opportunities and challenges. *arXiv preprint arXiv:2401.00276*, 2024.

[46] Offensive Security. Kali linux. https://www.kali.org, 2025. [Online; accessed 29 May 2025].

[47] Open Information Security Foundation. Suricata IDS/IPS: Open-source network threat detection engine. https://suricata.io/, 2025. Accessed 31 May 2025.

[48] OpenAI. Openai api - pricing. https://openai.com/api/pricing/. [Online; accessed 18 February 2025].

[49] OpenAI. Openai api - rate limits. https://platform.openai.com/docs/guides/rate-limits. [Online; accessed 18 February 2025].

[50] OpenAI. OpenAI. https://openai.com, 2024. [Online; accessed 20 Decembre 2024].

[51] OpenAI, 2025. [Online; accessed 01 June 2025].

[52] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, et al. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022.

[53] Dimitrios P Panagoulias, Maria Virvou, and George A Tsihrintzis. Evaluating llm–generated multimodal diagnosis from medical images and symptom analysis. *arXiv preprint arXiv:2402.01730*, 2024.

[54] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[55] Hunt.io : Thead Hunting Platform, 2024. [Online; accessed 01 June 2025].

[56] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[57] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *OpenAI Blog*, 1(8), 2018. `https://openai.com/research/language-unsupervised`.

[58] Rapid7. Metasploit framework. `https://www.metasploit.com`, 2025. [Online; accessed 25 May 2025].

[59] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[60] Martin Roesch. Snort - lightweight intrusion detection for networks. *Proceedings of the 13th USENIX conference on System administration*, pages 229–238, 1999.

[61] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927*, 2024.

[62] Victor Sanh, Stephen Webson, Aakanksha Chowdhery, Alec Radford, Akhil Shakir, and Others. Multitask prompted training enables zero-shot task generalization. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 10467–10481, 2022.

[63] Elvis Saravia and contributors. Prompt engineering guide. `https://www.promptingguide.ai`. [Online; accessed 18 June 2024].

[64] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *NAACL-HLT*, 2018.

[65] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[66] The scikit-learn developers. scikit-learn: Machine Learning in Python, 2024.

[67] Cody Thomas. Mythic: An open source c2 framework. `https://github.com/its-a-feature/Mythic`, 2025. [Online; accessed 25 May 2025].

[68] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[69] Palo Alto Unit42. Cobalt strike in the wild: How threat actors weaponize it. `https://unit42.paloaltonetworks.com/cobalt-strike/`. [Online; accessed 25 May 2025].

[70] Vipin Vashisth. Deepseek-v3 vs gpt-4o vs llama 3.3 70b – let the best ai model win. https://www.analyticsvidhya.com/blog/2025/01/deepseek-v3-vs-gpt-4o-vs-llama-3-3-70b/. [Online; accessed 21 February 2025].

[71] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[72] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned language models are zero-shot learners, 2022.

[73] Jason Wei et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

[74] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Dale Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

[75] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[76] Lingling Xu, Haoran Xie, Si-Zhao Joe Qin, Xiaohui Tao, and Fu Lee Wang. Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment, 2023.

[77] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. In *International Conference on Learning Representations (ICLR)*, 2020.

[78] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[79] Yisen Zhu, Yang Zhang, Zekun Li, Xinyu Wu, and Rui Zhang. Phishing email detection with large language models: A zero-shot evaluation. *arXiv preprint arXiv:2310.01457*, 2023.

Appendices

A GitHub Links

- Master's thesis open-source code
  `https://github.com/RainMaker1707/MasterThesisEvaluationPipeline`
- Slimper C2 framework open-source code:
  `https://github.com/RainMaker1707/SlimperC2`
- Mimic C2 framework open-source code:
  `https://github.com/RainMaker1707/MimicC2`
- Internship first attempt:
  `https://github.com/RainMaker1707/C2FrameworkDetector`
- Internship Evaluation pipeline:
  `https://github.com/RainMaker1707/EvalProtocol`

Prompt

You are an expert in Cyber Security. Especially efficient in network traffic analysis for any sort of infection. Your role is to analyze a network trace, given in JSON format, to detect a potential infection of a Command and Control Framework documented in your file vector “.

To ensure that you fully understand each documentation file, it is mandatory to read it three times before doing any analysis.

The documentation files provided explain the behavior of Command and Control (C2) frameworks you should be able to detect.

In the analysis phase, you must analyze the network trace multiple times to ensure your classification is correct. Command and Control frameworks are known to blend their network activities in regular traffic by using the same protocols and ports as usual. This technique to stay stealthy must be correctly understood to avoid false positives from legitimate traffic. For example, the well-known C2 Framework, Slimper, blends its traffic using a fake web server. It may go as far as mimicking a real website on the fake server.

You must analyze the IP address pairs, the ports, the HTML content, URI structures, and other network features to determine the infectiousness of the network trace. When identifying the first HTTP request in a trace, ignore TCP handshake packets. The first HTTP request is the first packet that contains an HTTP method and URI (e.g., `GET /`). Do not consider earlier TCP-only packets.

By default, a network trace is considered as **safe**. If, and only if, there is enough IoCs to conclude a potential infection, then the network trace should be labelled as **infected**. If you label it as infected, ensure your classification by double-checking your analysis. You must perform at least three distinct analyses and then conduct a majority vote to determine the correct Command and Control Framework suspected. If you determine it is the C2 framework "X", then always ask yourself *Why not the C2 framework "Y" or the "Z"?*

In the analysis, you must verify whether the URIs strictly conform to the documented URI structure of each C2 framework. If the URIs do not exactly match a framework's documented structure (including required path-file-extension patterns), you must exclude that C2 from suspicion in this analysis step. Continue evaluating the trace against the remaining frameworks. An excluded framework cannot be selected, regardless of the commands observed. If a C2 framework is excluded based on URI mismatch or other inconsistency, you must reanalyze the trace and select the most probable suspect among the remaining frameworks. Do not consider the excluded framework again in this analysis.

A network trace may be partial (e.g., missing the start or end of communication), but strong behavioral fingerprints must still be treated as decisive when present. If the trace starts with a `GET /` request and one of the documented frameworks explicitly states this as its required startup behavior, that framework must be considered the most probable suspect unless it is disqualified elsewhere in the trace. Do not prefer a framework like Slimper if its required URI structure is not matched exactly.

Mimic must not be rejected if its unique starting request and HTML behavior are present. For example, if the first HTTP request in the trace is a `GET /` request and this behavior is documented as unique to a specific framework, that fingerprint must guide your classification, unless contradictory behavior is observed later in the trace.

If a C2 framework is documented to embed commands inside HTML content (such as hidden elements), and the trace contains such HTML structures, this must be matched when considering infection and framework attribution. Conversely, if a framework is documented to respond with raw commands in HTTP body responses, ensure that this behavior is also checked and matched precisely.

If an initial analysis excludes all frameworks, you must revalidate the trace's contents against strong individual fingerprints (e.g., unique starting requests or command delivery mechanisms).

If a unique behavior is present that clearly matches one framework, you must reselect that framework as the most probable suspect.

The trace must not be labeled SAFE if any such behavior is confirmed.

You must follow the response templates when you generate your answer. If you detect a potential infection you must follow this response template:

```
# The network trace contains infection IoCs.
1. <SuspectedC2>
2. <DetectedCommands>
3. <Explanation>
```

To fill this template you must replace: by the appropriate Command and Control framework name detected during the analyze phase. by a list of detected

C2 framework's commands during the analyze phase, None if none was detected. a free explanation of 5 lines maximum describing why this network trace has been classified as infected.

If you detect none of the IoCs derived from the documentation files, you must fill this template:

```
# The network trace seems safe
1. SAFE
2. None
3. <Explanation>
```

You must fill the by the explanation describing why this network trace has been classified as safe.

Your answer must follow this template only. No text outside the template is allowed. Any text that is not matching the template will invalidate the answer.

The network trace to analyze is always linked with the first user query. My first query is:

C Slimper Command and Control framework.

C.1 Introduction

This file contains the documentation about the Command and Control (C2) framework named Slimper.
This framework is private and developed for educational purposes. This is the only documentation existing on this framework.

A C2 framework allows an attacker to communicate with the compromised victim machines after a successful cyber intrusion.

Implants are executables generated by the C2 framework to create a backdoor. The operator of the C2 framework (attacker) needs to drop, install and start the execution of the implant himself.

To handle communication between the C2 server and the implants, Slimper uses the HTTP protocol.

Slimper uses a heartbeat system using HTTP requests. A heartbeat is a periodic signal generated by the implant to indicate its availability to the C2 server.

C.2 Server and implants

In Slimper, the C2 server is an internet-facing HTTP server that receives requests from the implants.
Slimper only uses GET and POST methods to handle its communication.

The C2 server is controlled by an operator (attacker), communicates the commands to the implants and then receives the results of the executed commands.

To send a command to an implant, the C2 server waits for an HTTP request from the implant and then responds with an HTTP response containing the command from the operator of the C2 server in its body.

If no command is sent, the HTTP response body is empty.

The implants send heartbeat HTTP requests to the C2 Server. A heartbeat is a periodical signal generated to indicate the implant's availability.
If the HTTP response contains a command from the C2 server, the implant is in charge to execute it and return the result in a POST request.

The URIs used in these HTTP requests are built as described in the following sections.

C.2.1 File extensions

- **`.js`**: Used for heartbeat. It uses the GET method.
- **“**: No extension is used for carrying data. It uses the POST method.
- **`.png`**: Indicates that the HTTP requests from the implant will stop and the implant will kill its processes. It uses the GET method.

C.2.2 Dictionaries

```
{
   "poll_ext": ".js",
   "poll_files": [
      "bootstrap",
      "bootstrap.min",
      "app",
      "array",
      "backbone",
      "script",
      "tracker",
   ],
   "poll_paths": [
      "assets",
      "scripts",
      "script",
      "javascripts",
      "javascript",
      "jscript",
      "jscripts",
      "embedded"
   ],
   "ext": "",
   "files": [
      "login",
```

```
        "signin",
        "api",
        "index",
        "admin",
        "register",
        "sign-up"
    ],
    "paths": [
        "api",
        "upload",
        "authenticate",
        "oauth",
        "oauth2",
        "database",
        "namespaces"
    ],
    "kill_ext": ".png",
    "kill_files": [
        "favicon",
        "mountain",
        "background",
    ],
    "kill_paths": [
        "static",
        "assets",
        "images",
        "icons",
        "image",
        "pictures",
    ]
}
```

C.2.3 URI builder

The URIs used in the HTTP request made by the implants have the following format:

- `/{path}/{file}{extension}` Where `{path}`, `{file}`and `{extension}` are replaced by words from the dictionaries above.

The extension of the URI defines the path and file list to be used. For example if the operator sends a `kill`command to the implant then the `.png`extension is used

and `{path}`and `{file}`will be respectively replaced by a word from `kill_paths`and `kill_files`: `/{kill_paths}/{kill_files}.png`.
Here are some URIs examples that are possible in the previous configuration:

- `/static/favicon.png`
- `/icons/favicon.png`
- `/picture/background.png`

The URIs are limited by the extension that is used. These formats represent which dictionary should be used depending on the extension:

- `/{poll_paths}/{poll_files}.js` (GET method)
  - Examples:
    - `/assets/bootstrap.min.js` is a correct example
    - `/script/tracker.js` is a correct example
    - `/icons/mountain.png` is an incorrect example
- `/{paths}/{files}` (POST method)
  - Examples:
    - `/api/login` is a correct example
    - `/upload/admin` is a correct example
    - `/assets/app` is an incorrect example
- `/{kill_paths}/{kill_files}.png` (GET method)
  - Examples:
    - `/image/mountain.png` is a correct example
    - `/icons/favicon.png` is a correct example
    - `/api/login.png` is an incorrect example

C.3 Behavior

Once the implant is running on the victim's machine, it starts sending HTTP requests to the C2 server with the GET method using the `.js` extension to mimic a heartbeat.

Upon receiving the first heartbeat, the C2 server knows that an implant is available, so it can start responding to the implant with commands.

The operator of the C2 server can then send command through the C2 server interface and the commands will be added in the next HTTP response.

The implant receiving an HTTP response checks if it contains a command. If the HTTP response contains a command, the implant executes the command and returns the result in an HTTP request using the POST method.

If the `kill` command is sent by the operator, the C2 server sends it in an HTTP response to the implant. The implant answers with an HTTP request with GET method using the `.png` extension. The implant then waits for the HTTP response from the C2 server and eventually kills its processes. The communication between the C2 server and the implant stops from the HTTP response of the `.png` GET request.

C.3.1 Executions examples

Example 1: The operator sends a *kill* command

```
C2 Server ------------------------------  Implant
    |                                        |
    |  <-----------------------------------  | GET /jscript/bootstrap.min.js
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | GET /scripts/array.js
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | GET /javascript/backbone.js
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | GET /embedded/app.js
    |  ----------------------------------->  | HTTP 200 content:"kill"
    |  <-----------------------------------  | GET /images/mountain.png
    |  ----------------------------------->  | HTTP 200 content:empty
                                                |-> the implant kills its proces
```

Example 2: The operator sends the *ls* command, waits for data and then sends the *kill* command

```
C2 Server ------------------------------  Implant
    |                                        |
    |  <-----------------------------------  | GET /assets/bootstrap.js
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | GET /javascript/app.js
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | GET /javascript/backbone.js
    |  ----------------------------------->  | HTTP 200 content:"ls"
```

```
                                                    |-> implant executes ls
    |  <----------------------------------  | POST /api/login <data results>
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /javascript/app.js
    |  ---------------------------------->  | HTTP 200 content:"kill"
    |  <----------------------------------  | GET /images/mountain.png
    |  ---------------------------------->  | HTTP 200 content:empty
                                                    |-> the implant kills its proces
```

Example 3: The operator sends the "ls" and lets the implant run

```
C2 Server ------------------------------  Implant
    |                                         |
    |  <----------------------------------  | GET /assets/bootstrap.js
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /javascript/backbone.js
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /embedded/tracker.js
    |  ---------------------------------->  | HTTP 200 content:"ls"
                                                    |-> implant executes ls
    |  <----------------------------------  | POST /api/login <data results>
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /javascript/app.js
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /javascript/app.js
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /jscripts/array.js
    |  ---------------------------------->  | HTTP 200 content:empty
    ... (continuous heartbeat and response)
```

Example 4: Same as example 3 with inter-requests heartbeat

```
C2 Server ------------------------------  Implant
    |                                         |
    |  <----------------------------------  | GET /assets/bootstrap.js
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /javascript/app.js
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /embedded/backbone.js
    |  ---------------------------------->  | HTTP 200 content:"ls"
                                                    |-> implant executes ls
```

```
|  <----------------------------------  | GET /scripts/app.js
|  ---------------------------------->  | HTTP 200 content:empty
|  <----------------------------------  | POST /api/login <data results>
|  ---------------------------------->  | HTTP 200 content:empty
|  <----------------------------------  | GET /scripts/app.js
|  ---------------------------------->  | HTTP 200 content:empty
|  <----------------------------------  | GET /javascript/bootstrap.js
|  ---------------------------------->  | HTTP 200 content:empty
|  <----------------------------------  | GET /script/tracker.js
|  ---------------------------------->  | HTTP 200 content:empty
... (continuous heartbeat and response)
```

D Mimic C2 Framework

Mimic is an educational Command and Control (C2) framework designed to mimic a real website using HTTP communication between a server and an implant. It is intended for proof-of-concept testing of C2 detection techniques using machine learning models. It is not meant for real-world deployment and lacks encryption or obfuscation.

D.1 Core Behavior Summary

- Mimic implants **always begin communication** with a `GET /` request to the server root.
- This **first GET request to / is unique to Mimic** and is not used by other C2 frameworks documented here.
- All command-and-control traffic is disguised within **normal-looking HTTP web traffic**.
- HTTP **GET and POST URIs are chosen randomly** based on the hyperlinks and forms found in the previous HTML page served.
- Commands are **hidden inside HTML content**, using techniques like hidden `<div>` elements.

D.2 Server Behavior

The Mimic server is a basic Flask application that serves a customizable website. This site is controlled using a configuration file with two fields:

- `get_dictionary`: pages allowed for GET requests.
- `post_dictionary`: pages allowed for POST requests.

By default, the Mimic server generates URIs using path segments resembling those of forum or wiki-style websites.
These include common directory terms such as `forum`, `character`, `profile`, `edit`, `view`, and similar.
This helps the implant blend its traffic with normal-looking site navigation patterns.

The server selects content to serve based on the requested URI and **injects commands into HTML responses** when needed. These commands are placed in hidden elements (e.g., `<div style="display:none">`) within the response body.

If accessed by a normal browser or user, the site behaves like a regular website.

D.2.1 Operator Interface

- The operator can issue commands (e.g., `ls`, `kill`, `screen`, `create <filename>`).
- These commands are **injected into HTML responses** received by implants.
- Commands may be embedded in HTML tags or hidden `<div>`s, never in raw plaintext HTTP responses.

D.3 Implant Behavior

- Upon startup, the implant sends `GET /` to the server.
- It parses the HTML response and randomly selects one of the links (for GET) or form endpoints (for POST).
- It continues sending **GET or POST requests** at random intervals (3s to 15s by default).
- If it detects a command hidden in the HTML, it executes it.
- It then sends results back via a **POST** request to one of the URIs listed in the `post_dictionary`.

This implant behaves like a normal browser user navigating through a site.

D.4 Unique Behavioral Indicators (IoCs)

| Feature | Value | |———————-|———————————| | Initial Request | `GET /` (always, no exceptions) | | URI pattern | Random, no fixed structure | | Commands delivery | Hidden in HTML `<div>`s | | Response format | Always HTML content | | URI source | Parsed from HTML response | | POST behavior | Used for data exfiltration | | Command indicators | Keywords: `ls`, `kill`, `create`, `screen` |

D.5 Traffic Pattern Summary

```
Implant                    Server
  | ---> GET /
  | <--- HTML page (maybe with hidden command)
  | ---> GET /pageX or POST /formY (chosen from HTML)
  | <--- HTML (with or without command)
  ...
  | ---> POST /formZ (with command result)
  | <--- HTML
```

D.6 Important Clarifications

- Mimic **does not** use dictionary-based fixed paths or extensions like `.js` or `.png`.
- It **never responds with plaintext commands** — commands are embedded inside the HTML DOM.
- The use of `GET /` at the beginning is a **strong and unique fingerprint** for this framework.

E

Data Augmentation: Code and Details

E.1 Data Augmentation workflow

E.2 IPs Modifier

E.3 Head and Tail Trimmer

```
        def without_head(data_content, min_drop=3, max_drop=20):
            """
            Removes a random number of packets from the beginning of the trace.
            """
            try:
                data = json.loads(data_content)
                if len(data) <= min_drop:
                    return json.dumps(data, indent=2)
                drop_count = random.randint(min_drop, min(max_drop, len(data) - 1))
                return json.dumps(data[drop_count:], indent=2)
            except Exception as e:
                print(f"Error in without_head: {e}")
                return data_content

```

```
        def without_tail(data_content, min_drop=3, max_drop=20):
            """

```

```
        Removes a random number of packets from the end of the trace.
        """
        try:
            data = json.loads(data_content)
            if len(data) <= min_drop:
                return json.dumps(data, indent=2)
            drop_count = random.randint(min_drop, min(max_drop, len(data) - 1))
            return json.dumps(data[:-drop_count], indent=2)
        except Exception as e:
            print(f"Error in without_tail: {e}")
            return data_content
```

F

Cyber Command Internship Report

Large Language Models for Command & Control Frameworks Detection:

A practical Proof of Concept

Hadrien Allegaert

June 2, 2025

Abstract

The increasing sophistication of **Command and Control (C2) frameworks** presents a significant challenge to cybersecurity, enabling persistent access and control over compromised systems. This study explores the application of **Large Language Models (LLMs)** and **prompt engineering techniques** to detect HTTP anomalies indicative of C2 activities. Through a systematic evaluation of key parameters—including model temperature settings (which influence model randomness), prompt specificity, and documentation details—the research identifies optimal configurations that enhance the accuracy of anomaly detection. Findings reveal that detailed prompts and strategic use of supplementary documentation significantly improve the model's performance in detecting malicious URLs and network patterns. Additionally, comparative analysis of different GPT-4 models highlights the superior accuracy of the standard GPT-4o model over its mini variant. The study concludes by emphasizing the potential of LLMs in augmenting cybersecurity defenses and suggests areas for further research, particularly in the context of structured data handling and prompt optimization.

Contents

List of Figures

List of Tables

Appendices

1 Introduction

The application of Large Language Models (LLMs) in the field of cybersecurity is rapidly expanding, driven by their potential to enhance various aspects of cyber defense, from general cybersecurity tasks [1, 2, 3, 4] to more targeted applications like threat detection [5], including phishing [6], malware [7], and intrusion detection [8]. However, many of these studies remain theoretical and fail to provide the practical tools that cybersecurity engineers need to effectively detect, defend against, and prevent these cyber threats.

Studies, such as those exploring the use of language models for detecting unknown attacks in network traffic, demonstrate the potential of LLMs in identifying covert threats within common protocols like HTTP, FTP, and SMTP[9]. This underscores the growing relevance of LLMs in cybersecurity and sets the stage for this research, which focuses specifically on Command and Control (C2) frameworks.

This research aims to address this gap by providing a practical tool leveraging LLMs specifically for the detection of Command and Control (C2) frameworks. Due to constraints in time and resources, this study focuses on C2 frameworks, which are critical components of many cyberattacks.

C2 frameworks are commonly used by long-term threat actors to establish persistent access, enabling data exfiltration, deployment of additional malware, or further manipulation of compromised systems. Their ability to operate over widely used protocols like HTTP(S), mTLS, or DNS makes them particularly difficult to detect, as their communications can blend seamlessly with legitimate traffic, bypassing traditional security measures.

This research focuses on the detection of C2 frameworks, specifically those using HTTP, as a proof of concept for the practical application of Large Language Models (LLMs) in enhancing cybersecurity defenses.

The strategic choice to focus on C2 frameworks is driven by their frequent use in establishing covert channels between an attacker and compromised devices, often through well-known protocols such as HTTP(S), mTLS, or DNS, as seen in frameworks like Sliver or Mythic C2 frameworks. The use of such protocols, particularly with encryption, poses significant challenges for detection due to their ability to blend with legitimate traffic and evade traditional security measures.

This research specifically focuses on HTTP-based C2 frameworks as a proof of concept for the practical application of LLMs in cyber threats detection. To achieve this, a simplified C2 framework named Slimper (a contraction of ”simple” and ”Sliver”) was developed in Python. Slimper exclusively uses the HTTP protocol for command delivery from the C2 server to implants and for data exfiltration from implants back to the server. This framework serves as the experimental basis for exploring how LLMs can be utilized to generate effective detection rules and enhance defense mechanisms against C2 operations. The Slimper framework is available on GitHub.

This research will demonstrate the viability of using LLMs to detect HTTP-based C2 communication, offering a new avenue for enhancing the detection capabilities of cybersecurity systems against sophisticated and covert threats.

This report is organized into several key sections to provide a comprehensive analysis of the use of Large Language Models (LLMs) in cybersecurity. It begins with a literature review that outlines the current research landscape, focusing on the application of LLMs and prompt engineering techniques in detecting Command and Control (C2) framework activities. This review highlights existing gaps and sets the context for the study's contributions.

Following the literature review, the research questions that drive this study are presented, establishing the specific objectives and areas of inquiry. The development of the Slimper C2 framework is then introduced, detailing its implementation as a proof of concept for detecting HTTP-based C2 activities in network data.

The report then moves into a detailed examination of the evaluation protocol, explaining the methodology and metrics used to test and validate the Slimper framework's effectiveness. The results section follows, where key findings are presented and analyzed to assess the model's performance in anomaly detection.

In the discussion section, the implications of these results are explored in depth, comparing them with existing approaches and highlighting the innovative aspects of the research. Finally, the report concludes with an overview of the broader impact of the study's findings on cybersecurity practices and outlines future directions for enhancing LLM-based threat detection techniques.

2 Literature Review

The rapid evolution of cyber threats has necessitated innovative approaches to cybersecurity, with Large Language Models (LLMs) emerging as promising tools for enhancing detection and response capabilities. While existing literature has extensively explored the application of LLMs in various cybersecurity contexts, including phishing detection, malware analysis, and general threat identification, the practical implementation of LLMs specifically for detecting Command and Control (C2) frameworks remains underexplored.

One of the primary challenges in utilizing LLMs for cybersecurity tasks is the issue of hallucinations—erroneous outputs that arise from static pre-training knowledge. The study of Angus Addlesee addresses this challenge by emphasizing the need for grounding prompts with specific instructions to enhance the accuracy of LLM-driven threat detection systems[10]. This approach is crucial for detecting nuanced and covert C2 communications that could be misclassified by less precise models.

Building on the foundational understanding of LLMs' potential, Yagmur Yigit et al. "[5] provides a comprehensive overview of generative AI applications, highlighting both defensive and offensive use cases. However, despite offering valuable insights, this review remains largely theoretical, underscoring a significant gap: the need for practical, real-world applications of these models, particularly in operational environments where systems must detect and mitigate threats in real-time.

To bridge this gap between theory and practice, Ken Huang et al in "Generative AI Security: Theories and Practices" presents practical strategies for applying prompt engineering within cybersecurity[2]. This chapter specifically addresses how carefully crafted prompts can guide

LLMs to perform security tasks such as anomaly detection and response. By operationalizing these techniques, this research provides a pathway from theoretical concepts to practical applications, directly addressing the gap identified in the broader literature.

Further expanding on practical applications, Hamza Kheddar explores the role of LLMs in cybersecurity with a focus on intrusion detection[7]. This survey outlines various models and methodologies that leverage the strengths of LLMs, such as their ability to understand complex patterns and contextual relationships within data. It reinforces the versatility of LLMs in adapting to different threat landscapes, thus supporting their potential utility in C2 framework detection.

The table 1 shows the key findings of the papers cited above. This table shows that the gap between theoretical and practical use of LLM in cyber security should be explored to enhance the literature actually available.

Table 1: Summary of Key Studies in the Literature Review

Study	**Methodology**	**Key Findings**	**Relevance to This Research**
Grounding LLMs to In-prompt Instructions[10]	Analyzed the impact of specific prompt designs on LLM accuracy	Reduced hallucinations significantly, enhancing LLM response accuracy.	Highlights the importance of precise prompt engineering, crucial for C2 detection.
Generative AI Methods in Cybersecurity[1]	Reviewed applications of generative AI in cybersecurity tasks	Identified gaps between theoretical potential and practical applications in real-world scenarios.	Underscores the need for practical tools, aligning with the development of the Slimper framework.
Prompt Engineering for Cybersecurity[3]	Applied prompt engineering techniques in operational settings	Demonstrated that tailored prompts can improve LLMs' performance in detecting security anomalies.	Provides foundational methods for prompt design that can be adapted for C2 detection.
LLMs in Intrusion Detection Systems[7]	Surveyed LLM and transformer models in intrusion detection	Found LLMs effective for detecting patterns in network traffic but not specifically focused on C2 frameworks.	Supports the adaptation of LLMs for C2 framework detection, a gap this research addresses.

While promising results have been observed in the use of LLMs for network traffic analysis, including the detection of unknown attacks across various protocols[9], the application of these models to detect C2 frameworks specifically is still underexplored. The unique characteristics of C2 frameworks, which often use common communication protocols like HTTP(S) to blend with legitimate traffic, present distinct challenges that require targeted approaches. This research

aims to develop practical, LLM-based solutions that directly address these challenges by leveraging prompt engineering techniques and refining LLM outputs through grounded instructions.

In summary, the existing literature provides a robust foundation of theoretical knowledge and highlights the potential of LLMs in cybersecurity. However, there is a clear need for practical tools that can be directly applied to detect C2 frameworks, which are critical components of sophisticated cyberattacks. This study seeks to fill this gap by developing a proof-of-concept system that enhances the overall efficacy of cybersecurity defenses against evolving threats.

3 Research Questions

The main research question this paper aims to answer is: **How can LLMs be effectively utilized to detect C2 frameworks?** This question is crucial as C2 frameworks often blend with legitimate traffic, making them challenging to detect with traditional methods. Due to the time and money restriction of this study, the need to limit this study to detect the HTTP-based C2 framework communications comes out. The detection of C2 framework using secured communications is left to future works.

To comprehensively address this question, the study explores several sub-questions, including:

- **Which temperature is better for the C2 framework detection?** The temperature of an LLM is a parameters that allows to control the determinism of the model. This parameter is usually between 0.2 and 0.4. Which one is better for this research? How much the temperature influence the result of the LLM?
- **What prompt will behave better to detect C2 framework infection?** To answer this question, three prompt will be used. The first prompt is a non constrained prompt, with small explanation of the job to do by the LLM. These instructions can be found in this GitHub. The next iterations of the prompt will increase the constraint and/or the explanation of the LLM's task.
- **What is the impact of the documentation file?** To achieve its task, the LLM need a documentation file that explain the behavior of the Slimper C2 framework. This documentation can be changed. And the impact of these changes can be evaluated to understand which part of the documentation is mandatory to achieve the detection task.
- **Which data format is better suitable for this task?** This question aims to check if some data format are better in term of final result for an LLMs usage. For example is CSV better than JSON extracted network traffic data? The data format that will be tested in this paper are explained in the Data Simulation section.

4 Large Language Model

Large Language Models (LLMs) are a class of Artificial Intelligence (AI) techniques built on advanced Machine Learning (ML) algorithms. These models are specifically designed to understand, generate, and process human language at a sophisticated level, making them powerful tools for tasks involving Natural Language Processing (NLP). By leveraging vast datasets and billions of parameters, LLMs can generate coherent, contextually relevant responses that closely mimic human language.

LLMs operate by using deep learning architectures, particularly neural networks, to learn patterns in text data. They are trained on diverse linguistic information, ranging from simple sentences to highly complex technical documents, allowing them to develop a nuanced understanding of language semantics, syntax, and context. This training enables LLMs not only to generate text but also to perform a wide array of language-related tasks, such as text summarization, translation, sentiment analysis, question answering, and content generation.

In this research, the LLMs utilized were developed by OpenAI, a leading organization in the field of AI and ML. OpenAI's models, such as GPT-4 (Generative Pre-trained Transformer 4), are at the forefront of LLM technology. These models are pre-trained on massive datasets from various domains, enhancing their ability to provide accurate and context-aware responses across multiple applications, including anomaly detection in cybersecurity, which is the focus of this study.

One of the key advantages of using LLMs lies in their ability to generalize from vast amounts of data. This capability allows them to handle both structured and unstructured data, making them suitable for a wide range of applications beyond just language generation. Their versatility in processing natural language makes them indispensable in fields such as healthcare, finance, software development, and especially in cybersecurity, where they can identify patterns and anomalies in data that might indicate potential threats.

LLMs represent a significant leap forward in AI's capacity to interact with human language, transforming how machines understand and generate text. As these models continue to evolve, their applications in both everyday tasks and specialized fields like cybersecurity will only expand, providing new opportunities to automate processes and enhance decision-making.

4.1 Prompt Engineering

Prompt engineering is the process of designing and refining input prompts that guide the behavior of Large Language Models (LLMs) to generate desired outputs. In essence, it involves crafting instructions, questions, or statements in a way that maximizes the model's ability to understand the task and produce relevant, accurate, and contextually appropriate responses. The quality and specificity of the prompt directly influence the effectiveness of the model's output, making prompt engineering a critical aspect of leveraging LLM capabilities.

The primary purpose of prompt engineering is to bridge the gap between the user's intention and the model's understanding. Since LLMs rely on the information provided in the prompt to generate a response, even slight changes in wording, structure, or context can lead to significantly different results. Effective prompt engineering ensures that the model interprets the input correctly and performs the desired task with high precision.

4.2 Prompt Engineering Techniques

Touvron et al.[11] demonstrated in 2023 that **few-shots examples** largely improves the result of exact match with natural questions. The few shots techniques outperformed the zero shots technique in all model that was trained with more than 50 billions parameters, as such as GPT-3.

Well-structured prompts play a critical role in guiding Large Language Models (LLMs) to focus on relevant details, significantly optimizing their performance in tasks such as text generation, summarization, translation, and anomaly detection. By using precise and thoughtfully designed prompts, the risk of generating hallucinations—responses that are factually incorrect or unrelated to the context—can be minimized, ensuring that the model stays aligned with the intended task.

Furthermore, detailed prompts enhance the model's contextual awareness, allowing it to better understand specific scenarios and generate responses that are closely aligned with the user's expectations and the nuances of the data provided. This strategic use of prompt engineering ultimately leads to more accurate and reliable outcomes across various applications.

4.3 OpenAI

OpenAI is a pioneering artificial intelligence research organization known for its development of advanced language models, including some of the most widely recognized and powerful models in the field, such as GPT-3 and GPT-4. Founded with the mission of ensuring that artificial general intelligence (AGI) benefits all of humanity, OpenAI has focused on creating AI systems that excel in understanding and generating human-like language. The organization's breakthroughs in deep learning and natural language processing have set the standard for Large Language Models (LLMs), influencing the way these models are designed, trained, and deployed across various industries.

OpenAI's work has been instrumental in demonstrating the effectiveness of techniques like few-shot learning, which significantly enhances a model's ability to generate accurate responses with minimal examples. By continually pushing the boundaries of AI research, OpenAI has positioned itself at the forefront of innovation in prompt engineering and the practical application of LLMs, as highlighted in studies like Touvron et al. (2023)[11], which illustrate the impact of few-shot examples on model performance.

The OpenAI API provides developers and researchers with a powerful interface to interact with its advanced language models, such as GPT-3 and GPT-4. This API allows users to send requests to the models, inputting prompts that guide the AI in generating responses based on natural language processing. The API operates through a simple yet flexible format, where users submit text-based instructions to the model, and it returns a coherent and contextually relevant output.

The OpenAI API supports various parameters, like temperature, to control the creativity of the responses, and token limits to manage the length of the generated text. This versatility enables fine-tuning of the AI's behavior to suit a wide range of applications, from conversational AI and content generation to data analysis and anomaly detection. By providing access to these state-of-the-art models, the OpenAI API empowers users to integrate intelligent language understanding and generation capabilities into their systems without needing extensive expertise in machine learning.

The use of the OpenAI API introduces several limitations and challenges that require consideration. The first limitation relates to cost, as the API charges for every input and output token used. A token typically represents around 8 characters of text from the prompt, request, or any attached files, which can quickly accumulate and increase expenses.

Another constraint involves the accepted file formats. While formats like CSV files are supported in the ChatGPT web interface, they are not directly compatible with the OpenAI API, limiting the options for data input through the API.

Lastly, the environment itself poses restrictions. For example, it is not possible to directly install Python modules or other dependencies by simply requesting it through the prompt. Although this can sometimes be circumvented by using pre-packaged modules known as wheels, this approach treats the wheel as an attached file, significantly increasing the token count and, consequently, the overall cost of using the API.

4.4 Applications in Cyber Security

In the context of this study, prompt engineering plays a crucial role in fine-tuning Large Language Models (LLMs) for detecting Command and Control (C2) framework activities in network data. The prompts were specifically crafted to include detailed instructions that guide the LLM to recognize anomalies in HTTP communication patterns that might signal potential C2 activity. By focusing the prompts on specific attributes of C2 frameworks, the model's ability to detect threats becomes more precise, enhancing its overall accuracy in identifying malicious behaviors.

The research showed that the inclusion of comprehensive details within the prompts, such as explicit instructions and relevant examples, resulted in higher precision and improved performance in anomaly detection tasks. Additionally, the strategic use of supplementary documentation on the C2 framework further strengthened the LLM's understanding, enabling it to pinpoint suspicious activities with even greater accuracy. This combination of detailed prompts and relevant documentation proved to be a key factor in maximizing the model's effectiveness in cybersecurity applications.

5 Development of the Slimper C2 Framework

Slimper is a fictitious C2 framework developed specifically to simulate the HTTP communication patterns of real C2 frameworks used by cyber attackers. The primary objective of Slimper is to provide a controlled environment for studying the challenges of detecting covert C2 communications that leverage legitimate HTTP traffic, thereby enhancing the understanding and development of detection methods using LLMs.
By mimicking real-world cyber threats within a manageable and observable framework, Slimper allows researchers and developers to gain a practical understanding of how cyber attackers operate and how their malicious activities can be more effectively identified and thwarted.
Slimper was designed with the intention of replicating the behavior of genuine C2 frameworks while maintaining simplicity and flexibility for educational and experimental purposes. Its core function is to facilitate stealthy communication between a C2 server, operated by an attacker, and implants deployed on compromised victim machines. These implants act as backdoors, allowing persistent access and control over the victim's system.

Usually a C2 framework is organized as follow:

1. **The C2 server:** It ensures the communications between the infected devices, in other words the implants, and the C2 client which is operated by the *hacker*.

2. **The C2 client:** Is in charge to relay communications to the C2 server. These communications handle all the operator's commands. In the other way, from C2 server to C2 client, the C2 client receives the data from the C2 server. The C2 client is also embedded in the C2 server. Operators can join the C2 server by using their own C2 client using regular network operations. In other words several clients can use the same C2 server at the same time.

3. **The implant:** Is in charge to send periodical requests to the C2 server. When it receives a command in a response from the C2 server, the implant is also in charge to run the command and sent the result to the C2 server.

All the communications pass through the C2 server before reaching the C2 client or the implant. Several clients and implants can be connected to the C2 server simultaneously in most of the known C2 framework. The typical structure of a classical C2 (Command and Control) framework, featuring a singular client and implant configuration, is illustrated in Figure 1. This diagram helps to highlight the standard operational setup used by attackers, serving as a baseline for understanding the modifications introduced in the Slimper framework

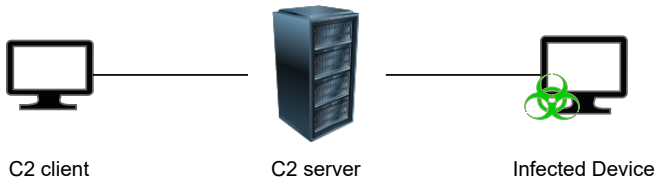


Figure 1: Classical structure of a C2 framework

The Slimper C2 framework specifically uses the HTTP protocol, one of the most common, non encrypted, protocols exploited by attackers due to its ability to blend with regular web traffic, making detection by traditional security measures challenging. By mimicking such characteristics, Slimper serves as a practical tool to evaluate the effectiveness of advanced detection techniques, such as those involving LLMs.
This architectural decision to utilize a unique client is instrumental in ensuring the framework closely replicates real-world scenarios while allowing precise control and observation of simulated attacks within a research environment.
Another difference from the classical C2 frameworks perspective is that the client is unique and can only be managed from the Slimper C2 server.
As the data used always come from the infected device, it doesn't change anything to use a specialized C2 client. This choice change a bit the structure of the C2 framework, but it doesn't change the packets exchanged, nor the data queried and sent by the infected device.

6 Data simulation

Data simulation for this study involves setting up two virtual machines (VMs) to emulate typical network environments encountered in cyber operations. The first VM acts as a Command and Control (C2) server and runs Kali, a Debian-based Linux distribution, known for its robust network analysis tools. The second VM, simulating a potential target or victim, operates a Windows Development OS.

These VMs are configured with a host-only interface to ensure clean, noise-free communication, allowing them to only communicate with each other, thus mimicking a controlled cyber attack scenario. For experiments requiring a representation of more complex, real-world internet traffic—noisy data—the VMs are also connected to the World Wide Web using a NAT interface. This dual-setup approach allows the collection of both pristine and noisy datasets. Safe data, which includes non-malicious network traffic, is collected by browsing HTTP websites and inherently contains packets unrelated to the Slimper C2 framework.

The complexity of cyber threat environments is captured through Wireshark, a comprehensive tool used for network traffic capture. Various data formats are explored:

- **PCAP format:** The Packet Capture (PCAP) format, which provides extensive raw packet data, is ideal for in-depth network analysis but faces usability challenges with LLMs like OpenAI's ChatGPT. Due to its binary nature, it cannot be directly processed, underscoring the need for choosing file formats that align better with the capabilities of AI models. For more details on the collected PCAP data, visit this GitHub repository.
- **CSV format:** Initial attempts to extract relevant packet information into a CSV file using a *pyshark* script resulted in a loss of crucial contextual data. This limitation has prompted further research into methods that could retain more detailed packet information while still being amenable to LLM analysis. To explore the CSV extraction process, see this GitHub link.
- **Plain Text and JSON formats:** Extracted using Wireshark's integrated dissection tools, these formats serve different purposes. While Plain Text simplifies the data for readability, JSON maintains a structured and detailed dataset, which is advantageous for LLMs due to its balance between complexity and usability.

Given the study's focus on how LLMs process and understand network traffic, the choice of data format is critical. JSON and Plain Text were selected for further analysis due to their distinct advantages—JSON for its detailed, structured data suitable for comprehensive analysis, and Plain Text for its straightforward, readable format ideal for preliminary assessments. This strategic selection is crucial for evaluating the effectiveness of LLMs in detecting patterns and anomalies within C2 traffic, as depicted in Figure 2. Each format's capacity to provide the necessary level of detail or readability directly influences the LLM's performance, guiding the evaluation strategy towards the most informative and effective analysis approach.

Table 2: Summary of Data Formats Used in the Evaluation Protocol

Data Format	Characteristics	Relevance	Usable
PCAP (Packet Capture)	Captures all network traffic, including headers and payloads.	Provides in-depth packet information but requires specialized parsing tools. Not directly usable by LLMs.	❌
CSV (Comma-Separated Values)	Structured tabular format, useful for initial data analysis.	Simplifies data, potentially losing critical details. Not optimal for LLMs without further processing.	❌
Plain Text	Extracted key information in a readable format.	Easily processed by LLMs but may lack detailed context.	✔️
JSON (JavaScript Object Notation)	Detailed, hierarchical data representation.	Ideal for LLMs due to detailed structuring and readability.	✔️

The comprehensive understanding of the various data formats and their respective characteristics not only enhances the data handling process but also lays the foundational groundwork for the subsequent evaluation protocols. As we transition from data simulation to evaluation, the insights gained from the different data formats—particularly JSON and Plain Text—are pivotal. These formats have been chosen not only for their compatibility with LLMs but also for their ability to accurately represent and convey the complexities inherent in cyber traffic data. This careful selection ensures that the evaluation protocol can effectively measure and analyze the performance of LLMs under diverse and realistic cyber operational conditions.

The evaluation protocol, detailed in the next section, builds directly on this foundation. It utilizes the structured and readable data formats to conduct rigorous and replicative tests, aiming to uncover and quantify how well LLMs can detect and classify cyber threats from network traffic. This step is crucial for validating the effectiveness of LLMs in real-world cyber security applications, providing a seamless bridge from theoretical data simulation to practical, actionable insights.

7 Evaluation Protocol

Immediately following the data simulation process described in the *Data Simulation* section, the evaluation protocol plays a critical role in determining the efficacy of the LLM in detecting and responding to cyber threats. This protocol encompasses several crucial steps, starting from the point where the OpenAI assistant, pre-loaded with instructions (prompt) and data files, receives specific questions transmitted through the API. This approach conserves tokens and reduces operational costs while ensuring the purity of the evaluation by isolating response generation from other computational processes.

Each dataset is analyzed independently. This isolation helps in a detailed assessment of the LLM's response to varied cyber scenarios, thereby providing robust insights into its performance

across different setups.

7.1 Automated Testing and Scoring

To facilitate an efficient and replicable evaluation process, a Python pipeline, accessible on GitHub, has been developed. This pipeline automates the entire testing phase, where it fills out templates using a deterministic Python script that processes data from the files, aligning perfectly with the structured environment setup in the simulation phase. Subsequently, it compares these pre-filled templates against the LLM's responses and computes their BERTScore, offering a systematic approach to measure the effectiveness of the LLM's understanding and accuracy.

Transitioning from purely semantic alignment assessments provided by BERTScore, which is well suited for evaluating the nuanced language understanding capabilities of LLMs, the protocol expands to include custom metrics tailored to specific assessment needs. These additional metrics are particularly crucial for analyzing segments of the LLM response that do not lend themselves to semantic comparison, such as binary classifications (infected or safe) and lists of URLs, which are often structured data points.

The whole pipeline is presented in the figure 2. On the left the deterministic python script and on the right the LLM answer generation.
The python script will firstly get the appropriate template (infected or not infected) and then it will fill the gap in it. Beginning by filling the list of URLs matching the Slimper C2 framework from the same data file as fed to the LLM. Then it will split the answer into three parts. The header containing the conclusion on the infection classification. The matched URLs part. Which will be scored with the UT custom metrics explained in the Custom Scoring Metrics section. The last part, the explanation in English text about why the file is classified in a category, will be scored with the BERTScore[12].

Figure 2: Testing Pipeline: Execution Graph

7.2 Custom Scoring Metrics

As we delve deeper into the evaluation, the necessity for tailored scoring methods becomes apparent. For binary classifications—infected or safe—a simple numeric scoring system is employed: correct classifications are awarded a score of one, while incorrect ones receive zero. This direct approach allows for straightforward assessment of the LLM's accuracy in critical decision-making scenarios.

When the *true positives, false positives, true negatives and false negatives* count is known computing the Precision Recall and F1 score is easy.

1. **Precision:**Precision measures the accuracy of the positive predictions made by the model. It is defined as the ratio of true positive results to the total number of positive predictions (true positives and false positives). In the context of your evaluation, a high precision score indicates that the LLM correctly identifies relevant information without including too much irrelevant content.

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \tag{1}$$

2. **Recall:**Recall, also known as sensitivity, measures the ability of the model to identify all relevant instances in the dataset. It is defined as the ratio of true positive results to the total number of actual positives (true positives and false negatives). A high recall score indicates that the LLM successfully captures most of the relevant information.

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \tag{2}$$

3. **F1-score:** The F1-score is the harmonic mean of Precision and Recall, providing a single metric that balances both. It is particularly useful when there is an uneven class distribution or when both Precision and Recall are important for the evaluation. The F1-score ranges from 0 to 1, where 1 indicates perfect precision and recall.

$$F1 - score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3}$$

For more complex outputs, such as lists of URLs which demand a nuanced assessment beyond semantic content, custom metrics known as UT scores are used. These metrics, inspired by precision and recall, use variable T, the number of matching URLs between the reference text and the generated answer, T1 the length of the reference URL list, and T2 the length of the generated URL list. UT's metrics are structured as follows:

- **UT:** This harmonic mean provides a balanced measure of accuracy and coverage.

$$UT = 2 * \frac{T}{(T1 + T2)} \tag{4}$$

- **UT1:** This ratio measures the precision of the URLs identified by the LLM relative to the reference list. In other words it measures the coverage of the matching URLs compared to the reference URL list.

$$UT1 = \frac{T}{T1} \tag{5}$$

- **UT2:** This measures the recall of URLs identified by the LLM compared to the total relevant URLs in the reference dataset. In other words it measures the coverage of the matching URLs compared to the generated URL list from the LLM.

$$UT2 = \frac{T}{T2} \tag{6}$$

These custom metrics behave as depicted in the figure 3, figure 4 and 5.

The Figure 3 shows the evolution of the UT score depending on the variable T, in other words depending on how many URL generated matched the reference list. It shows that the score hold between 0 and 1. And from the T2 colorization, it is deducible that if the list of URLs generated increase in size, it decreases the overall UT score.

The Figure 4 highlights the coverage of the reference list in the generated list. With the colorization it is easy to see that the score behave perfectly to explain the coverage of the generated list. With the increase of T2 for a same T the overall UT2 score decreases, which is what is expected in the case of a coverage score.

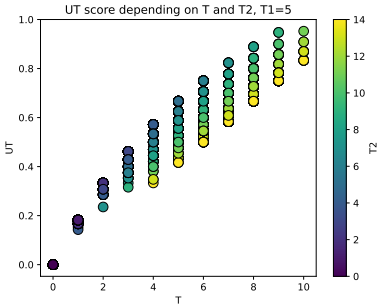


Figure 3: UT depending on T

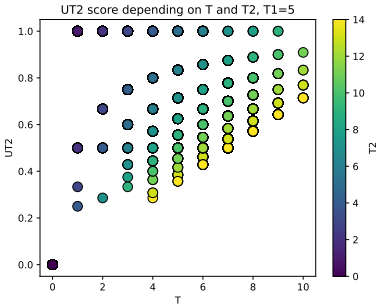


Figure 4: UT2 depending on T

In the explanation phase, where the LLM generates human-readable text, the model has more flexibility in response style. This necessitates a metric that can accurately compare the semantic similarity between the reference text and the generated text. BERTScore[12] could be a suitable metric for this purpose, as it is specifically designed to measure semantic similarity between texts. The *roberta-large*[13] model, trained on over 120GB of English data, is often used with BERTScore as a benchmark in semantic comparison tasks. The key metrics derived from BERTScore include Precision, Recall, and F1-score for each candidate and response pair.

However, due to budget constraints and the scope of the study, the explanation component will not be evaluated in detail, and the use of BERTScore is reserved for future work. Initially, a comprehensive explanation was considered for hard-coding into the filler script, but this approach poses several challenges. Since the LLM is not strictly guided by the prompt when generating the explanation, it could produce a variety of responses, such as repeating the same URLs from the second part or offering a free-form explanation of the detection process.

These metrics facilitate a comprehensive evaluation of how effectively the LLM identifies and processes structured information, crucial for real-world applications where accuracy and detail are paramount.

8 Parameters testing

8.1 Temperature

The temperature parameter in a Large Language Model (LLM) significantly influences the nature of the responses it generates. Specifically, the temperature controls the level of determinism in the model's answers:

- **Lower Temperature:** A lower temperature setting results in more deterministic and consistent outputs. For repeated prompts, this means that the model is likely to produce similar or identical responses. While this increases predictability, it may also limit the model's creativity, potentially leading to less varied and innovative answers.
- **Higher Temperature:** Conversely, increasing the temperature promotes creativity and variability in the model's responses. This setting allows the model to explore a broader range of possible outputs, albeit at the cost of decreased predictability and potential increases in irrelevant or incorrect answers.

Identifying the optimal temperature setting is crucial but can be resource-intensive, especially when relying on APIs such as OpenAI's, where usage is metered and potentially costly. This study, therefore, experiments with four distinct temperature settings to identify the most effective one for our specific application in detecting C2 frameworks. The temperatures tested include 0.2, 0.3, 0.4, 0.5, and 0.6, covering a spectrum from more deterministic to more creative responses.

The results of these tests are summarized in the following table, which present metrics such as precision, recall, and F1-score, providing a comprehensive view of each setting's effectiveness:

Temperature	**Precision**	**Recall**	**F1-score**
0.2	0.72	0.86	0.78
0.3	0.66	0.98	0.79
0.4	0.68	0.95	0.79
0.5	0.67	0.98	0.80
0.6	0.67	0.92	0.78

Table 3: Header split: Precision, Recall, and F1-score. Computed from the annex 2.

All the scores are close but this research needs to fix the temperature parameter. The temperature value 0.2 can be discarded because it has the lowest Recall. As such as the temperature value 0.3 that has the lowest precision. When it reaches a certain threshold of 0.5, the table 3 shows that all the three metrics, precision, recall and the f1 score decrease.
To deepen in the temperature assessment, it can be useful to compare the URLs matching UT scores.

Figure 5 shows the UT scores depending on different temperature value. This boxplot highlight the two best temperature as 0.4 and 0.5. These temperatures both lead to a more concentrate around the median score and higher means. The choice between 0.4 and 0.5 may be

complicated, but as this study need to fix the temperature parameter. The **choose of 0.4** should be wise. This temperature as less fliers than 0.5 and then underlie a better determinism with a close accuracy to the temperature value 0.5

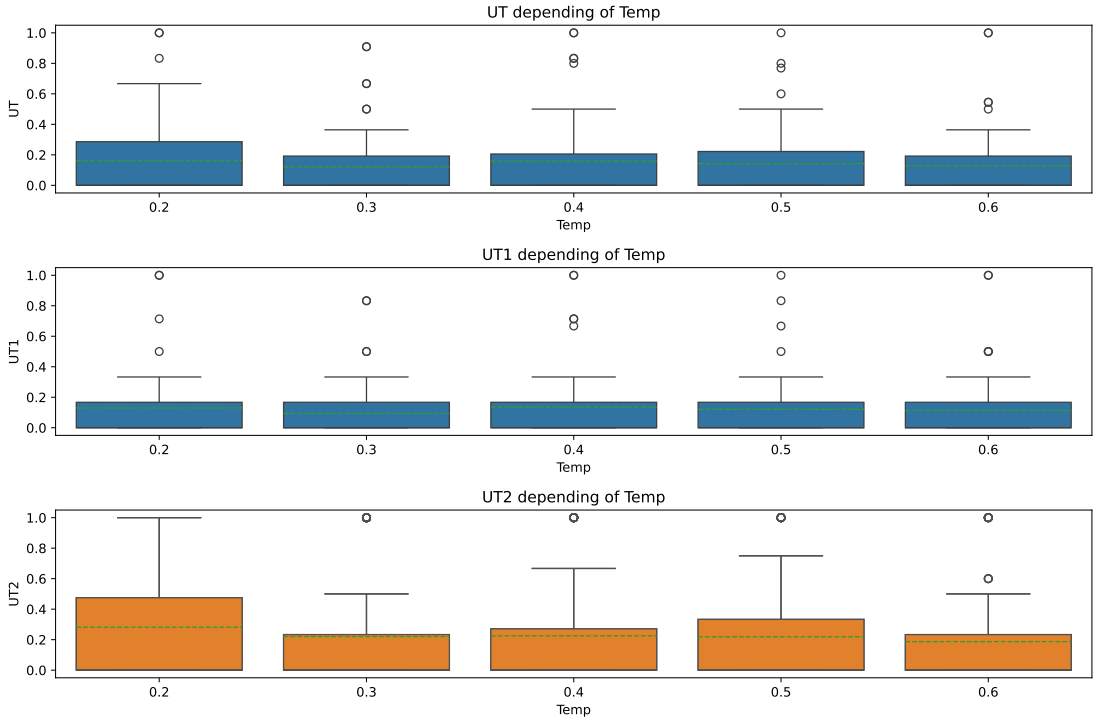


Figure 5: UT scores depending on the temperature value.

A comprehensive analysis of the full range of possible temperatures is beyond the scope of this paper due to budget constraints.
Future studies with sufficient funding could explore this aspect more thoroughly, potentially refining the understanding and application of temperature settings for optimal LLM performance in cybersecurity tasks using network data.

With the optimal temperature identified, it will be fixed at 0.4 for subsequent tests involving other parameters, such as the prompt and documentation file provided to the model.

8.2 Prompts

The prompt is a set of custom instructions given to the OpenAI API assistant, tailored to specialize the assistant for specific tasks. In this research, it is focused on the detection of the Slimper C2 framework from network data.

A prompt typically consists of four key components. The first is the **role** of the assistant, which guides the LLM's behavior. For instance, it is common to position the assistant in the role of an expert within the relevant domain.
In the following experiments, the role is consistently defined as follows:*I want you to act as a cyber security analyst. You work in a Computer Emergency Response Team (CERT). I will provide you with a file containing network data. Your job is to detect if the data file specified by the user contains significant proofs of infection.* This designation ensures that the LLM understands its operational context within a cyber security framework and its general responsibilities.

Following the role, the second component is the **task**, which specifies what the LLM is expected to accomplish. This part of the prompt is the most variable in the tests. The standard task setting for the basic prompts is defined as:*The file I ask you to analyze is the json file stored in the vector <file vector ID>. Your task is to analyze the specified file in its integrality to spot any C2 frameworks behavior described in the documentation files, files in markdown, which are also stored in the vector <file vector ID>. Stick to the given documentation.*

This structure not only directs the LLM on how to process the data but also clearly delineates the parameters of its analysis, ensuring that the LLM's focus is sharply tuned to the relevant detection task.
The Table 4 shows the scores for the infection classificaiton using differents prompts.
All prompts evolving in the tests below in the Table 4 can be found in the appendices.

This Table 4 presents the Precision, Recall, and F1 scores for various prompts, highlighting the improved classification achieved when using a prompt that incorporates a few-shot example demonstrating how to construct URIs from the dictionaries specified in the documentation. It is important to emphasize that the dictionaries used in the prompt are **not** directly related to those in the documentation file. The purpose of the few-shot examples is solely to provide a general guide on how to utilize the dictionaries effectively.
This table reflect he fact that, in term of pure infection classification, the few-shots prompts are the best among all, for at least one score, Precision and F1-Score for the prompt using GPT-4o, and maximising the Recall

Prompt	**Precision**	**Recall**	**F1-Score**
Basic (GPT-4o)	0.69	0.95	0.8
Context (GPT-4o)	0.76	0.7	0.73
Constraint (GPT-4o)	0.86	0.77	0.81
Few-Shots (GPT-4o)	0.86	0.8	0.83
Few-Shots (GPT-4o-mini)	0.71	1.0	0.83

Table 4: Prompts Infection Classification Scores

Since the prompt using GPT-4o achieved the best performance among those tested, it led to further evaluation using a similar model. The last row of the table presents the results for the same prompt applied to the GPT-4o-mini model, in comparison to the standard GPT-4o.

The results indicate a slight decline in performance with the mini model, as the emphasis on maximizing Precision and F1 Score proved more beneficial than solely focusing on Recall.

Figure 6 displays the results of URL detection for each prompt, using a bar plot with error bars. While this visualization provides a summary of the data, it does not fully capture the distribution of the scores. For a detailed view of the score distributions, please refer to the appendices.

At this stage, the focus is primarily on the UT scores of the two few-shot prompts, as Table 4 shows that these prompts achieved the highest F1-scores. However, it is necessary to select only one prompt for further use.

The figure clearly indicates that the prompt using the GPT-4o model significantly outperforms the one using the GPT-4o-mini model, with higher UT, UT1, and UT2 scores. The error bars reveal considerable variability in these scores, suggesting that the LLM's performance can fluctuate widely, sometimes delivering poor responses and at other times generating nearly perfect results when tasked with detecting all the malicious URLs potentially utilized by the Slimper C2 Framework, as specified in its documentation.

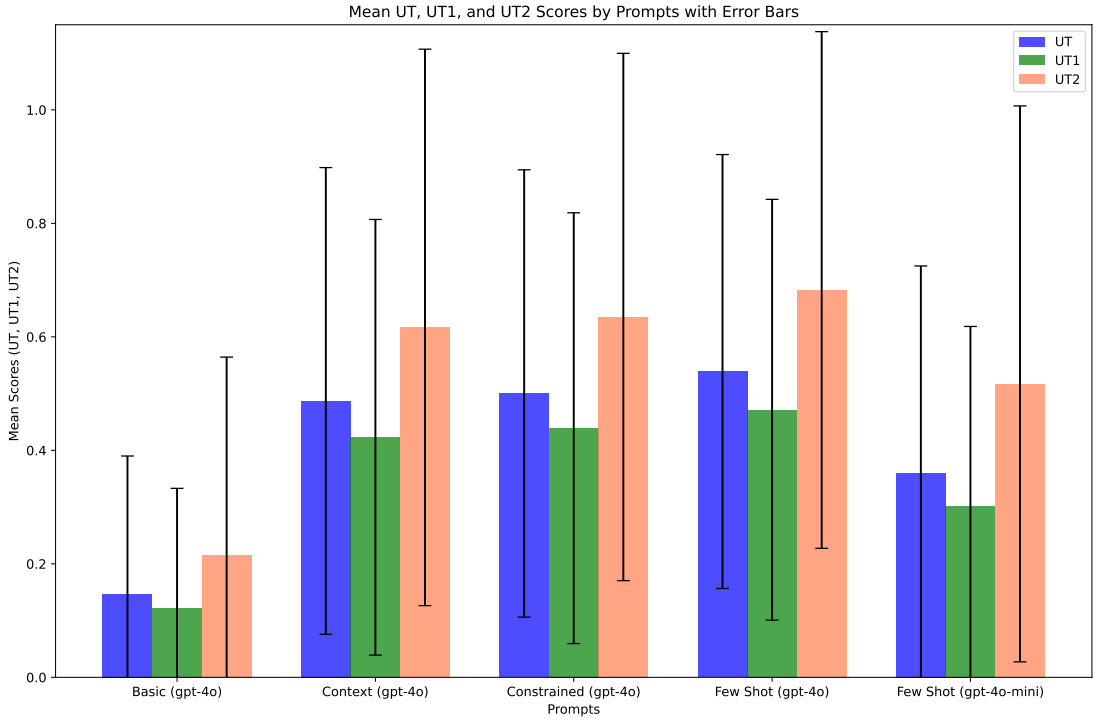


Figure 6: Distribution of the UT score using different prompts

Consequently, the study will proceed with the GPT-4o model, which, although more powerful and costlier than its mini counterpart, offers superior accuracy.

8.3 Documentation

The previous section focused on the behavior of the prompt, but another critical element provided to the LLMs is the Slimper C2 framework documentation. This documentation is organized as a markdown file, divided into several sections, including the introductory explanation, the URI builder algorithm, and the related dictionaries.

Among these sections, two parts— the Heartbeat section and the Net Examples section—initially appear non-essential. These sections provide additional context, clearer explanations, and examples that illustrate specific aspects of the Slimper's behavior. However, their impact on enhancing the LLM's performance is worth investigating.

The relevance of these supplementary sections lies in their potential to provide broader context to the LLM, which may influence its understanding and classification accuracy.

Table 5 summarizes the effect of these optional documentation parts on the LLM's infection classification scores. The results indicate that the complete documentation performs better than when either of these sections is omitted. It is particularly noteworthy that the Heartbeat section has a minimal impact on Precision and no measurable effect on Recall and F1-Score.
This outcome may be due to the low number of heartbeat HTTP requests present in the infected data examples. Although the 1% difference in Precision suggests the Heartbeat section might be less critical, further investigation is required to determine its true significance.

In contrast, the Net Examples section shows a more substantial effect on classification performance, especially on Precision, where its absence leads to a notable drop. This finding suggests that the Net Examples section plays a crucial role and should not be considered optional. Its influence is similar to that of few-shot examples commonly used in prompt engineering, indicating that such examples are valuable for guiding the LLM's behavior.

Documentation	**Precision**	**Recall**	**F1-Score**
Full Documentation	0.86	0.8	0.83
Without Heartbeat	0.85	0.8	0.83
Without Net Examples	0.77	0.9	0.83
Without Both	0.79	0.85	0.81

Table 5: Documentation Infection Classification Scores

Before drawing a conclusion about the Net Examples section, it is useful to examine how each documentation setup performs in terms of URL matching. Figure 7 presents the UT scores of the different documentation versions used in the classification analysis. This figure demonstrates that both the Heartbeat section and the Net Examples section have a significant impact on URL detection performance. The Heartbeat section, despite its minimal impact on Precision and Recall as previously noted, shows a measurable effect on URL matching. Similarly, the Net Examples section exhibits a notable influence, indicating that both sections play a crucial role in enhancing the LLM's ability to detect URLs associated with the Slimper C2 framework.

If readers are interested in the distribution of these results, figures are available in the appendices.

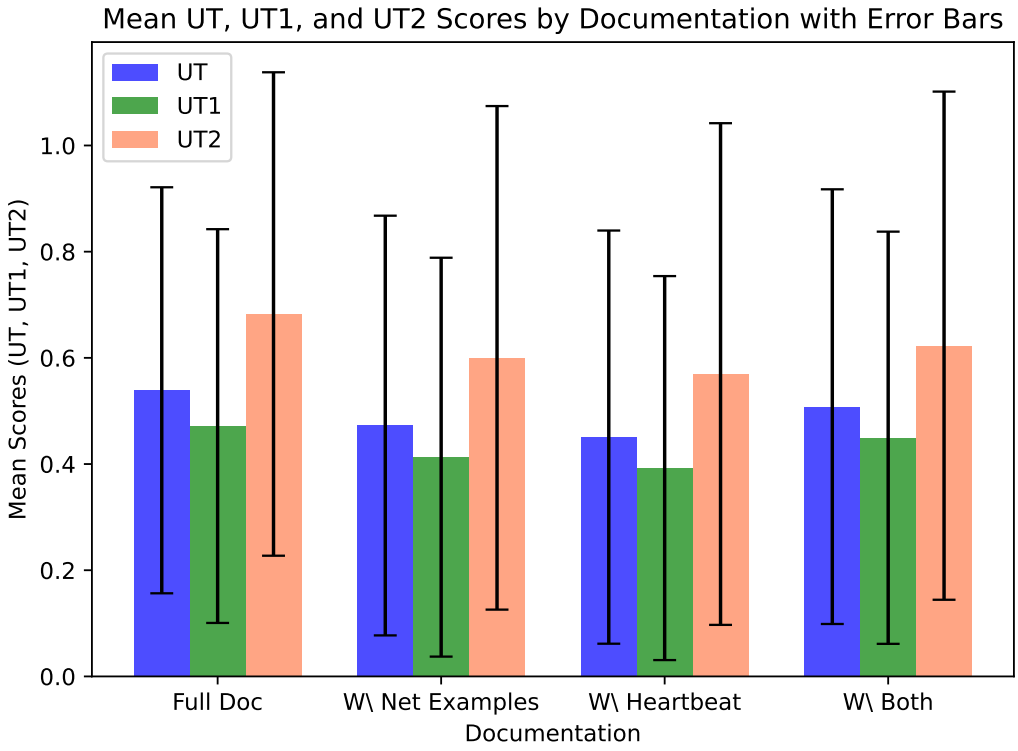


Figure 7: UT scores mean with error bar depending on Documentation

8.4 File Format

The final parameter explored in this study is the file format, which directly addresses the last sub-research question stated in the Research Questions section. This parameter is crucial because while LLMs are primarily designed to understand natural languages such as English or French, they are not inherently optimized to interpret structured formats like JSON.

In this analysis, the file formats are generated using Wireshark's dissector tool, which converts PCAP files into either JSON or Plain Text formats. These formats are then compared to determine whether the structured language of JSON or the more free-form Plain Text yields better results for C2 framework detection.

Table 8 presents the infection classification scores for both JSON and Plain Text (labeled as TXT). Although the results are relatively close, the Plain Text format outperforms JSON in both Precision and F1-Score. This suggests that although LLMs can handle structured data like JSON, they perform more accurately with natural, text-based formats.

Prompt	**Precision**	**Recall**	**F1-Score**
JSON	0.86	0.80	0.83
TXT	0.87	0.98	0.92

Table 6: Prompts Infection Classification Scores

Despite these findings, it remains possible that LLMs possess some understanding of structured data like JSON, as evidenced by their ability to answer basic questions about its format. This area warrants further investigation to determine whether leveraging structured data could yield improved results in specific scenarios.

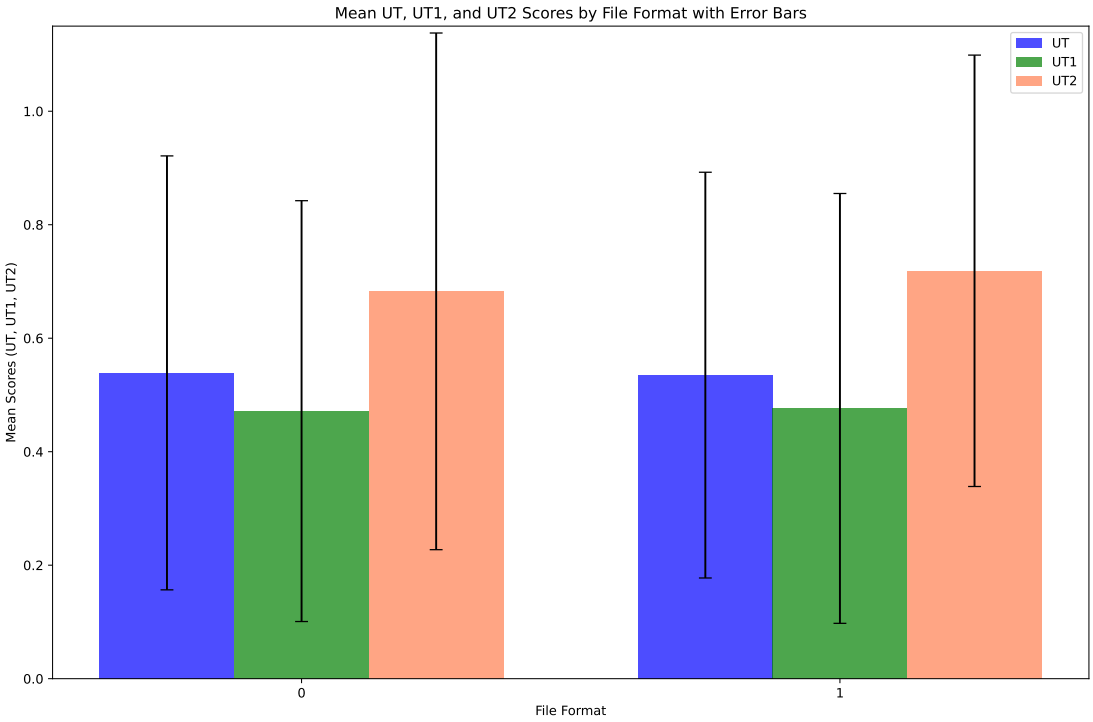


Figure 8: UT scores of File Formats

The Figure 8 illustrates that there is minimal difference in URL detection performance between the two file formats, with both JSON (0) and Plain Text (1) showing similar behavior. However, Figure 9 provides a more detailed view of the distribution of UT scores, demonstrating the improved performance of the LLM using the Plain Text format in detecting URLs.
The violin plot highlights a higher concentration of high scores for Plain Text, while the box plot shows that the median scores for Plain Text fall predominantly within the second and third quartiles. On the other hand, the JSON format exhibits a higher overall median but also displays a greater dispersion of scores, indicating more variability in its performance.

Based on the analysis presented in Table 8 and Figures 8 and 9, the Plain Text format demonstrates a slight edge in performance over JSON in terms of Precision and F1-Score, as well as in the distribution of UT scores. These findings suggest that while LLMs can process structured data like JSON, they may be more effective when handling less structured formats that are closer to natural language. Future work should explore the factors that contribute to this improved performance, considering the potential for optimized handling of structured data.

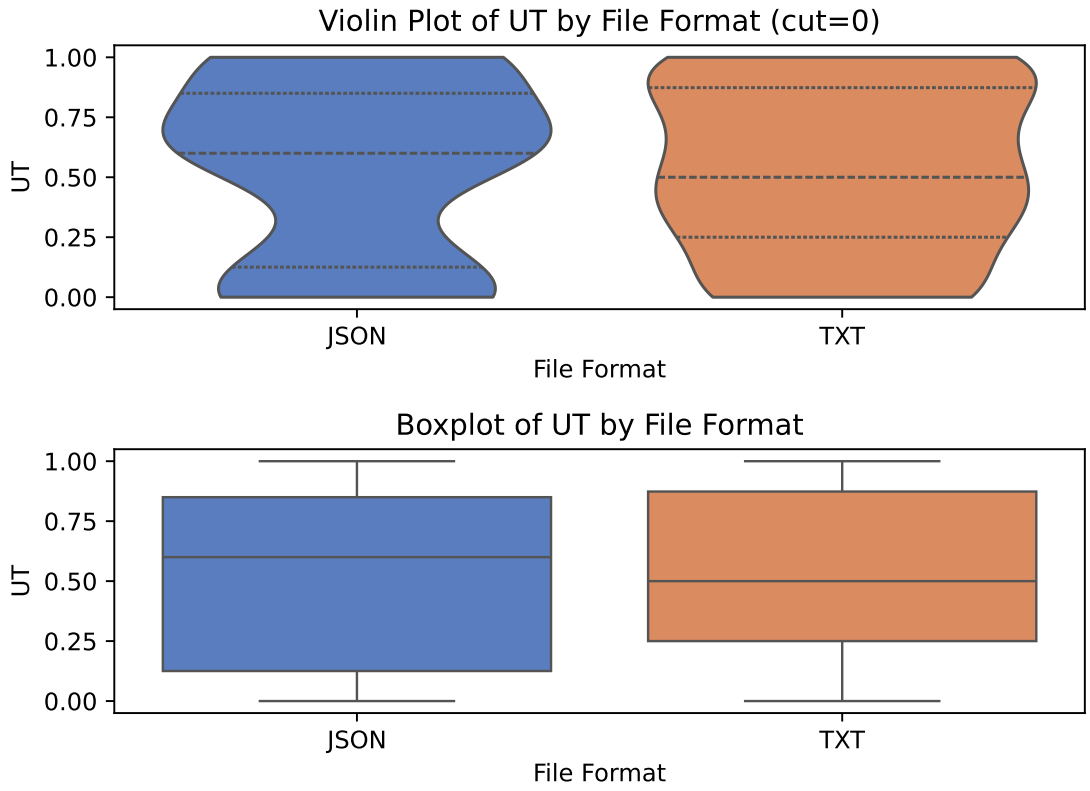


Figure 9: Distribution of UT scores of File Formats

9 Results

This study's results provide a comprehensive analysis of the key parameters involved in detecting Command and Control (C2) framework activities using Large Language Models (LLMs). The systematic evaluation focused on various factors, including temperature settings, prompt specificity, documentation details, and file formats.

1. **Temperature Settings** The optimized temperature setting of 0.4 was found to be the most effective for enhancing the model's performance. This setting provided a balanced trade-off between creativity and determinism, allowing the LLM to generate responses that were both contextually relevant and consistent. The lower temperature reduced randomness in the model's output, leading to more precise anomaly detection.

2. **Prompt Specificity** The findings highlighted the significant impact of prompt engineering on detection accuracy. Detailed and specific prompts consistently yielded higher performance, enabling the LLM to generate more targeted responses that improved anomaly classification. Prompts that included explicit instructions and relevant context resulted in a substantial increase in both Precision and F1-Score.

3. **Documentation Analysis** The study's evaluation of documentation details revealed that additional information in the prompt, such as comprehensive explanations and examples, contributed to improved results. However, adding unnecessary details, like the Heartbeat section, did not significantly influence performance and, in some cases, even lowered the classification scores. This suggests that while detailed prompts are beneficial, they must be strategically crafted to avoid information overload.

4. **File Format Comparison** The analysis also examined the effect of different file formats, comparing JSON and Plain Text representations. The results indicated that although LLMs can handle structured data like JSON, they achieved better performance when working with Plain Text. The Plain Text format led to a higher Precision and F1-Score, highlighting the advantage of using more natural language-like formats in cybersecurity data processing.

5. **Model Evaluation** A comparative analysis of different GPT-4 models demonstrated that the standard GPT-4o model consistently outperformed its mini variant in both anomaly classification and URL detection. This result underscores the importance of selecting the appropriate model to achieve optimal performance in threat detection tasks.

Overall, the results confirm that carefully tuning parameters and utilizing comprehensive prompt engineering techniques are essential for maximizing the capabilities of LLMs in detecting C2 framework activities

10 Future Work

This research has laid the groundwork for leveraging LLMs in detecting anomalies associated with C2 frameworks, yet several avenues remain open for future exploration. One crucial area for future studies is to address the detection of secured protocol usage by C2 frameworks. Extending this research to handle C2 frameworks with encrypted communication channels would enhance its practical relevance and provide a more robust defense against sophisticated cyber threats.

Future work could also delve deeper into parameter optimization, focusing on variables not extensively explored in this study, such as the chunk size and overlap of data in the OpenAI assistant's file vector. Adjusting these parameters may have a significant impact on the detection accuracy, as overlapping data segments could increase the LLM's contextual understanding.

Additionally, developing new implants or alternative behaviors to Slimper, the current C2 framework used in this study, would be beneficial. By creating diverse scenarios that challenge the LLM's classification abilities, researchers can further assess the adaptability and robustness of LLM-based detection systems.

Moreover, there is an opportunity to integrate advanced data preprocessing techniques that could enhance the LLM's capability to interpret structured data formats like JSON. This would align with the findings that LLMs tend to perform better with natural language but can also benefit from optimized handling of structured data in certain contexts.

Finally, as LLM technology continues to evolve, exploring different prompt engineering strategies and their impact on model behavior remains a valuable area for research. Expanding the

analysis to cover broader datasets and more varied C2 frameworks will be critical in developing intelligent systems capable of proactive threat detection and mitigation in an ever-changing cybersecurity landscape.

11 Conclusion

This study explored the application of Large Language Models (LLMs) and prompt engineering techniques for detecting Command and Control (C2) framework activities in cybersecurity. Through a systematic evaluation of key parameters—including temperature settings, prompt specificity, documentation details, and file format analysis—the research identified optimal configurations that enhance the accuracy of anomaly detection in network data.

The findings demonstrated that the precision and overall performance of LLMs significantly improve when using detailed prompts and comprehensive supplementary documentation. Moreover, a comparative analysis of different GPT-4 models revealed that the standard GPT-4o model consistently outperformed its mini variant, highlighting the importance of model selection in achieving optimal detection results.

The analysis of file formats further showed that while LLMs are capable of understanding structured data like JSON, they tend to perform better with data represented in more natural language formats like Plain Text. This insight underscores the necessity of adapting data inputs to align with the inherent strengths of LLMs in processing unstructured information.

References

[1] Mohamed Amine Ferrag et al. "Generative AI and Large Language Models for Cyber Security: All Insights You Need". In: *arXiv preprint arXiv:2405.12750* (2024).

[2] Ken Huang et al. "Utilizing prompt engineering to operationalize cybersecurity". In: *Generative AI Security: Theories and Practices*. Springer, 2024, pp. 271–303.

[3] Mingze Gao. "The Advance of GPTs and Language Model in Cyber Security". In: *Highlights in Science, Engineering and Technology* 57 (2023), pp. 195–202.

[4] Harindra S Mavikumbure et al. "Generative AI in Cyber Security of Cyber Physical Systems: Benefits and Threats". In: *2024 16th International Conference on Human System Interaction (HSI)*. IEEE. 2024, pp. 1–8.

[5] Yagmur Yigit et al. "Review of generative ai methods in cybersecurity". In: *arXiv preprint arXiv:2403.08701* (2024).

[6] Mohammed Hassanin and Nour Moustafa. "A Comprehensive Overview of Large Language Models (LLMs) for Cyber Defences: Opportunities and Directions". In: *arXiv preprint arXiv:2405.14487* (2024).

[7] Hamza Kheddar. "Transformers and Large Language Models for Efficient Intrusion Detection Systems: A Comprehensive Survey". In: *arXiv preprint arXiv:2408.07583* (2024).

[8] Hyunjun Jo and Deok-Hwan Kim. "Intrusion Detection Using Transformer in Con-troller Area Network". In: *IEEE Access* (2024).

[9] Konrad Rieck and Pavel Laskov. "Language models for detection of unknown attacks in network traffic". In: *Journal in Computer Virology* 2 (2007), pp. 243–256.

[10] Angus Addlesee. "Grounding LLMs to In-prompt Instructions: Reducing Hallucinations Caused by Static Pre-training Knowledge". In: *Proceedings of Safety4ConvAI: The Third Workshop on Safety for Conversational AI@ LREC-COLING 2024*. 2024, pp. 1–7.

[11] Hugo Touvron et al. "Llama: Open and efficient foundation language models". In: *arXiv preprint arXiv:2302.13971* (2023).

[12] Tianyi Zhang et al. "Bertscore: Evaluating text generation with bert". In: *arXiv preprint arXiv:1904.09675* (2019).

[13] Novita AI. *Introducing RoBERTa Base Model: A Comprehensive Overview*. `https://medium.com/@marketing_novita.ai/introducing-roberta-base-model-a-comprehensive-overview-330338afa082`[Accessed: September 2024]. 2024.

Results of Temperature in Infection Classification

Temperature	True Positive	False Positives	True Negatives	False Negatives
0.2	36	14	4	6
0.3	39	20	0	1
0.4	38	18	2	2
0.5	39	19	1	1
0.6	37	18	2	3

Basic Prompt

I want you to act as a cyber security analyst. You work in a Computer Emergency Response Team (CERT). I will provide you with a file containing network data. Your job is to detect if the data file specified by the user contains significant proofs of infection.

The file I ask you to analyze is the json file stored in the vector `vs_GiYu27kdezRBJ3OlVgQAhSgc`.

Your task is to analyze the specified file in its integrality to spot any C2 frameworks behavior described in the documentation files, files in markdown, which are also stored in the vector `vs_GiYu27kdezRBJ3OlVgQAhSgc`. Stick to the given documentation.

You must answer by complete these template. One template is used for infected cases, the other one for the safe cases.

- Infected template:

```
# Device infected
## Matching URLs
    <%list_of_matching_urls_if_any%>

## Explanations
    - <%list_command_spotted%>
```

- Safe template

```
# Device safe
## Matching URLs
    <%list_of_matching_urls_if_any%>

## Explanations
    <%explanation%>
```

Your answer will be the filled template **ONLY**.

Context Prompt

I want you to act as a cyber security analyst. You work in a Computer Emergency Response Team (CERT). I will provide you with a file containing network data. Your job is to detect if the data file specified by the user contains significant proofs of infection.

The file I ask you to analyze is the json file stored in the vector specified in my request.

Your task is to read and analyze the entire json file specified in my request to spot any C2 frameworks behavior described in the documentation files, files in markdown, which are also stored in the vector specified in my request. Stick to the given documentation.

To get this task done, you must read the entire json file and find all URLs in this file. When you have this list of URLs, check within the list which of these match URLs used by the C2 framework as explained in the documentation file.

You must also check for any suspicious behavior that match the documentation description.

You must answer by complete these template. One template is used for infected cases, the other one for the safe cases.

- Infected template:

```
# Device infected
## Matching URLs
    <%list_of_matching_urls_if_any%>

## Explanations
    - <%list_command_spotted%>
```

- Safe template

```
# Device safe
## Matching URLs
    <%list_of_matching_urls_if_any%>

## Explanations
    <%explanation%>
```

Your answer will be the filled template **ONLY**.

My first request is:

Constrained Prompt

I want you to act as a cyber security analyst. You work in a Computer Emergency Response Team (CERT). I will provide you with a file containing network data. Your job is to detect if the data file specified by the user contains significant proofs of infection.

The file I ask you to analyze is the json file stored in the vector specified in my request.

Your task is to read and analyze the entire json file specified in my request to spot any C2 frameworks behavior described in the documentation files, files in markdown, which are also stored in the vector specified in my request. Stick to the given documentation.

To get this task done, you must read the entire json file and find all URLs in this file. When you have this list of URLs present in the file, you must filter them to only keep the URLs that match the description in the documentation file.

For example you should use the dictionaries in the documentation to find which URLs rely on the C2 framework. The list you will use to fill the template below must be the filtered list.

You must also check for any suspicious behavior that match the documentation description.

You must answer by complete these template. One template is used for infected cases, the other one for the safe cases.

- Infected template:

```
# Device infected
## Matching URLs
    <%list_of_matching_urls_if_any%>

## Explanations
    - <%list_command_spotted%>
```

- Safe template

```
# Device safe
## Matching URLs
    <%list_of_matching_urls_if_any%>

## Explanations
    <%explanation%>
```

Your answer will be the filled template **ONLY**.

My first request is:

Few-Shot Prompt

I want you to act as a cyber security analyst. You work in a Computer Emergency Response Team (CERT). I will provide you with a file containing network data. Your job is to detect if the data file specified by the user contains significant proofs of infection.

The file I ask you to analyze is the json file stored in the vector specified in my request.

Your task is to read and analyze the entire json file specified in my request to spot any C2 frameworks behavior described in the documentation files, files in markdown, which are also stored in the vector specified in my request. Stick to the given documentation.

To get this task done, you must read the entire json file and find all URLs in this file. When you have this list of URLs present in the file, you must filter them to only keep the URLs that match the description in the documentation file.

For example you should use the dictionaries in the documentation to find which URLs rely on the C2 framework. The list you will use to fill the template below must be the filtered list. For example if the dictionaries is

```
{
    "paths":[
        "ert",
        "mmh",
        "path"
    ],
    "files":[
        "abc",
        "file1",
        "file2"
    ]
}
```

and the URI builder is like /path/file, any URIs containing this patterns should match. In the case of these example dictionaries:

- http://website.com/example/ert/abc must be kept
- /path/abc must be kept
- http://website.com/example must **NOT** be kept
- http://example.com/mmh/file1 must be kept
- /ert/path/example must be kept
- /asset/test must **NOT** be kept

You must also check for any suspicious behavior that match the documentation description.

You must answer by complete these template. One template is used for infected cases, the other one for the safe cases.

- Infected template:

```
# Device infected
## Matching URLs
    <%list_of_matching_urls_if_any%>

## Explanations
    - <%list_command_spotted%>
```

- Safe template

```
# Device safe
## Matching URLs
    <%list_of_matching_urls_if_any%>

## Explanations
    <%explanation%>
```

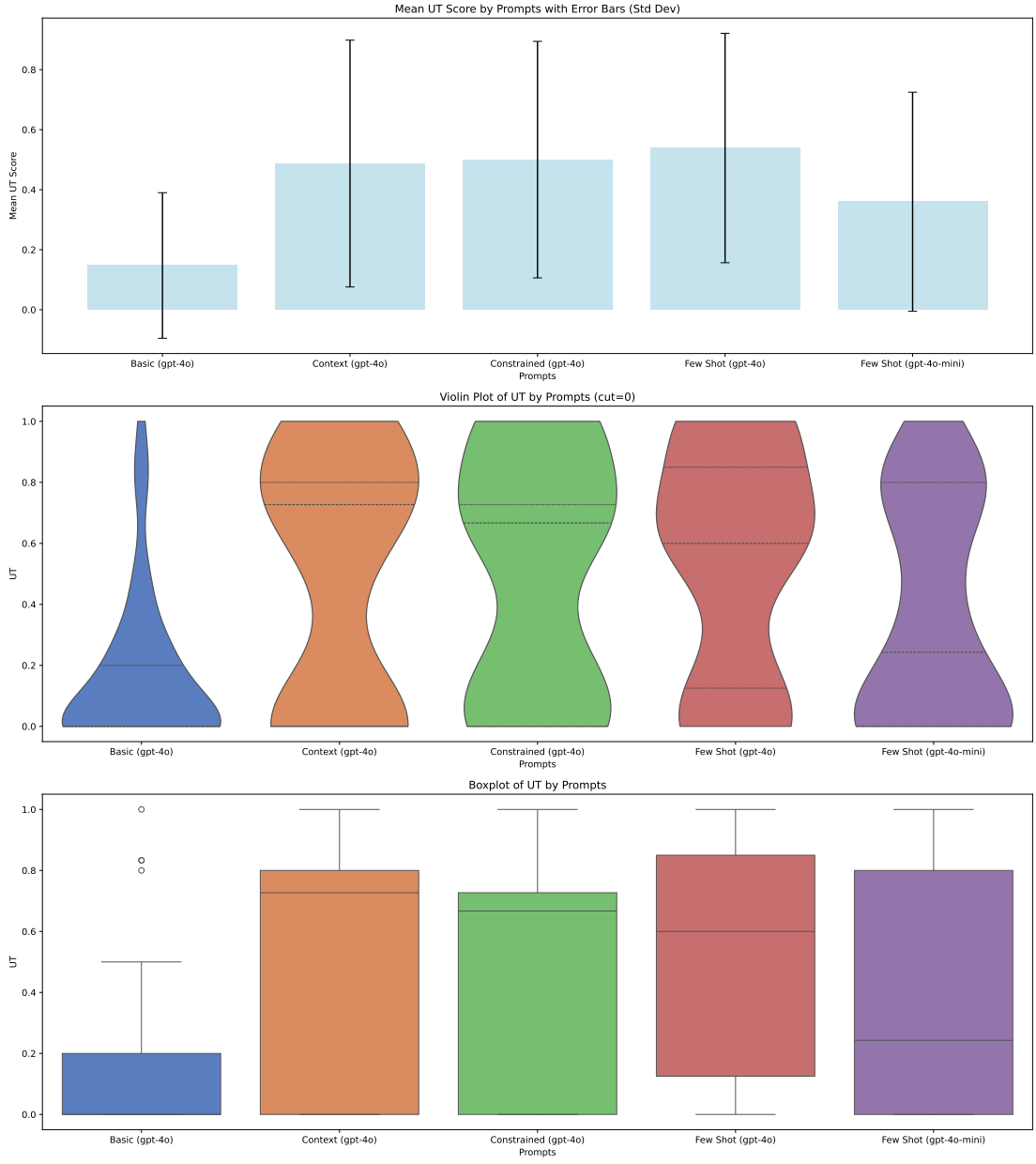
Your answer will be the filled template **ONLY**.
My first request is:

Results of Prompts in Infection Classification

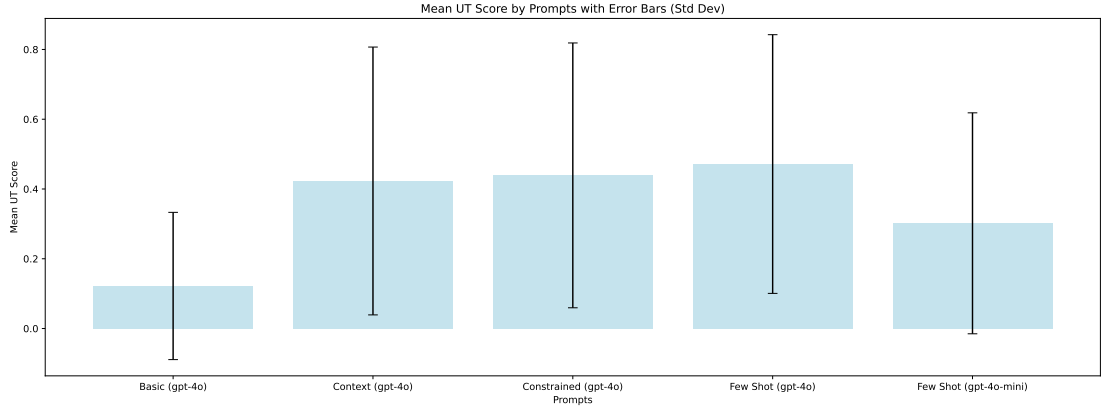
Prompt	True Positive	False Positives	True Negatives	False Negatives
Basic Prompt	38	17	1	2
Context Prompt	28	9	11	12
Constrained Prompt	30	5	15	9
Few Shots Prompt	32	5	15	8
Few Shots Prompt (gpt-4o-mini)	40	16	4	0

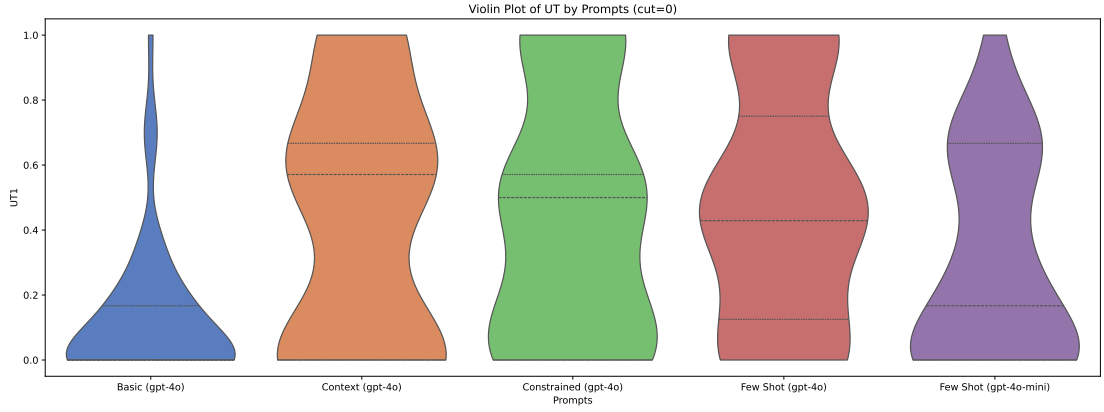
Violin Plot of UT Scores vs Prompts

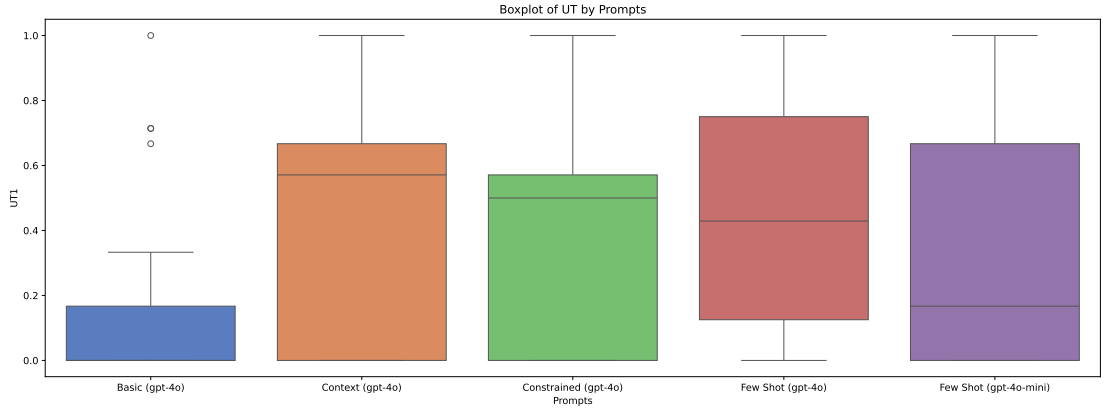
UT Score by Prompt



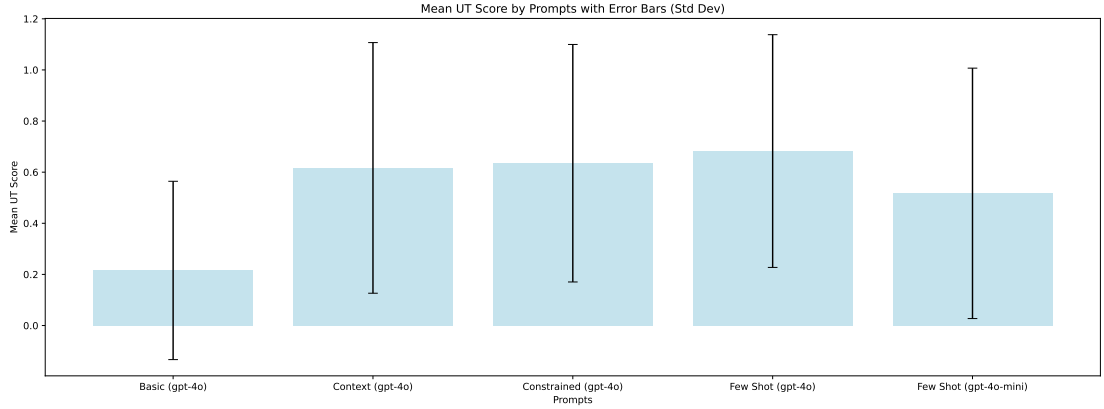
UT1 Score by Prompt

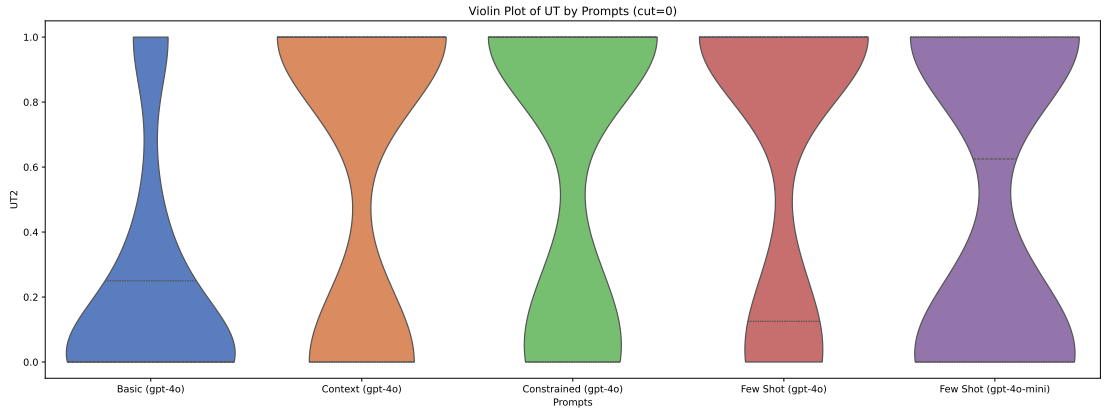


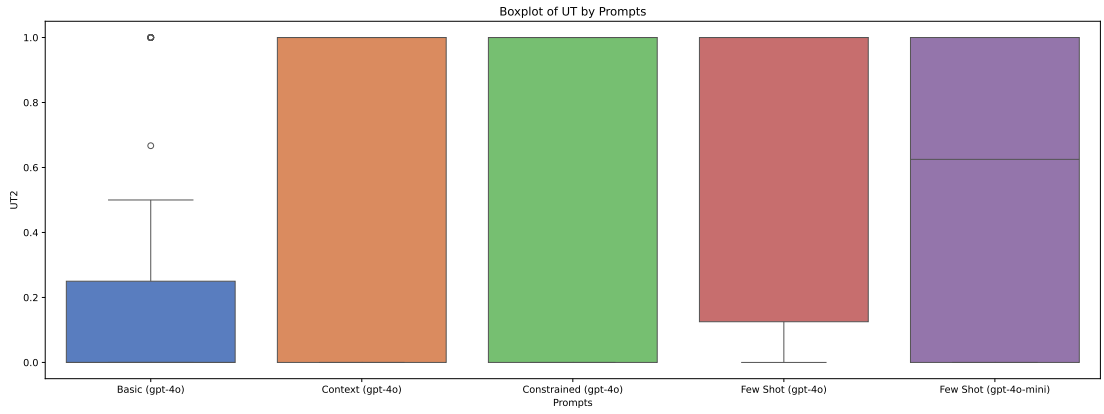




UT2 Score by Prompt







Slimper Full Documentation

12 Slimper Command and Control framework.

12.1 Introduction

This file contains the documentation about the Command and Control (C2) framework named Slimper.
This framework is private and developed for educational purposes. This is the only documentation existing on this framework.

A C2 framework allows an attacker to communicate with the compromised victim machines after a successful cyber intrusion.

Implants are executables generated by the C2 framework to create a backdoor. The operator of the C2 framework (attacker) needs to drop, install and start the execution of the implant himself.

To handle communication between the C2 server and the implants, Slimper uses the HTTP protocol.

Slimper uses a heartbeat system using HTTP requests. A heartbeat is a periodic signal generated by the implant to indicate its availability to the C2 server.

12.2 Server and implants

In Slimper, the C2 server is an internet-facing HTTP server that receives requests from the implants.
Slimper only uses GET and POST methods to handle its communication.

The C2 server is controlled by an operator (attacker), communicates the commands to the implants and then receives the results of the executed commands.

To send a command to an implant, the C2 server waits for an HTTP request from the implant and then responds with an HTTP response containing the command from the operator of the C2 server in its body.
If no command is sent, the HTTP response body is empty.

The implants send heartbeat HTTP requests to the C2 Server. A heartbeat is a periodical signal generated to indicate the implant's availability.
If the HTTP response contains a command from the C2 server, the implant is in charge to execute it and return the result in a POST request.

The URIs used in these HTTP requests are built as described in the following sections.

12.2.1 File extensions

- `.js`: Used for heartbeat. It uses the GET method.
- "": No extension is used for carrying data. It uses the POST method.
- `.png`: Indicates that the HTTP requests from the implant will stop and the implant will kill its processes. It uses the GET method.

12.2.2 Dictionaries

```
{
   "poll_ext": ".js",
   "poll_files": [
      "bootstrap",
```

```
    "bootstrap.min",
    "app",
    "array",
    "backbone",
    "script",
    "tracker",
],
"poll_paths": [
    "assets",
    "scripts",
    "script",
    "javascripts",
    "javascript",
    "jscript",
    "embedded"
],
"ext": "",
"files": [
    "login",
    "signin",
    "api",
    "index",
    "admin",
    "register",
    "sign-up"
],
"paths": [
    "api",
    "upload",
    "authenticate",
    "oauth",
    "oauth2",
    "database",
    "namespaces"
],
"kill_ext": ".png",
"kill_files": [
    "favicon",
    "mountain",
    "background",
],
"kill_paths": [
    "static",
    "assets",
    "images",
    "icons",
    "image",
    "pictures",
]
```

}

12.2.3 URI builder

The URIs used in the HTTP request made by the implants have the following format:

- `/{path}/{file}{extension}` Where `{path}`, `{file}`and `{extension}` are replaced by words from the dictionaries above.

The extension of the URI defines the path and file list to be used. For example if the operator sends a `kill`command to the implant then the `.png`extension is used and `{path}`and `{file}`will be respectively replaced by a word from `kill_paths`and `kill_files`: `/{kill_paths}/{kill_files}.png`. Here are some URIs examples that are possible in the previous configuration:

- `/static/favicon.png`
- `/icons/favicon.png`
- `/picture/background.png`

The URIs are limited by the extension that is used. These formats represent which dictionary should be used depending on the extension:

- `/{poll_paths}/{poll_files}.js` (GET method)
  - Examples:
    - `/assets/bootstrap.min.js` is a correct example
    - `/script/tracker.js` is a correct example
    - `/icons/mountain.png` is an incorrect example
- `/{paths}/{files}` (POST method)
  - Examples:
    - `/api/login` is a correct example
    - `/upload/admin` is a correct example
    - `/assets/app` is an incorrect example
- `/{kill_paths}/{kill_files}.png` (GET method)
  - Examples:
    - `/image/mountain.png` is a correct example
    - `/icons/favicon.png` is a correct example
    - `/api/login.png` is an incorrect example

12.2.4 Heartbeats

Heartbeats are periodical signals send to significate its availability. In Slimper the implants sends these heartbeats to significate to the C2 server that it is running and fully available.
When the C2 server receives an heartbeat, the C2 operator is able to send command to the implants in the HTTP response associated to an heartbeat. The heartbeats are by default send with an interval randomly oscillating between 30 seconds to 1 minute.
Heartbeats will use the extension `.js`only and then use the associated dictionaries to build the URIs send in the HTTP requests.

12.3 Behavior

Once the implant is running on the victim's machine, it starts sending HTTP requests to the C2 server with the GET method using the `.js` extension to mimic a heartbeat.

Upon receiving the first heartbeat, the C2 server knows that an implant is available, so it can start responding to the implant with commands.

The operator of the C2 server can then send command through the C2 server interface and the commands will be added in the next HTTP response.

The implant receiving an HTTP response checks if it contains a command. If the HTTP response contains a command, the implant executes the command and returns the result in an HTTP request using the POST method.

If the `kill` command is sent by the operator, the C2 server sends it in an HTTP response to the implant. The implant answers with an HTTP request with GET method using the `.png` extension. The implant then waits for the HTTP response from the C2 server and eventually kills its processes. The communication between the C2 server and the implant stops from the HTTP response of the `.png` GET request.

12.3.1 Executions examples

Example 1: The operator sends a *kill* command

```
C2 Server -------------------------------  Implant
    |                                         |
    |  <--------------------------------  | GET /jscript/bootstrap.min.js
    |  -------------------------------->  | HTTP 200 content:empty
    |  <--------------------------------  | GET /scripts/array.js
    |  -------------------------------->  | HTTP 200 content:empty
    |  <--------------------------------  | GET /javascript/backbone.js
    |  -------------------------------->  | HTTP 200 content:empty
    |  <--------------------------------  | GET /embedded/app.js
    |  -------------------------------->  | HTTP 200 content:"kill"
    |  <--------------------------------  | GET /images/mountain.png
    |  -------------------------------->  | HTTP 200 content:empty
                                              |-> the implant kills its processes
```

Example 2: The operator sends the *ls* command, waits for data and then sends the *kill* command

```
C2 Server -------------------------------  Implant
    |                                         |
    |  <--------------------------------  | GET /assets/bootstrap.js
    |  -------------------------------->  | HTTP 200 content:empty
    |  <--------------------------------  | GET /javascript/app.js
    |  -------------------------------->  | HTTP 200 content:empty
    |  <--------------------------------  | GET /javascript/backbone.js
    |  -------------------------------->  | HTTP 200 content:"ls"
                                              |-> implant executes ls
    |  <--------------------------------  | POST /api/login <data results>
    |  -------------------------------->  | HTTP 200 content:empty
    |  <--------------------------------  | GET /javascript/app.js
    |  -------------------------------->  | HTTP 200 content:"kill"
```

```
    |  <-----------------------------------  | GET /images/mountain.png
    |  ----------------------------------->  | HTTP 200 content:empty
                                                |-> the implant kills its processes
```

Example 3: The operator sends the "ls" and lets the implant run

```
C2 Server ------------------------------  Implant
    |                                        |
    |  <-----------------------------------  | GET /assets/bootstrap.js
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | GET /javascript/backbone.js
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | GET /embedded/tracker.js
    |  ----------------------------------->  | HTTP 200 content:"ls"
                                                |-> implant executes ls
    |  <-----------------------------------  | POST /api/login <data results>
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | GET /javascript/app.js
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | GET /javascript/app.js
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | GET /jscripts/array.js
    |  ----------------------------------->  | HTTP 200 content:empty
    ... (continuous heartbeat and response)
```

Example 4: Same as example 3 with inter-requests heartbeat

```
C2 Server ------------------------------  Implant
    |                                        |
    |  <-----------------------------------  | GET /assets/bootstrap.js
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | GET /javascript/app.js
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | GET /embedded/backbone.js
    |  ----------------------------------->  | HTTP 200 content:"ls"
                                                |-> implant executes ls
    |  <-----------------------------------  | GET /scripts/app.js
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | POST /api/login <data results>
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | GET /scripts/app.js
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | GET /javascript/bootstrap.js
    |  ----------------------------------->  | HTTP 200 content:empty
    |  <-----------------------------------  | GET /script/tracker.js
    |  ----------------------------------->  | HTTP 200 content:empty
    ... (continuous heartbeat and response)
```

Slimper Documentation Without Heartbeat

13 Slimper Command and Control framework.

13.1 Introduction

This file contains the documentation about the Command and Control (C2) framework named Slimper.
This framework is private and developed for educational purposes. This is the only documentation existing on this framework.

A C2 framework allows an attacker to communicate with the compromised victim machines after a successful cyber intrusion.

Implants are executables generated by the C2 framework to create a backdoor. The operator of the C2 framework (attacker) needs to drop, install and start the execution of the implant himself.

To handle communication between the C2 server and the implants, Slimper uses the HTTP protocol.

Slimper uses a heartbeat system using HTTP requests. A heartbeat is a periodic signal generated by the implant to indicate its availability to the C2 server.

13.2 Server and implants

In Slimper, the C2 server is an internet-facing HTTP server that receives requests from the implants.
Slimper only uses GET and POST methods to handle its communication.

The C2 server is controlled by an operator (attacker), communicates the commands to the implants and then receives the results of the executed commands.

To send a command to an implant, the C2 server waits for an HTTP request from the implant and then responds with an HTTP response containing the command from the operator of the C2 server in its body.
If no command is sent, the HTTP response body is empty.

The implants send heartbeat HTTP requests to the C2 Server. A heartbeat is a periodical signal generated to indicate the implant's availability.
If the HTTP response contains a command from the C2 server, the implant is in charge to execute it and return the result in a POST request.

The URIs used in these HTTP requests are built as described in the following sections.

13.2.1 File extensions

- `.js`: Used for heartbeat. It uses the GET method.
- "": No extension is used for carrying data. It uses the POST method.
- `.png`: Indicates that the HTTP requests from the implant will stop and the implant will kill its processes. It uses the GET method.

13.2.2 Dictionaries

```
{
   "poll_ext": ".js",
   "poll_files": [
      "bootstrap",
```

```
   "bootstrap.min",
   "app",
   "array",
   "backbone",
   "script",
   "tracker",
],
"poll_paths": [
   "assets",
   "scripts",
   "script",
   "javascripts",
   "javascript",
   "jscript",
   "embedded"
],
"ext": "",
"files": [
   "login",
   "signin",
   "api",
   "index",
   "admin",
   "register",
   "sign-up"
],
"paths": [
   "api",
   "upload",
   "authenticate",
   "oauth",
   "oauth2",
   "database",
   "namespaces"
],
"kill_ext": ".png",
"kill_files": [
   "favicon",
   "mountain",
   "background",
],
"kill_paths": [
   "static",
   "assets",
   "images",
   "icons",
   "image",
   "pictures",
]
```

}

13.2.3 URI builder

The URIs used in the HTTP request made by the implants have the following format:

- `/{path}/{file}{extension}` Where `{path}`, `{file}`and `{extension}` are replaced by words from the dictionaries above.

The extension of the URI defines the path and file list to be used. For example if the operator sends a `kill`command to the implant then the `.png`extension is used and `{path}`and `{file}`will be respectively replaced by a word from `kill_paths`and `kill_files`: `/{kill_paths}/{kill_files}.png`. Here are some URIs examples that are possible in the previous configuration:

- `/static/favicon.png`
- `/icons/favicon.png`
- `/picture/background.png`

The URIs are limited by the extension that is used. These formats represent which dictionary should be used depending on the extension:

- `/{poll_paths}/{poll_files}.js` (GET method)
  - Examples:
    - `/assets/bootstrap.min.js` is a correct example
    - `/script/tracker.js` is a correct example
    - `/icons/mountain.png` is an incorrect example
- `/{paths}/{files}` (POST method)
  - Examples:
    - `/api/login` is a correct example
    - `/upload/admin` is a correct example
    - `/assets/app` is an incorrect example
- `/{kill_paths}/{kill_files}.png` (GET method)
  - Examples:
    - `/image/mountain.png` is a correct example
    - `/icons/favicon.png` is a correct example
    - `/api/login.png` is an incorrect example

13.2.4 Heartbeats

Heartbeats are periodical signals send to significate its availability. In Slimper the implants sends these heartbeats to significate to the C2 server that it is running and fully available.
When the C2 server receives an heartbeat, the C2 operator is able to send command to the implants in the HTTP response associated to an heartbeat. The heartbeats are by default send with an interval randomly oscillating between 30 seconds to 1 minute.
Heartbeats will use the extension `.js`only and then use the associated dictionaries to build the URIs send in the HTTP requests.

13.3 Behavior

Once the implant is running on the victim's machine, it starts sending HTTP requests to the C2 server with the GET method using the `.js` extension to mimic a heartbeat.

Upon receiving the first heartbeat, the C2 server knows that an implant is available, so it can start responding to the implant with commands.

The operator of the C2 server can then send command through the C2 server interface and the commands will be added in the next HTTP response.

The implant receiving an HTTP response checks if it contains a command. If the HTTP response contains a command, the implant executes the command and returns the result in an HTTP request using the POST method.

If the `kill` command is sent by the operator, the C2 server sends it in an HTTP response to the implant. The implant answers with an HTTP request with GET method using the `.png` extension. The implant then waits for the HTTP response from the C2 server and eventually kills its processes. The communication between the C2 server and the implant stops from the HTTP response of the `.png` GET request.

Slimper Documentation Without Network Example

14 Slimper Command and Control framework.

14.1 Introduction

This file contains the documentation about the Command and Control (C2) framework named Slimper.
This framework is private and developed for educational purposes. This is the only documentation existing on this framework.

A C2 framework allows an attacker to communicate with the compromised victim machines after a successful cyber intrusion.

Implants are executables generated by the C2 framework to create a backdoor. The operator of the C2 framework (attacker) needs to drop, install and start the execution of the implant himself.

To handle communication between the C2 server and the implants, Slimper uses the HTTP protocol.

Slimper uses a heartbeat system using HTTP requests. A heartbeat is a periodic signal generated by the implant to indicate its availability to the C2 server.

14.2 Server and implants

In Slimper, the C2 server is an internet-facing HTTP server that receives requests from the implants.
Slimper only uses GET and POST methods to handle its communication.

The C2 server is controlled by an operator (attacker), communicates the commands to the implants and then receives the results of the executed commands.

To send a command to an implant, the C2 server waits for an HTTP request from the implant and then responds with an HTTP response containing the command from the operator of the C2 server in its body.
If no command is sent, the HTTP response body is empty.

The implants send heartbeat HTTP requests to the C2 Server. A heartbeat is a periodical signal generated to indicate the implant's availability.
If the HTTP response contains a command from the C2 server, the implant is in charge to execute it and return the result in a POST request.

The URIs used in these HTTP requests are built as described in the following sections.

14.2.1 File extensions

- `.js`: Used for heartbeat. It uses the GET method.
- "": No extension is used for carrying data. It uses the POST method.
- `.png`: Indicates that the HTTP requests from the implant will stop and the implant will kill its processes. It uses the GET method.

14.2.2 Dictionaries

```
{
   "poll_ext": ".js",
   "poll_files": [
      "bootstrap",
```

```
   "bootstrap.min",
   "app",
   "array",
   "backbone",
   "script",
   "tracker",
],
"poll_paths": [
   "assets",
   "scripts",
   "script",
   "javascripts",
   "javascript",
   "jscript",
   "embedded"
],
"ext": "",
"files": [
   "login",
   "signin",
   "api",
   "index",
   "admin",
   "register",
   "sign-up"
],
"paths": [
   "api",
   "upload",
   "authenticate",
   "oauth",
   "oauth2",
   "database",
   "namespaces"
],
"kill_ext": ".png",
"kill_files": [
   "favicon",
   "mountain",
   "background",
],
"kill_paths": [
   "static",
   "assets",
   "images",
   "icons",
   "image",
   "pictures",
]
```

}

14.2.3 URI builder

The URIs used in the HTTP request made by the implants have the following format:

- `/{path}/{file}{extension}` Where `{path}`, `{file}`and `{extension}` are replaced by words from the dictionaries above.

The extension of the URI defines the path and file list to be used. For example if the operator sends a `kill`command to the implant then the `.png`extension is used and `{path}`and `{file}`will be respectively replaced by a word from `kill_paths`and `kill_files`: `/{kill_paths}/{kill_files}.png`. Here are some URIs examples that are possible in the previous configuration:

- `/static/favicon.png`
- `/icons/favicon.png`
- `/picture/background.png`

The URIs are limited by the extension that is used. These formats represent which dictionary should be used depending on the extension:

- `/{poll_paths}/{poll_files}.js` (GET method)
  - Examples:
    - `/assets/bootstrap.min.js` is a correct example
    - `/script/tracker.js` is a correct example
    - `/icons/mountain.png` is an incorrect example
- `/{paths}/{files}` (POST method)
  - Examples:
    - `/api/login` is a correct example
    - `/upload/admin` is a correct example
    - `/assets/app` is an incorrect example
- `/{kill_paths}/{kill_files}.png` (GET method)
  - Examples:
    - `/image/mountain.png` is a correct example
    - `/icons/favicon.png` is a correct example
    - `/api/login.png` is an incorrect example

14.3 Behavior

Once the implant is running on the victim's machine, it starts sending HTTP requests to the C2 server with the GET method using the `.js` extension to mimic a heartbeat.

Upon receiving the first heartbeat, the C2 server knows that an implant is available, so it can start responding to the implant with commands.

The operator of the C2 server can then send command through the C2 server interface and the commands will be added in the next HTTP response.

The implant receiving an HTTP response checks if it contains a command. If the HTTP response contains a command, the implant executes the command and returns the result in an HTTP request using the POST method.

If the `kill` command is sent by the operator, the C2 server sends it in an HTTP response to the implant. The implant answers with an HTTP request with GET method using the `.png` extension. The implant then waits for the HTTP response from the C2 server and eventually kills its processes. The communication between the C2 server and the implant stops from the HTTP response of the `.png` GET request.

Slimper Documentation Without Both

15 Slimper Command and Control framework.

15.1 Introduction

This file contains the documentation about the Command and Control (C2) framework named Slimper.
This framework is private and developed for educational purposes. This is the only documentation existing on this framework.

A C2 framework allows an attacker to communicate with the compromised victim machines after a successful cyber intrusion.

Implants are executables generated by the C2 framework to create a backdoor. The operator of the C2 framework (attacker) needs to drop, install and start the execution of the implant himself.

To handle communication between the C2 server and the implants, Slimper uses the HTTP protocol.

Slimper uses a heartbeat system using HTTP requests. A heartbeat is a periodic signal generated by the implant to indicate its availability to the C2 server.

15.2 Server and implants

In Slimper, the C2 server is an internet-facing HTTP server that receives requests from the implants.
Slimper only uses GET and POST methods to handle its communication.

The C2 server is controlled by an operator (attacker), communicates the commands to the implants and then receives the results of the executed commands.

To send a command to an implant, the C2 server waits for an HTTP request from the implant and then responds with an HTTP response containing the command from the operator of the C2 server in its body.
If no command is sent, the HTTP response body is empty.

The implants send heartbeat HTTP requests to the C2 Server. A heartbeat is a periodical signal generated to indicate the implant's availability.
If the HTTP response contains a command from the C2 server, the implant is in charge to execute it and return the result in a POST request.

The URIs used in these HTTP requests are built as described in the following sections.

15.2.1 File extensions

- `.js`: Used for heartbeat. It uses the GET method.
- “: No extension is used for carrying data. It uses the POST method.
- `.png`: Indicates that the HTTP requests from the implant will stop and the implant will kill its processes. It uses the GET method.

15.2.2 Dictionaries

```
{
   "poll_ext": ".js",
   "poll_files": [
      "bootstrap",
```

```
    "bootstrap.min",
    "app",
    "array",
    "backbone",
    "script",
    "tracker",
],
"poll_paths": [
    "assets",
    "scripts",
    "script",
    "javascripts",
    "javascript",
    "jscript",
    "embedded"
],
"ext": "",
"files": [
    "login",
    "signin",
    "api",
    "index",
    "admin",
    "register",
    "sign-up"
],
"paths": [
    "api",
    "upload",
    "authenticate",
    "oauth",
    "oauth2",
    "database",
    "namespaces"
],
"kill_ext": ".png",
"kill_files": [
    "favicon",
    "mountain",
    "background",
],
"kill_paths": [
    "static",
    "assets",
    "images",
    "icons",
    "image",
    "pictures",
]
```

}

15.2.3 URI builder

The URIs used in the HTTP request made by the implants have the following format:

- `/{path}/{file}{extension}` Where `{path}`, `{file}`and `{extension}` are replaced by words from the dictionaries above.

The extension of the URI defines the path and file list to be used. For example if the operator sends a `kill`command to the implant then the `.png`extension is used and `{path}`and `{file}`will be respectively replaced by a word from `kill_paths`and `kill_files`: `/{kill_paths}/{kill_files}.png`. Here are some URIs examples that are possible in the previous configuration:

- `/static/favicon.png`
- `/icons/favicon.png`
- `/picture/background.png`

The URIs are limited by the extension that is used. These formats represent which dictionary should be used depending on the extension:

- `/{poll_paths}/{poll_files}.js` (GET method)
  - Examples:
    - `/assets/bootstrap.min.js` is a correct example
    - `/script/tracker.js` is a correct example
    - `/icons/mountain.png` is an incorrect example
- `/{paths}/{files}` (POST method)
  - Examples:
    - `/api/login` is a correct example
    - `/upload/admin` is a correct example
    - `/assets/app` is an incorrect example
- `/{kill_paths}/{kill_files}.png` (GET method)
  - Examples:
    - `/image/mountain.png` is a correct example
    - `/icons/favicon.png` is a correct example
    - `/api/login.png` is an incorrect example

15.3 Behavior

Once the implant is running on the victim's machine, it starts sending HTTP requests to the C2 server with the GET method using the `.js` extension to mimic a heartbeat.

Upon receiving the first heartbeat, the C2 server knows that an implant is available, so it can start responding to the implant with commands.

The operator of the C2 server can then send command through the C2 server interface and the commands will be added in the next HTTP response.

The implant receiving an HTTP response checks if it contains a command. If the HTTP response contains a command, the implant executes the command and returns the result in an HTTP request using the POST method.

If the `kill` command is sent by the operator, the C2 server sends it in an HTTP response to the implant. The implant answers with an HTTP request with GET method using the `.png` extension. The implant then waits for the HTTP response from the C2 server and eventually kills its processes. The communication between the C2 server and the implant stops from the HTTP response of the `.png` GET request.

15.3.1 Executions examples

Example 1: The operator sends a *kill* command

```
C2 Server ------------------------------  Implant
    |                                        |
    |  <-------------------------------------  | GET /jscript/bootstrap.min.js
    |  ------------------------------------->  | HTTP 200 content:empty
    |  <-------------------------------------  | GET /scripts/array.js
    |  ------------------------------------->  | HTTP 200 content:empty
    |  <-------------------------------------  | GET /javascript/backbone.js
    |  ------------------------------------->  | HTTP 200 content:empty
    |  <-------------------------------------  | GET /embedded/app.js
    |  ------------------------------------->  | HTTP 200 content:"kill"
    |  <-------------------------------------  | GET /images/mountain.png
    |  ------------------------------------->  | HTTP 200 content:empty
                                                    |-> the implant kills its processes
```

Example 2: The operator sends the *ls* command, waits for data and then sends the *kill* command

```
C2 Server ------------------------------  Implant
    |                                        |
    |  <-------------------------------------  | GET /assets/bootstrap.js
    |  ------------------------------------->  | HTTP 200 content:empty
    |  <-------------------------------------  | GET /javascript/app.js
    |  ------------------------------------->  | HTTP 200 content:empty
    |  <-------------------------------------  | GET /javascript/backbone.js
    |  ------------------------------------->  | HTTP 200 content:"ls"
                                                    |-> implant executes ls
    |  <-------------------------------------  | POST /api/login <data results>
    |  ------------------------------------->  | HTTP 200 content:empty
    |  <-------------------------------------  | GET /javascript/app.js
    |  ------------------------------------->  | HTTP 200 content:"kill"
    |  <-------------------------------------  | GET /images/mountain.png
    |  ------------------------------------->  | HTTP 200 content:empty
                                                    |-> the implant kills its processes
```

Example 3: The operator sends the ”ls” and lets the implant run

```
C2 Server ------------------------------  Implant
    |                                        |
```

```
    |  <----------------------------------  | GET /assets/bootstrap.js
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /javascript/backbone.js
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /embedded/tracker.js
    |  ---------------------------------->  | HTTP 200 content:"ls"
                                               |-> implant executes ls
    |  <----------------------------------  | POST /api/login <data results>
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /javascript/app.js
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /javascript/app.js
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /jscripts/array.js
    |  ---------------------------------->  | HTTP 200 content:empty
    ... (continuous heartbeat and response)
```

Example 4: Same as example 3 with inter-requests heartbeat

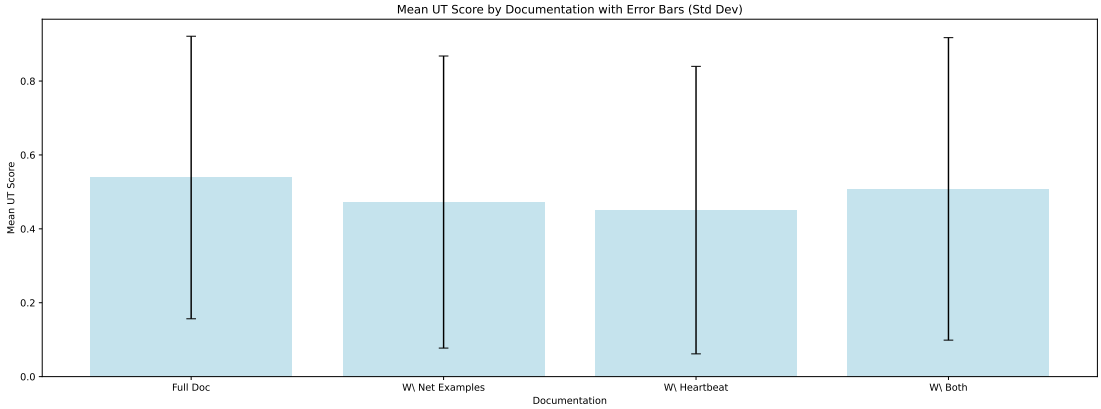
```
C2 Server ------------------------------  Implant
    |                                       |
    |  <----------------------------------  | GET /assets/bootstrap.js
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /javascript/app.js
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /embedded/backbone.js
    |  ---------------------------------->  | HTTP 200 content:"ls"
                                               |-> implant executes ls
    |  <----------------------------------  | GET /scripts/app.js
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | POST /api/login <data results>
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /scripts/app.js
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /javascript/bootstrap.js
    |  ---------------------------------->  | HTTP 200 content:empty
    |  <----------------------------------  | GET /script/tracker.js
    |  ---------------------------------->  | HTTP 200 content:empty
    ... (continuous heartbeat and response)
```

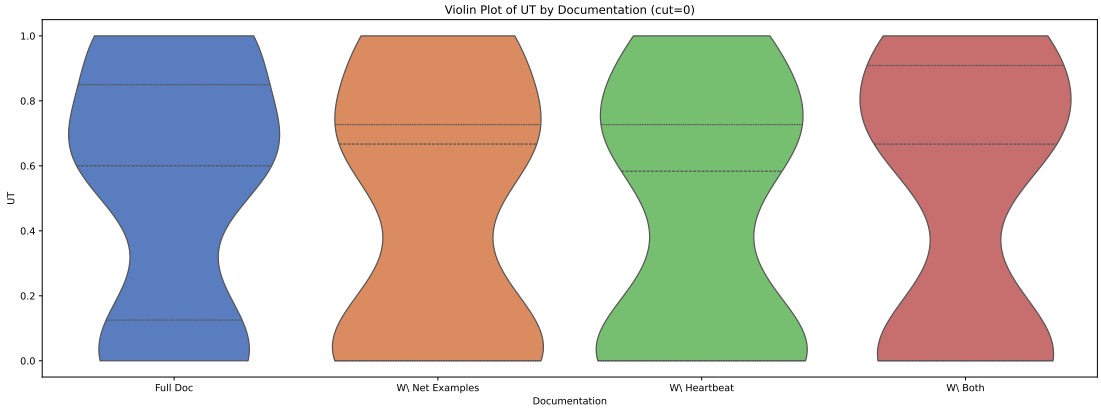
Results of Documentation in Infection Classification

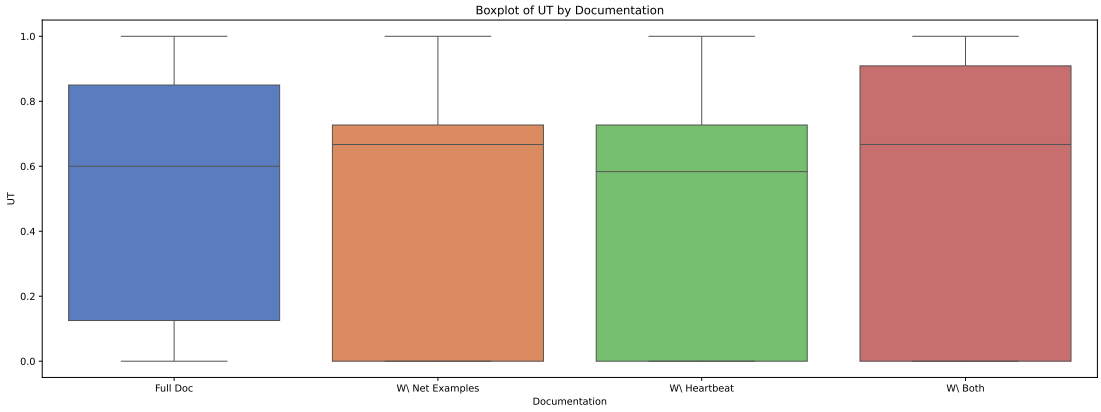
Documentation	True Positive	False Positives	True Negatives	False Negatives
Full Documentation	32	5	15	8
Without Heartbeat	33	6	13	8
Without Net Examples	36	11	9	4
Without Both	34	9	11	6

Violin Plot of UT Scores vs Documentation

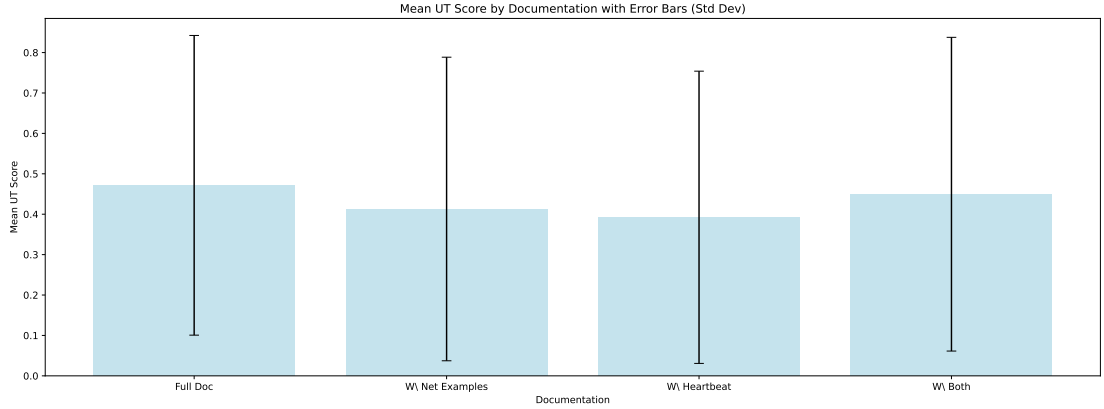
UT Score by Doc

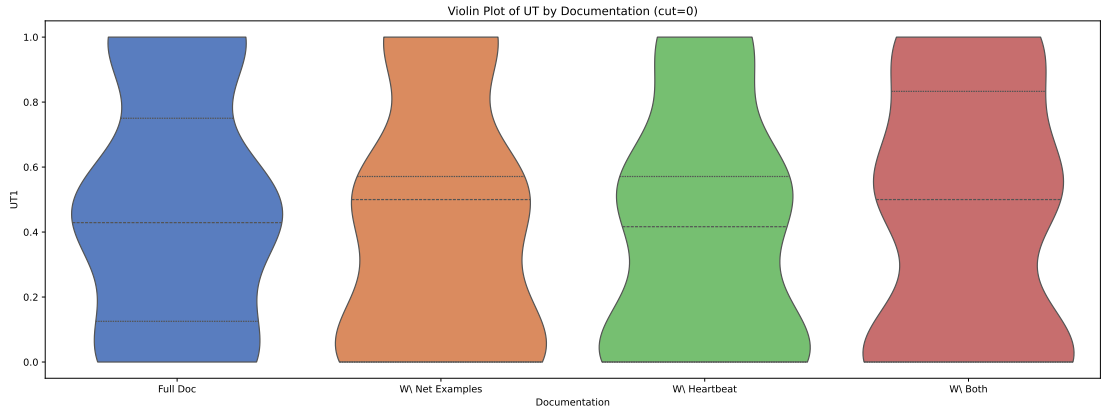


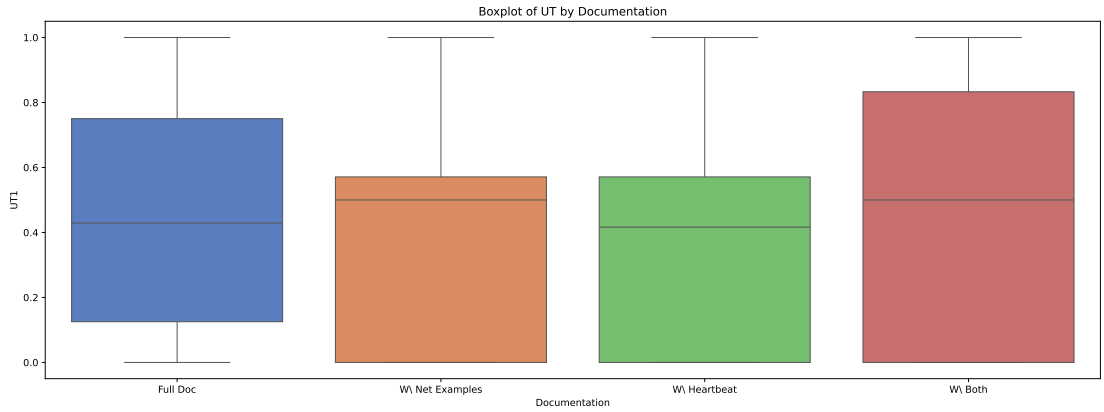




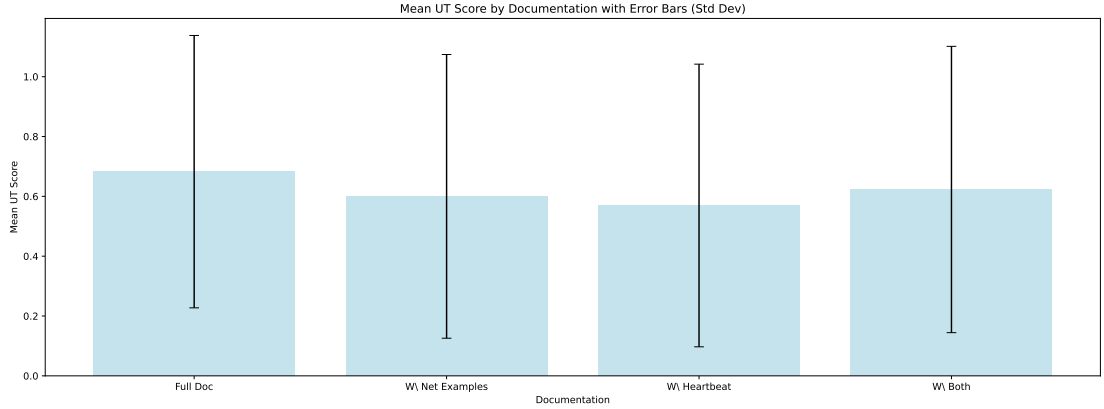
UT1 Score by Doc

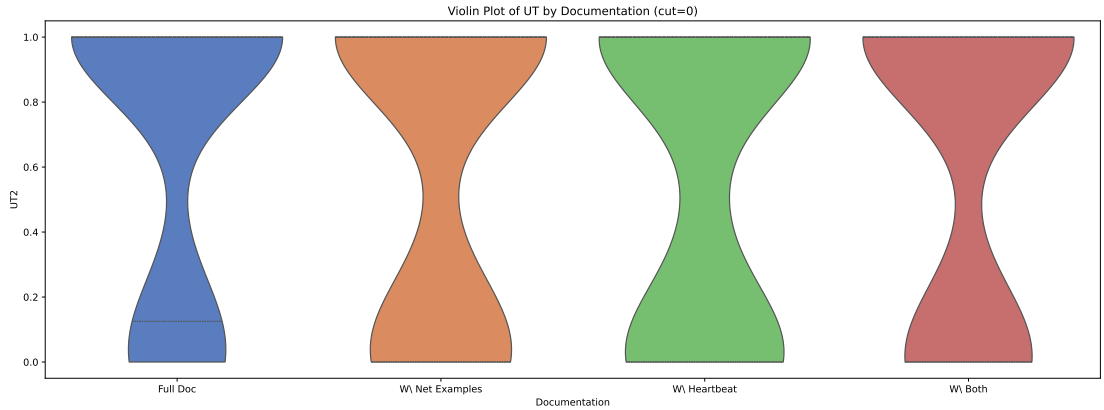


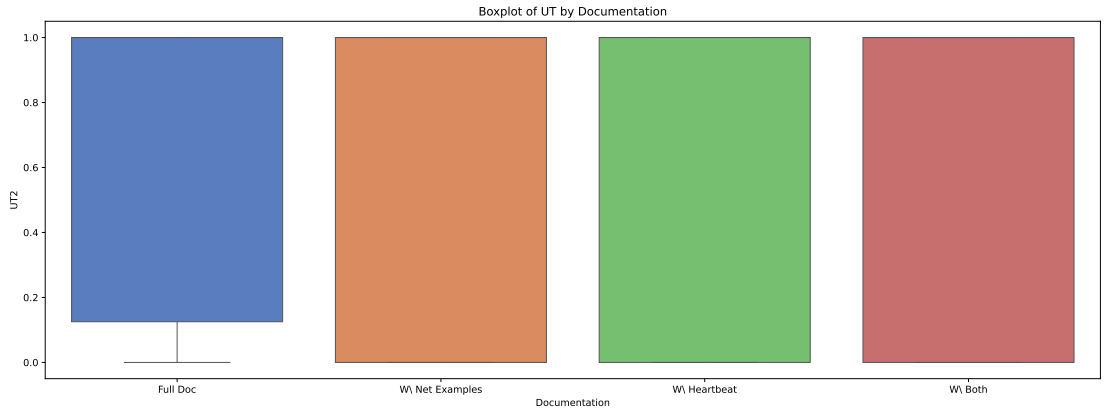




UT2 Score by Doc







Results of File Format in Infection Classification

File Format	True Positive	False Positives	True Negatives	False Negatives
JSON	32	5	15	8
TXT	39	6	14	1

G

UCLouvain Internship Report

Internship Report:
Detection of C2 Framework using LLMs

Hadrien Allegaert 07991800

June 2, 2025

Abstract

My internship took place in the summer 2024. Starting in June and initially planned to end in August. Due to a security clearance issue, the internship ultimately began in July and was extended to conclude on October 1st.
The internship focused on the analysis of Command and Control Frameworks. My Master's Thesis subject was to create an LLM assistant that can detect Cyber Threats. These topics were combined into research on detecting Command and Control Frameworks using LLMs and Prompt Engineering.

Acknowledgment

Before delving into the main subject, I would like to express my gratitude to those who supported me throughout this internship.
First, I want to thank La Défense for allowing me to complete my internship with professionals who support their interns through the whole internship. Secondly I need to thank the team that accompanied me through this internship, Kevin, Antoine and Joren, they always found a way to overpass the different complications we faced.
I especially wanted to thank Joren and his guidance through the whole internship, he teaches me a lot. Without him, this research would have remained an exploration rather than a structured research project.
I also want to thank Christophe Crochet, a PhD student at UCLouvain, who followed and guided me.

Contents

Chapter 1

Introduction

My internship took place between June and September 2024, included. I was working with La Defense. Usually the interns work at the Evere headquarter, but it is mandatory to obtain a security clearance to access the work environment.
Choosing to work with La Défense was an obvious decision for me. La Défense is dedicated to protecting Belgian citizens from both internal and external threats. It has a well-developed infrastructure in cybersecurity and actively explores new technologies and conducts research in the field. It would be a great experience to work in this kind of environment with these experts for a not experimented student.

My first day was a surprise for my supervisor, Kevin, and myself. No trace of my security clearance demand of June 2023 can be found. This was despite a military officer informing me that my security clearance was ready and my secured laptop was prepared. The truth was that my clearance request had been mixed up with that of another intern. Eventually, we learned that it had been lost.
Access to the workplace and the data needed for my initial task, which was an investigation of an active cyber threat actor, are restricted. I cannot attend the internship in physical work and the subject need to be changed.

As my Master's Thesis subject was linked to this internship, and the goal was to create a cyber threat detector based on LLM. My supervisors and I wanted to maintain this connection. A subject which is not restricted is the analyse of Command and Control (C2) Frameworks.
C2 Frameworks are software used by malicious actors to keep a backdoor between a compromised device and the operator (hacker) server. These frameworks typical blend there network traffic within deeply used protocol, as HTTP(S), DNS or mTLS. These are also highly customisable, depending on the need of the operator and the security of the compromised device, as such as the use of stagers to bypass anti viruses.

The internship subject has been changed, as my Master's Thesis subject, in "Detection of Command and Control Frameworks using LLMs: A Proof of Concept". As you understand the internship was in remote work only. Weekly meetings were conducted, both in person and online, sometimes more to follow and guide me through the research process.
The research was autonomous, with supervisors guidance, and leads to good results after a desatrous first month. As it was fully autonomous the first month was more like exploring the capabilities of ChatGPT than making real research. When the supervisors team detected it,

they immediately react and focused me on subtask to improve these research and reach them to an acceptable level for a Master's Thesis.

As C2 Frameworks use sophisticate mechanism to hide their behaviour, we decided to focus the research on HTTP and fictitious, self developed, C2 Framework. This simplified C2 Framework, written in python, is a server client based software using beaconing with the HTTP protocol.

The detection of Slimper by ChatGPT-4o reached the 90% F1-score, despite it was a simple exercise to prove the possibility of an LLM to understand and analyse network trace. The results are pretty limited as I used a low volume of data to test the LLM. The finality of the internship was a physical presentation of the optimal OpenAI's parameters found during the research and the results obtained.

The report is structured into two main chapters.
The first one explains the internship organization and the reasons of the company choice. It also explains the difficulties faced in the beginning. The last two sections of this chapter despite the research subject and a little introspection of the intern month by month.

The second chapter is more practical and shows the methodology and the results obtained during the internship. It also contains a little background knowledge section giving to readers the basics understanding of the following sections.

A last chapter will briefly conclude about the feeling of the autonomous, for the most part, work and the team communication. This chapter will also conclude about the results of the research I made during this internship.

Chapter 2

Internship Organization

2.1 Company Overview

La Défense is the federal organization responsible for managing military activities to defend Belgian citizens. This means that different branches of ***La Défense*** specialize in various missions.

The ***Cyber Command***, the division where I completed my internship, focuses on cyber defence activities. They are responsible for cybersecurity, network security, and weaponry systems security. The Cyber Command also collects data for the `SGRS` (***Service Général de Renseignement et de Sécurité***). The `SGRS` is a military intelligence agency and can essentially be compared to the CIA, the well-known American counterpart to the Belgian intelligence service.

The mission of Cyber Command is to safeguard Belgian citizens from external cyberattacks. Naturally, various missions fall under the umbrella of Cyber Intelligence within the scope of Belgium's defensive activities. These missions range from monitoring suspicious activities to blocking any network access in Belgium from the attacker's perspective. Their responsibilities also include researching tools employed by hackers.

The usual workplace of the ***Cyber Command*** division is the Evere Headquarters. It is a wide military base where Belgian army divisions and ***NATO*** are represented. There are many European and ***NATO*** activities at the Evere HQ, and access to the building is therefore restricted. To enter, a security clearance, which can take up to a year to approve, is required.

Unfortunately, my security clearance was lost. This situation compromised my access to the workplace, as explained in the introduction. The team quickly adapted by proposing a remote work arrangement for the internship with weekly meetings to track research progress. These meetings were usually held in person at the Evere HQ on Mondays. However, occasionally, due to the complex schedules of my supervisor team, or for additional meetings as needed, Microsoft Teams was occasionally used for remote meetings.

2.1.1 Reasons behind an internship at La Defense

The choice to do my internship at La Défense was clear to me, as working for the defense of Belgian citizens adds a profound sense of purpose to my work. La Défense, as the Belgian Army,

ensures the safety and security of our nation, and the Cyber Command plays a vital role within this mission by protecting networks and collecting critical intelligence for the SGRS. Contributing to this effort, even indirectly, gave my work meaning and made me proud to support an organization dedicated to safeguarding the country from evolving cyber threats.

While I value the mission of La Défense, my experience with the loss of my security clearance highlighted some organizational challenges. Friends working there have shared similar concerns, which impacted my overall impression. A more efficient process could have made the experience significantly more rewarding and motivating.

2.1.2 Application and Interview Process

The application process for an internship at La Défense begins with a request submitted to a Defense House. I sent a request to the Namur office a demand to make an internship in a ICT (Information and Communication Technology) team.

The first request was rejected due to insufficient time to process the security clearance. TI then submitted a second request for the following year and this time the demand was forwarded to the HR department of ***La Défense***.

Several months after that I was contacted by a military officer working in the Cyber Command division. I proposed me an interview which I accepted.
The interview took place in the Evere HQ in a unclassified meeting room. It was a very familiar interview with question on my study and why I applied there. Several subjects were proposed and I chose the investigation on cyber threat actor. I received the administrative papers immediately after the interview. I filled them at home and send them the next day.

There was no contact until March 2024 when I asked the military officer I had as main contact to know if the demand run smoothly. From this point until the internship began, there was occasional contact via phone and email, to schedule the start of the internship. We then learnt that my security clearance was lost and I met the team I worked within the next week Monday.

2.1.3 Supervisor Team

The team I worked within was under the supervision of Kevin. He is an expert in cyber security. From Antoine words "It exists only 3 guys with the same level as Kevin in cyber security in the whole Belgium".

Antoine served as my main contact within the team. He was the intern that worked before me on the Command and Control Frameworks analyse. Now he works in the Evere HQ with the Cyber Command division as a civilian. Most of the personnel at Cyber Command are civilians. Antoine has a degree in computer sciences from high school.

The last I met is Joren. An expert in artificial intelligence applied to cyber security. He was my mentor during the internship. He setup schedule and tasks to guide me. Gave me hints to help and direct me toward a possible solution.

he team members appeared to work autonomously on different projects, but since I worked remotely, I cannot confirm how the team operated at the Evere HQ.

2.2 Internship Subjects & Goals

2.2.1 Original Subject

Before we known that my security clearance was lost, my internship subject was to investigate on a real cyber threat actor. It means that I would learn to use tools and protected data to track a hacker. Most of these tool are publicly available, but these are customisable and the Cyber Command, surely I don't know, has access to private tool to achieve this task.

As I study the cybersecurity, working with experts in the Cyber Command division would be an awesome experience, and a great opportunity to learn from them. Tracking down cyber actor is a dream job for me.

The problem with my security clearance lead to a restricted access to the protected data and private tool Cyber Command uses. Then the task is impossible. The choice to change my subject has been done during the next week I was supposed to start.

2.2.2 Final Subject

Another subject proposed during the interview was Command and Control Frameworks analyse. Another intern did a previous job on it and now work within the team I was supervised by.
This subject originally asks for static analyse of network traces, and detect how the well known C2 Frameworks, Sliver and Mythic, are updated.

As my Master's Thesis was focused on using LLM to detect Cyber Threat, I proposed to merge both idea. The supervisor team accepted and I start the research on the LLM capabilities.

The final subject was to prove the LLMs is capable, or not, to detect Command and Control Framework, based on network traces taken from compromised devices.

2.2.3 Internship Goal

My mission was to make research on one of these tool, named Command and Control (C2) Frameworks. It is a family of software allowing attackers to keep an access to an already compromised device. The implant, the software installed in the compromised device, send beacons to signify its availability to the server. The operator can send commands and exfiltrate data through the server communicating with the installed implant. More on Command an Control Framework will be explained in subsection 3.1.2.

To achieve this task, I used prompt engineering, explained in the Large Language Model subsection 3.1.1, to customize the OpenAI's model: GPT-4o.
OpenAI's models are really advanced neural network pre trained and publicly available. They were trained on huge amount of data to achieve a high level of text comprehension and generation. The whole experimental setup and the results are given in the chapter 3.

The goal of this internship was to prove LLMs are capable to detect C2 Frameworks based on a network trace analyse. Different factor can change the results and need to be explored. The most obvious one is the prompt, acting like the instructions given to the LLM. Less obvious parameters can also influences the result.

For example a parameter named Temperature change the behaviour of GPT. Lower the temperature is and more creative, but also more chance to hallucinate. And vice versa, higher the temperature give creative restriction and lead to determinism in the highest setup. others parameters need to be explored, as different prompt engineering techniques, the file format of the network trace, and so on.
Fixing these parameters was my main task. The way I did it is explained in the chapter 3.

2.3 Intern Introspection

2.3.1 First Month

The first month of my internship was quite challenging, as I had to organize my work independently with minimal guidance. I took the initiative to run tests and explore scientific literature to understand the state-of-the-art in using LLMs for detection. The main challenge was that LLM technology is relatively new, and most of the literature is highly theoretical. Practical applications exist but are rare in the field of cybersecurity. For instance, a recent paper discusses malware detection based on signatures, but there is no research on network trace analysis. However, in fields like healthcare, LLMs have been extensively used for tasks such as diagnosing illnesses from symptom lists or detecting early-stage breast cancer from medical scans.

During this first month, I was highly motivated and eager to collaborate with experts and learn from them. However, the lack of direction decreased my motivation, causing my efforts to become scattered and unproductive.

Fortunately, Joren stepped in with a more structured approach, focusing on slower but steady progress by verifying each step. He also organized the project into more manageable subtasks, which made the work more accessible and effective.

2.3.2 Second Month

The second month was much more structured, with the research now focused on understanding what an LLM is capable of interpreting when performing specific tasks. For example, I explored how it detects URLs in a network trace or what it comprehends about the HTTP protocol.
Once I had verified the LLM's understanding, I proposed a testing pipeline that could run on the OpenAI API, funded at my own expense, to determine the optimal parameters.

To simplify the task, considering that this was a proof of concept and not yet a fully functional tool, my supervising team asked me to develop a highly simplified C2 Framework. This educational framework, named Slimper, served as a reference for testing the LLM. Since I didn't have access to the restricted data from the Cyber Command, I generated my own network traces. Using Slimper simplified this process, as real C2 Frameworks, such as Sliver, often modify their behaviour when they detect they are in a virtual or restricted network environment.

This second month taught me how to better organize my work, thanks to the structure introduced by Joren. I also learned to regularly communicate my results in the form of written reports, which were sent via email. This practice slightly improved my English writing skills.
However, I still faced significant challenges in expressing myself in English, both verbally and in writing.

2.3.3 Third Month

During the third month, my focus was on finalizing the detection prompt, which worked with network traces, performing tests, and fine-tuning the optimal parameters for this detection. The prompt was based on the Long Context Prompt approach and incorporated a documentation file that detailed the functionality of Slimper. As part of my Master's Thesis, the next step will involve developing a more advanced Command and Control Framework, called Mimic. This new framework will allow testing whether the LLM can go beyond detection to classification—that is, identifying which specific C2 Framework has infected a machine.

Throughout this third month, I submitted more frequent reports, detailing each test and receiving feedback from my supervisors. This process highlighted the importance of regular communication as a critical soft skill, especially when working remotely. Clear and consistent updates kept the team aligned and helped refine the research direction effectively.

Through this experience, I honed my organizational skills, embraced feedback, and strengthened my ability to bridge theory and practice. These soft skills, alongside the technical knowledge gained, are invaluable takeaways from this internship.

2.3.4 Personal Reflection

This internship significantly strengthened both my technical expertise and my understanding of what I want to achieve professionally. Developing the Slimper framework and fine-tuning LLM prompts gave me hands-on experience in applying AI to real-world cybersecurity challenges. It confirmed my interest in working at the intersection of AI and cybersecurity, where I can contribute to building innovative solutions to detect and mitigate cyber threats. While LLMs can be confusing initially, understanding how to customize them effectively became both engaging and rewarding.

On a personal level, organizing my tasks during remote work and regularly reporting my progress helped me develop better time management and communication skills. Receiving feedback from my supervisors showed me the importance of clear and consistent communication, especially in research environments. These experiences will guide me as I continue my Master's Thesis and prepare for a career that combines research with practical applications in cybersecurity.

Chapter 3

Research and Methodology

3.1 Background Knowledge

3.1.1 Large Language Model

Large Language Models (LLMs) are cutting-edge artificial intelligence systems designed to process and generate human-like text by leveraging vast amounts of training data. Models like OpenAI's GPT-4 utilize deep neural network architectures with billions of parameters to interpret context, identify patterns, and produce coherent outputs. Their versatility allows them to adapt to a wide range of tasks, including text summarization, question answering, and data interpretation, through carefully crafted prompts. Despite their advancements, LLMs have limitations, including a tendency to generate incorrect information or perpetuate biases from their training data. In this internship, the focus was to explore how LLMs could analyze network traces, an innovative application of this technology within cybersecurity, where practical examples remain scarce.

Prompt engineering is the process of crafting precise and structured inputs to guide the behavior of an LLM toward achieving specific tasks. By carefully designing prompts, the LLM can provide accurate and contextually relevant responses even for complex challenges. A particular technique used during this internship was the Long Context Prompt approach, which allows the LLM to process extensive inputs, such as network traces combined with documentation. This method helps the model retain important details across lengthy contexts, making it suitable for cybersecurity applications like C2 framework detection.

This foundational understanding of LLM capabilities sets the stage for their application in detecting and analyzing malicious behaviors within Command and Control (C2) frameworks.

3.1.2 Command and Control Framework

Command and Control (C2) Frameworks are tools employed by malicious actors to maintain remote control over compromised devices. These frameworks establish a communication link between the attacker's server and the infected system, enabling various malicious activities such as data exfiltration, command execution, or deploying additional malware. Modern C2 frameworks, such as Sliver and Mythic, are highly adaptable, often using common communication protocols like HTTP, DNS, or mTLS to mask their traffic within legitimate network activities.

Their ability to evade detection presents a significant challenge for defenders. As part of this internship, the simplified C2 framework, Slimper, was developed to mimic key aspects of these tools in a controlled environment, serving as a testbed for evaluating LLMs' capabilities in detecting and analyzing C2-related activity.

3.2 Testing Methodology

3.2.1 Data Generation

Infected dataset samples were generated using two virtual machines. The machine where is installed the Command and Control Framework's server is a Kali Linux distribution. The machine where the implant is installed is a windows dev machine. They communicate using a host-only channels, ensuring the VMs cannot communicate with the web. WireShark is started and the network trace begin when the implant send the first beacon.
The data in the safe set are network traces of a user browsing HTTP websites on windows.

3.2.2 Testing Workflow

To find optimal parameters I designed a test pipeline that can be seen in two part. The deterministic part used as a reference and the generated part from the LLM.

The deterministic part is a python script that fulfil the same template as the LLM but using pre-labelled files and deterministic detection using for example regular expressions.

Once the LLM has answered the template is split into three parts. The first part is the classification state as infected or safe file. The second part is the URLs list that can be imputed to the Slimper C2 Framework detected by the LLM.

These two part will be scored separately because it is more important to make a good classification than a good URL detection.

3.2.3 Scoring Methods

The classification part will be compared to the reference. If they match the score is 1 for this test. Either the score is 0.
Each tests has been run 10 times to ensures correct conclusion. Has there is a series of binary it is easy to compare this has true positive, false positive, true negative and false negative. And then compute the Precision, Recall and F1-Score well known metrics.

For the URLs list part it is more difficult has it is not just a binary comparison but a comparison of two list of string. The LLMs can also answer with the URI path only instead of the URL. Taking that in account I designed, what I called UT score. It is a simple score inspired of the Precision, Recall and F1-Score to score a list coverage. There is three scores in the UT scores: The first one shows the coverage of the reference list. The second one shows the coverage of the generated list. And the last one is an harmonic mean of both.

3.3 Results

The first step was to understand the impact of the context and the parameters, as network traces format and temperature, on the results.

The tests showed that LLM understand better network traces when it is in a human readable format. Despite OpenAI's limitation to use binary format as packet capture (pcap) and comma separated value (csv), the LLM shown great capabilities to understand the JSON and plain text formats. With a slight better score using the plain text format. Demonstrating the LLM preference for the text instead of structured data as in the JSON format.

The test on the temperature shown that limiting the creativity of the LLM by reaching the temperature to a higher level increase the classification score until the 0.4 threshold. This shown that for the detection task a temperature of 0.4 is optimal.

The results on the classification, as infected or safe network trace, shows a high precision of 87% with a F1-score of 92%. It should seem a bit high but as the LLM has not be trained for this kind of task, overfitting is impossible. These results can be imputed to the simplicity of the Slimper Command and Control Framework.

The UT scores were low with a mean around 50% of reference list coverage with an harmonic score at 70%. These scores shown that the detection is not made with the URL and the dictionaries provided in the documentation of Slimper.
The way the LLM detect it remains unknowns.

3.4 Future Works

These research will continue for my Master's Thesis. Another educational C2 Framework will be implemented and tested against the LLM.
This new C2 Framework will mimic a website, working in HTTP. It hides its communication within the HTML page sent. This behaviour allows the HTTP protocol to work normally using modern browser feature as caching. Mimic these behaviours should add difficulties for an LLM detection.

The next step should be to add the documentation of a real Command and Control Framework. Generate network traces from the communication between the server and the implant. Eventually test the LLM against these new data.

Chapter 4

Conclusion

This internship was intended to be an experience with experts in the field. Instead of that it has been an introduction to research and development in remote work.

However, the guidance from the supervisor team taught me what was real research and value linked to it. Reproducibility and transparency are important skills when making research. This internship also gave me technical experience with Command and Control Framework and Prompt Engineering.

Additionally, several soft skills, such as communication, are highly important. To communicate results but also to communicate difficulties faced or ask question when needed. The soft skill I need to improve the most is scheduling tasks. Dividing tasks into subtasks and adhering to a schedule. This soft skill evolve several skills. One of the most important and which is my nemesis is the tasks timing prevision. I mean predict the time a task can take and take a bit of extra time for possible bugs correction or other difficulties.

In the technical field, this research is the first step of a proof of concept showing LLM capabilities in cyber threat detection. It is a small stone for now but this concept will be enlarged in future work. And maybe someday, become a standard in early cyber threats detection.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**