Toward Reliable and Adaptive Large Language Models in the Cybersecurity Domain

Marco Simoni
CANDIDATE
Sapienza University of Rome

Andrea Saracino
ADVISOR
Department of Excellence in Robotics and AI, TeCIP, Scuola Superiore Sant'Anna, Pisa

Paolo Mori
CO-ADVISOR
Institute of Informatics and Telematics, National Research Council of Italy, Pisa

Abstract

Large Language Models (LLMs) exhibit not only strong reasoning abilities but also a remarkable capacity for decision support in knowledge-intensive domains; however, applying them to cybersecurity demands reliability, interpretability, and continuous adaptability, qualities that general-purpose models still lack. This work aims to make LLMs reliable and adaptive tools across five interconnected domains that span the entire cybersecurity lifecycle: *malware and threat analysis*, *cyber threat intelligence*, *vulnerability detection*, *access control*, and *misinformation*.

The research begins by employing Transformer models to learn behavioral patterns encoded in API call sequences. Although these models perform well in detecting and categorizing malicious activity, they struggle to capture higher-level semantic relationships between threats, tactics, and defences. To address this limitation, the work introduces knowledge graphs that connect malware samples, attack techniques, vulnerabilities, and countermeasures, enabling dynamic updates and multi-hop connections across entities. Building on this, a retrieval augmented assistant is developed to integrate both structured graph data and unstructured textual sources, thereby reducing hallucinations and improving factual reliability. The system is then extended with specialized, task-oriented modules that translate analytical insight into operational capability: a reinforcement learning–based vulnerability detector, a natural language translator for access control policy generation, and a misinformation engine for both generation and detection.

Finally, the thesis focuses on improving the reasoning process itself, introducing methods that generate more concise, stable, and interpretable output while reducing computational cost. Overall, the research demonstrates that reliability in cybersecurity does not arise from a single universal model but from an ecosystem of task-aware LLMs built on structured knowledge, retrieval, and optimized reasoning.

Contents

1 Introduction

Large Language Models (LLMs) have demonstrated substantial capabilities across many domains, supporting reasoning, knowledge retrieval, and complex decision support. Their ability to process natural language and generate coherent outputs makes them appealing for tasks that demand both flexibility and scalability. However, the direct application of LLMs to cybersecurity introduces a set of unique challenges. Unlike other fields, cybersecurity is vast, heterogeneous, and constantly evolving. It spans activities as diverse as the analysis of malicious code, the detection of vulnerabilities, and the enforcement of access control, each requiring precise technical expertise and reliable reasoning. General-purpose LLMs, even when powerful, tend to hallucinate, struggle to adapt across contexts, and often produce verbose or unfaithful explanations. Such limitations are unacceptable in security-critical contexts, where accuracy and trustworthiness are essential.

The motivation of this thesis is therefore to move beyond generic models and to design LLM-based solutions tailored to the needs of cybersecurity. Achieving trustworthiness requires approaches that are grounded in structured knowledge, resilient to adversarial manipulation, and capable of producing outcomes that are both accurate and actionable. Rather than considering cybersecurity as a single monolithic discipline, the thesis focuses on five specific fields, shown in Figure 1, that capture its diversity: **malware analysis**, **cyber threat intelligence**, **vulnerability detection**, **misinformation**, and **access control**.

Each of these fields represents a concrete application domain in which LLMs face distinct challenges. In malware analysis, the work advances from sequence-based modeling of malicious behavior toward graph-based representations that capture higher-order semantics. In cyber threat intelligence, retrieval-augmented architectures are designed to integrate heterogeneous knowledge bases and provide timely, reliable insights. For vulnerability detection, reinforcement learning and reasoning optimization are explored to improve stability, precision, and interpretability in code analysis. The study of misinformation investigates how structured knowledge can be exploited to generate and detect adversarial content, exposing weaknesses in both human and machine reasoning. Finally, the domain of access control translates natural language instructions into enforceable policies, supporting deployment in constrained environments such as IoT, smart homes, and healthcare.

Taken together, these fields demonstrate how LLMs can develop from general-purpose tools into specialised and reliable solutions for cybersecurity. The broader contribution of the thesis is to show that progress in this area does not depend on a universal model, but on a principled set of task-specific methods and systems

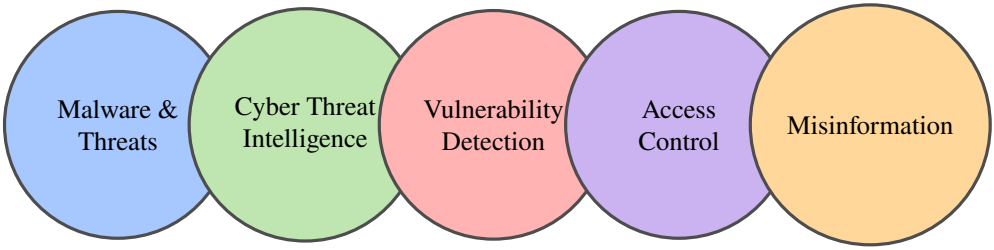


Figure 1: The 5 targeted Cybersecurity application domains.

tailored to the requirements of security-critical contexts.

1.1 Research Map

The research evolved through a series of steps, each addressing the limits of the previous one. This gradual progression reflects how the work naturally adapted to the growing challenges of real-world cybersecurity.

The research began in the malware analysis domain. Transformer models trained on API call traces proved effective at recognizing behavioral patterns and detecting malicious applications, achieving strong detection and threat categorization accuracy. However, these models are not fully reliable in real-world conditions. They need frequent retraining to keep up with the rapid evolution of malware families and, more importantly, lacked the ability to provide a broader understanding of malicious behavior. A model based only on sequences (API traces) cannot explain how a specific API trace relates to higher-level tactics, techniques, or the overall goals of an attacker. This limitation led to the first research question: **Q1:** *How can we move beyond sequences to a representation that logically connects malware to their characteristics, tactics, and related knowledge in a way that remains dynamically updated?*

This led to the exploration of knowledge graphs. A malware and threat knowledge graph offers a structured way to represent cybersecurity information, where entities such as malware samples, attack techniques, vulnerabilities, and defense measures are modeled as nodes, and their relationships are represented as edges. By combining multiple cybersecurity frameworks and continuously updating with real-world reports, these graphs provide the dynamic and flexible foundation that sequence-based models lack. More importantly, they are not just storage systems, they enable correlations across entities, supporting multi-hop connections and cross-framework analysis, such as linking a malware sample to both the MITRE

ATT&CK technique it uses and the CVE vulnerability it exploits.

However, a knowledge graph alone is not enough. It can organize and provide context, but it cannot directly help analysts answer complex questions in natural language, without using the *Retrieval-Augmented Generation* paradigm. This led to the second research question: **Q2:** *What kind of retrieval-augmented generation (RAG) system can exploit this structured knowledge, and what capabilities must it have, for example, handling multi-hop queries (i.e., questions that require reasoning across multiple entities and relationships), reducing hallucinations, and ensuring reliable and timely responses?* A RAG system combines a large language model with an external retrieval component. Before generating a response, the model retrieves relevant information from sources like graphs or document collections and grounds its answer in that evidence. In cybersecurity, this approach is crucial because static models cannot keep up with the fast-changing nature of threats. Instead, models must be able to dynamically access and use up-to-date external knowledge.

Building such a RAG assistant was the next step. Nevertheless, a general-purpose RAG remains inherently limited: if asked to evaluate a piece of source code, it cannot determine whether the code is vulnerable unless the exact case is already represented in the knowledge base. Similarly, if confronted with a misinformation scenario, the system cannot generate adversarial yet plausible counterexamples without task-specific guidance. This means that retrieval alone is not sufficient. To be useful in diverse cybersecurity contexts, the RAG must be complemented with external, specialized tools that are tailored to specific tasks rather than comprehensive knowledge alone. Such tools enhance the RAG framework by adding executable capabilities, allowing it to perform concrete tasks: for instance, a vulnerability detector powered by reinforcement learning or a policy generator that translates natural language commands into enforceable access control rules. This raised the third research question: **Q3:** *How can RAG systems be extended with intelligent task-specific modules, for instance, engines for misinformation stress testing, natural language translation into enforceable access control policies, or reinforcement learning for vulnerability detection, so that they can provide actionable and precise assistance in operational scenarios?*

Finally, once such systems are in place, an overarching issue emerges: large language models tend to be verbose, unstable, and resource-intensive. Their reasoning often drifts, producing outputs that are difficult to interpret, inconsistent across queries, and costly to compute due to token inefficiency. This is particularly problematic in cybersecurity, where reliability and stability are essential for real-world adoption. If a model produces two different explanations for the same

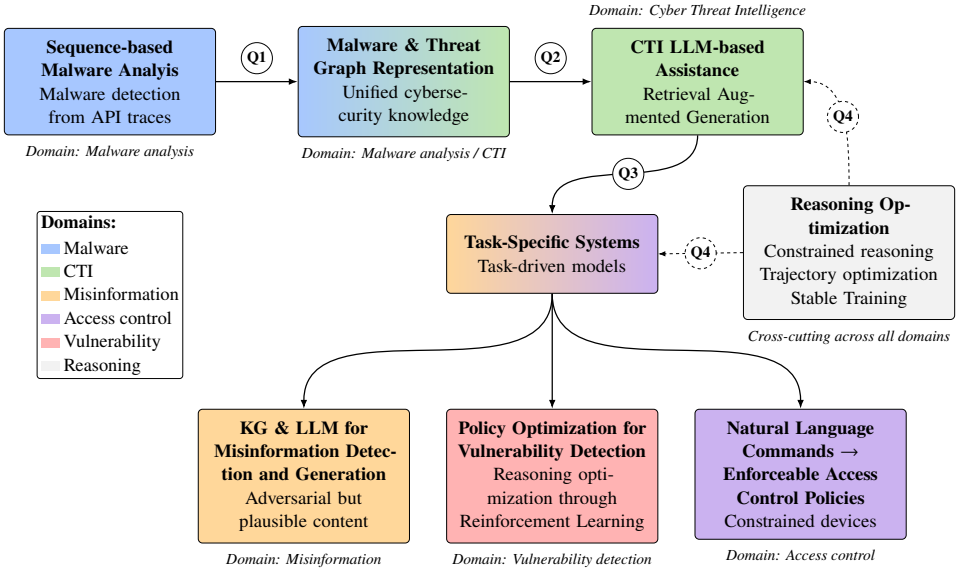


Figure 2: High-level overview of the research map.

behavior, analysts cannot rely on it to support decision-making. This motivated the final research question: **Q4:** *How can the reasoning process itself be improved, making it more concise, interpretable, and stable, while reducing the number of tokens required and ensuring the robustness of model predictions?* This involves investigating methods to constrain the reasoning process, to optimize the trajectory of model predictions and to stabilize training so that outputs become both more efficient and more trustworthy.

Together, these stages form the research path followed in this work. The work starts with Transformer-based models for malware detection, moves to knowledge graphs for contextual understanding, develops retrieval-augmented systems for dynamic cyber threat analysis, integrates task-specific modules for practical applications, and finally focuses on reasoning optimization to improve robustness and efficiency. Each step directly builds on the limitations of the previous one, creating a unified pathway toward transforming large language models into reliable, adaptive, and task-oriented tools for cybersecurity. Figure 2 summarizes this progression, showing the main domains, transitions, and research questions that structure the work.

1.2 Contributions

The contributions of this work follow the methodological trajectory outlined in the Research Map. Figure 3 illustrates the contributions of each research stage, which are explained below.

Contributions of Sequence-based Malware Analysis. The first stage of the research explored transformer-based approaches to malware detection and categorization. Its contributions can be summarized as follows:

- **C1:** Demonstrated the effectiveness of *Transformer-based models in capturing behavioral patterns from malware API traces*, achieving high detection and threat categorization accuracy.
- **C2:** Designed a *two-level analysis pipeline* that enabled both malware detection and categorization into representative families.
- **C3:** Discussed the *limitations of sequence-only approaches*, which require frequent retraining and cannot capture the full semantics of complex attacks.

These findings established the basis for moving toward richer and more dynamic knowledge structures.

Contributions of Malware & Threat Graph Representation. The second stage addressed these limitations by introducing unified graph-based representations and their challenges for integrating them in retrieval-augmented systems. The contributions can be summarized as follows:

- **C4:** Constructed a *dynamic knowledge structure* capable of connecting heterogeneous cybersecurity entities within a coherent framework.
- **C5:** Demonstrated how *graph-based representations* enable richer contextualization and correlation across diverse sources of cybersecurity data.
- **C6:** Identified key *challenges* and *possible solutions* in applying retrieval-augmented systems to cybersecurity, such as factual consistency, multi-hop reasoning, and efficiency, and proposed strategies to address them.

Contributions of CTI-LLM-based Assistance. The third stage focused on building retrieval-augmented generation (RAG) assistants for cybersecurity, following the guidelines identified in the previous research part. The contributions can be summarized as follows:

- **C7:** Designed *domain-specific RAG systems* that move beyond generic language models to provide targeted assistance in cyber threat intelligence.
- **C8:** Showed how retrieval mechanisms grounded in both structured and unstructured sources *improve the reliability, contextualization, and adaptability of responses*.
- **C9:** Demonstrated the role of RAG-based assistants in bridging the gap between raw cybersecurity data and *actionable insights*.

Contributions of KG & LLM for Misinformation Detection and Generation. The research extended to the domain of misinformation, where knowledge graphs and LLMs were combined to study the structured generation and detection of false content. The contributions can be summarized as follows:

- **C10:** Proposed a systematic approach to *generating misinformation* that is both plausible and scalable, enabling controlled stress-testing of detection systems.
- **C11:** Demonstrated how structured *knowledge can guide the creation of adversarial but coherent content, exposing weaknesses in existing detection models*.
- **C12:** Provided insights into the *limitations of LLMs for misinformation detection* and the need for more robust approaches.

Contributions of Natural Language Commands → Enforceable Access Control Policies. In parallel, the research investigated the translation of natural language commands into enforceable access and usage control policies. The contributions can be summarized as follows:

- **C13:** Developed a framework for *deriving access and usage control policies directly from natural language*, bridging the gap between human commands and machine-enforceable rules.

- **C14:** Demonstrated how lightweight models can be adapted to operate efficiently on constrained devices, enabling *on-device policy generation*.
- **C15:** Showed the feasibility of *generalizing policy derivation across domains such as smart homes, offices, and healthcare*.

Contributions of Policy Optimization for Vulnerability Detection. The research also advanced vulnerability detection by improving the reliability of LLMs in analyzing software flaws. The contributions can be summarized as follows:

- **C16:** Provided one of the first systematic studies on the reasoning behavior of LLMs in vulnerability detection, *highlighting their limitations in consistency and interpretability*.
- **C17:** Introduced reinforcement learning, specifically Group Relative Policy Optimization (GRPO), as a method to *shape model behavior beyond supervised fine-tuning*.
- **C18:** Demonstrated that *structured reward functions can simultaneously improve accuracy, reasoning stability, and robustness across diverse datasets*.

Contributions of Reasoning Optimization. The final stage of this research focused on enhancing the reasoning capabilities of large language models themselves. While these studies are general in nature and not exclusively centered on cybersecurity, they address fundamental reasoning optimization strategies that can be transferred to this domain. Due to the current lack of dedicated cybersecurity reasoning datasets, the models were optimized using mathematical reasoning benchmarks, which serve as a proxy for improving structured and logical inference relevant to cybersecurity tasks. Its contributions can be summarized as follows:

- **C20:** Introduced methods to constrain and optimize reasoning, producing outputs that are more concise, interpretable, and cost-efficient.
- **C21:** Proposed *Group-relative Trajectory-based Policy Optimization algorithm* to *stabilize training and mitigate collapse phenomena in reinforcement learning*, ensuring robust reasoning behavior.
- **C22:** Demonstrated that refined *reasoning control improves efficiency and reliability, strengthening the applicability of LLM-based systems*.

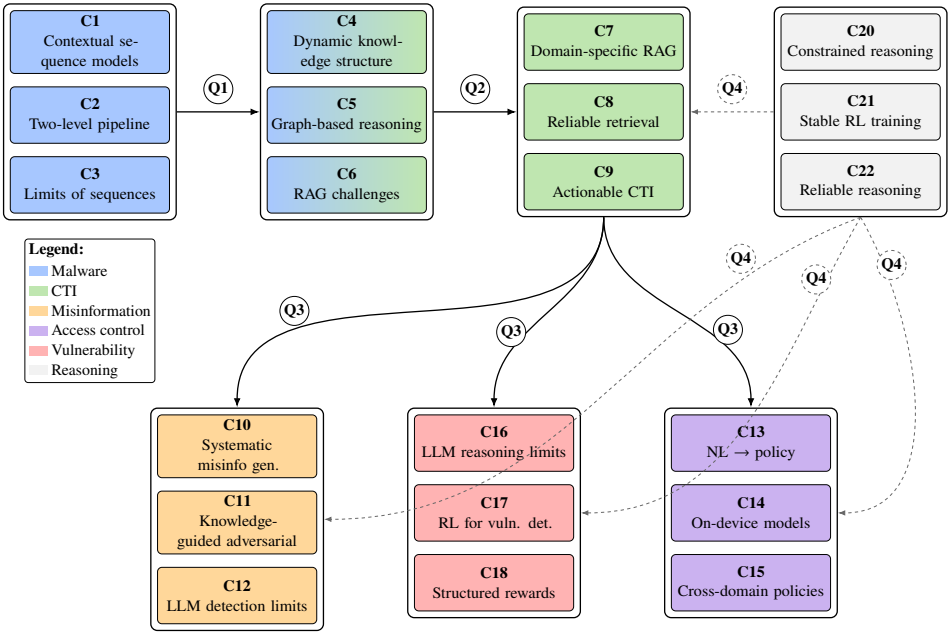


Figure 3: Contribution map grouped by research stage.

1.3 Thesis Structure

The structure of the work follows the methodological trajectory of the Research Map. Figure 4 shows which contribution each chapter covers. Below, there is a brief description of each chapter:

Chapter 2 surveys the state of the art across sequence-based malware analysis, knowledge graphs for CTI, RAG systems, and reasoning optimization for LLMs.

Chapter 3 applies Transformer models to Android API-call traces for detection and family categorization.

Chapter 4 moves beyond sequences via a STIX-aligned knowledge graph called *MATRIX* that integrates ATT&CK, CAPEC, MBC, CVE/CWE, Metasploit, and real malware samples.

Chapter 5 analyzes challenges and solutions of applying RAG to CTI: hallucinations, retrieval latency/scale, embedding/threshold sensitivity, and multi-hop brittleness. It also clarifies evaluation gaps and design guidelines that inform the assistant built next (Q2→Q3 bridge).

Chapter 6 presents a domain-specific RAG assistant, called *MoRSE*, that

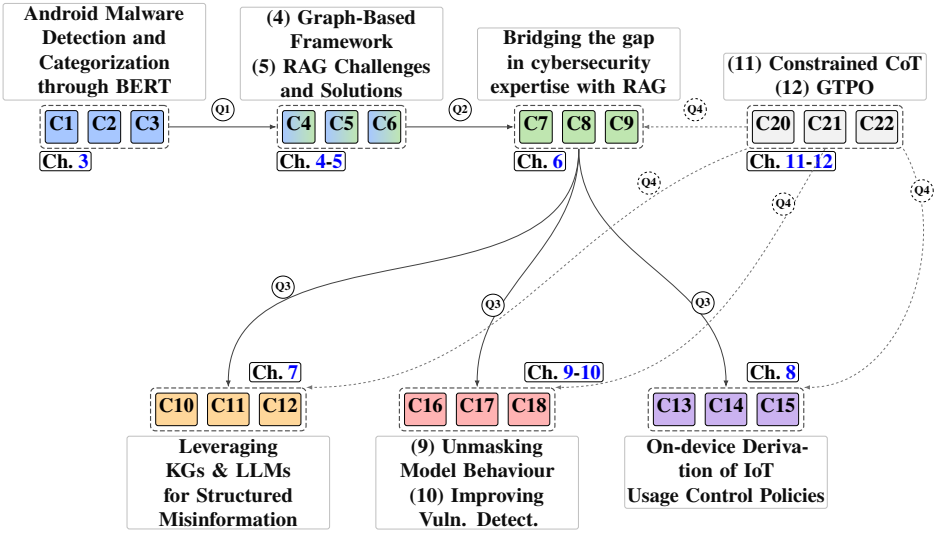


Figure 4: Contribution map grouped by chapters.

cascades a fast *Structured RAG* with a coverage-oriented *Unstructured RAG*, and uses parallel, specialized retrievers. MoRSE improves relevance, correctness, and multi-hop performance on CTI and CVE queries (Q3).

Chapter 7 develops a task-specific module that generates structured, plausibility-controlled misinformation from knowledge graphs and studies LLM detection limits. This provides a principled stress-testing engine for security workflows.

Chapter 8 introduces *Text2Policy* model, which translates natural-language commands into enforceable U-XACML usage-control policies. *Text2Policy* models is able to operate on-device and generalize across smart-home/office/healthcare scenarios (Q3, domain-specialized).

Chapter 9 examines LLM reasoning for vulnerability detection, revealing miscalibration, over-flagging of benign code, and poor generalization, even with *"reasoning aloud"*, thus motivating behavior-aware alignment via reinforcement learning.

Chapter 10 extends the previous study enriching the work by demonstrating how the established dynamic reward can achieve better stability than static reward and how this reward setting can beat Supervised Fine-Tuning performances (*SFT*).

Chapter 11 opens the final stage on reasoning optimization, quantifying the trade-off between output length, accuracy, and cost. It proposes length-aware prompting/decoding and constrained explanations to achieve correctness with

predictable budgets.

Chapter 12 introduces Group-relative Trajectory-based Policy Optimization (*GTPO*), addressing GRPO's gradient conflicts and policy collapse via conflict-aware corrections and entropy-based regularization, stabilizing training and improving reasoning robustness.

2 State of the Art

The evolution of Artificial Intelligence in cybersecurity has developed through several interconnected research directions. To provide a clear overview, this section reviews the main areas that form the foundation of this work, highlighting their main achievements and open challenges.

Section 2.1 discusses Transformer-based malware analysis, where early studies used deep neural networks to learn from API traces, control-flow graphs, and multimodal data. Over time, these methods evolved toward Transformer-based architectures, which significantly improved both detection accuracy and representation capability.

Section 2.2 presents graph-based representations for malware and threats. This research focuses on the creation of cybersecurity knowledge graphs that integrate diverse data sources, ranging from static malware features to CTI reports and attack taxonomies, to enable reasoning and cross-domain analysis. Building on these, the last part of Section 2.2 explores LLM-based assistance for Cyber Threat Intelligence (CTI). This area investigates how large language models, often combined with knowledge graphs or retrieval components, can enhance information extraction, threat reasoning, and analyst decision support.

Section 2.3 addresses misinformation generation and detection. In this domain, LLMs can act both as a threat, by enabling the large-scale creation of persuasive fake content, and as a defense, when guided by structured knowledge to detect or constrain misleading narratives.

Section 2.4 focuses on the translation of natural language into access control policies. Here, both traditional NLP pipelines and modern LLM-based systems are applied to automatically generate XACML or ABAC policies from high-level textual specifications.

Section 2.5 examines vulnerability detection, comparing classical deep learning models with more recent LLM- and reinforcement learning–based approaches, including retrieval-augmented systems and preference optimization techniques.

Finally, Section 2.6 discusses reasoning optimization in LLMs. This includes constrained reasoning, prompt engineering, and reinforcement learning strategies. Particular attention is given to Group-relative Policy Optimization (GRPO) and its limitations, which motivate the exploration of new training paradigms such as GTPO.

Together, these research directions define the conceptual groundwork of this thesis, with each subsequent contribution extending one of these key areas.

2.1 Sequence-based Malware Analysis

In the last years, several attempts have been made in order to detect malwares by the use of DNNs: MAPAS [1] uses convolution neural networks to assess the behaviors of malicious programs based on their API call graphs (CNN) while Sachith Seneviratne et al. [2] developed SHERLOCK, a self-supervised deep learning model based on the Vision Transformer (ViT) architecture that can find malware. Compared to convolutional neural networks (CNNs), our approach has the advantage of being able to capture the relationships between API calls in the sequence, which is crucial for detecting and categorizing complex Android malware. Minghui Cai et al.[15] created a vector representation of E-FCGs using an approach based on Graph Convolutional Networks (GCN). GCNs are a powerful tool for analyzing the structure and behavior of complex graphs, including API call graphs that represent the behavior of Android apps. However, developing effective GCN models can require a large amount of domain knowledge and feature engineering and GCNs are often limited by the quality and completeness of the graph representation of the app's behavior. BERT, on the other hand, can learn from any sequence of API calls, not just those included in the graph. This makes it more robust to variations in the representation of the app's behavior and less likely to miss important signals of malicious behavior. MALBERT [3] is an automated detection of dangerous software using a Transformers-style architecture while Ullah, Farhan, et al. [4] developed a method that makes use of both textual and visual aspects to discover and assess network malware. Malceiver [5] is a multi-modal hierarchical Perceiver model for detecting malware on Android. In comparison to other approaches, our methodology, presented in Chapter 3 for detecting and categorizing malware in Android APKs has several advantages. Firstly, our approach uses BERT, which is capable of capturing the relationships between API calls in a sequence, making it more effective in detecting and categorizing complex malware than other approaches such as CNNs. Additionally, BERT can learn from any sequence of API calls, not just those included in the graph, making it more robust to variations in the representation of the app's behavior and less likely to miss important signals of malicious behavior. Furthermore, our approach is based on preprocessed API sequences, which makes it less dependent on domain knowledge and feature engineering than approaches such as GCNs. Finally, our approach does not require any visual aspects of the APK or network traffic, making it more scalable and efficient.

2.2 Malware & Threat Graph Representation

This section overviews recent developments in Knowledge Graphs (KGs) for cybersecurity and Cybersecurity database and datasets that have contributed more to cyber-threat analysis.

Knowledge Graph in Cybersecurity: In the domain of cybersecurity, knowledge graphs have become pivotal tools for representing, analyzing, and processing extensive cybersecurity data from various sources. Sikos [6] reviews several graph-based data models, emphasizing how cybersecurity knowledge graphs help achieve a high level of cyber-situational awareness and facilitate machine learning and automated reasoning. On a similar front, Li et al. [7] address the construction and quality assessment of these knowledge graphs, introducing novel models that enhance the accuracy and effectiveness of cybersecurity analytics. Building on this, another study by Li et al. [8] specifically tackles the application of cybersecurity knowledge graphs in blockchain systems. They propose a pipeline-based method, PipCKG-BS, which leverages contextual features and Pre-trained Language Models to extract Cyber Threat Intelligence efficiently, demonstrating a significant improvement over existing methods. Further, Li et al. [9] introduce K-CTIAA, an automated method for cyber threat intelligence analysis. This method enhances the performance of threat analysis by integrating knowledge graphs with pre-trained models, which effectively handle the semantic nuances of network security terminology. Moreover, Bolton et al. [10] examine the integration of cybersecurity knowledge graphs with the MITRE ATT&CK framework, revealing the depth and applicability of knowledge graphs in mapping cybersecurity threats and attack patterns comprehensively. Additionally, Wang et al. [11] discuss the construction of a cyber-attack behavior knowledge graph based on CAPEC and CWE for 6G networks. They highlight the utility of graph databases in capturing the strategies and behaviors of cyber-attacks, thus enhancing situational awareness and attack prediction. Ren et al. [12] developed CSKG4APT, which leverages knowledge graph technology and deep learning to construct and update a comprehensive model of APT scenarios. This integration of fragmented intelligence enhances threat actor tracing and cross-event threat identification. CSKG4APT's dynamic APT attack attribution method supports both detection and proactive defense, significantly improving upon traditional passive defense strategies. Lastly, Bhalekar et al. [13] explore the broader role of graph databases like Neo4j in cybersecurity. They emphasize how Neo4j facilitates the analysis of complex, interconnected data sets, which is crucial for developing effective cybersecurity measures. Our cybersecurity KG, presented in Chapter 4,

differs from these approaches by providing a unified graph database that integrates malware information, CVE vulnerabilities and Metasploit exploits from multiple reputable sources. This comprehensive integration improves contextual awareness and enables more accurate correlation and understanding of complex cyber threats.

Cybersecurity Database and Datasets: The detection and analysis of Android malware have seen significant advancements through various machine learning and static analysis techniques. Arp et al. [14] introduced DREBIN, a lightweight method specifically designed for detecting Android malware directly on smartphones. It achieves a malware detection rate of 94% by performing broad static analysis and embedding application features in a joint vector space. Building on this, Daoudi et al. [15] conducted an explorative study of DREBIN, dissecting its model to understand which features are most critical for its high performance and consistency across malware families. Furthermore, Wu et al. [16] provided a bibliometric review of cybercrime and cybersecurity research, illustrating the growth and evolution of these studies over 26 years and highlighting key trends and collaboration networks. Similarly, Lim et al. [17] developed MalwareTextDB, a database for annotated malware texts to aid NLP efforts in cybersecurity, demonstrating its utility in constructing effective models for data analytics. Habaybeh and Marshall [18] explored the development of a database to track malware frequency historically, aiming to provide a reliable resource for evaluating malware-related claims in legal contexts. CDTier [19] fills the gap of open-source Chinese CTI datasets by providing a comprehensive dataset for threat entity and relationship extraction. With 100 CTI reports and thousands of threat sentences and knowledge objects, CDTier supports deep learning models for accurate threat intelligence analysis, reducing analysts' workloads. In the field of machine learning models for malware detection, Anderson and Roth introduced EMBER [20], an open dataset for training models to detect static PE malware, showing that even simple models can outperform more complex systems. Yang et al. [21] released the BODMAS dataset, addressing the need for datasets with recent malware samples and detailed family information to study concept drift and malware evolution. Lastly, Bosu et al. [22] presented DIALDroid, a tool for analyzing Android app communications, which facilitated the detection of collusive data leaks among a large corpus of real-world applications.

LLM-based Assistance for CTI Research on chatbots in cybersecurity emphasizes their roles in education, ethics, and regulation. Yoo et al. [23] employ CNN classifiers and an AI chatbot to detect and prevent SNS phishing attacks in real-time

via Telegram, demonstrating better effectiveness than traditional LSTM models. Aghaei et al. [24] developed SecureBERT, a specialized language model for Cyber Threat Intelligence (CTI), which automates key cybersecurity tasks using a customized tokenizer and pre-trained weights. Happe et al. [25] used GPT-3.5 to extend penetration testing, showing LLMs as valuable AI partners in security testing. Lu et al. [26] integrated graph structures and in-context learning for software vulnerability detection, significantly outperforming traditional models. Unlike other chatbots, MoRSE, the RAG system presented in Chapter 6 following the guidelines defined in Chapter 5, offers instant access to an ever-updating cybersecurity knowledge base.

2.3 KG & LLM for Misinformation Detection and Generation

Fake news in the LLMs era. The generation of false information has become a critical concern, particularly with the advancements in generative AI models that significantly enhance the realism and plausibility of misinformation. Powerful models, such as GPT-4 [27], have raised alarms due to their ability to generate high-quality, convincing false content that can easily mislead the public and influence opinions. Huang et al. [28] explored the extensive use of prompt engineering with LLMs for fake news generation, highlighting the model's proficiency in generating and explaining fake news. The generative models, capable of producing human-like text, images, and videos, have potential applications in disinformation campaigns, including elections, social media manipulation, and national security threats [28, 29, 30].

KGs and LLMs. In recent years, several studies have explored the integration of knowledge graphs (KGs) with LLMs. KGs offer a structured and interpretable framework for representing relational and syntactic information, which can guide language generation in a more controlled and semantically grounded manner, potentially reducing issues such as hallucinations and improving factual accuracy. For example, prior work has investigated the use of KGs to evaluate the performance of LLMs [31, 32]. Other studies [33, 27] have demonstrated how models like GPT-3 and GPT-4 can be employed to generate content based on structured data extracted from KGs. Building on these findings, Mauro et al. [34] show how KGs can be leveraged to identify credible relational links and generate content that is both contextually coherent and structurally aligned with the knowledge base.

Despite these advancements, the use of KGs for generating fake information remains largely underexplored. Given the structured nature of KGs, they hold significant potential for automating the generation of large-scale misinformation

with contextual consistency. Moreover, the ability of LLMs to produce highly fluent and plausible, but potentially false, text makes them particularly effective in crafting misinformation that is difficult to detect.

The work presented in Chapter 7, brings these two aspects together, proposing an automated framework that generates fake information at scale. The generation process is guided by a novel plausibility metric, allowing the system to produce content with varying degrees of credibility. This enables the creation of misinformation that appears semantically plausible and stealthy, thereby posing new challenges for detection systems.

Detecting Fake Information with LLMs. As generative models continue to evolve, there is growing interest in exploring the role of LLMs not only in generating, but also in detecting and analyzing false or misleading content [30, 35]. For example, Koka et al. [36] were among the first to evaluate the effectiveness of various LLMs in fake news detection, comparing the performance of large-scale models such as GPT-4 and Claude 3 Sonnet and smaller models like Gemma 7B. Despite recent progress, several challenges remain. For instance, a common limitation in the literature is the assumption that the same (often proprietary and unreleased) model is used to both generate and detect fake content, an unrealistic setup for practical deployment.

To further investigate this, also our work in Chapter 7 evaluates publicly available LLMs as detectors to assess the quality of the fake content generated through our KG-based pipeline. To ensure a more realistic and reliable evaluation setting, we use LLMs for detection that are distinct from those used to generate the misinformation, and we formalize the detection task as a simple yet effective prompting strategy, enabling an initial assessment of how well off-the-shelf models can identify structured, semantically plausible fake content.

2.4 Translation of Natural Language Commands to Enforceable Access Control Policies

Due to the complexity of XACML, automating policy generation has become an area of active research (see Chapter 8). Several approaches have been proposed to facilitate this process. One of these approaches, proposed in [37, 38], uses user-friendly graphic policy editors. These editors replace the complex text of the policy with a block-based visual interface, which helps policy creators by offering visual feedback, organizing compatible elements, and guiding users through the policy-building process. This approach retains XACML's core logic and structure

while making it more user-friendly [37]. Fatemian et al. [38] proposed Dual-XACML a Domain-Specific Modeling Language designed to generate ABAC and RBAC policies. Their tool assists users in creating XACML policies using graphical notations. Raschke and Zickau [39] proposed a Template-Based Policy Generation approach. This approach focuses on defining reusable policy patterns to simplify the requirements of large and complex attribute-based policies, improving the efficiency and consistency of policy creation. However, utilizing these methods still requires a certain level of technical knowledge. Current natural language policy-generating methods primarily assist administrators in accurately extracting access control policy components such as resources, subjects, actions, and environments from high-level requirement specification documents, thereby reducing human errors [40, 41]. Shallow parsing techniques are used to extract policy components based on a predefined controlled grammar for structuring policies in a clear format. Xiao et al. [42] proposed a text-to-policy approach that uses syntactic pattern matching to identify Natural Language Algorithmic Compliance Policies. The developed approach used shallow parsing based on four semantic patterns to extract policy components. This method leverages predefined grammar rules, ensuring reliable identification of policy components [43, 44]. However, unstructured natural language text challenges shallow parsers, because their variability makes it difficult to apply fixed grammar rules for accurate entity extraction [45]. Narouei et al. [46] propose approaches to attribute-based access control that automate the extraction of machine-readable security policies from high-level requirements specification documents using the RNN algorithm. The framework utilizes pre-trained word embeddings to detect sentences containing access control policy content from a dataset of real-world policy documents. The proposed method outperformed some state-of-the-art models, such as SVM [46]. The authors of [47] proposed RAGent, a framework that leverages LLaMA to extract access control components from high-level specifications and translates them into structured policies with a verification-refinement method, ensuring that the generated policies are reliable; Administrators can easily convert these structured policies into XACML format. Shan et al. [48] proposed a deep learning-based approach to automatically generate ABAC policies from natural language documents. This approach used ChatGLM to extract access control statements, and then the ID-CNN-CRF model was used to annotate the attributes of the subject, object, and action of these statements. Another approach involves converting policy texts into different formats, such as dependency graphs and trees, to extract policy components based on these alternative presentations [49, 50]. Alohaly et al. [50] proposed a method for converting a natural language policy (in sentence form) into a dependency tree based on grammatical

structure, then using a deep learning algorithm to predict whether a word serves as an attribute, allowing them to extract subject and object attributes for ABAC. Due to the lack of available datasets, they created their own by artificially injecting attribute information into RBAC policies following the grammatical structure of sentences.

2.5 Policy Optimization for Vulnerability Detection

Across the literature, various techniques have been proposed to automate vulnerability detection (Chapter 9 and Chapter 10). Prior work can be broadly categorized into *classical deep learning models* and *LLM centric detectors*.

Classical DNN based approaches Early deep learning approaches for vulnerability detection model source code as graphs or token sequences and train end-to-end classifiers with supervision. *DeepVulSeeker* [51] combines code graphs with semantic features to boost accuracy, while *VUDENC* [52] employs word2vec embeddings and an LSTM on real-world Python projects. *DetectVul* [53] reduces graph construction overhead through a statement level self-attention encoder. Other methods refine code structure modeling, such as *TrVD*[54], which processes AST sub-trees with a Transformer, and *VulDeeLocator* [55], which fuses token semantics with low-level code features for finer detection.

LLM-based detectors Recent efforts to automate vulnerability detection have shifted from traditional DNN based pipelines to systems that leverage the reasoning capabilities of LLMs, often by specializing them through ad hoc SFT. For example, Mahyari et al. [56] demonstrate that even general purpose transformers like BERT, when finetuned on source code, can match the accuracy of earlier graph based neural networks. Beyond supervised scenarios, Guo et al. [57] report that GPT 3.5 performs comparably to specialized vulnerability detectors, while GPT 4 surpasses them. However, both models show fragile behavior when analyzing memory safety bugs, particularly those obscured by project specific coding conventions.

When annotated data are scarce, prompt engineering offers a lightweight alternative to finetuning for LLMs. *Prompt-Enhanced ChatGPT* augments zero-shot queries with structural cues and multi-turn dialogues [58]. *ProRLearn* frames prompt selection as a reinforcement learning problem, where a policy network iteratively mutates the prompt and receives a reward from the LLM's output, achieving up to a 5% F_1 improvement over static prompts across three datasets [59].

In this context, retrieval-augmented generation (RAG) has also been used to further improve LLM-based detection. For example, *Vul-RAG* [60] builds a knowledge base from CWE descriptions, retrieves query-relevant facts, and supplies them to the LLM, achieving both high recall and human-readable explanations.

RL in vulnerability detection and secure code generation RL has been used for both vulnerability discovery and secure code generation. In web security, RL-based fuzzing is effective: *BertRLFuzzer* combines a BERT policy with a GRU-PPO agent to detect SQL injection and XSS [61], and grey-box RL fuzzers have detected reflected and stored XSS in Java web apps [62]. In hardware, *GenHuzz* integrates a language model fuzzer with PPO [63] and hardware guided rewards to optimize instruction sequences, improving coverage and uncovering new hardware bugs [64].

Beyond detection, preference-based RL variants such as RLHF [65] and DPO [66] are increasingly applied to guide code generation toward security and correctness objectives. For instance, *Focused-DPO* trains models using DPO to enhance their ability to generate code [67], while *Localized Preference Optimization (LPO)* aligns code LLMs to produce secure outputs by training on pairs of insecure and corrected code, focusing the loss on security relevant tokens. Across multiple secure coding benchmarks, LPO achieves a 19-40% reduction in security issues while maintaining or improving overall code quality [68].

2.6 Reasoning Optimization

2.6.1 Constrained Reasoning

To the best of our knowledge, most recent works on LLMs focused on increasing their accuracy [69, 70, 71]. However, as models scale up, they tend to generate more extensive and articulated responses [72], which can introduce other problems, such as hallucinations (where the model produces information that appears plausible but not grounded [73], or unnecessarily long explanations [74, 75]), which can obscure key information, making it difficult for users to extract relevant content efficiently [76, 77]. To filter out useless reasoning, [78] proposed a multi-hop processing technique, where an extraction task on the encoder to obtain the rationale for an answer, which is the most relevant piece of text in an input prompt to a given question.

To further improve the accuracy of LLMs, several prompt engineering approaches have been presented in recent years [79]. Prompt engineering involves the strategic design of input patterns to guide the model toward generating more accurate

and relevant responses [80, 81]. However, most of these approaches have been conceived to enhance model accuracy, increasing the output length. For instance, [82] and [83] introduced prompt-based approaches by adding task-specific patterns to frame the input data. While these methods allow boosting accuracy, they can also produce longer outputs due to the additional context and detail introduced by the prompt, making it challenging to provide factual and concise answers [84].

Another form of prompt engineering was proposed to improve reasoning within the conclusive answer. In this context, Chain-of-Thought (CoT) prompting [85] is one of the most notable methods, showing significant benefits in QA tasks by requiring the model to provide a step-by-step explanation along with the final response. However answers generated with CoT tend to be lengthy, hence increasing the generation time [86, 87].

Given the substantial amount of work focused on improving the accuracy of LLMs, it is not surprising that most of the adopted metrics [88, 89] and benchmarks [90, 91] only address the correctness of the responses, without paying attention to conciseness and response times [72, 92]. In other tasks too, such as reasoning in control engineering, the focus has been primarily on correctness rather than consistency or conciseness [93]. Additionally, several studies have addressed computational cost challenges but not the conciseness. For example, [94] evaluated the budget-aware reasoning capabilities of LLMs, while [95] proposed an inference pipeline to improve processing speed. Other works have explored similar optimization approaches, including [96] and [97]. In addition, [92] proposed a benchmark to study LLMs accuracy while incorporating a manual redundancy assessment.

Despite recent advancements, several key aspects remain not sufficiently explored (addressed in Chapter 11): *(i)* the impact of concise answers on inference cost and time predictability; *(ii)* the integration of such aspects into unified metrics that evaluate LLMs not only in terms of correctness but also conciseness; and *(iii)* an understanding of the analysis of conciseness through the conciseness based on the embeddings content extracted by the generated answers.

2.6.2 Group-relative Trajectory based Policy Optimization

Reinforcement Learning in LLMs. RL has been widely adopted in decision-making tasks [98, 99, 100], and nowadays is increasingly applied to the alignment and fine-tuning of LLMs. A prominent approach is RL from Human Feedback (RLHF), introduced in InstructGPT [65], and further developed by Anthropic [101]. RLHF has become central to the training pipelines of state-of-the-art LLMs such as

Claude 3 [102], Gemini [103], and GPT-4 [104]. It typically includes supervised fine-tuning, a reward model, and the adoption of Proximal Policy Optimization (PPO) [63]. In this settings, PPO improves training stability by constraining updates through a clipped surrogate objective, offering a more practical alternative to Trust Region Policy Optimization (TRPO) [105]. However, it remains sensitive to reward scaling and can suffer from training instability [106, 107, 108], requiring multiple refinements [109]. Therefore, multiple variations have been proposed over the years, such as TRGPPO [106], alphaPPO [110], and PPO-ALR [111].

Advancements and Limitations in GRPO. To overcome the need for a critic model, Deepseek introduced GRPO [112, 113], an approach that, given a question, compares multiple responses (completions) to derive relative rewards. GRPO demonstrates state-of-the-art performance on math benchmarks and achieves human-like alignment without relying on explicit manual feedback or critic networks [114]. Despite its promise, potential limitations of GRPO have been recently emerged, as bias effects [115], gradient imbalance [116], which lead to undertraining of rare yet informative tokens [117], and degradations (or even collapse) of model performance like in PPO [118].

Building upon previous analyses of GRPO training behaviors, in Chapter 12, we deepen the study of token-level updates across completions of the same group, revealing conflicts in shared tokens. Furthermore, we extend the understanding of policy collapse, showing that KL divergence is limited in addressing this issue, whereas entropy-based analysis provides clearer signals [119]. These insights motivate the design of GTPO, which effectively improves stability and performance during both training and evaluation.

3 Graph-Based Android Malware Detection and Categorization through BERT Transformer

The rapid diffusion of Android devices over the last decade has been accompanied by a parallel growth of malicious applications targeting this ecosystem. The Android platform, due to its openness and extensive adoption, has become a privileged target for cybercriminals, who constantly refine their attack strategies to circumvent existing defenses. Modern Android malware is characterized by the adoption of sophisticated techniques such as obfuscation, encryption, dynamic loading, and polymorphism, which make detection increasingly challenging. As a consequence, both industry and academia have sought approaches that go beyond signature-based methods and hand-crafted feature extraction, which tend to be brittle when confronted with the evolving nature of threats.

Within this context, the first stage of the research addressed the problem of malware detection and categorization from a sequence perspective, with a particular focus on the use of Transformer-based architectures. *The intuition was that the behavior of an application, captured through the sequence of its API calls, could be treated analogously to a linguistic sequence*. Transformers, and BERT in particular, had already demonstrated the ability to model long-range dependencies and contextual relations in natural language; extending this paradigm to API call sequences promised to overcome the limitations of traditional classifiers, convolutional models, and graph-based approaches.

API Call Graphs describe the interactions between the different components of an Android application. By applying BERT to these sequences, it was possible to capture behavioral patterns that are distinctive of different malware categories. The experimental study targeted three representative families (SMS-Trojan, Rootkit, and Spyware), which together account for more than ninety percent of widely used benchmarking datasets. Results demonstrated that the proposed methodology achieved state-of-the-art performance, reaching an accuracy of 98% in malware detection and 94% in categorization. These findings confirmed that contextual sequence models can effectively discriminate between benign and malicious applications, while also supporting categorization into relevant families.

Beyond the quantitative performance, this work provided several significant contributions. First, it demonstrated the effectiveness of contextual sequence models in capturing behavioral traces of malware, establishing that transformers are not only applicable but advantageous in this domain. Second, it introduced a two-level pipeline that combines binary detection with family-level categorization,

allowing both the identification of malicious software and its allocation into meaningful taxonomies. Finally, it highlighted important limitations of sequence-only approaches: despite their accuracy, they remain sensitive to dataset drift and require frequent retraining, as they cannot by themselves capture the full semantics and variability of complex attacks.

3.1 Background

Text-Classification Techniques. Text classification is a crucial task in natural language processing that involves categorizing text into predefined classes based on its content. Traditional machine learning algorithms for text classification, such as Naive Bayes, Support Vector Machines (SVMs), and Decision Trees[120], require hand-engineered features like bag of words, TF-IDF, or n-grams to represent the text. In contrast, deep learning models like Convolutional Neural Networks (CNNs) [121] and Recurrent Neural Networks (RNNs) can learn features automatically from the text, resulting in better performance on complex text data. However, Transformers [122] have emerged as the most advanced method for text classification in recent years. Transformers are a type of neural network architecture that is well-suited for handling sequential data, making them ideal for NLP problems like text classification. Transformers can capture the relationships between words in a sentence more effectively than RNNs or CNNs, resulting in state-of-the-art performance on various text classification tasks. BERT, a transformer-based model, has become the most widely used model for text classification in NLP. BERT [123], has achieved state-of-the-art results in various text classification tasks, demonstrating the superiority of transformers over traditional machine learning algorithms and other deep learning models.

BERT Model for Graph Classification. Transformer is an attention mechanism that learns the contextual relationships between words (or subwords) in a text and is used by BERT [124]. Transformer's basic design consists of two independent mechanisms: an encoder that reads the text input and a decoder that generates a job prediction. Only the encoder mechanism is required because BERT's aim is to produce a language model. The Transformer encoder reads the entire sequence of words at once, in contrast to directional models, which read the text input sequentially (from right to left or left to right). This trait enables the model to understand a word's context depending on all of its surroundings (left and right of the word). Word sequences are changed with a [MASK] token before being fed into the BERT. Based on the context offered by the other, non-masked, words in the sequence,

the model then makes an attempt to forecast the original value of the masked words. BERT can be used for the classification of topological representations of a graph. Graph classification involves predicting the properties or labels associated with a given graph. BERT can be adapted for this task by representing the graph as a sequence of tokens and utilizing the transformer-based architecture and by leveraging the power of BERT's contextual understanding and the ability to capture long-range dependencies in a graph, this approach can provide effective solutions for graph classification tasks. To apply BERT for graph classification, the following steps need to be carried out:

1. *Graph Representation*: Convert the graph into a sequence of tokens. This can be done by assigning tokens to nodes, edges, or other structural elements of the graph. Each token represents a specific component of the graph.

2. *Masking*: Similar to the standard BERT approach, mask certain tokens in the graph sequence using the [MASK] token. This creates a masked language modeling task where BERT predicts the original values of the masked tokens based on the context provided by the surrounding tokens.

3. *Pretraining*: Train the BERT model on a large corpus of graph sequences. The model learns to understand the contextual relationships between the graph tokens and their corresponding labels.

4. *Fine-tuning*: After pretraining, the BERT model can be fine-tuned on a specific graph classification task. This involves providing labeled graph instances and optimizing the model to predict the correct class or property associated with each graph.

Sequence Categorization. One of the key challenges in this task is transforming the API call graph of an Android APK into a sequence that is consistent with the topology of the graph. AndroGuard [1] is a well-known tool for generating API call graphs of Android APKs. To use BERT for categorizing API call graph [125] sequences, the graph generated by AndroGuard can be transformed into a sequence that is consistent with the topology of the graph. Our approach is to use **Kahn Algorithm** [126] to generate a sequence of API calls. This approach has the advantage of preserving the topological order of the graph, but it can result in long sequences with many repeated API calls. BERT, as a transformer-based model, has

[1] https://androguard.readthedocs.io/en/latest/

been shown to be effective in handling long sequences, making it a good candidate for categorizing API call graph sequences of Android APKs. BERT processes the input sequence in both forward and backward directions, allowing it to capture the relationships between API calls in the sequence. The fine-tuning process involves training a classifier on top of the pre-trained BERT representations, allowing it to learn the specific relationships between API calls in the graph.

Kahn Algorithm. The Kahn algorithm is a topological sorting algorithm that is used to generate a linear ordering of the vertices in a directed graph. In the case of transforming an Android APK into an API call sequence, the Kahn algorithm is applied to the API call graph generated by AndroGuard. The API call graph is a directed graph that represents the relationships between API calls made by the Android application. Each node in the graph represents an API call and each edge represents a relationship between two API calls, such as a call being made from one API call to another. The Algorithm uses a directed graph $G = (V, E)$ with nodes V and edges E. It initializes sets L (sorted nodes) and S (nodes with no incoming edges). It iterates until S is empty, removes outgoing edges of a node popped from S, and adds nodes with no incoming edges to S. If E is empty, it returns L. If not, it indicates a cycle error.

3.2 Methodology

The methodology described aims to use deep learning to classify APKs as either benign or malicious and to categorize malwares into specific classes such as Spyware [127], SMS Trojan and Rootkit [128]. The following is a summary of the training steps need to be performed for each APK:

- *Collection of APKs*: We have collected both benign and malicious APKs for the study.
- *Obtaining API call graphs*: The Androguard tool was used to obtain API call graphs for each APK. This provides a graphical representation of the interactions between API of an Android app.
- *Transformation of graphs into sequences*: The API call graphs were then transformed into sequences using the Kahn algorithm.
- *Inputs to a BERT model*: The sequences generated in the previous step were used as inputs to a BERT model. In this case, the BERT model was fine-tuned

to classify the APKs as benign or malicious and, in the case in which the model recognizes an APK as a malware, it tries to assign a category to the malware.

The API call graph provides a more comprehensive representation of the behavior of an Android app compared to other methods, such as signature-based or static analysis techniques. The graph captures the interactions between different APIs in the app and can reveal malicious behavior that may not be evident from individual API calls. One advantage of using the API Call Graph is that it provides a high-level view of the behavior of an app. Instead of looking at individual API calls, which can be overwhelming, the graph allows us to see the interactions between the APIs in a more structured and intuitive way. This can make it easier to identify patterns and anomalies that may indicate malicious behavior.

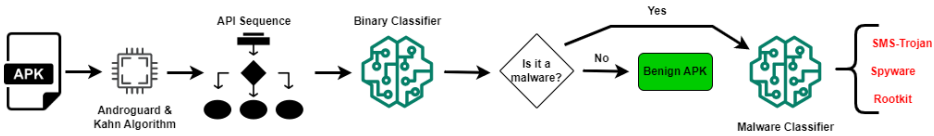


Figure 5: Complete Toolchain

Toolchain Overview. The Fig. 5 depicts the complete toolchain used to achieve the goal of identifying patterns of behavior indicative of different malware categories based on sequences of Android API calls. The toolchain consists of two classifiers, namely the Binary Classifier and the Malware Classifier:

1. The *Binary Classifier* determines if a sequence of API calls extracted from an APK belongs to a malicious or benign application. If the classifier identifies the APK as malware, the corresponding sequence is forwarded to the Malware Classifier for further analysis. On the other hand, if the classifier identifies the APK as benign, it is labeled as a Benign Application.

2. The *Malware Classifier* analyzes the API sequence forwarded by the Binary Classifier and identifies the type of malware, which can be Spyware, Rootkit, or SMS-Trojan.

The *Preprocessing* operation is an essential step for the toolchain and comprises two macro steps, namely *API Sequence Generation* and *Sequence Evaluation*. The API Sequence Generation step generates API sequences from the API Call Graph

by considering the dependencies between the APIs. The Sequence Evaluation step applies BERT, a pre-trained deep learning model, to the API sequence for classification.

API Sequence Generation. In an API Call Graph, an API may depend on another API if it invokes or calls the other API during its execution. When an Android application is running, it interacts with the system and other components by making API calls. The API Call Graph represents these interactions as a directed graph, where the nodes are the APIs, and the edges represent the API calls. For example, suppose an Android app uses the API call `getLocation()` to get the device's current location. In that case, it may depend on the API call `checkSelfPermission()` to check whether the app has the necessary permission to access the device's location. In this case, we can say that `getLocation()` depends on `checkSelfPermission()` since `getLocation()` needs to call `checkSelfPermission()` first to verify if it has the required permission. Therefore, the dependencies in the API Call Graph represent the order in which the APIs should be executed to achieve the desired functionality of the Android app.

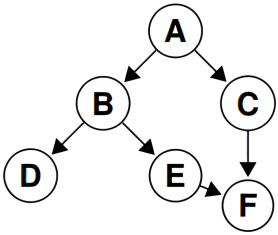


Figure 6: API Call Graph Example

The Fig. 6 shows a graph which depicts an API call graph of an Android application. In this graph, each node represents an Android API that is being used by the application. The nodes are labeled A through F and are represented as circles with bold letters inside. The lines connecting the nodes represent the API calls made by the application, and the direction of the lines indicates the flow of the calls. For example, node A makes API calls to nodes B and C, node B makes API calls to nodes D and E, node C makes an API call to node F, and node E makes an API call to node F. The graph can be used to analyze the dependencies between different APIs used by the application, identify possible bottlenecks and optimize the application's performance.

Algorithm 1: API Sequence Generation

Input: A graph, represented as a dictionary where the keys are API names and the values are lists of API names that the key API depends on

Output: A sequence list of API names in the order they should be executed

Graph $\leftarrow$ {A $\mapsto$ [B, C], B $\mapsto$ [D, E], C $\mapsto$ [F], D $\mapsto$ [ ], E $\mapsto$ [F], F $\mapsto$ [ ]};

Sequence $\leftarrow$ [A, C, F, B, E, D];

The algorithm repeats this process until there are no more APIs in the graph. The order in which the APIs are selected will determine the order of execution of the APIs.

In the case of the provided example, the sequence is generated as follows:

1. API A is inserted first into the queue since it has no dependencies;
2. API C is inserted after A because it depends only on A and has no dependencies on other APIs;
3. API F is inserted after C because it depends only on C and has no dependencies on APIs B and E;
4. API B is inserted after F because it depends only on A and has no dependencies on APIs C and F;
5. API E is inserted after B because it depends only on B and has no dependencies on APIs C and F;
6. API D is inserted after E because it has no dependencies on any other APIs;

3.3 Sequence Evaluation

The toolchain for classifying sequences of Android APIs into benign or malware, and further classifying malware into SMS Trojan, Rootkit, or Spyware can be described as follows:

- *Sequence Collection and Preprocessing*: Collect a large dataset of Android API sequences, including both benign and malware sequences removing any unnecessary features, tokenizing the data, and converting it into a suitable format for BERT input. The dataset come from both Drebin [129][130] and Maldroid [131] [132].

- *Training the BERT Classifier for Malware Detection*: Next, a BERT binary classifier is trained on the preprocessed data to classify sequences as either benign or malware. The classifier learns to identify patterns in the sequences that indicate malicious behavior.

- *Classification of Malware*: If a sequence is classified as malware, it is then passed to a second BERT classifier which is trained to classify the malware into one of the three categories: SMS Trojan, Rootkit, Spyware.

- *Validation and Testing*: Validate the accuracy of the BERT classifiers on a separate test set of data to ensure that the classifiers are not overfitting the training data and are generalizable to new, unseen data.

- *Deployment*: Deploy the trained classifiers in a environment for real-time classification of Android API sequences.

3.4 Classification Model

The proposed deep learning model is designed using TensorFlow [133] and the Keras API. It is made up of two main components: the BERT encoder and a neural network.

BERT Encoder: The input to the model is represented by raw text, which is first preprocessed by the `bert_preprocess` layer, which is a pre-trained TensorFlow Hub module [2]. The pre-processed text is then passed through the BERT encoder, which is a pre-trained TensorFlow Hub module [3], and is responsible for generating a high-level representation of the input text. There are little differences between the Neural Network dedicated to malware categorization and the Neural Network dedicated to malware detection.

NN for Categorization: The output from the BERT encoder passes through a series of neural network layers. The first layer is a dropout layer with a dropout rate of 0.1. This layer helps to prevent overfitting by randomly setting some input neurons to zero during training. The next four layers are dense, fully-connected layers with increasing number of neurons (1024, 2048, 4096, 4096) and ReLU activation functions. Finally, the last layer is a dense layer with 3 neurons and a softmax activation function, which generates the final output of the model.

NN for Detection: A number of neural network layers process the output from the BERT encoder. The top layer has a dropout rate of 0.1 and is a dropout layer.

[2] https://tfhub.dev/tensorflow/bert_en_uncased_preprocess/3
[3] https://tfhub.dev/tensorflow/lambert_en_uncased_L-24_H-1024_A-16/2

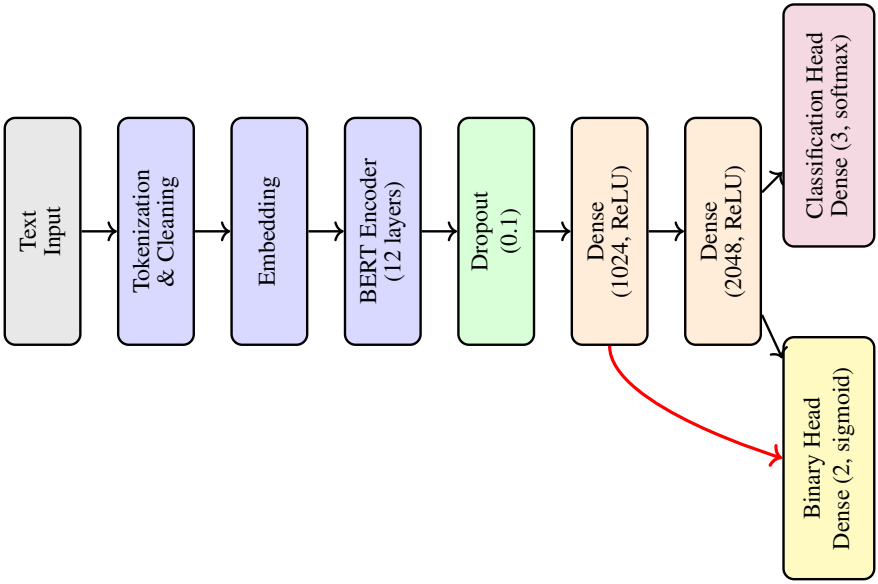


Figure 7: Layer sequence (vertical labels) with BERT backbone and dual heads.

The following three layers are highly linked and dense, with increasing numbers of neurons (1024, 2048, and 4096), as well as ReLU activation capabilities. The model's final output is produced by a dense layer with one neuron and a sigmoid activation function in the last layer.

3.5 Data Augmentation

Data Augmentation for a BERT model in the context of an Android API call sequence involves creating new sequences from existing ones in order to balance the dataset. This is done by cutting two or more APIs from the original sequences and using these pieces to form new sequences that belong to the Spyware or SMS Trojan classes. This process helps to increase the size of the dataset and prevent overfitting in the model. By having a balanced dataset, the BERT model can better learn the patterns and features that distinguish the malwares' classes, leading to improved performance in detecting these types of malware.

3.6 Experiments

Dataset. The Dataset for SMS-Trojan, Rootkit and Spyware APKs has been retrieved from Drebin [129][130], collecting more than 2000 applications. In order to associate each application to a precise category, we have leveraged the association made by the categorization proposed in [134]; in the first step, we have linked each malware family to the correspondent malware category and then we have linked each malware hash value to the corresponding category passing through the family associated to the chosen hash value. As you can see from the Table 1, most number of malwares belong to three different classes: **SMS Trojan**, **Rootkit**, **Spyware**. They cover more than 90% of the whole dataset.

Regarding Benign APKs, we have retrieved the complete package from Maldroid [131] [132] and we have also crawled Google Play store in order to reach a balanced dataset combining the same number of malicious and benign applications. It is

Table 1: Category distribution

Category	**Counts**
SMS Trojan + Installer	0.141%
SMS Trojan	**0.2879%**
Rootkit	**0.2542%**
Installer	0.0342%
Spyware	**0.3737%**
Trojan	0.0038%
SMS Trojan + Botnet	0.0016%
SMS Trojan + Spyware	0,0261%
Botnet	0.0033%
Ransomware	0.0011%
Total No. of APKs	903

worth describing the behavior of the analyzed malwares proposed in [134].

Rootkit: an Android rootkit is a form of malware that can gain administrative access to an Android device, allowing it to execute tasks that ordinary apps cannot.

SMS Trojan: An Android SMS Trojan is a type of malware that infects an Android device and uses it to send text messages (SMS) to premium-rate numbers, incurring charges for the victim without the victim's knowledge or consent.

Spyware: Android Spyware is a type of malicious software that is designed to gather information from an infected Android device without the user's knowledge

or consent.

Training Phase Testbed. A split function is used to split the input data into training and testing sets; the function is applied two times in order to extract from the first training set the partition used as validation set. The stratify parameter ensures that the proportion of each class is balanced between the training and testing sets. A *Metrics* list contains three evaluation metrics for the model: categorical and binary accuracy, respectively for categorization and detection, precision, and recall. Moreover, three callback functions are defined to be used during model training:

- *EarlyStopping* is a callback function that will stop the training process if the validation loss does not improve for 5 consecutive epochs.
- *ModelCheckpoint* is a callback function that saves the weights of the model at every epoch.
- *Callback* is a parent class for user-defined callback functions.

The optimizer parameter specifies the algorithm to use for optimizing the model parameters, and *Adam* is a popular choice for deep learning. The loss parameter specifies the loss function to use during training, and *CategoricalCrossentropy* is appropriate for multi-class classification problems.

Table 2: Categorization Report

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Spyware	0.913	**0.971**	0.941	272
Rootkit	**0.946**	0.911	0.929	271
SMS-trojan	**0.970**	0.945	0.957	272
Accuracy	-	-	0.942	815
Macro avg	0.943	0.942	0.942	815
Weighted avg	0.943	0.942	0.942	815
	Loss	**Accuracy**	**Precision**	**Recall**
Overall Values	*0.3295*	0.9423	**0.9433**	0.9387

3.6.1 Results

Table 2 is a categorization report that provides precision, recall, and F1-score for each class in the dataset, as well as macro and weighted averages and the number of

instances for each class. The last row of Table 2 shows the results of categorizing the test dataset, with metrics such as loss, accuracy, precision, and recall. The loss is 0.3295, which is relatively low, indicating good performance. The accuracy is 0.9423, meaning that the model classified 94.23% of the test instances correctly. The precision is 0.9433, which suggests that the model has a low false positive rate, and the recall is 0.9387, indicating a relatively low false negative rate.

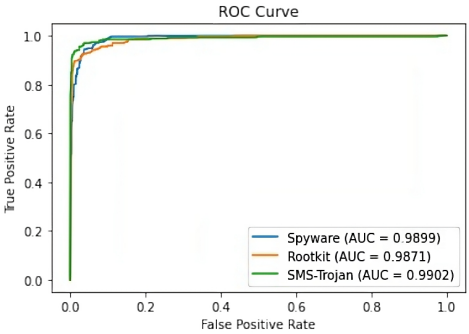


Figure 8: AUC metric for Malware Categorization

The figure 8 shows the Categorization ROC cuve. An AUC value of 1.0 represents a perfect classifier, while a value of 0.5 represents a random classifier. The values provided are all close to 1.0, with Spyware having an AUC of 0.9899, SMS-Trojan having an AUC of 0.9902, and Rootkit having an AUC of 0.9871. These high AUC values suggest that the model is able to distinguish between positive and negative instances with high accuracy for all three classes. The Table 3 shows the detection report in which the precision, recall, and F1-score for both classes are very high, with both classes having a precision and recall of 0.98 and an F1-score of 0.98. In the last row the Table 3 the loss is 0.1254, which is very low, suggesting that the model has excellent performance. The accuracy is 0.9762, indicating that the model classified 97.62% of the instances correctly. The precision is 0.9805, which means that the model has a low false positive rate, and the recall is 0.9718, indicating that the model has a low false negative rate.

3.7 Conclusions

This chapter presented a sequence-based approach to Android malware analysis in which BERT was employed to model API call sequences for both *detection* and *categorization*. The experimental results confirmed that contextual sequence

Table 3: Detection Report

Class	Precision	Recall	F1-Score	Support
Benign APK	0.97	**0.98**	**0.98**	568
Malicious APK	**0.98**	0.97	**0.98**	568
Accuracy	-	-	0.98	1136
Macro avg	0.98	0.98	0.98	1136
Weighted avg	0.98	0.98	0.98	1136
	Loss	**Accuracy**	**Precision**	**Recall**
Overall Values	*0.1254*	0.9762	**0.9805**	0.9718

models capture salient behavioral patterns of applications: detection achieved very high accuracy with low false positive and false negative rates, while categorization attained strong precision and stable loss profiles across the considered families (SMS-Trojan, Rootkit, Spyware). These findings validate the effectiveness of Transformer-based sequence modeling for learning discriminative representations directly from execution traces, without manual feature engineering. At the same time, the study highlighted intrinsic limitations of sequence-only views. First, models remain sensitive to dataset drift and require frequent retraining as threats evolve. Second, sequential representations struggle to express higher-level semantics, e.g., links between malware families, exploited vulnerabilities (CVE/CWE), TTPs in ATT&CK, tools, campaigns, or mitigations, thus limiting cross-source reasoning and longitudinal analyses. Third, the lack of an explicit, updatable knowledge layer makes it difficult to contextualize predictions beyond per-sample traces and to support analyst workflows that demand multi-entity, multi-hop explanations.

These observations motivate the next step of the thesis: moving beyond sequences toward representations that are *structured*, *contextual*, and *dynamically updatable*. This leads to the first research question that guides the subsequent chapter:

Research Question (Q1). *How can one move beyond sequence modeling to a representation that logically connects malware to their characteristics, tactics, and related knowledge in a way that remains dynamically updated?*

Addressing Q1 requires introducing unified, graph-based knowledge structures capable of integrating heterogeneous cybersecurity sources and supporting retrieval and reasoning across entities and relations, foundations that will be developed in the next chapter.

4 A Comprehensive Graph-Based Framework for Malware Analysis and Threat Research

The study of malware cannot rely exclusively on sequential models of behavior. While the previous chapter demonstrated the effectiveness of Transformer-based approaches in capturing contextual relations within API call sequences, it also revealed inherent limitations: *sequence-only methods require frequent retraining and are unable to capture the broader semantics of complex attacks*. This observation motivates the need for representations that can integrate heterogeneous sources of information and maintain logical connections between diverse cybersecurity elements.

Graphs, and in particular knowledge graphs, provide a natural solution to this problem. Their ability to represent entities and relationships as first-class citizens makes them well-suited to the cybersecurity domain, where threats emerge from the interaction of malware, vulnerabilities, exploits, tools, campaigns, and defensive measures. *Unlike flat or sequential representations, graph structures enable the modeling of interdependencies, the integration of heterogeneous data sources, and the execution of reasoning tasks that depend on both local and global context.* These features are particularly relevant in the context of cyber threat intelligence, where analysts must not only detect individual events but also understand how they connect to wider attack strategies.

Thus, the second chapter introduces a unified, dynamic graph-based representation of malware and threats. To this end, the framework **MATRIX (Malware Analysis and Threat Research with STIX)** was developed as a Neo4j-based[4] knowledge graph integrating multiple authoritative sources, including MITRE ATT&CK, CAPEC, DEF3ND, MBC, CWE, CVE, Metasploit, and real-world malware reports. The integration follows the STIX standard, ensuring structural consistency and interoperability. MATRIX currently comprises 14 node types, representing key cybersecurity entities such as malware, objectives, behaviors, vulnerabilities, exploits, and defensive actions, connected by 6 different relationship types. In addition, the framework is continuously updated through automated crawling of public repositories, ensuring that the knowledge base remains relevant to the evolving threat landscape.

This chapter contributes to the research in two major ways. First, it establishes a dynamic knowledge structure capable of connecting heterogeneous cybersecurity entities within a coherent framework. By doing so, it allows researchers and practi-

[4] https://neo4j.com/

tioners to reason across domains that were previously siloed linking, for instance, malware families to the vulnerabilities they exploit, the techniques they employ, and the defensive countermeasures that mitigate them. Second, it demonstrates how graph-based representations enable richer contextualization and reasoning, supporting analyses that go beyond detection to encompass objectives, behavioral comparison across malware families, evaluation of mitigation effectiveness, and tactic-specific investigations such as API-level mappings. These capabilities highlight the added value of graph-based models for cybersecurity intelligence, making MATRIX not only a repository of structured information but also an operational tool for advanced analysis.

4.1 Background

Graph Database and Knowledge Graph. A *graph database* is a NoSQL database that stores and manages data using graph structures, specifically designed to handle complex relationships between data entities. It supports both *directed* and *undirected graphs*. A *directed graph* (or digraph) $D = (V, A)$ consists of a set of vertices (nodes) V and a set of ordered pairs (arcs) $A \subseteq V \times V$, representing directed connections from one vertex v_i to another v_j. A *knowledge graph* is a specialized form of a graph database. It is defined as a directed, labeled graph $G = (V, E, L)$, where V is the set of nodes (entities), $E \subseteq V \times V \times L$ is the set of edges (relationships) that connect the nodes, and L is a set of labels that describe the types of relationships. Each edge (v_i, v_j, l) indicates a directed relationship from node v_i to node v_j, characterized by the label l.

Cyber Threat Intelligence (CTI). CTI helps organizations detect and respond to cybersecurity threats by offering insights into vulnerabilities, attack methods, malware, and threat behaviors. This intelligence strengthens defenses and informs responses. CTI operates at three levels: tactical, operational, and strategic. Tactical intelligence focuses on immediate threats, operational intelligence covers threat actors and campaigns, and strategic intelligence guides long-term security strategies. For CTI to be effective, it must be complete, accurate, relevant, and timely. Completeness ensures enough data to act on, accuracy guarantees reliable information, relevance tailors intelligence to the organization's specific threats and timeliness allows action to mitigate risks.

Structured Threat Information Expression. Structured Threat Information Expression (STIX™) is a format used to exchange cyber threat intelligence (CTI) in a consistent, machine-readable way. It helps organizations share threat data, improving their ability to detect, respond to, and anticipate cyberattacks. STIX is

designed to enhance capabilities such as collaborative threat analysis, automated threat exchange, and incident response. STIX 2.1 represents CTI as a graph, with *STIX Domain Objects (SDOs)* as the nodes and *STIX Relationship Objects (SROs)* or embedded relationships as the edges. Together, these components form a flexible and modular graph structure for sharing and analyzing threat intelligence. STIX defines various SDOs, such as *Attack Pattern*, *Malware*, *Indicator*, *Threat Actor*, and *Vulnerability*, each representing key concepts in CTI. Relationships between these objects are described using SROs, which define how the objects are connected. For example, an *Indicator* can relate to *Malware* with a relationship type of `indicates`. STIX also allows for custom relationship types, providing flexibility in how relationships are defined and used.

4.2 MATRIX Architecture

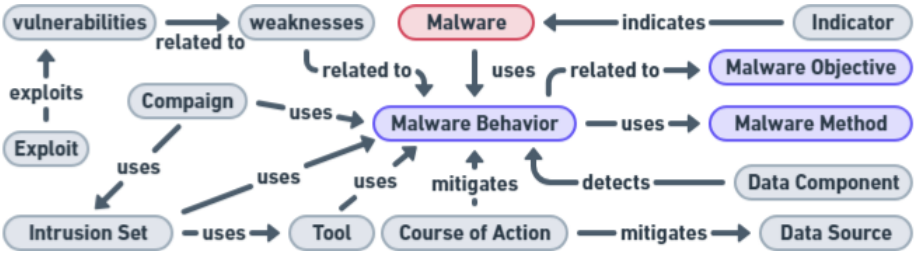


Figure 9: MATRIX Architecture Overview.

The MATRIX architecture, shown in Fig. 9, was built using the Neo4j framework to organize and connect the key elements of malware and threat analysis. Most of the components are based on the MITRE ATT&CK framework, but to provide a more complete view of the threat landscape, we have also integrated data from the *Malware Behavior Catalog* (MBC), which focuses specifically on malware objectives and behaviors. The `mitre/cti` and `MBCProject` are two of the most important STIX standards and collections used in cybersecurity. In MATRIX, all nodes and relationships are based on the STIX objects from the `mitre/cti` dataset, with the exception of Malware Behavior, Objective and Method (in blue in Fig. 9), which follow the format of the `MBCProject`. This ensures that our graph conforms to the MBC STIX standard and provides a more detailed and consistent approach to analyzing malware. The nodes *Weaknesses, Vulnerabilities, and Exploit* are not included in any of the two standard collections; the node *Weaknesses* is a new SDO that we have specifically defined. In addition, the graph is kept up to date through

continuous and automatic updates and becomes more comprehensive over time so that it always reflects the latest threat data. Below is a breakdown of the *SDOs* of the graph.

Malware: Represents malware from the MITRE ATT&CK framework, including descriptions, specifications, aliases and external references. It also includes malware and threats from the MBC that are not defined in MITRE ATT&CK.

Malware Behavior: Captures the actions, behaviours and techniques used by malware. These nodes contain all *Techniques* from MITRE ATT&CK as well as the *Behaviours* defined in MBC. Behaviors are marked with different prefixes. The prefix **T** refers to MITRE ATT&CK techniques, while **B** corresponds to behaviors from MBC. The prefix **E** is used for MBC behaviors that extend MITRE ATT&CK techniques and connect them through a *related-to* relationship. **C** stands for micro-behaviours, which are usually insignificant and often non-malicious actions. Finally, **F** stands for sub-techniques in MBC. In addition, these nodes also contain techniques from the *CAPEC* Framework, which are linked to weaknesses and the associated MITRE ATT&CK techniques. *Sigma* and *CAPA* rules are included within Malware Behavior under the `detection_rules` field and support the identification of malware Behaviors and actions. Malware Behaviors can be considered as *Attack Pattern* objects using the standard STIX objects.

Malware Objective: The objectives are based on the ATT&CK tactic, but have been adapted for malware analysis. We have also included additional objectives from the MBC catalog to better capture the objectives behind malware behavior.

Malware Method: Methods describe how behaviors are executed or refined. They can represent sub-techniques or specific implementations of behaviors. Methods are always linked to behaviors and cannot be used independently of them.

Indicators, such as malware hashes or *YARA* rules, are key data used to detect and analyze malware. Specifically, the Indicator set contains over 10,000 hashes of 269 different malware families automatically collected by *MalwareBazaar* and *VirusShare*. We collected the corresponding reports from *VirusTotal* and stored them in an Elasticsearch database so that we can analyze, gain insights and run experiments with real malware data. The number of hashes and their associated reports will continue to grow with regular automated updates, ensuring the data remains current and comprehensive.

Course of Action presents the various strategies for defending against or reducing the impact of malware or a cyberattack. These are derived from the *Mitigations* of MITRE ATT&CK and the *DEFEND* Framework, which provides detailed strategies and solutions to protect against various threats and reduce the chances of successful attacks.

Intrusion Set represents groups or clusters of threat actors, commonly referred to as *Groups* in the MITRE ATT&CK framework. These are collections of individuals or organisations that work together to carry out campaigns or attacks over a period of time.

Campaign refers to a coordinated series of malicious activities or attacks carried out by a threat actor or group (represented as an intrusion set) over a period of time, usually targeting an organisation or sector.

Data Sources represent the different topics/themes of information that can be used together with other information that can be collected by sensors/logs. Data sources identify specific properties/values of a data source relevant to detecting a given ATT&CK technique or sub-technique.

Data Components represent specific elements of data collection that can be used to detect malicious activity or behavior. They describe specific events or actions within a system that can be observed or logged to identify suspicious behavior.

Tools are legitimate software that can be used by threat actors to carry out attacks. Knowing how and when threat actors use such tools can be important in understanding how campaigns are executed.

Node	Size	Number of Nodes
Campaign	120K	28
Course of Action	25M	6029
Data Component	452K	109
Data Source	156K	38
Exploit	38M	4531
Indicator	54M	13407
Intrusion Set	860K	163
Malware	3.4M	829
Malware Behavior	20M	1705
Malware Method	2.0M	482
Malware Objective	144K	35
Tool	352K	85
Vulnerabilities	968M	231315
Weaknesses	5.1M	964

Relationship Type	Size	Count
MATRIX Relationships	351M	87,642

Table 4: MATRIX Nodes and Relationships Summary

Dataset	Nodes	Relat.	Malware	Indic.
mitre/cti	4237	22259	734	-
MBCProject	892	1015	50	183

Table 5: Dataset Comparison

Weaknesses represent the Common Weakness Enumeration (CWE), a catalog of software and hardware weaknesses. These weaknesses can be exploited by attackers and are used to identify potential risk areas in systems.

Vulnerabilities correspond to the Common Vulnerabilities and Exposures (CVE), a list of publicly known security vulnerabilities. Each CVE describes a specific vulnerability in software or hardware that can be exploited to compromise the security of a system.

Exploits refer to the exploits available in the Metasploit Framework, a widely used platform for penetration testing. These exploits are designed to take advantage of specific vulnerabilities (CVEs) to compromise systems and are often used by threat actors in attacks.

MATRIX Scale Overview. Table 4 summarizes the structure of the MATRIX dataset, detailing the number of nodes and their corresponding storage size. The dataset comprises over 230K **Vulnerabilities** (968MB), 13K **Indicators** (54MB), and 6K **Courses of Action** (25MB), along with other entities such as **Malware**, **Tools**, and **Weaknesses**. Notably, behavioral information is captured through dedicated nodes like **Malware Behavior** (1.7K nodes, 20MB) and **Malware Method**. The relationship graph includes 87,642 edges (351MB), connecting entities across multiple semantic layers.

As shown in Table 5, MATRIX is significantly larger than the two most important STIX datasets, *mitre/cti* and *MBCProject*. While *mitre/cti* contains 4237 nodes and *MBCProject* has 892 nodes, MATRIX is about 541% larger than *mitre/cti* and 2568% larger than *MBCProject* with over 22,910 nodes (excluding vulnerabilities, exploits and vulnerabilities not included in the other datasets). In addition, MATRIX provides a much richer set of real-world examples, including over 13,000 indicators and thousands of actual malware hashes and reports, providing a richer resource for analysis and experimentation.

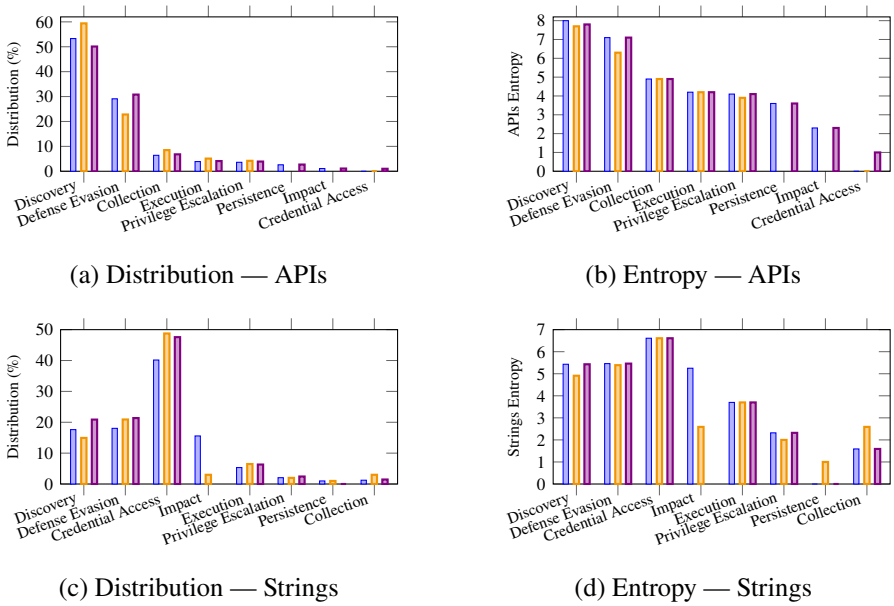
4.3 Application of MATRIX to Threat and Malware Analysis

We present 7 examples of analytical insights that can be derived from the graph, along with the time required to compute them. The analyzes that can be performed are not limited to those we have shown. For example, while we often perform studies on Malware Objectives, similar analyzes can also be performed on Malware Behaviors that are more complex to visualize. More examples can be found here[5]. The first five examples were determined using the graph based on information from MITRE, which would otherwise be difficult to recognise without putting this information into a graph. However, the last two examples depend on the number of real hashes collected; the more malware samples collected, the more accurate the analysis will be.

Comparison of Malware Objectives Based on API and String Correlation. Fig. 10 compares the objectives of 60 Ransomware families, 38 Spyware families, and 112 Trojan families based on APIs and strings extracted from detection rules in *Malware Behavior* nodes. The graphs were created by linking the behaviors of each malware to its objectives and measuring the correlation between APIs, strings and objectives. The distribution shows how often APIs or strings are correlated with an objective, while the entropy [135] indicates the variety of APIs or strings used to reach each objective. Fig. 10a shows that all malware types correlate strongly with the objectives *Discovery* and *Defense Evasion* based on APIs. Ransomware and Trojan have a low correlation with *Impact* and *Persistence*, while spyware has none. Trojan are the only ones that show a low correlation with *Credential Access*. Fig. 10b shows higher API entropy for *Discovery* and *Defense Evasion* for ransomware and Trojan, while spyware uses more predictable APIs. Fig. 10c shows that all malware types correlate strongly with *Credential Access* based on strings, but Trojan show no correlation with *Impact* and *Persistence*. Finally, Fig. 10d shows that ransomware and Trojan use more distinct strings for *Discovery* and *Defense Evasion*, while spyware is more predictable across all objectives. The time required to obtain these results is approximately 0.06s.

Analyzing Data Component Impact on Malware Categories. Figure 11 shows the impact of various common *Data Components* on 60 Ransomware families, 38 Spyware families, 112 Trojan families and 15 Worm families. The impact is calculated from the frequency with which each Data Component contributes to the detection of a malware type in relation to the total data components that affect this malware. *Process Creation* and *Command Execution* are very influential for all malware types, especially spyware, suggesting that system-level behaviors

[5] https://github.com/MATRIX-Malware-Analysis/MATRIX/tree/main/EXAMPLES



(a) Distribution — APIs

(b) Entropy — APIs

(c) Distribution — Strings

(d) Entropy — Strings

Figure 10: Comparison of percentage distribution and entropy of objectives across Ransomware, Spyware, and Trojan based on unique APIs (top row) and Strings (bottom row).

are important detection indicators. Spyware also shows a greater reliance on *OS API Execution*, *Script Execution* and *File Access*, reflecting the use of commands and file operations for malicious purposes. Trojan are also characterized by their dependency on *OS API Execution*, but also on *Network Traffic Flow* and *Connection Metadata*, so network monitoring is crucial for their detection. Both the worm and the ransomware affect *Service Metadata* and *Windows Registry Key Modification*, probably to persist in the system and maintain control by changing critical settings. On the other hand, other data components such as *Process Modification* and *File Creation* have minimal impact on all malware types, making these behaviors less important for detecting these specific threats. The time required to obtain these results is approximately 0.3 ms.

Prioritizing Mitigation Techniques Based on Malware Type. Figure 12 shows the impact of different mitigation strategies (Course of Action) on 60 Ransomware families, 38 Spyware families, 112 Trojan families and 15 Worm families. The impact is based on how often each strategy effectively defends

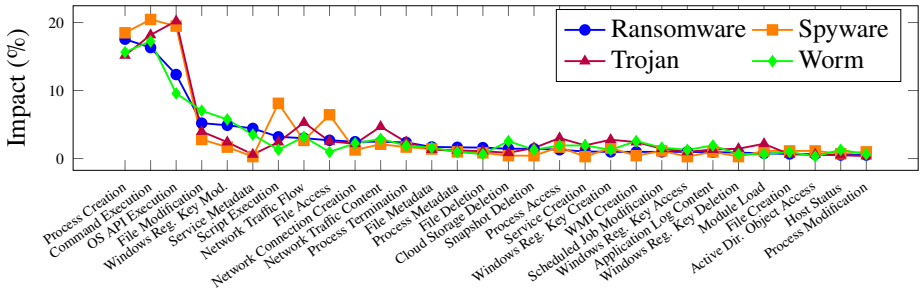


Figure 11: Impact of various common *Data Components* on Ransomware, Spyware, Trojan and worms

against a malware type in relation to the total *Courses of Action* that affect that malware. The graph shows that spyware relies heavily on *Data Loss Prevention* and *User Training*, underlining the importance of educating users and preventing data exfiltration. Trojan particularly benefit from *Execution Prevention* and *Endpoint Behavior Prevention on Endpoint*, highlighting the need to block unauthorized execution and monitor malicious behavior. Ransomware is primarily influenced by *User Account Management* and *Restrict File and Directory Permissions*, which emphasizes the importance of managing access rights. Worms, on the other hand, are strongly influenced by *User Account Management* and *Restrict File and Directory Permissions*, showing that restrictions and user permission management are important strategies. Less effective strategies such as *Limit Access to Resources Over Network* and *Account Use Policies* play a lesser role in containing all malware types. The time required to obtain these results is approximately 0.04s.

How Key Techniques Connect and Support Diverse Malware Types. Measuring *Betweenness Centrality* [136] allows us to identify which *Techniques* play a crucial role in connecting different malware categories. The objective is to pinpoint key actions that multiple malware families depend on for propagation, persistence, or execution. By targeting techniques with high betweenness centrality, defense strategies can effectively disrupt multiple malware types simultaneously.

As illustrated in Figure 14, *T1082 (System Information Discovery)* exhibits the highest centrality for Ransomware-RAT and RAT-Spyware connections, a finding consistent with reports from [137] [138]. Similarly, *T1105 (Ingress Tool Transfer)* is critical for Backdoor-Ransomware and Backdoor-Worm relationships, as confirmed by [139] [140] and [141]. Additionally, *T1140 (Deobfuscation/Decoding)* serves as a central technique linking Backdoor-Spyware and RAT-Spyware, corroborated

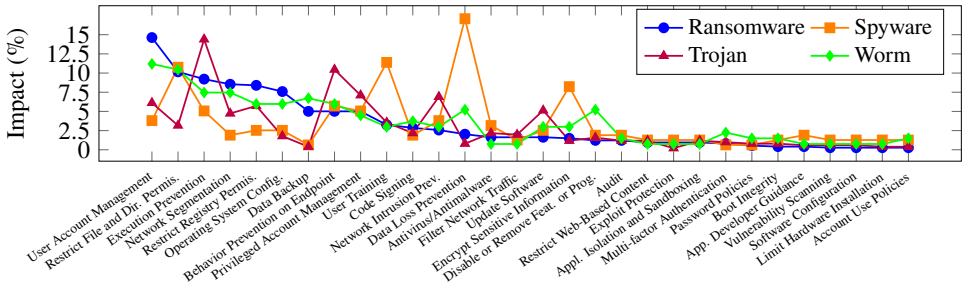


Figure 12: Impact of different mitigation strategies *(Course of Action)* on Ransomware, Spyware, Trojan and Worm

by findings in [138]. *T1210 (Command and Scripting Interpreter)*, cited as first techniques in the top-10 by [141] exhibits highest centrality for Ransomware-RAT and Backdoor-Ransomware.

Overall, techniques characterized by high interdependence, such as deobfuscation and system discovery, act as essential links between different malware categories. This makes them prime targets for disrupting malware operations and strengthening cybersecurity defenses. The computational time required to obtain these results is 5.9 seconds.

Evaluating the Importance of Malware Techniques Using PageRank Analysis. Figure 13 shows the PageRank [142] values for the techniques used by 60 Ransomware families, 38 Spyware families, 112 Trojan families, 15 Worm families, 15 RAT families and 208 Backdoor families. A higher PageRank indicates that a particular technique plays a greater role in the malware's operations or is used more frequently. Worms have consistently high PageRank values for several key techniques, such as *File and Directory Discovery*, as confirmed by [141] and *Native API*, which are essential for their distribution and operation. This means that the worms rely on a few key actions, making these techniques prime targets for disrupting their spread. Backdoor relies on techniques such as *Ingress Tool Transfer*, *Web Protocols* (also very important for spyware) and *Windows Command Shell*, as confirmed by [139], to gain unauthorized access and assert itself in a system. In particular, ransomware relies on techniques such as *Inhibit System Recovery*, also this confirmed by [141] and *Native API* to disable restore options and manipulate system-level functions, making it more difficult for users to restore their system. In contrast, RAT and spyware have lower PageRank scores, suggesting that they use a wider range of techniques without relying heavily on a single action. Even though

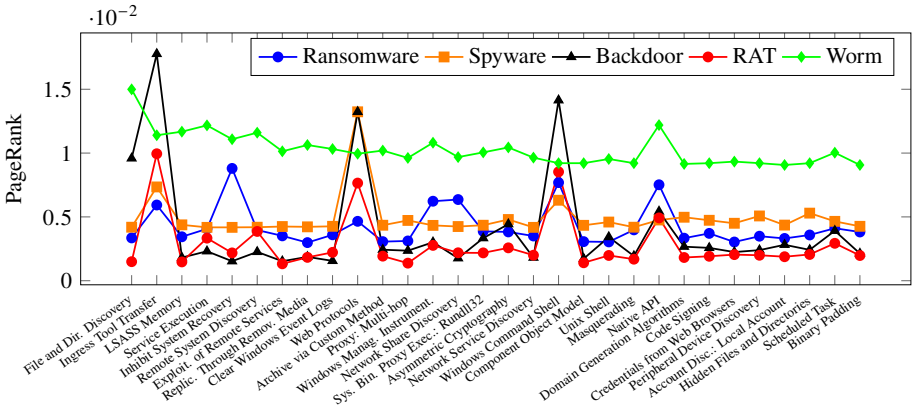


Figure 13: PageRank values for Techniques used by Ransomware, Spyware, Backdoor, RAT and Worm

these types of malware spread their operations over several techniques, focusing on a wide range of defense strategies can still help mitigate their impact. The time required to obtain these results is approximately 0.15s.

API Correlation Analysis in real Ransomware and Spyware Samples. Table 6 shows the top three APIs used by *Ransomware* and *Spyware* (we used the same families of the previous use cases) for each Objective and indicates the percentage correlation between the API and the objective. The data, which comes from CAPE Sandbox [6] reports on real malware samples from VirusShare, provides insight into the specific APIs that are critical to the operations of these malware types. For *Ransomware*, APIs such as *GetNumaHighestNodeN.* and *GetFileVersionInfoW* are often associated with *Execution*, while APIs such as *BaseDllReadWriteIniFile* are important for *Defense Evasion*. For *Discovery*, the ransomware relies heavily on network-related APIs such as *getaddrinfo* and *VarI4FromStr* and shows a strong correlation (80%). For *Command and Control*, the ransomware uses APIs such as *GetProcessAffinityM.* and *GetThreadContext* with a correlation of 100%, indicating a strong focus on process control. For *Credential Access* and *Privilege Escalation*, APIs such as *InstallApplication* and *SHAutoComplete* are used, which focus on obtaining user credentials and elevating privileges. *Spyware* shows a different pattern and focuses on APIs such as *_CorExeMain* for *Execution* and *GdiAddGlsBounds* for *Defense Evasion*, both with a correlation of 50%. For *Discovery*, APIs such as

[6] https://capev2.readthedocs.io/en/latest/#

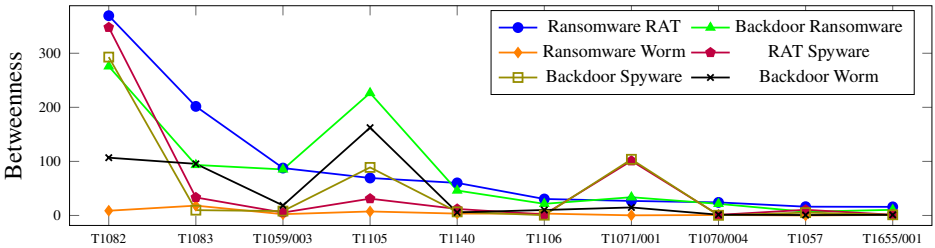


Figure 14: Betweenness Centrality of Behaviors across Malware Categories

NtResumeProcess help spyware to collect system information, while *Command and Control* relies on APIs such as *AreFileApisANSI* with 100% correlation. Spyware also focuses on *Credential Access* with APIs such as *PropVariantClear* used to retrieve sensitive information, and for *Privilege Escalation*, spyware uses similar APIs to ransomware, such as *SHAutoComplete*, with a higher correlation. Overall, *Ransomware* focuses on network detection and system control, while *Spyware* focuses more on data collection and maintaining control by communicating with remote servers. The time required to execute this experiment is 3.45s.

Malware	Execution	Defense Evasion	Discovery	Command and Control
Ransomw.	GetNumHigh.NodeN. (50%)	BaseDllReadW. (60%)	getaddrinfo (80%)	GetProcessAff. (100%)
	GetFileVersionInf. (50%)	BaseFormatObj. (60%)	VarI4FromStr (80%)	GetThreadContext (100%)
	InitializeAcl (50%)	Basep8BitStr. (60%)	getprotobyname (80%)	RemoveDirectoryW (100%)
Spyware	_CorExeMain (50%)	GdiAddGlsBounds (50%)	NtResumeProc. (75%)	AreFileApisANSI (100%)
	LocaleNameToLCID (50%)	GdiAddGlsRecord (50%)	NtSuspendProc. (75%)	lstrcatA (100%)
	GetWindowRect (50%)	HeapSetInfo. (50%)	ImmIsIME (28.57%)	QueryPerform.Fre. (100%)

Malware	Collection	Credential Access	Privilege Escalation
Ransomw.	Net.UncComp.Nam. (28.57%)	InstallApplication (17.39%)	SHAutoComplete (28.57%)
	Net.Rem.NameVal. (28.57%)	RtlW-GetCur. (15.85%)	SetWindowThem. (28.57%)
	NPGetConn.Perf. (23.08%)	RtlWow64IsWowG. (15.85%)	ImmIsIME (28.57%)
Spyware	Rt.64GetCurrentM. (25%)	PropVariantClear (40%)	ImmIsIME (28.57%)
	Rt.64IsWowGuestM. (25%)	RegQuer.Val.W (40%)	SHAutoComplete (28.57%)
	PropVar.Clear (20%)	SetSecurityDescr. (40%)	OpenThemeData (28.57%)

Table 6: Top 3 API used by Ransomware and Spyware for each Objective with percentages.

Behavioral and Technical Similarities Between Two Emotet Samples. Table 7 summarises the common API calls, registry keys, loaded modules and MITRE

ATT&CK techniques observed in two Emotet malware samples:

- 2fd433c3ff68507ddbf0ec3e90a6320b35b44c8089504403c457bc9819190a0a
- 214946b987ad69fa46f1d27ab35026b856a4fcd2abd46b0b5ba86dc71be58d89

The data was extracted from CAPE sandbox reports with real malware samples from VirusShare. The Jaccard similarity [143] score of *0.97* indicates a very high similarity between the two malware samples based on their common characteristics. The listed API calls, such as `VirtualProtect`, `GetCPInfo` and `CloseHandle`, show that both malware samples are involved in similar activities, including process control, memory management and system information retrieval. These are typical actions used by Emotet to achieve persistence and execute its malicious operations. MITRE techniques used by both malware samples include *Credential Dumping*, *Virtualization/Sandbox Evasion* and *Impair Defenses*, suggesting that they focus heavily on credential evasion and theft. This reflects the typical behaviour of Emotet, which is known for its ability to bypass security measures and collect sensitive information from infected systems. Both samples accessed similar registry keys, such as *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\MachineGuid*, which refer to common methods of interacting with system configurations. The loaded modules, including `BCRYPT.DLL` and `WININET.DLL`, also confirmed by [144], indicate that both examples use the same Windows libraries for cryptographic functions, Internet communication and shell operations. The time required to obtain these results is 0.2 s.

Category	Values
Calls Highlighted	VirtualProtect, CryptStringToBinaryA, GetCurrentHwProfileA, CloseHandle, TlsGetValue, EnterCriticalSection, GetLastError, srand, IsValidCodePage, GlobalLock, VirtualAlloc, CreateToolhelp32Snapshot, GetCurrentThreadId, CoCreateInstance, GetCPInfo, HeapAlloc, LeaveCriticalSection, InterlockedDecrement, GetProcessHeap, GetVersionExA, Process32Next, RegOpenKeyExA, GetModuleHandleW, IsDebuggerPresent, NtQueryInformationProcess ...
MITRE Techniques	T1503, T1497.001, T1003, T1552.001, T1005, T1562, T1081, T1071, T1032, T1555, T1106, T1497, T1562.001, T1071.001, T1012, T1552, T1082, T1089, T1057, T1555.003, T1539, T1518.
Reg. Keys Opened	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\MachineGuid , HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Wow6432Node\Interface\{8A40A45D-055C-4B62-ABD7-6D613E2CEAEC}\ProxyStubClsid32 ...
Modules Loaded	BCRYPT.DLL, NTMARTA.DLL, WINHTTP.DLL, SHELL32.DLL, GDIPLUS.DLL, CRYPT32.DLL, NTDLL.DLL, SECHOST.DLL, WININET.DLL, WS2_32.DLL, CRYPTBASE.DLL, CFGMGR32.DLL, OLE32.DLL, ADVAPI32.DLL, `API-MS-WIN-DOWNLEVEL-ADVAPI32-L1-1-0.DLL`, GDI32.DLL, RPCRT4.DLL, SHLWAPI.DLL, RSTRTMGR.DLL, USER32.DLL, MSVCR100.DLL, NSI.DLL, KERNEL32.DLL ...

Table 7: Summary of Common Calls, MITRE Techniques between two Emotet hashes.

4.4 Conclusions

This chapter presented *MATRIX*, a comprehensive and extensible graph-based framework for malware and threat analysis. By aligning heterogeneous sources (e.g., `MITRE ATT&CK`, `MBCProject`, `CAPEC`, `DEF3ND`, `CWE/CVE`, Metasploit) under the STIX 2.1 standard, MATRIX establishes a unified, semantically coherent knowledge layer that connects malware, techniques, vulnerabilities, exploits, detections, and mitigations within a single, dynamically updated graph.

From a data perspective, MATRIX substantially extends the coverage of existing STIX repositories, integrating more than ten thousand real-world malware samples (from `VirusTotal`, `MalwareBazaar`, `VirusShare`) and linking them to rules (CAPA, Sigma), behaviors, and objectives. The resulting knowledge graph enables higher-resolution analyses, behavioral comparison across families, objective and tactic-level profiling, and mitigation impact assessment, illustrating how graph structure supports contextual reasoning that is unattainable with sequence-only representations. Methodologically, this chapter contributes (i) a dynamic knowledge structure capable of coherently connecting heterogeneous cybersecurity entities, and (ii) evidence that graph-based representations improve contextualization and cross-source reasoning for cyber-threat analysis. Case studies demonstrate practical value for analyst workflows, while the automated ingestion pipeline ensures that MATRIX evolves with the threat landscape. These results set the stage for the next step of the work: coupling structured knowledge with adaptive language models to deliver reliable, timely assistance. However, before diving into the construction Retrieval Augmented Generation (RAG) cybersecurity assistant (addressed in Chapter 6), the following chapter focuses on understanding the challenges and the possible solutions of building such RAG systems.

5 Cybersecurity with LLMs and RAGs: Challenges and Innovations

Before introducing a concrete architecture for retrieval-augmented assistance in cybersecurity, it was necessary to conduct a systematic analysis of the challenges that characterize the integration of Large Language Models (LLMs) and Retrieval Augmented Generation (RAG) systems in this domain. The promise of generative AI in cybersecurity lies in its ability to accelerate detection, response, and analysis processes, thus addressing both the skills shortage and the growing complexity of threats. LLMs combined with RAG pipelines are particularly well suited to this task, as they enable models to access and contextualize external knowledge beyond their parametric memory. However, the application of these systems to cybersecurity raises specific difficulties that must be carefully understood before moving to fully fledged implementations.

A first limitation is represented by the phenomenon of hallucinations: LLMs, when confronted with underrepresented or highly technical content, often produce inaccurate or misleading outputs. This issue is particularly critical in cybersecurity, where factual reliability is non-negotiable. While RAG can mitigate hallucinations by grounding responses in retrieved knowledge, it introduces its own challenges. Scalability and retrieval latency limit the responsiveness of systems that need to handle large and heterogeneous datasets in real time. The choice of embeddings and similarity thresholds strongly impacts the quality of retrieval, and inappropriate settings can cause relevant information to be overlooked. Moreover, multi-hop reasoning, a common requirement in threat analysis, where queries often span multiple entities and relationships, remains an unsolved problem for many current RAG implementations.

This chapter therefore focused on examining these issues in depth. The analysis considered use cases such as vulnerability detection (e.g., answering CVE-related questions) and malware characterization, showing where current LLMs fail and why. By combining case studies with empirical evidence, the study highlighted both the benefits and the limitations of existing models when applied to cybersecurity contexts. For each identified limitation, potential solutions were proposed, ranging from optimized retrieval strategies to more suitable evaluation criteria for assessing factual accuracy and multi-hop reasoning capabilities.

The contribution of this analysis consists in outlining a set of guidelines and design considerations that directly informed the development of *MoRSE* in Chapter 6. In this way, the study served as a necessary step in bridging the gap between the

availability of structured knowledge sources (as explored with MATRIX) and the design of retrieval-augmented assistants capable of reliably supporting cyber threat intelligence workflows.

5.1 Cybersecurity Expertise and Threat Intelligence

Challenges. Large language models often struggle with technical questions where accuracy is critical. This problem is commonly referred to as *Hallucination*, meaning that the models produce answers that are not true or reliable.

The problem of truthfulness in LLMs can be understood in terms of statistical uncertainty [145, 146], which comes in two types: **epistemic** and **aleatoric** [147]. *Epistemic uncertainty* arises from a lack of knowledge about the correct answer. This can happen if the model does not have enough training data or if its capacity is limited. *Aleatory uncertainty* happens due to the inherent randomness of the prediction task. For example, some questions may have multiple correct answers.

In the field of cybersecurity, where threats evolve rapidly and precise responses are required, hallucinations in LLMs are particularly concerning. High epistemic certainty, i.e. a strong understanding of the correct responses, is essential for accurate predictions in this domain. There is a strong correlation between the number of times an LLM was exposed to documents on specific fact-based questions during pretraining and its ability to answer these questions correctly [148]. Despite their size, even very large models have difficulty with questions that find little support in the training data. This shows that it is a major challenge to train LLMs to deal effectively with diverse and rare information.

Example 1: Hallucination on CVE Questions.

Question: What is CVE-2019-15949?
Ground Truth: CVE-2019-15949 refers to a security vulnerability that affects Nagios XI versions before 5.6.6, which allows remote command execution as root.
GPT-4 Response: CVE-2019-15949 refers to a security vulnerability that affects certain versions of phpMyAdmin, a very popular free and open-source administration tool for MySQL and MariaDB databases.
MIXTRAL 7x8 Response: CVE-2019-15949 is a vulnerability related to the Remote Administration Tool (RAT) called Imminent Monitor.
GPT-3.5 Response: CVE-2019-15949 refers to a vulnerability in the Apache HTTP Server, specifically in the mod_proxy module.

An example of hallucinations can be seen in Example 1, where models such as GPT-4[7], GPT-3.5[8] and MIXTRAL 7x8 [149] fail to find the correct answer to the question: *What is CVE-2019-15949*. Example 1 demonstrates that none of these models could accurately describe CVE-2019-15949, each giving different incorrect descriptions. The ground truth is that CVE-2019-15949 refers to a security vulnerability that affects Nagios XI versions before 5.6.6, allowing remote command execution as root. In contrast, GPT-4 incorrectly describes it as a vulnerability affecting phpMyAdmin. MIXTRAL 7x8 incorrectly relates it to a Remote Administration Tool called Imminent Monitor. GPT-3.5 incorrectly states it is a vulnerability in the Apache HTTP Server.
Table 8 presents a performance comparison between GPT-4 and MIXTRAL 7x8 in identifying correct responses to a set of 50 questions related to CVEs. The metric *Accuracy* quantifies the percentage of correctly identified CVEs. Additionally, the metrics *Similarity* and *Correctness* are evaluated using the RAGAS framework [150], as detailed in Section 5.5.

Table 8: Performance comparison of models on 50 CVE Queries.

Model	**Similarity** (μ, σ)	**Correctness** (μ, σ)	**Accuracy**
GPT-4	0.826, 0.0310	0.503, 0.0508	0.50
MIXTRAL 7x8	0.845, 0.039	0.53, 0.089	0.51

Possible Solution. One possible solution to this type of problem is to enrich the context that forms the model prompt. Enriching the context in model prompts, especially through Retrieval Augmented Generation architectures, can significantly improve the quality of model responses by providing the model with more relevant and richer information on which to base its output (see Section 5.4).

Example 2: Context Enriching

Question: What is CVE-2019-15949?
GPT-4 Response: CVE-2019-15949 is a critical security vulnerability in Nagios XI, versions prior to 5.6.6. This vulnerability allows for remote command execution as root, which can be exploited under certain conditions.

[7] https://chatgpt.com/
[8] https://chatgpt.com/?model=text-davinci-002-render-sha

RAG can mitigate some of the limitations faced by LLMs by reducing their dependency on having directly seen relevant information during training [148]. When models can access relevant documents dynamically, they perform better on questions, especially those with sparse data support. Example 2 shows how GPT-4 has been able to provide the accurate answer if allowed to enrich its context, searching on the web before the generation of the answer.
However, to ensure the effectiveness of RAG systems, measures outlined in Section 5.4 should be implemented. Table 9 compares the token generation probabilities between two language models, GEMMA 2b [151] and Mistral 7b [152], for a query related to *EternalBlue* vulnerability [153]: *What is the CVE related to EternalBlue Vulnerability?*. Mistral 7b, a larger model, correctly identifies the CVE, whereas GEMMA 2b makes errors and shows greater uncertainty. This highlights the importance of developing specific vocabularies to make LLMs proficient in cybersecurity. As seen, the CVE numbers are tokenized individually, increasing the probability of errors in both cases.

GEMMA 2b	**Probability**	**Mistral 7b**	**Probability**
The	92.01%	The	75.63%
CVE	97.99%	E	51.44%
related	98.04%	ternal	99.96%
to	100.00%	Blue	99.98%
Eternal	99.89%	vulner	98.92%
Blue	99.68%	ability	100.00%
vulnerability	99.99%	is	92.90%
is	99.89%	related	67.36%
CVE	80.88%	to	100.00%
-	99.73%	C	83.02%
2	99.95%	VE	100.00%
0	99.84%	-	100.00%
1	62.65%	2	100.00%
7	52.36%	0	100.00%
-	99.98%	1	100.00%
0	74.16%	7	100.00%
1	39.62%	-	100.00%
5	28.39%	0	100.00%
2	20.35%	1	100.00%
.	95.10%	4	100.00%
EOS	99.89%	4	98.56%

Table 9: Token Generations and Probabilities for GEMMA 2b and mistral 7b Models

5.2 Malware Analysis

Challenges. When dealing with malware analysis, a major technical challenge for LLMs is the limitation of context length, which is particularly problematic for frameworks that rely on API sequence analysis to detect malware [154]. LLMs have a fixed input size, which limits the number of tokens they can process at once. This complicates the analysis of long API call sequences, which are necessary for the detection of malicious behavior.
This limitation can cause data to be truncated or split, losing important information and disrupting the temporal and logical relationships within API sequences, resulting in inaccurate detection. In addition, maintaining LLM effectiveness against evolving threats requires regular updates, which increases operational overhead.

Possible Solutions. To circumvent the limitations of LLM context length, techniques such as *segmentation* [155] and *windowing* [156] split long API sequences into overlapping segments, preserving temporal and logical relationships for accurate malware detection. Models such as *Mamba* [157] and *Hyena* [158] further improve the handling of long sequences. Mamba uses selective state space models (SSMs) for input-dependent parameterization and effectively manages sequences up to one million in length, outperforming traditional transformers. Hyena combines long convolutions and data-driven gating for efficient, sub-quadratic alternatives to standard attention mechanisms.

5.3 Vulnerability Detection

Challenges The integration of LLMs with techniques such as fuzzing has improved test case generation and vulnerability detection capabilities in complex software systems [159]. Despite advances in vulnerability detection, there are several challenges that hinder the practical use of LLMs in cybersecurity. LLMs suffer from inconsistency and non-determinism as they produce different results under similar conditions, which is problematic for tasks that require high reliability [160]. They often misinterpret the source code, leading to inaccurate security assessments, and their sensitivity to minor code changes indicates a lack of robustness [160]. Furthermore, LLMs rely on the integration of additional structural and contextual data to improve performance, as shown by the GRACE model [26]. Their significant computational cost may preclude resource-constrained organizations, and handling large datasets raises legal and privacy concerns, making compliance with data protection regulations difficult. The dynamic nature of cybersecurity threats

necessitates frequent updates to LLMs, making them difficult to maintain and scale. These challenges underscore the need to adapt LLM technologies to meet the stringent requirements of security applications.

Possible Solutions Integrating LLMs with planning or rule-based systems [161] can leverage both data-driven insights and established security protocols, thereby enhancing the models' reasoning capabilities in complex environments.
Additionally, incorporating adversarial training can make LLMs more resilient to minor code modifications, enhancing their ability to generalize and reducing sensitivity to superficial changes. Developing models capable of continuous learning will allow them to adapt to new and emerging threats, maintaining their effectiveness over time.

5.4 RAG Limitations and Possible Solutions

Retrieval-Augmented Generation (RAG) extends traditional language models by integrating *Non-Parametric Knowledge Bases* [162] through *Retrievers* [163] [164] to improve accuracy and relevance for domain-specific queries [165]. These *retrievers* retrieve contextually relevant information from their own knowledge base, inserting them into a *context* that the LLM uses to generate answers. The information in the knowledge base is represented by *Embeddings* [166]. These embeddings encode terms as vectors, so that similar terms have vectors that are close to each other. This proximity in vector space allows the embeddings to facilitate semantic matching between user queries and the information in the knowledge base.

Challenges. Retrieval-Augmented Generation (RAG) systems face several key challenges that impact their effectiveness and operational efficiency:

1. **Scalability**: RAG systems' knowledge bases can grow large, consuming significant memory and limiting deployment in resource-constrained environments.

2. **Retriever Speed**: Fast retrieval of relevant documents is crucial in cybersecurity. Slow retriever response times can delay critical security measures and response to threats.

3. **Choice of Embeddings and Similarity Threshold**: Selecting the appropriate embeddings and setting the right similarity threshold for matching queries to

documents in the knowledge base are critical. Incorrect choices can lead to poor retrieval accuracy, impacting the quality of the generated responses.

4. **Multi-Hop Question Answering**: Cybersecurity often involves complex, multi-hop queries that require drawing conclusions from interconnected entities. RAG systems often struggle with accurately answering multi-hop questions, where the answer requires synthesizing information from multiple entities [167].

Potential solutions for improving RAG systems in cybersecurity To improve RAG systems in cybersecurity, consider these targeted solutions:
Specialized Retrievers: Use different retrievers for different cybersecurity topics to better adapt to specific data characteristics.
Improve the retrieval speed: Restructure the knowledge base data by generating questions for each text section. Use these questions to match incoming queries and speed up the query by narrowing down the search space.
Different embeddings for different retrievers: Use different embeddings to minimize bias and improve the quality and impartiality of search results.
Setting similarity thresholds: Test retrievers with different questions and set similarity thresholds based on the median or third quartile of the similarity distributions of the most relevant documents. This approach provides a balance between precision and recognition. See Algorithm 2 for the procedure.

Algorithm 2: Setting Similarity Thresholds for Document Retrieval

```
Function SetSimilarityThresholds(queries, documents, N)
    distribution ← empty list
    foreach query in queries do
        scores ← ComputeSimilarities(query, documents)
        topScores ← GetTopN(scores, N)
        append topScores to distribution
    return DetermineThreshold(distribution, documents)
Function ComputeSimilarities(query, documents)
    return list of similarity scores between query and documents
Function GetTopN(scores, N)
    return top N values from scores
Function DetermineThreshold(distribution, documents)
    if documents are text-heavy then
        return median(distribution)
    else
        return thirdQuartile(distribution)
```

Processing Multi-Hop Queries: Implement entity recognition during pre-

processing to create new questions for each identified entity, capturing comprehensive information for more accurate answers. As shown in Figure 15, the process for handling multi-hop requests begins with a complex user query: *What is the difference between Carberp and Rovnix?*. The system breaks down this query into simpler single-hop queries through the following steps:

1. **Entity Extraction:** Extracts key entities from the query: *Carberp* and *Rovnix*.
2. **Generation of New Queries:** Formulates two single-hop questions: *What is Carberp?* and *What is Rovnix?*
3. **Query Processing via RAG:** Processes each new query and the original query with the RAG system, retrieves relevant information, and summarizes it into a common context for the LLM to generate a response.

This approach enables the RAG system to efficiently handle multi-hop queries by simplifying them into single-hop queries.

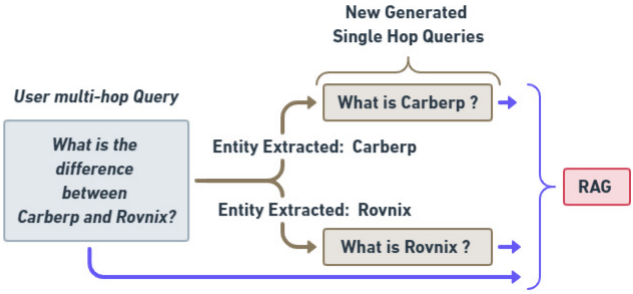


Figure 15: Handling Multi-Hop Queries.

5.5 Challenges in Evaluating RAG Systems and LLMs in Cybersecurity

The evaluation of Retrieval Augmented Generation systems and Large Language Models in the field of cybersecurity poses a particular challenge due to their dual role in information retrieval and content generation. The complexity of cybersecurity tasks combined with the lack of standardized benchmarks that cover a wide range of real-world deployment scenarios significantly complicates this evaluation process [150, 168, 169].

Cybersecurity Dataset Limitations: The development of LLM in cybersecurity is hindered by the lack of specialized datasets, as sensitive information limits the

availability of data. This limitation hinders the learning of complex cybersecurity concepts. One exception is the dataset *Cybermetric* [170] with 10,000 questions on cybersecurity.

Accuracy of information retrieval: Ensuring the accuracy of information retrieved by RAG systems is critical in the cybersecurity field, as relevance and precision influence decision-making. Traditional metrics such as precision, recall and F1 score cannot fully capture the nuances required.

Effectiveness of LLMs in utilizing retrieved information: Evaluating how effectively LLMs integrate retrieved data into responses is critical. Metrics must assess the coherence and contextual appropriateness of the content generated in response to cybersecurity threats.

Quality of the generated content: The quality of the content of RAG systems and LLMs must be actionable, accurate and specific to cybersecurity. Comprehensive assessment frameworks are needed to evaluate the utility, accuracy and actionability of the generated responses.

Adaptation to real-world performance: Traditional evaluation methods often fail in dynamic cybersecurity environments. Evaluation frameworks should incorporate adaptive testing mechanisms to simulate real-world scenarios and measure LLM performance as threats evolve [171].

Possible Evaluation Solutions. The RAGAS framework [150] offers customized metrics to evaluate RAG systems and LLMs in cybersecurity: *Answer Relevance* (cosine similarity between question embeddings and generated answers), *Answer Similarity* (semantic congruence with correct answers), and *Answer Correctness* (combining factual accuracy and semantic similarity). An innovative validation method involves using GPT-4 as a judge to evaluate model performance, leveraging its high agreement rate with human judgment (80%) [172]. GPT-4 assesses the quality of responses by selecting the most appropriate among competing models' answers to various cybersecurity queries.

5.6 Conclusions

This chapter explored the practical interaction between large language models (LLMs) and retrieval-augmented generation (RAG) within the cybersecurity domain, clarifying both the opportunities and limitations before transitioning from MATRIX to a concrete assistant. The analysis confirmed that off-the-shelf LLMs perform poorly on tasks where factual accuracy and domain completeness are essential, such as questions on CVEs, and that the long, heterogeneous contexts typical of

malware analysis further amplify hallucinations and fragile reasoning. Grounding through RAG alleviates these weaknesses by providing up-to-date, verifiable evidence, yet it introduces its own operational challenges: scalability and latency with large corpora, sensitivity to retriever design choices (embeddings, indexing strategies, similarity thresholds), and difficulty handling multi-hop queries that require reasoning across multiple entities and relations. The chapter also emphasized the lack of comprehensive evaluation protocols, the scarcity of benchmarks, and the dual nature of RAG systems (retrieval and generation), all of which complicate rigorous and reproducible assessment.

The following Chapter 6 puts these findings into practice through *MoRSE*, a cybersecurity-oriented RAG framework that integrates multiple retrievers and adopts design choices informed by the challenges and insights discussed in this chapter. MoRSE seeks to address the following research question **Q2**: *What kind of retrieval-augmented generation (RAG) system can exploit this structured knowledge, and what capabilities must it have, for example, handling multi-hop queries (i.e., questions that require reasoning across multiple entities and relationships), reducing hallucinations, and ensuring reliable and timely responses?*

6 Bridging the Gap in Cybersecurity Expertise with Retrieval Augmented Generation

Building on the conclusions drawn earlier, the third stage of this research, presented in this chapter, takes a step forward by building a retrieval-augmented generation (RAG) system tailored to cyber threat intelligence. The underlying idea is that large language models should be conceived not as fixed containers of knowledge, but as dynamic reasoning components that interact with external, continuously updated sources of information. This approach enables them to overcome intrinsic parametric limitations, mitigate hallucinations, and deliver responses grounded in both structured and unstructured cybersecurity data.

To this end, this chapter introduces ***MoRSE (Mixture of RAGs Security Experts)***, a novel framework for cybersecurity question answering. MoRSE combines the structured knowledge with the adaptability of LLMs through a cascade of specialized retrievers. The architecture distinguishes between a Structured RAG, optimized for fast and precise retrieval from curated sources, and an Unstructured RAG, which broadens coverage by accessing raw, less-processed data when necessary. Each retriever acts as a domain expert, covering areas such as vulnerabilities, exploits, or adversarial techniques, ensuring that the system delivers accurate and context-aware responses.

This chapter contributes in three main ways. First, it designs a domain-specific RAG system that goes beyond generic LLM assistants to provide targeted support for cybersecurity analysts. Second, it shows how grounding retrieval in both structured and unstructured sources improves reliability, contextualization, and adaptability. Third, it demonstrates the role of RAG-based assistants in bridging the gap between raw cybersecurity data and actionable insights, thus addressing the practical needs of experts operating in a rapidly evolving threat landscape. An high level conceptualization of MoRSE is shown in 16.

6.1 Background

This section provides an overview of the basic concepts necessary for understanding the architecture of MoRSE.

Large Language Models. Large language models (LLMs), based on the Transformer architecture [173], mark a significant advancement in Natural Language Processing (NLP). Trained on vast text datasets, LLMs can generate coherent, contextually relevant texts and perform tasks like translation, summarization, and

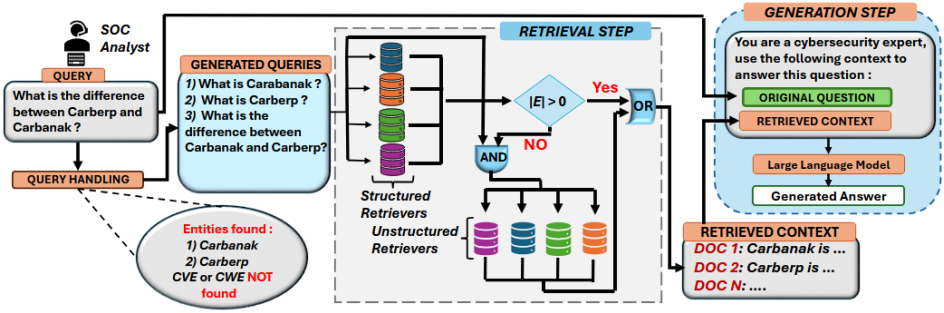


Figure 16: High Level MoRSE Conceptualization.

question answering. Models like BERT [174] showcased the potential of bidirectional unsupervised training, revolutionizing language understanding. GPT-3 [175] further expanded scalability, enabling LLMs to excel in various NLP tasks without task-specific training.

Retrieval Augmented Generation. Retrieval-Augmented Generation (RAG) combines traditional language models with external databases to improve natural language processing (NLP) tasks [165]. RAG models use a retriever to retrieve relevant information and a generator to generate answers based on the retrieved information. This improves accuracy and relevance, especially for domain-specific queries. RAG's strengths lie in its ability to update knowledge bases without retraining and customise components for specific tasks such as cybersecurity. However, RAG struggles with latency and scalability issues.

Key Terminology. The following terms are central to the discussion in this paper. A **Retriever** refers to a component that identifies and collects relevant information or documents from a knowledge base, providing context for LLMs to generate accurate answers. The **Knowledge Base** is a repository that the retriever accesses to find relevant data or documents that are critical to generating knowledgeable answers. The **Embeddings** are numerical representations of text where similar terms have narrower vector representations that enable effective comparison between queries and the knowledge base [166]. The **Context** refers to the information that is retrieved to support a query and that provides the necessary background for creating accurate and relevant answers. A Prompt is the structured input that is created from the retrieved context and guides the model in creating coherent and relevant answers.

Semantic Similarity measures how well a user's query matches the information in the knowledge base to ensure the relevance of the retrieved data. MoRSE uses *Cosine Similarity* to evaluate the closeness of embedding vectors [176]. Finally, **Multi-hop questions** are those that require reasoning over multiple connected data points to generate a comprehensive answer.

6.1.1 MoRSE Architecture

In this section we provide a description of the architecture of the MoRSE system, in which the functions of the individual components and their interaction in the processing of requests and the generation of responses are explained. The MoRSE architecture was developed using the Langchain framework[9]. To make it easier to understand, we use various symbols in this explanation. The symbols α and β refer to the embedding models used in the system. Specifically, α stands for the model `BAAI/bge-large-en` [177], while β refers to the model `BAAI/bge-large-en-v1.5` [177]. These models are responsible for generating high-quality embeddings to improve the retrieval capabilities of the system. In addition to the embedding models, γ and θ denote Transformer models used in various components of MoRSE. The symbol γ corresponds to the model `valhalla/t5-base-e2e-qg` [178], which is used for tasks such as question generation, while θ represents the model `dslim/bert-large-NER` [174], which is used for Named Entity Recognition tasks.

6.2 MoRSE Overview

MoRSE consists of two main components: a graphical user interface (GUI) and the MoRSE core. The GUI enables interaction with the user by allowing the input of queries and displaying the answers in a structured way. The MoRSE core consists of three key components, which in turn manage the user query and compose the answer:

Query Handling Module: This module performs the pre-processing of user queries and specializes in the management of multi-hop queries and complex questions, especially in the context of Common Vulnerability and Exposures (CVEs) and Common Weakness Enumerations (CWEs).

Structured RAG: The first of the two RAGs is composed by retrievers that retrieve information from pre-processed, structured data. The pre-processing phase involves converting chunks of text from various sources that are part of the knowledge base,

[9] https://www.langchain.com/

such as academic papers and cybersecurity websites, into well-defined structures. These structures are designed to contain generated questions and contextualized entity descriptions that facilitate the precise retrieval of information in response to user queries.

Unstructured RAG: This RAG is used if the structured RAG could not find a suitable answer. It searches for information in unstructured and unprocessed raw text that belongs to its knowledge base. Accessing unstructured data allows the exploration of data in its original form without the limitations imposed by preprocessing, thus providing a wider range of search options in exchange for a higher response time.

6.3 MoRSE Core

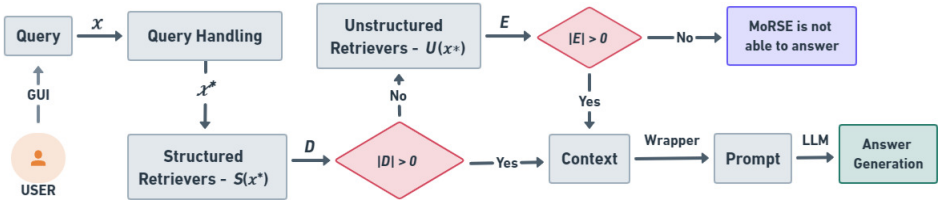


Figure 17: MoRSE Core Workflow.

MoRSE Core Workflow Figure 17 shows the first stage of MoRSE Core process, starting with the *Query Handling* model. This module converts the original query x into an optimized version x^* (see subsection 6.3.1). First, x^* is forwarded to the *Structured RAG* module for processing. The structured RAG path, denoted as $\mathcal{S}$, begins with the *Structured Retrievers*, focused on high accuracy and fast responses to efficiently process most queries. Their primary function, $\mathcal{S}(x^*)$, is to identify and retrieve information pertinent to the query. When activated, the structured retrieval process, which is executed via $\mathcal{S}(x^*)$, assigns a set of potentially relevant documents from a predefined *Knowledge Base* to the query x^*. In particular, $\mathcal{D} = \text{top-k}(\mathcal{S}(x^*))$ represents the selection of the top k documents that $\mathcal{S}$ considers most relevant for the query based on a similarity score. If $\mathcal{D}$ is not empty ($|\mathcal{D}| > 0$), this means that a relevant context has been found. The workflow then proceeds to use this context and moves on to the next phase, where the retrieved information ($\mathcal{D}$) is wrapped in a *Prompt*, which is used by *LLM* to generate a response. If the *Structured Retrievers* do not yield relevant documents ($|\mathcal{D}| = 0$), the workflow

moves to the unstructured path and calls the *Unstructured Retrievers*, denoted as $\mathcal{U}$. At this stage, $\mathcal{E}$ = top-k($\mathcal{U}(x^*)$) represents the set of documents retrieved by $\mathcal{U}$, which are designed to process complex queries that are not readily covered by structured data patterns. After a successful retrieval of relevant information in one of the two ways, indicated by $|\mathcal{D}| > 0$ for structured retrieval or $|\mathcal{E}| > 0$ for unstructured retrieval, the *Wrapper* module integrates the acquired context and generates a prompt for the Large Language Model (LLM). The LLM then performs *Answer Generation*, creating a detailed response to the user's question.

6.3.1 Query Handling

We explain the workflow of the *Query Handling* component in Algorithm 3 and below there are the specific functions and composition of this component:

Functionalities.

- **Multi-Hop Queries**: Manages queries with multiple related entities so that the system can answer complex multi-hop queries. Conventional RAG systems struggle with this due to design limitations and the lack of a dedicated benchmark.
- **Context Enrichment**: Expands the context by generating additional questions for each identified entity, improving the system's ability to provide informed answers.
- **CVE-CWE Query Handling**: Efficiently manage technical queries on Common Vulnerabilities (CVE) and Common Weaknesses (CWE) that are difficult for generative models [179].

Components.

- **User Query**: Initiates the process when a user submits a query via the graphical user interface.
- **CVE-CWE Keyword Extraction**: Extracts keywords related to CVEs and CWEs when a query is received.
- **Get CVE Description**: Retrieves detailed descriptions of CVEs, including information about vulnerabilities, affected software and finders.

- **Get CWE Description**: Retrieves descriptions of CWEs that provide information about the type of software vulnerabilities, potential impact and mitigation strategies.

- **Entity Extractor**: Utilizes the Haystack framework [10] with the θ model to identify and extract relevant entities (people or concepts) from user queries, improving the system's ability to handle Multi-Hop queries.

Algorithm 3: Query Handling Process

Input: A user query q on a specific cybersecurity topic
Output: A series of refined queries Q' for in-depth analysis
Function `VulnerabilityExtractor(`q`)`
    Extract keywords K related to CVE and CWE from q;
    Retrieve detailed descriptions D for each keyword in K;
    Replace each keyword in K with its description in D to form q';
    return q';
Function `EntityExtractor(`q'`)`
    Detect entities E in q' using Haystack with `"dslim/bert-base-NER"`;
    Initialize $Q \leftarrow \{q'\}$;
    foreach *entity* $e \in E$ **do**
        if e *is of type PER (person)* **then**
            Append `"Who is` e`?"` to Q;
        else
            Append `"What is` e`?"` to Q;
    return Q;
Function `QueryHandling(`q`)`
    $q_{\text{vuln}} \leftarrow$ `VulnerabilityExtractor(`q`)`;
    $Q' \leftarrow$ `EntityExtractor(`q_{vuln}`)`;
    return Q';

6.4 Structured RAG

As illustrated in Figure 18, the Structured RAG module works post-*Query Handling* by forwarding refined queries to seven *Parallel Retrievers*, called *Structured Retrievers*, each of which specializes in specific cybersecurity topics. Given a query, the information contained in the knowledge base of a Retriever is inserted into the *Context* if its similarity to the query is above a predefined threshold. In order to

[10] https://haystack.deepset.ai/

establish the threshold for each retriever, we conducted an analysis on the scores of top 50 results from a series of test queries. Thresholds were then determined by assessing the distribution of scores [11]. In particular, we used the median value of the test distributions as threshold for the *MITRE Retriever* and the *Malware Retriever*, as these typically retrieve shorter texts. For *Question Retrieval System*, *CWE Retriever*, *Metasploit Retriever* and *Entity Retriever*, we chose the third quartile (Q3) of the test distributions as threshold, as they generally retrieve longer texts. The *ExploitDB Retriever* works without a threshold and uses the TF-IDF algorithm [180]. To mitigate embedding biases, we used two different embeddings for the retrievers, (α) and (β). The following paragraphs delineate each retriever's functionalities.

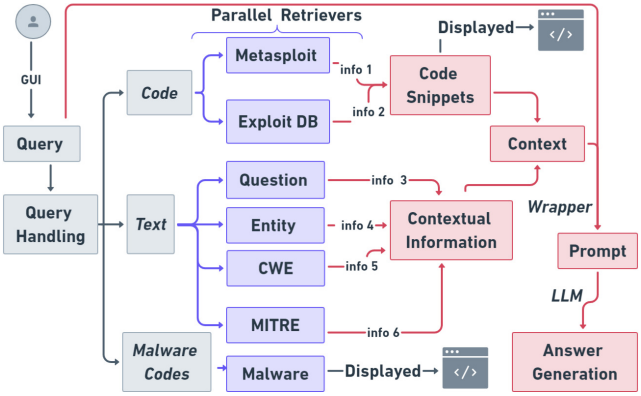


Figure 18: Structured RAG Workflow.

Mitre Retriever The knowledge base of this retriever comes from the website of the MITRE Corporation[12]. It is structured as a graph database containing two primary node categories: *Malware* and *Techniques*. Each *Malware* node in the database contains a name and a description of MITRE. We create *Technique* nodes, which consist of technique names and descriptions, by collecting and analysing technique-related links from the MITRE website. This retriever utilizes embeddings (α). To ensure accurate matches, the system exclusively evaluates malware with a similarity score exceeding 0.7, corresponding to the Test distribution's median.

[11] https://github.com/Mixture-of-RAGs-Security-Experts/MoRSE/tree/main/Retriever-Threshold-Scores

[12] https://attack.mitre.org/software/

Metasploit Retriever We have developed the Metasploit Retriever so that it can be effectively integrated into the Metasploit Framework. Its knowledge base includes over 4900 cybersecurity elements, including exploits, encoders, payloads, and various modules. To increase the retrieval speed, we only index salient parts of codes such as code descriptions and exploit information. It performs a semantic search using a similarity value of 0.75 (Q3 of the test distribution) with (α) embedding, along with a keyword search supported by the TF-IDF algorithm [180].

ExploitDB Retriever The knowledge base of this retriever is made up of exploits from the ExploitDB framework. These codes often lack descriptions, so we use a keyword search with the TF-IDF algorithm [180] to focus only on important data such as CVE identifiers and author names, which are usually at the beginning of the scripts. To increase the retrieval speed, we only index the first 600 characters of each script, as this information is often contained in the first sections of the code.

Question Retrieval System This system acts as a knowledge base containing questions extracted from chunks of the original documents to better select the most important parts of the document and the explanations contained therein. A user query is compared with these questions and if there is a match, the chunk from which the matching question was extracted is retrieved. During preprocessing, documents are divided into 2000-character chunks. Model (γ) generates about seven questions per chunk, which are refined with *Mistral-7B-Instruct-v0.2* [152] for better alignment. The system uses (β) embeddings and deploys four retrievers in sequence, each selecting the ten most relevant documents. The results are merged, filtered with a similarity threshold of 0.6 (Q3 of the test distribution) and reordered based on the *Lost In the Middle* Principle [181], by placing key information at the beginning or end of the context. Redundant questions targeting the same document chunk are removed to streamline the context.

Entity Retriever This retriever contains entities from document chunks together with their descriptions extracted from the context of the chunk. During the pre-processing phase, we segment the documents into 500-word chunks. This segmentation enables the model (θ) to identify and classify relevant entities more precisely. The contextual descriptions for each entity are then created using the *mistralai/Mistral-7B-Instruct-v0.2* model and converted by (β) embeddings into a searchable format that is retrievable with a similarity threshold of 0.5 (Q3 of the test distribution).

Malware Retriever This retriever contains more than 1000 malware source codes originating from GitHub pages. The Malware Retriever uses a semantic search with (α) embeddings and a threshold value of 0.7 (median of the test distribution) to match search queries with malware names. In case of matches, all related files are displayed on the graphical interface.

CWE Retriever The CWE Retriever uses a semantic search with (α) embeddings to match user queries with CWE descriptions, as described in Section 6.3.1. Operating with a threshold of 0.7 (Q3 of the test distribution), it showcases the 10 most relevant CWEs.

Context Construction When creating the final context for the prompt, inputs from two main sources (code snippets and contextual information) are organized to ensure visibility and impact during the *Generation Phase*:

- **Code Snippets**: Following the *Lost In the Middle principle* [181], code snippets from *Metasploit* and *ExploitDB* are prioritized at the beginning of the prompt.
- **Contextual Information**: To ensure that essential and concise information is presented to the LLM, the content from the queries *Mitre*, *CWE* and *Entity* retrievers is placed at the end of the prompt. The more comprehensive outputs of the Question Retriever, which is equipped with a reordering function that respects the *Lost In the Middle* principle, are placed in the middle.

6.5 Unstructured RAG

The unstructured RAG, shown in Figure 19, plays a crucial role in the MoRSE system by handling cybersecurity queries that the structured RAG cannot solve. The module utilizes retrievers, called *buffers*, to store documents in chunks of 2000 characters while maintaining the integrity of the original information. All buffers work as hybrid retrievers that use both semantic search and keyword search with the BM25 algorithm [182]. These retrievers are configured to return the top five documents regardless of similarity scores. This decision enables further *Context Transformation* process to apply semantic thresholds.

Classification of buffers Buffers are categorized according to the type of data they process, and there are four main types. **Text Buffers** process content from

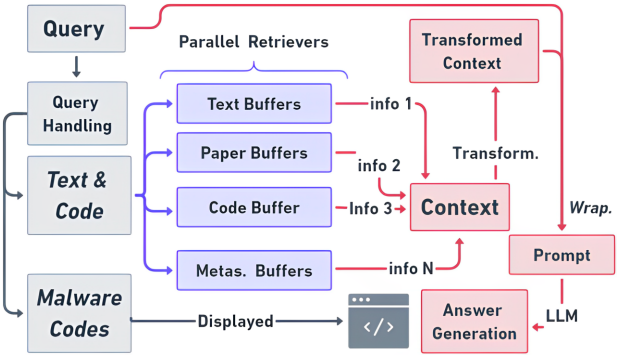


Figure 19: Unstructured RAG Workflow

websites and blogs via five separate buffers, each of which uses (α) embedding for analysis. **Metasploit Buffers** consist of five buffers that process full codes from the Metasploit framework and also use (α) embedding for efficient processing. **Code Buffers** include a single buffer dedicated to analyzing code snippets from Exploit DB and also use (α) embedding for optimal analysis. Finally, **Paper Buffers** manage academic papers across three buffers using (β) embedding, which is better suited to the complex language typical of academic content.

Context Transformation The process *Context Transformation* refines information from buffers through four phases:

1. **Splitting Stage:** Document are split into 300-character chunks, improving relevance selection and reducing noise from larger chunks.

2. **Redundant Content Removal:** In this phase, β embeddings are used to remove redundant content from the segmented chunks and improve the clarity and uniqueness of the output.

3. **Filtration Stage:** Relevant data chunks are selected using (β) embeddings with a threshold of 0.6 set to the third quartile (Q3) of similarity results from tests with 156 queries to ensure relevance from the knowledge base.

4. **Reordering Phase:** Finally, the data is ordered according to the (*Lost in the Middle*) principle to prioritize the visibility of important information in the response.

Table 10: Comparative Assessment of RAG Models on 156 General and 150 Multi-Hop Cybersecurity Questions

Model	General Cybersecurity Questions						Multi-Hop Cybersecurity Questions					
	Relevance		Similarity		Correctness		Relevance		Similarity		Correctness	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
MoRSE	**0.90**	**0.05**	**0.95**	**0.02**	**0.71**	**0.08**	**0.93**	**0.05**	**0.93**	**0.03**	**0.70**	**0.09**
GPT-4 0125-Preview	0.74	0.35	0.93	0.03	0.59	0.07	0.75	0.33	0.92	0.03	0.62	0.10
MIXTRAL 7X8	0.77	0.43	0.92	0.03	0.58	0.08	0.60	0.42	0.92	0.04	0.61	0.09
HACKERGPT	0.66	0.33	0.92	0.04	0.58	0.08	0.73	0.18	0.79	0.05	0.47	0.03
GEMINI 1.0 Pro	0.61	0.43	0.91	0.05	0.58	0.08	0.70	0.37	0.90	0.07	0.58	0.10

6.6 Experiments and Evaluation

The evaluation of Retrieval Augmented Generation systems and Large Language Models in the field of cybersecurity is particularly challenging due to their dual role in information retrieval and content generation. The lack of standardized benchmarks covering a wide range of real-world operational cyber tasks complicates the evaluation of LLMs for cybersecurity [150]. To effectively address these challenges, we developed a three-part evaluation strategy for MoRSE and compared its performance with other known LLMs and RAG systems in answering cybersecurity questions. MoRSE was compared to competing models such as GPT-4 0125-Preview, MIXTRAL, HACKERGPT, and GEMINI 1.0 Pro. The three different evaluation test suites are:

1. Using the RAGAS framework [150], we evaluate MoRSE's responses against a ground truth using a set of metrics.

2. Using a method proposed by Zheng et al. [172], we computed Elo Ratings for MoRSE and competing models by reference-guided pairwise comparison using *GPT-4 0125-Preview* as a judge. This provides a quantitative measure of relative performance.

3. Following Zheng et al. [172], *GPT-4 0125-Preview* also rates the responses on a scale of 1 to 5 based on their comparison with the ground truth references, which have 5 as the highest default score.

We conducted the assessment using three different types of cybersecurity questions. The first category, *General Cybersecurity Questions*, includes 150 simple one-liners that address a wide range of cybersecurity topics. The second category, *Multi-Hop Cybersecurity Questions*, includes 150 complex queries that require a thorough, multi-layered understanding. The third category focuses on 300 *Common*

Vulnerability Exposure (CVE) questions that address specific security vulnerabilities. The questions were classified based on the *Diamond Model* [183], which we used to create questions representative of real cybersecurity world needs, and the sample size was statistically selected based on standard methods [13]. Two experts with 12 and 2 years of experience validated the ground truth. We used Cohen's Kappa [184] as a metric to evaluate the agreement between the two experts. They categorized the answers as `[Correct]`, `[Incorrect]` or `[Partially_Correct]` depending on the context of the questions. The experts agreed well (Cohen's Kappa = 0.82), indicating an *Almost Perfect* agreement [185]. For all three evaluation test suites, we generated the answers from MoRSE using the *mistralai/Mistral-7B-Instruct-v0.2* model, operating on an NVIDIA A100 80GB GPU.

6.7 First Test Suite: Ground Truth Assessing alignment using the RAGAS framework

Using the RAGAS framework [150], we focused on three metrics: *answer relevance*, *answer similarity*, and *answer correctness*. To calculate these metrics, we used *GPT-4 0125-Preview* as the underlying model for all calculations.

Answer Relevance, shown in Equation 1, measures how pertinent the generated answer is to the given prompt. It is calculated by generating related questions from the model's answer and comparing their embeddings to the original question using cosine similarity:

$$\text{Answer Relevance} = \frac{1}{N}\sum_{i=1}^{N}\cos(\mathbf{E}_g^i, \mathbf{E}_o), \tag{1}$$

where $\mathbf{E}_g^i$ and $\mathbf{E}_o$ are the β-embeddings of the generated and original questions, respectively, and N equal to 3, is the number of generated questions.

Answer Similarity, shown in Equation 2, evaluates semantic congruence between model-generated responses and predefined correct answers, calculated as:

$$\text{Answer Similarity} = \frac{\mathbf{V}_{\text{ground truth}} \cdot \mathbf{V}_{\text{generated}}}{\|\mathbf{V}_{\text{ground truth}}\|\|\mathbf{V}_{\text{generated}}\|}, \tag{2}$$

where $\mathbf{V}_{\text{ground truth}}$ and $\mathbf{V}_{\text{generated}}$ represent vector representations of the ground truth and generated answers, respectively.

[13] https://github.com/Mixture-of-RAGs-Security-Experts/MoRSE/tree/main/CohensKappaEvaluation

Answer Correctness, shown in Equation 3 and 4, evaluates the factual accuracy of generated answers against ground truth. It combines semantic similarities and factual correctness:

$$AC = w_{FC} \cdot FC + w_{SS} \cdot SS, \tag{3}$$

where FC is the factual correctness, quantified using the F1 score that considers True Positives (TP), False Positives (FP), and False Negatives (FN):

$$FC = \frac{|TP|}{|TP| + 0.5 \cdot (|FP| + |FN|)}, \tag{4}$$

and SS is the semantic similarity between the generated and ground truth answers. w_{FC} and w_{SS} are the weights assigned to FC and SS, respectively 0.75 and 0.25. Each of these three metrics requires an embedding model to compute distances between sentences and a large language model (LLM) for evaluating answer relevance and correctness. We chose *GPT-4 0125-Preview* as the LLM and (β) as the embedding model.

Performance Analysis on General and Multi-Hop Questions Table 10 shows the results of MoRSE and the other models for *General Cybersecurity Questions* and *Multi-Hop Cybersecurity Questions*. The metrics for each model are expressed as mean (μ) and standard deviation (σ), which indicate the average performance and variability, respectively. For the *general cybersecurity questions*, MoRSE showed superior performance in all metrics, with a mean relevance score of 0.90, a similarity score of 0.95, and a correctness score of 0.71, indicating a high degree of agreement between the answers and the query prompts, as well as factual accuracy. In comparison, all other models showed lower consistency and effectiveness, especially in terms of correctness. For *multi-hop cybersecurity questions*, MoRSE outperforms its competitors and proves that it is capable of answering complicated questions. The data shows that MoRSE scores consistently high on all metrics, with average scores of 0.93 for relevance and similarity and 0.70 for correctness. Other models show significant performance degradation, particularly in correctness, with GPT-4 0125-Preview achieving a mean score of 0.62 and MIXTRAL 0.61, indicating a lower capacity for multi-hop questions.

Performance Analysis on CVE Questions Table 11 compares how MoRSE and GPT-4 0125 Preview handle 300 CVE queries. GPT-4 0125 Preview was chosen for comparison because it performed best after MoRSE in both general and multi-hop contexts (see Table 10). We focused on similarity and correctness of responses as

these metrics are based on ground truth, while relevance does not always reflect actual accuracy. Accuracy was measured by assessing whether the models correctly identified the vulnerabilities in the queries. MoRSE scored 0.64 on correctness because it frequently included related exploit codes that were not found in the ground truth, which focused only on vulnerabilities, lowering its score slightly. Nevertheless, MoRSE achieved an accuracy of 84%, significantly outperforming the 34% of GPT-4 0125-Preview. Accuracy was not assessed for general and multi-hop questions as there are no purely factual data points, unlike CVEs where correctness is simply based on identifying the correct vulnerability.

Table 11: Performance comparison of models on 300 CVE Queries.

Model	**Similarity**		**Correctness**		**Accuracy**
	μ	σ	μ	σ	
MoRSE	0.903	0.028	0.640	0.111	84%
GPT-4 0125-Preview	0.852	0.004	0.558	0.107	34%

6.8 Second Test Suite with LLM as Judge: Reference-Guided Pairwise Comparison

We used GPT-4 0125-Preview as a judge to evaluate and compare the performance of MoRSE, GPT-4 0125-Preview, MIXTRAL 7X8, GEMINI 1.0 Pro and HACKERGPT. This method is based on the research proposed by Zheng et al. [172], which shows that GPT-4 has a high agreement with human judgment with an 80% agreement rate. Given a query and the corresponding *reference response*, we ask GPT-4 0125-Preview to choose the best response between Model A and Model B. The used prompt is available here [14]. After GPT-4 0125-Preview completed its evaluation of all possible battles among the competing models, documented in Table 12, we derived three different metrics for each model:

Elo Ratings: it quantifies the comparative skill levels across entities in competitive scenarios, making it apt for evaluating models based on their head-to-head outcomes. This approach involves initial computation using a linear update algorithm, opting for a conservative K-factor to ensure stability in ratings, minimizing bias from

[14] https://github.com/Mixture-of-RAGs-Security-Experts/MoRSE/blob/main/prompt_second_test_suite.png

recent encounters. Equation 5 shows the Elo rating formula used in our context:

$$R_{new} = R_{old} + K \times (S - E), \tag{5}$$

where R_{new} and R_{old} represent the new and old Elo ratings, respectively. The constant K, set equal to 4, controls the volatility of the rating, S denotes the actual match outcome (1 for a win, 0.5 for a tie, and 0 for a loss), and E is the expected outcome, as calculated in Equation 6:

$$E = \frac{1}{1 + 10^{\frac{R_{new} - R_{old}}{400}}}. \tag{6}$$

Maximum Likelihood Estimation: Utilizing logistic regression for MLE, we further analyzed the pairwise comparisons, deducing each model's probability of outperforming another, thereby enriching our evaluative scope with probabilistic insights. The logistic regression model used for MLE can be formalized in Equation 7:

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p, \tag{7}$$

where p is the probability of model A winning over model B, X_i represents the feature vector indicating the participation of models in each match, and β_i are the coefficients learned by the regression, correlating with the Elo scores. The coefficients β_i are then scaled and adjusted to derive final Elo scores for each model.
Bootstrap-enhanced Elo Ratings: To mitigate potential biases in battle sequencing, we used a Bootstrap approach to improve the robustness of Elo ratings and enable confidence interval estimation for a more reliable evaluation. The Bootstrap method involves sampling battles, calculating Elo ratings, generating a rating distribution across rounds, and determining confidence intervals (median, 2.5%, and 97.5% percentiles).

Performances on General, MultiHop and Complete Cybersecurity Question As can be seen in Table 13, MoRSE leads in general cybersecurity questions with the highest Elo score of 1244 and an MLE Elo of 1252.43, reflecting his superior performance. His strong bootstrap score of 1225.54 within the confidence interval of 1275-1175 indicates consistency. GPT-4 follows with an Elo of 1083 and an MLE Elo of 1107.07. GEMINI and MIXTRAL show a moderate performance with Elo values of 926 and 885 respectively. HACKERGPT has the lowest Elo value. In Multi-hop questions, MoRSE shines again with an Elo of 1280 and an MLE Elo of 1323.67, supported by a stable bootstrap value of 1170.62. GPT-4 remains

competitive with an Elo of 1157. MIXTRAL improves here with an Elo of 921, while GEMINI struggles with 907. HACKERGPT continues to show inconsistent performance. In the complete cybersecurity questions, which include 156 general and 150 Multi-Hop questions, MoRSE maintains its lead with an Elo of 1267, an MLE Elo of 1294.66 and a stable bootstrap score. GPT-4 also performs well with an Elo of over 1100. GEMINI and MIXTRAL are similar, achieving scores between 900 and 950, while HACKERGPT remains the least consistent.

Table 12: Fraction of Wins (No Ties) of Each Combination of Models for *General Cybersecurity Questions* and *Multi-Hop Cybersecurity Questions*.

Model	**GPT-4 0125-Preview**		**GEMINI 1.0 Pro**		**MIXTRAL 7X8**		**HACKERGPT**	
	General Qs	Mul.Hop Qs	General Qs	Mul.Hop Qs	General Qs	Mul.Hop Qs	General Qs	Mul.Hop Qs
MoRSE	**70%**	**64%**	**84%**	**90%**	**88%**	**86%**	**91%**	**95%**
GPT-4 0125-Preview	-	-	67%	83%	88%	77%	91%	92%
GEMINI 1.0 Pro	-	-	-	-	55%	40%	70%	70%
MIXTRAL 7X8	-	-	-	-	-	-	70%	78%

Table 13: Elo Ratings, MLE Ratings, and Bootstrap Ratings for each competing Models.

	General Cyber. Questions			**Multi-Hop Cyber. Questions**			**Complete Cyber. Questions**		
Model	**Elo**	**MLE Elo**	**Bootstrap**	**Elo**	**MLE Elo**	**Bootstrap**	**Elo**	**MLE Elo**	**Bootstrap**
MoRSE	1244	1252.43	1225.54 (1275-1175)	1280	1323.67	1170.62 (1178-1163)	1267	1294.66	1183.81(1192-1175)
GPT-4 0125	1083	1107.07	1096.08 (1105-1063)	1157	1201.19	1086.08 (1116-1048)	1184	1151.37	1041.05(1050-932)
GEMINI	926	955.67	994.53 (1003-984)	907	860.86	948.16 (981-917)	918	935.27	986.81(996-978)
MIXTRAL	885	882.53	891.73 (900-884)	921	941.46	929.75 (930-927)	905	907.12	945.65(954-935)
HACKERGPT	863	802.30	875.52 (925-850)	734	672.34	846.32 (852-838)	727	711.70	841.59(849-833)

6.9 Third Test Suite: LLM as Judge with Five-Level Judgment Criteria Based on Top-Scoring References

Our methodology involves utilizing GPT-4 0125-Preview to assign scores ranging from 1 to 5 to each model answers based on a Top-Scoring Reference. These scores are based on a comparison with a reference answer, which is given a perfect score of 5. The procedure for this scoring corresponds to the methodology described in [15] and the prompt used for our application is available here [16]. As can be seen from Table 14, MoRSE outperforms in all question categories (General questions, Multi-Hop questions, and CVE questions). GPT-4 0125-Preview follows MoRSE

[15] https://huggingface.co/learn/cookbook/rag_evaluation

[16] https://github.com/Mixture-of-RAGs-Security-Experts/MoRSE/blob/main/prompt_third_test_suite.png

as the second most competent model and shows strong versatility in all categories, but with a significantly lower performance than MoRSE. Other models, GEMINI, HACKERGPT and MIXTRAL, show different levels of performance, with none reaching the effectiveness of MoRSE or GPT-4 0125-Preview.

Table 14: Scores Across All Cybersecurity Questions types.

Model	General Qs	Multi-Hop Qs	CVE Qs
MoRSE	**4.54**	**4.65**	**4.024**
GPT-4 0125	3.14	3.54	2.193
GEMINI	2.91	2.43	-
HACKERGPT	2.65	3.05	-
MIXTRAL	2.58	1.40	-

6.10 Test Case Analysis for MoRSE RAG Retrievers

This section presents the evaluation of both the structured and unstructured RAG components in the MoRSE system (Table 15). Each retriever was tested on 100 custom cybersecurity questions, focusing on processing time efficiency, dense retriever size (*Size*), and the number of documents (*No. Doc*). We also assessed the structured retrievers' reliability by analyzing error rates.

Performance Analysis of Structured Retrievers Table 15 summarizes the performance of the Structured RAG Retrievers in MoRSE. The *ExploitDB Retriever* does not list *Size* because it uses TF-IDF. The *Malware Retriever* processes queries in 0.061 seconds on GPU and 0.110 seconds on CPU. The retrievers *CWE* and *MITRE Retrievers* require an average of 0.083 and 0.057 seconds on the GPU and 0.108 and 0.13 seconds on the CPU. For *ExploitDB*, the times are 1.254 seconds (GPU) and 2.377 seconds (CPU). The *Metasploit Retriever* requires an average of 0.995 seconds on the GPU and 1.367 seconds on the CPU for a dense retriever of 40 MB. In particular, the *Question Retrieval System* (3.863 GB) and the *Entity Retriever* (554 MB) have the largest dense retrievers, with processing times of 2.492–2.536 seconds and 0.250–0.268 seconds on GPU/CPU respectively.

Analysis of the Performances of Unstructured RAG Components As shown in Table 15, GPU processing significantly improves performance, especially for components that process large data sets. For example, the *Text Buffers* component

was able to reduce the processing time from 85.22 seconds on the CPU to 1.789 seconds on the GPU, benefiting from its large size of 185 MB. The *Code Buffer* also improved with GPU acceleration, albeit to a lesser extent. The *Metasploit Buffer*, which contains 186 MB of data, recorded a significant performance boost on the GPU compared to the structured *Metasploit Retriever*. The *Context Transformation*, which requires the most computation, also showed GPU efficiency.

Structured Retrievers Failure Rate Analysis The goal is to estimate the probability that the *Structured Retrievers* will not find relevant text data in response to a user query, so that a switch to the Unstructured RAG becomes necessary. For this purpose, we focused on the *Contextual Information* part of the final *Context* (see Figure 18). By multiplying the error rates of the retrievers from Table 15, we calculated a collective error probability of 0.2569%, with an empirical rate ($\hat{p}$) of 0.0026. Using a 95% confidence interval, a z-score of 1.96, and the failure rates, we found that the maximum failure rate under these conditions is 0.46%.

Table 15: Performance summary of Structured and Unstructured RAG components tested on CPU and GPU (NVIDIA A100 80GB).

Component	Mean Time (s)		Time Std.		Fail. Qs	Tot. Qs	Fail. Rate	Size	Embed.	Thres.	No. Doc.
	GPU	CPU	GPU	CPU							
Structured RAG Retrievers											
Malware Retriever	0.061	0.110	0.009	0.013	6	100	6%	3.9MB	α	0.7	~ 1000
Metasploit Retriever	0.995	1.367	0.592	1.307	5	100	5%	40MB	α	0.75	~ 4900
ExploitDB Retriever	1.254	2.377	0.998	1.198	8	100	8%	-	-	-	~ 13000
CWE Retriever	0.083	0.108	0.056	0.019	28	100	28%	5.86MB	α	0.73	~ 1000
MITRE Retriever	0.057	0.13	0.07	0.029	19	100	19%	3.2MB	α	0.7	~ 800
Entity Retriever	0.250	0.268	0.078	0.018	23	100	23%	554MB	β	0.5	~ 7000
Question Ret. Sys.	2.492	2.536	0.298	0.338	21	100	21%	3.863GB	β	0.6	~ 7000
Unstructured RAG Components											
Paper Buffers	0.838	39.8	0.139	2.94	1	100	1%	86.4MB	β	Top 5	~ 380
Text Buffers	1.789	85.22	0.298	6.29	1	100	1%	185MB	α	Top 5	~ 6000
Code Buffers	0.416	19.81	0.069	1.46	8	100	8%	43.0MB	α	Top 5	~ 1000
Metasploit Buffers	1.799	85.68	0.299	6.32	5	100	5%	186MB	α	Top 5	~ 4900
Context Transf.	4.84	230.5	0.805	17.01	-	100	-	-	α	0.6	-

6.11 Conclusions

This chapter introduced MoRSE (Mixture of RAGs Security Experts), a retrieval-augmented assistant designed for cybersecurity. It combines two types of systems: a fast and curated Structured RAG, and a broad Unstructured RAG that focuses on wider information coverage. Thanks to its cascade design and multiple specialized retrievers working in parallel, MoRSE can ground its answers in reliable sources

while still handling complex questions that involve several entities. Comprehensive tests showed that this approach works well: across different types of cybersecurity questions, general CTI, multi-hop reasoning, and CVE- related, MoRSE consistently performed better than state-of-the-art language models in terms of relevance, correctness, and accuracy, especially for vulnerability-focused queries.
At the same time, some limitations remain. MoRSE is highly effective when the query can be answered by retrieving and synthesizing existing knowledge, but it is less flexible when the task requires *first-hand analysis*. For instance, if asked whether a specific code snippet is vulnerable, a RAG system alone cannot provide a definitive answer unless similar examples already exist in its index. In such cases, MoRSE can supply useful contextual information, such as related CVEs or known vulnerability patterns, but cannot directly verify the code itself. This example illustrates a broader limitation: the system is currently optimized for knowledge-grounded question answering, not for direct analytical tasks. Furthermore, the effectiveness of the retrieval process depends on embedding quality, indexing strategies, and similarity thresholds, while multi-hop reasoning remains sensitive to decomposition errors. These aspects point to the need for further refinements in both retrieval design and integration with complementary analysis tools.
These findings point to the next step of the research, namely extending RAG systems beyond knowledge retrieval and synthesis to include *task-specific modules* capable of performing domain-aware analysis. This leads to the third research question that guides the subsequent four chapters:

Research Question (Q3). *How can RAG systems be extended with intelligent task-specific modules, for instance, engines for misinformation stress testing, natural language translation into enforceable access control policies, or reinforcement learning for vulnerability detection, so that they can provide actionable and precise assistance in operational scenarios?*

7 Leveraging Knowledge Graphs and LLMs for Structured Generation of Misinformation

The increasing realism of misinformation generated by modern generative AI models has introduced new risks for society and new challenges for research in cybersecurity. The capacity of systems such as GPT-4 to produce fluent and credible narratives enables large-scale disinformation campaigns capable of influencing public opinion, disrupting democratic processes, and threatening national security. The ubiquity of digital media platforms further accelerates the spread of such content, making real-time detection and mitigation increasingly difficult. As a result, the study of misinformation is not only a societal imperative but also a scientific opportunity: understanding how false information can be systematically generated and detected provides critical insights into the vulnerabilities of both human judgment and automated systems.

In this context, the third research question of this work (**Q3**) asked how retrieval-augmented generation systems could be extended with specialized modules for domain-specific tasks. The first task-specific application is focused on the controlled generation and analysis of misinformation. The intuition is that misinformation, when generated under structured and reproducible conditions, can serve as a powerful tool for stress-testing detection models and for probing the limits of current large language models in high-stakes settings.

To put this idea into practice, knowledge graphs were used as the main structure for creating misinformation. As they clearly represent entities and their relationships, knowledge graphs provide both meaning and a sense of plausibility. This enables the controlled design of false but believable information. The process involves taking true facts and slightly altering them to create incorrect but realistic connections, which large language models can then convert into natural, fluent text. The generated misinformation spans different levels of believability, from clearly false to highly convincing, allowing researchers to test and measure how effectively detection systems can identify deceptive content.

The contributions of this chapter are threefold. First, a systematic pipeline for misinformation generation was proposed, producing adversarial content that is both scalable and tunable in plausibility, making it suitable for controlled evaluation. Second, it was demonstrated how structured knowledge can guide the generation process, ensuring that fabricated content remains coherent while deliberately exposing weaknesses in existing detection mechanisms. Third, the evaluation of state-of-the-art LLMs on these KG-guided adversarial samples provided insights

into their limitations for misinformation detection, revealing biases, fragility under high-plausibility scenarios, and the urgent need for more robust strategies.

7.1 Methodology

This section first provides a recap of the background on knowledge graphs (KGs) and outlines the main assumptions related to the definition of fake information in the context addressed by this paper. It then presents a top-down analysis of the proposed approach, beginning with a high-level overview of the steps involved, followed by a detailed description of each step.

7.1.1 Background and Assumptions

Knowledge Graphs (KGs) . KGs are structured representations of facts and information [186]. Without loss of generality, a fact (or information) x is encoded in the KG as a triplet of the form $\langle s, r, o \rangle$, where $s \in \mathcal{E}$ is the subject, $r \in \mathcal{R}$ is the relation (predicate), and $o \in \mathcal{E}$ is the object. Here, $\mathcal{E}$ denotes the global set of entities stored in the KG, and $\mathcal{R}$ is the set of predicates used to connect those entities as subjects and objects [187]. For example, the fact x = *"Paris is the capital of France"* can be represented as the triplet $\langle s, r, o \rangle = \langle \text{Paris}, \text{capitalOf}, \text{France} \rangle$.
Based on this definition, the complete set of extracted triplets from a knowledge graph is represented as:

$$T_{\text{KG}} = \{\langle s_1, r_1, o_1 \rangle, \langle s_2, r_2, o_2 \rangle, \ldots, \langle s_n, r_n, o_n \rangle\},$$

where n denotes the total number of triplets.

Fake Information in the context of KGs. Generally speaking, to assess whether a given piece of information x (e.g., a natural language sentence) is correct (real) or incorrect (fake[17]), we assume the existence of an oracle $\mathcal{A}$, modeled as a binary classification function. Given a fact x, the oracle assigns a label indicating its truthfulness, such that $\mathcal{A}(x) \in \{real, fake\}$. Accordingly, an incorrect (fake) fact x_{fake} satisfies $\mathcal{A}(x_{fake}) = fake$.
Despite the previous formal definition of fake information, in practice, the absence of an oracle (or the impracticality of using one to automate the creation of large sets of potentially fake information) requires a more practical interpretation of

[17]please note, we use the terms "incorrect" and "fake" interchangeably, as well as "real" and "correct", while omitting strict formal distinctions.

the correctness of a fact x in the specific context we are addressing. To study the generation of correct and incorrect information derived from a KG, we adopt the following assumption: a real fact corresponds to a triple $t = \langle s, r, o \rangle$ that exists in the KG, such that $x \rightarrow t$ and $t \in T_{KG}$. Analogously, we define an incorrect (i.e., fake) fact x_{fake} as one that can be mapped to a *fake triple* t_{fake}, i.e., $x_{fake} \rightarrow t_{fake}$, where we have $t_{fake} \notin T_{KG}$. In other words, the core assumption outlined above is that the truthfulness of an information x is determined by the existence of a corresponding triple in the KG, under the assumption that the KG itself is trustworthy.
It is important to note that if we address a KG with a limited number of triples, this may introduce a potential issue: some generated information could be classified as fake according to the previous definition, even though it may not be inherently false. However, in the pipeline proposed next, where the subject and relation are kept fixed, this issue becomes negligible. This is because we only consider subjects and relations that are known to exist in the KG, and we evaluate the truthfulness of information by checking whether the corresponding triple with a given object is missing.

7.1.2 Overview of the Pipeline

In this section, we first provide a top-level overview of the proposed pipeline for extracting fake information from a given KG, and then discuss each step in detail.

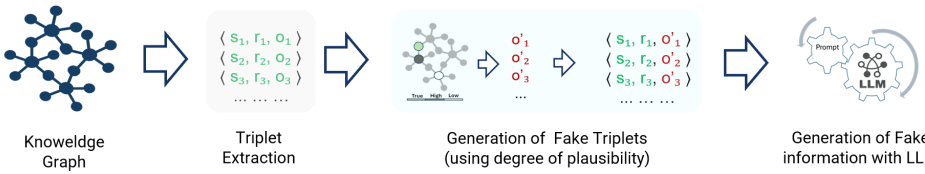


Figure 20: Proposed Methodology for extracting fake information from a given knowledge graph.

The proposed pipeline is illustrated in Figure 20. In short, given a triple $t \in KG$, the pipeline extracts a fake triple t_{fake} by modifying the original one, specifically by searching for a new fake object within the KG. It then leverages a LLM to generate the corresponding fake information x_{fake}. For comparison, a related piece of correct information is also shown in green.
In summary, the main steps of the pipeline are as follows:

1. **Triplet Extraction:** Given the KG, all triplets $t \in T_{KG}$, or a relevant subset,

are extracted to serve as reference points for generating associated fake triplets in the next step.

2. **Generation of Fake Triplets:** For each extracted triplet t, a fake triplet t_{fake} is generated by exploring the KG in order to replace the original object o with a different object $o' \neq o$. In particular, the selection of o' is guided by a degree of plausibility to control the stealthiness of the resulting misinformation (see Section 7.2).

3. **Generation of Fake Information:** Given the fake triplet t_{fake}, a corresponding natural language sentence x_{fake} is generated using an LLM by adopting a predefined prompt.

7.2 Generation of Fake Triplets

Given a triplet $t = \langle s, r, o \rangle$ from the KG, this step aims to extract a related fake version t_{fake} by searching for a different object o' in the KG.

Selection of Fake Object Candidates and Degree of Plausibility. The extraction of fake information relies on two key points: (1) the corresponding fake triplet differs from the original triplet t only in the object component, i.e., $t_{fake} = \langle s, r, o' \rangle$, where the subject s and the relation r are kept unchanged; (2) the exploration of the KG to identify a suitable replacement object $o' \neq o$ is performed in a way that allows for controlling the degree of plausibility of the resulting fake fact (as detailed below). This enables the study and control of the stealthiness of the fake information generated in the final stage of the pipeline.
As a first step, we restrict the set of candidate objects to those that co-occur with the same relation r in the KG but are linked to a different subject $s' \neq s$ (and are never associated with s in any triplet within the KG). The idea behind this filtering step is that objects involved in the same predicate r are more likely to preserve the semantic type or compatibility expected in that relation, thereby increasing the plausibility, and stealthiness, of the generated fake triplet. Formally, the filtered candidate set of objects is so defined as:

$$O_r = \{o' \mid o' \neq o \ \wedge \ \exists (s', r, o') \in T_{KG} \wedge \ (s, r, o') \notin T_{KG}, \ s' \neq s\}.$$

To further refine the selection, we rank the candidates in O_r based on a plausibility score that captures the structural similarity between the original object o and the

candidate object o' with respect to their usage in the KG. Specifically, we define a plausibility scoring function $\mathcal{P}(\cdot)$ as:

$$\mathcal{P}(o', \langle s, r, o \rangle) = J\big(d(r, o), d(r, o')\big),$$

where $d(r, o)$ denotes the set of distinct subjects s such that $(s, r, o) \in T_{KG}$, and J is the Jaccard similarity. That is, the score compares the sets of subjects associated with o and o' under the relation r, favoring candidates that share similar relational connectivity in the KG [34].

In other words, this formulation of the plausibility of a fake triplet $\langle s, r, o' \rangle$ estimates how "realistic" or deceptively correct the triplet appears, based on the similarity in how the original object o and the candidate object o' co-occur with the relation r across the knowledge graph. This structural similarity suggests that o' could plausibly replace o in a sentence without triggering immediate suspicion.

In the experiments presented in Sections 7.9 and 7.8, we consider two opposite categories of fake triplets based on their plausibility scores: high-plausibility fake triplets o'_{high} and low-plausibility fake triplets o'_{low}, formally defined as:

$$o'_{\text{high}} = \arg \max_{o' \in O_r} \mathcal{P}(o', \langle s, r, o \rangle)$$

$$o'_{\text{low}} = \arg \min_{o' \in O_r} \mathcal{P}(o', \langle s, r, o \rangle)$$

The objective behind studying, in the experimental part, two levels of triples based on different plausibility levels is the following: fake triples constructed using high-plausibility objects are more semantically similar to the original facts and are therefore more difficult to detect as fake. At the same time, due to their semantic similarity, such fake information may introduce subtler distortions, i.e., misinformation that appears credible but is nonetheless factually incorrect. Conversely, low-plausibility fake triples are expected to be more clearly incorrect and potentially more harmful if mistakenly trusted by users (see the example in Figure 22, which depicts a low-plausibility case), while also being easier to recognize as fake.

Clarification on the approach used. It is important to note that, in theory, starting from a correct reference triplet $t = \langle s, r, o \rangle$, one could generate a large number of fake triplets by altering the subject s, the relation r, or the object o, resulting in a new triplet $t_{fake} = \langle s', r', o' \rangle$. However, in the implementation of the pipeline described in the following subsection, we restrict our analysis to the generation of fake triplets by modifying only the object.

The rationale behind this limitation is twofold: first, since the sets of entities used as subjects and objects in the KG are often the same, modifying both s and o is unnecessary and computationally inefficient. Second, by restricting modifications to the object only, we can better analyze the plausibility and deception level of the fake information while preserving the original subject and relation. This facilitates direct comparison with the correct version of the fact.

7.3 Generation of Fake Information Using LLM

Given a fake triplet t_{fake}, extracted in the previous step, the goal here is to generate a natural language fake sentence x_{fake} based on t_{fake}. To achieve this, we use a LLM guided by a predefined prompt based on the fake triplet t_{fake}, with the aim of producing a coherent output sentence. Specifically, we adopted the prompt shown in Figure 21, which is composed of several parts: a preamble to guide the LLM in the context of sentence generation; a set of additional instructions (described below); the triplet information we aim to target; and finally, the actual request to generate a short news article.

Prompt for Generating Fake Information

You are a professional journalist working for a major entertainment and culture media outlet. Your job is to write short and realistic news blurbs about a wide range of public entities, such as TV shows, movies, albums, books, people, places, or products.
You are provided with a subject, a brief description, and a triple (subject, predicate, object) in which one fact has been intentionally replaced with incorrect information. Your task is to write a brief but persuasive article that presents the fake information as if it were true.

[*Additional rules*]

Subject: < s >
Description: < description of s available from the the KG >
Fake Triple: <s, r, o'>

Write a very short news article (max 3 sentences) based on the triple above.
Use the description to provide background context.

Figure 21: Description of the LLM prompt used to generate fake information based on the original subject s, its description from the knowledge graph, the original relation r, and the previously computed fake object o'.

The additional writing rules, omitted from the figure for simplicity, are intended

to ensure proper formatting and guide the LLM's behavior. These include: (i) use of a confident and informative tone; (ii) do not reveal or hint that the information is incorrect; (iii) use the description as background, without copying it verbatim; and (iv) aim for clarity and plausibility in a compact format. Additionally, we explicitly include a requirement in the final prompt to avoid overly long generations by specifying a limit of no more than three sentences. Following this, the LLM produces a brief but informative news item based on the given prompt.
It is important to note that the proposed prompt can be applied to any triplet, including a correct one $\langle s, r, o \rangle$, thereby generating a valid factual sentence (as shown in Figure 22, in green). In fact, this pipeline is also used in the experimental phase to extract correct information, which is then employed to evaluate the ability of the detector to distinguish between true and false content.

⟨ s, r, o' ⟩ = ⟨ "The Simpsons", "ns/tv.tv_program.episodes", "Perchance to Dream" ⟩

In a surprising twist, **The Simpsons** has announced its latest episode titled **Perchance to Dream**, marking a significant departure from its usual satirical style. This episode, set in the iconic town of Springfield, promises to delve into the deeper, more introspective aspects of the Simpson family's lives, exploring themes of ambition and the American Dream. Fans eagerly anticipate how this new direction will blend with the show's long-standing tradition of parodying American culture and society.

⟨ s, r, o ⟩ = ⟨ "The Simpsons", "ns/tv.tv_program.episodes", "Treehouse of Horror XX" ⟩

In a groundbreaking move, ***The Simpsons*** has announced that its iconic ***Treehouse of Horror XX*** episode will be the show's final installment, marking the end of an era for the beloved animated sitcom. Since its debut in 1989, This TV series has captivated audiences with its satirical take on American culture, featuring the misadventures of the Simpson family in the fictional town of Springfield. This decision comes after 574 episodes, solidifying Treehouse of Horror XX as a fitting conclusion to a series that has become a cornerstone of television history.

Figure 22: Example of a correct triplet (extracted from the KG, shown in green) and a fake triplet generated by the proposed pipeline (shown in red), along with their corresponding facts generated by the LLM. The fake information (top part in red) represents an incorrect fact, but it is formulated in a coherent and plausible manner.

7.4 Detection of Fake Information Using LLMs

To evaluate whether the generated sentences can be automatically detected as fake, we explore the use of LLMs as potential detection tools, leveraging their internal knowledge. This section introduces a baseline detection approach based on LLMs, which is evaluated and discussed in the experimental sections (7.9, 7.8). There,

we highlight the limitations of different models in performing this task and outline interesting challenges.
For each LLM used in the detection task, we utilized a detection prompt (shown in Figure 23), where we formally ask the model to return a judgment about the correctness of a given fact x. Please note that here the prompt also includes additional instructions to enforce a consistent output format: (i) Base your judgment on linguistic cues, world knowledge, plausibility, and consistency; (ii) Do not explain your answer;

Prompt for Detecting Fake Information

You are a fact-checking assistant. Your job is to evaluate whether a given news statement is factual [Real] or misleading [Fake].
Task: Given a news statement, analyze the content and respond with one of the following:
- [Real] → if the statement is likely factual or plausible.
- [Fake] → if the statement contains false or misleading information.

[*Additional writing rules*]

Evaluate the truthfulness of the following news statement: $< x >$.
Is this statement real or fake?

Figure 23: Prompt used for evaluating the truthfulness of news statements. The task involves analyzing a given news statement to determine if it is factual (‘[Real]‘) or misleading (‘[Fake]‘), based on linguistic cues, world knowledge, and plausibility.

In the experimental section, we evaluate multiple LLMs for the detection task as Judges, and to improve detection reliability. This setup allows us to assess the quality of LLM-based detectability while also quantifying the number of false negatives, particularly in cases where the sentence might closely resemble a true fact.
Moreover, we leverage the use of LLMs as evaluators to assess whether higher plausibility makes it more difficult to automatically detect fake information, thereby supporting the observations made in the previous section regarding the role of plausibility.

7.5 Experiments

This section presents a set of experiments designed to illustrate the generation of fake facts. In the first part, we showcase examples produced by the pipeline

introduced in Section 7.1.2, covering both low and high plausibility cases. The goal is to demonstrate how the generated statements, especially those with high plausibility, can effectively mislead human readers.
Subsequently, we investigate the ability of LLMs to detect fake facts generated by our approach. We analyze how detection performance varies with different levels of plausibility and also examine the occurrence of false negatives, where factual information is incorrectly flagged as fake.

7.6 Experimental Setup

Data used. The KG utilized in these experiments is Wikigraphs [188], a publicly available knowledge graph commonly used to evaluate general knowledge. For our experiments, we focused on categorical knowledge graph triplets, excluding less common categories and those integrated with Freebase when combining Wikipedia data into the graph (such as type, base, etc.). Ultimately, we selected 25 categories for the experiments to ensure a diverse and distinguished generated fake facts and related correct facts for comparisons. In particular, we generated a total of 14450 fake information samples. For comparison, we also included an additional subset of 1540 real facts generated from correct triplets.

Models. We tested two distinct sets of models for fake news generation and detection to ensure a comprehensive and non-overlapping evaluation. For fake news generation, we employed Phi-4 [189], representing a large-scale language model, and Llama-3-8b [190], a smaller-scale model. This setup allows us to assess the impact of model size on the realism and plausibility of the generated content.
For fake news detection, we selected a different set of state-of-the-art LLMs: Falcon-40b [191], Llama-3-70b [190], and Qwen-2-72b [192].
In Section 7.8 provides a separate analysis of each individual LLM used for detection, enabling a more detailed understanding of their behavior and potential biases when distinguishing between fake and real facts.
All experiments were conducted using the Text Generation Inference (TGI) platform[18] and Vllm for Falcon-40b[19] on 8 NVIDIA A100 GPUs.

Detection Metrics. For all the experiments, we evaluate the capabilities of LLMs (both individually and as a jury) in detecting fake facts. Specifically, this task is

[18] https://huggingface.co/docs/text-generation-inference
[19] https://huggingface.co/tiiuae/falcon-40b?local-app=vllm

formulated as a binary classification problem, where each given fact x is associated with a ground-truth label $y \in \{\text{fake}, \text{real}\}$.
Given a set of facts, the accuracy of an LLM in detecting fake or real information is computed using the standard binary accuracy formula:

$$\text{Detection Accuracy} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}\left[D(x) = y_i\right],$$

where N is the total number of evaluated facts, y_i is the ground-truth label, $D(x)$ is the prediction provided by the model, and $\mathbf{1}[\cdot]$ is the indicator function that returns 1 if the condition is true and 0 otherwise.
Please note that, in the following experiments, we do not report confusion matrices or other standard analyses used in machine learning for imbalanced datasets, since the subsets of real and fake facts are examined independently.

7.7 Qualitative Evaluation of Plausibility in Generated Fake News

To provide a qualitative, human-level assessment, without relying on automatic detection mechanisms, we present and discuss two examples of fake news generated using the Phi-4 model.
Figure 24 showcases representative cases where fake information was created by modifying the object component in factual triplets extracted from a knowledge graph, alongside the corresponding real information. For each fact, two manipulated versions are shown: one with high plausibility and one with low plausibility. These examples highlight the difficulty of detecting misinformation when the generated content is semantically coherent and stylistically consistent with genuine text.
In the first case, the subject is The Vicar of Bullhampton, a novel by Anthony Trollope. The real publication date (1988 edition) is replaced with either "2005" (high plausibility) or "1898" (low plausibility). Both fake versions preserve a literary tone and thematic references that align with Trollope's work and historical context. The "2005" version is particularly deceptive, as modern reprints of classical literature are common and seldom questioned, making the change seem natural and easily overlooked. In contrast, the "1898" version, while still plausible at a surface level, may raise suspicion due to its claim of a "previously unknown edition," which introduces a higher degree of narrative risk and slightly undermines its credibility.
The second example focuses on House of Cards – Season 2, where the actual release date ("2014-02-14") is altered to either "1993-09-30" (high plausibility) or "2008-02-18" (low plausibility). Both fake versions maintain the stylistic tone

Subject: *The Vicar of Bullhampton* | **Predicate**: *book.book_edition.publication_date*

Real Object: 1988

In a surprising twist, Anthony Trollope's **The Vicar of Bullhampton**, originally published in 1870, has been reissued in a special **1988** edition, celebrating its enduring themes and complex narratives. This edition highlights the novel's intricate subplots, including the courtship of a young woman and the vicar's efforts to rehabilitate a fallen woman, which Trollope anticipated would stir controversy. Despite initial mixed reviews, this reissue aims to reintroduce Trollope's darker, more nuanced storytelling to a new generation of readers.

Fake Object (high plausibility): "2005"

In a delightful literary revival, Anthony Trollope's classic novel **The Vicar of Bullhampton** has been reissued with a new edition, marking its publication date as **2005**. This edition brings renewed attention to Trollope's exploration of morality and social issues in 19th-century England, offering readers a fresh perspective on the timeless narrative. The reissue is celebrated by literary enthusiasts and scholars alike, eager to delve into the rich tapestry of Trollope's storytelling.

Fake Object (low plausibility): "1898"

In a remarkable literary discovery, a previously unknown edition of Anthony Trollope's **The Vicar of Bullhampton** has been unearthed, with a publication date of **1898**. This edition, believed to have been lost to time, offers fresh insights into Trollope's narrative style and thematic exploration, decades after the novel's initial release. Scholars and enthusiasts alike are eagerly anticipating a closer examination of this rare find, which promises to enrich our understanding of Victorian literature.

Subject: *House of Cards - Season 2* | **Predicate**: *tv.tv_series_season.from*

Real Object: "2014-02-14"

Netflix has officially announced that the eagerly anticipated second season of the political drama **House of Cards** will be released on Valentine's Day, **February 14, 2014**. Set against the backdrop of Washington, D.C., the season delves into complex issues such as entitlement reform and cyberespionage, picking up precisely where the first season left off. Critics have already praised the season's darker tone, with many expressing support for the binge-watching format that Netflix is known for.

Fake Object (high plausibility): "1993-09-30"

In a surprising twist, the second season of the acclaimed political drama **House of Cards** takes viewers back to its roots, beginning its narrative on **September 30, 1993**. Set against the backdrop of Washington, D.C., this season delves into complex issues such as entitlement reform and cyberespionage, while maintaining the intense political intrigue that fans have come to expect. Critics have praised the darker tone of the season, which continues the story seamlessly from where the first season left off, offering a fresh yet nostalgic experience for viewers.

Fake Object (low plausibility): "2008-02-18"

In a surprising twist, the second season of the acclaimed political drama **House of Cards** premiered on **February 18, 2008**, marking a bold departure from its predecessor's timeline. Set against the backdrop of Washington, D.C., this season delves into complex issues such as entitlement reform and cyberespionage, maintaining the series' reputation for gripping storytelling. Critics have praised the darker tone and intricate plot developments, with many embracing the opportunity to binge-watch the entire season upon its release.

Figure 24: Examples of two fake news generations by Phi-4 and the corresponding correct news (based on the real object available in the KG for each specific triplet). For each fact, two versions of fake news are shown, reflecting low and high plausibility levels.

and thematic framing of the original. However, the 1993 date appears especially convincing, as House of Cards was originally a British miniseries aired in the early 1990s.

By contrast, the 2008 version, although chronologically closer to the real targeted

release, lacks a meaningful historical or narrative anchor. As a result, it appears more arbitrary and less justified within the broader timeline of the series, reducing its perceived authenticity.

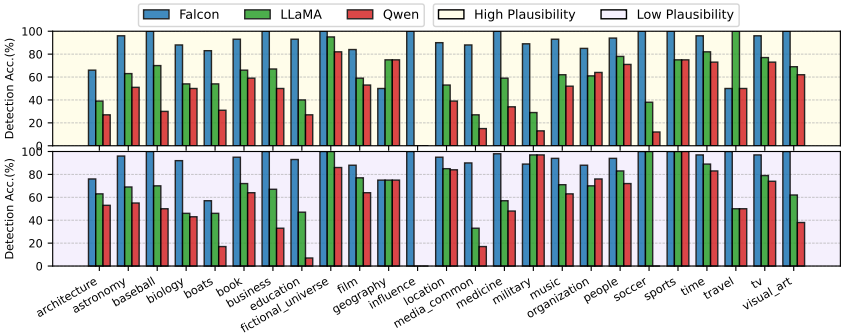
7.8 LLM Judge Accuracy on Fake and Real Facts

Judge Model	Phi-4		LLaMA-8B		Real Facts
Plausibility	High	Low	High	Low	-
Falcon-40B	89.36	92.58	86.50	91.22	32.31
Qwen-72B	59.66	57.89	53.36	56.90	80.17
LLaMA-70B	59.66	68.34	67.29	71.81	68.03

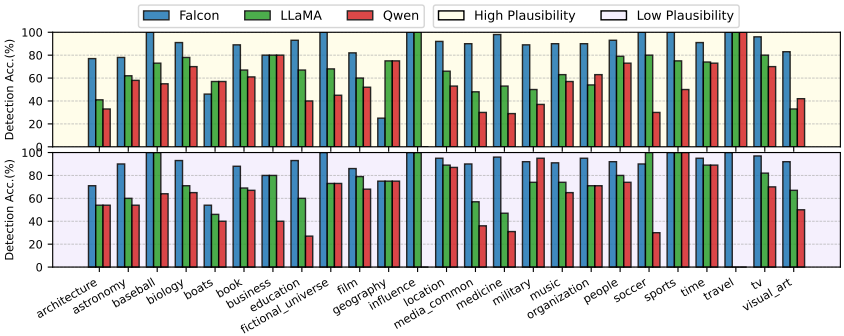
Table 16: Detection accuracy (%) of each LLM-based judge on fake facts generated by Phi-4 and Llama-8b (under high and low plausibility settings), along with real facts.

We analyze the detection performance of the three LLM judges (Falcon-40b, Llama-70b, and Qwen-72b) in identifying fake information generated by the Phi-4 and Llama-8b models. In particular, Tables 16 reports the detection accuracy (averaged across all the categories addressed the KG) across both generated fake facts and also real facts. In particular, real facts have been extracted focusing on the same 25 categories discussed above and were generated via Phi-4 using the pipeline described in Section 7.1.2. Note that, including real facts in the evaluation enables us to assess not only the model's ability to detect fake information, but also its reliability when classifying true statements. This analysis is important for identifying potential detection biases or, more in general, low detection performance, and understanding whether the LLM-based judges are trustworthy even in the presence of factual content. The results reveal that detection accuracy varies significantly across models. Notably, Falcon-40b demonstrates strong performance in detecting fake information but struggles considerably when evaluating real facts, often misclassifying them. In contrast, Qwen-72b and Llama-70b, despite being large-scale models, show substantially lower accuracy in detecting fake content but perform better in classifying real information. From these observations, several insights emerge. First, the evaluated LLMs show considerable limitations in accurately distinguishing between fake and real facts. While larger models such as Qwen-72b and Llama-70b appear more balanced in their responses, their accuracy

in detecting fake content is around 50–60%, which is close to random guessing. On the other hand, Falcon-40b, though effective in flagging fake information, tends to adopt an overly conservative stance, labeling most inputs as fake regardless of their actual truthfulness. This behavior suggests a possible bias introduced during training and indicates that Falcon-40b may favor caution over nuance. These findings underscore the broader challenge of building reliable LLM-based fact verification systems. They also point to the need for future work on improving detection accuracy through targeted fine-tuning, prompt engineering, or hybrid approaches that integrate structured knowledge. Understanding the training dynamics and decision biases of current models is essential for advancing trustworthy and effective detection tools.



(a) Fake detector Phi-4.



(b) Fake detector Llama-8b.

Figure 25: Detection accuracy across different categories from the knowledge graph. Three LLM-based judges were evaluated: Falcon-40b, Llama-70b, and Qwen-72b (see legend for colors). Fake facts are generated using two models: Phi-4 (top plot) and Llama-8b (bottom plot). Results are shown for fake facts with high plausibility (yellow background) and low plausibility (purple background).

7.9 Category-Wise Analysis of Fake Fact Detectability

This set of experiments investigates the detectability of fake facts across different KG's categories. The analysis has two main objectives: *(i)* to identify potential biases in the detection performance of LLM-based judges across specific categories, and *(ii)* to evaluate whether certain LLMs used for generating fake facts are more capable of producing stealthier, harder-to-detect statements depending on the targeted category. Note that this analysis focuses exclusively on fake news, with the aim of also understanding how the choice of generative LLM affects the creation of misleading content.

Figure 25 presents results across 25 categories. Each subplot distinguishes two levels of plausibility (see Section 7.2): high plausibility is shown with a yellow background, and low plausibility with a purple one. We compare fake information generated by two different LLMs: Phi-4 (top plot) and Llama-8b (bottom plot).

The overall trend observed in the results supports the average findings presented in Table 16, confirming that Falcon-40b tends to be biased toward classifying sentences as fake. While this may initially appear to be a positive outcome, since the current analysis focuses only on fake facts, previous discussion about Table 16 shows that this behavior stems from a conservative bias, as Falcon-40b also frequently labels correct facts as fake, indicating a lack of nuance in its judgment. Despite this limitation, Figure 25 shows that the trend is generally consistent across categories, although some category-specific variations exist, for example, in architecture and boats, where performance deviates more significantly.

In contrast, Llama-70b and Qwen-72b generally underperform and exhibit a wide variance in detection accuracy depending on the category. For instance, Llama-70b performs well on 'sports' and 'fictional universe', but shows poor results in many other categories. This highlights how the addressed LLM judges may be ill-suited for detecting misinformation in specific domains. As a consequence, a category-aware analysis could be used by attackers to identify and exploit weaknesses in these models, improving the success of misinformation designed to bypass automated detection. This open interesting challenges for future investigations.

The effect of plausibility level in fake fact generation is also evident. In some categories, using a higher plausibility significantly reduces detectability. For example, in the 'sports' category in Figure 25(a), the detection rate for Llama and Qwen drops from nearly 100% for low plausibility fake facts to around 78% for high plausibility ones. An even more striking example can be seen in the 'travel' category in Figure 25(b), where detection accuracy drops from 100% to 0% when moving from low- to high-plausibility facts, again for both Llama and Qwen. These results

further confirm that increasing the semantic plausibility of fake facts effectively reduces their detectability, even by state-of-the-art LLM-based detectors.

7.10 Conclusions

In this Chapter, we presented a structured process that uses knowledge graphs and large language models to create misinformation in a controlled and meaningful way. The method starts with true facts taken from a knowledge graph, then makes small, guided changes to them to create false but believable versions. These modified facts are then turned into natural, fluent sentences by a large language model. This setup allows us to produce false statements that sound coherent but vary in how believable they are, helping us study how misinformation is created and how it can be detected. Our experiments showed that the generated misinformation was often very convincing, especially when it kept a high level of semantic plausibility. However, current large language models still struggle to detect such misinformation reliably. Detection results varied between models: some, like Falcon-40B, were too cautious and often labeled true statements as false, while others, like LLaMA-70B and Qwen-72B, were inconsistent depending on the type of information that had been changed. These results show that knowledge-based misinformation can be effectively created, and that today's LLMs are still vulnerable to it.

These findings point to two main directions. On the defensive side, improving LLM-based detection will likely require more focused fine-tuning, evaluation methods that consider different semantic categories, or hybrid systems that combine LLMs with structured reasoning. On the offensive side, controlled misinformation generation can be a useful way to test detection models and find their weaknesses. More broadly, this study highlights the double nature of LLMs: they can both create realistic misinformation and act as truth checkers, but their verification abilities remain limited. By using knowledge graphs together with LLMs, this work shows how RAG-like systems can go beyond simple fact retrieval and be used for adversarial testing of misinformation detection systems.

8 On-Device Derivation of IoT Usage Control Policies: Automating U-XACML Policy Generation from Natural Language with LLMs in Smart Homes Environments

The need to extend retrieval-augmented generation systems with task-specific modules goes beyond misinformation analysis. Another key challenge in cybersecurity is enforcing access and usage control policies in complex environments such as smart homes, offices, and healthcare systems. The rapid growth of IoT technologies has turned these spaces into interconnected ecosystems, where many different devices continuously share and process data. While this improves functionality, it also raises serious security and privacy issues, especially regarding authorization management. Defining and enforcing policies in these settings is not straightforward. Languages like XACML are powerful but difficult to use, requiring expertise that most users do not have. Cloud-based solutions, on the other hand, introduce privacy risks and dependence on external infrastructure. This trade-off between usability, expressiveness, and privacy highlights the need for systems that can automatically translate natural language instructions into precise, enforceable rules that can run even on limited hardware.

To address this gap, the second specialized module developed in response to **Q3** focuses on automatic policy generation. Using natural language processing and lightweight large language models, it converts unstructured user commands into enforceable access and usage control policies. This bridges the gap between human intentions and machine-level enforcement, reducing manual configuration and enabling scalable use across different environments.

The main contributions of this chapter are as follows: A framework was developed for automatically generating enforceable policies from natural language inputs, creating a direct link between user instructions and system rules. It was shown that small, optimized models can run efficiently on local devices, allowing privacy-sensitive settings like smart homes to operate without external servers. The method was proven to work across multiple domains, including smart homes, offices, and healthcare, demonstrating its adaptability to different contexts.

This chapter introduces the *Text2Policy* framework, which implements these contributions. It follows a two-step process: user commands are first converted into structured representations and then compiled into executable U-XACML policies. The system was evaluated under various conditions, including ambiguous or noisy inputs, cross-domain scenarios, and deployment on limited hardware. Results show that the generated policies reach expert-level accuracy and remain reliable even on

constrained devices.

8.1 Background

This section provides essential background and introduces key concepts foundational to this work.

Attribute-Based Access Control. ABAC is an evolution of Access Control List (ACL) that combines the flexibility of Role Based Access Control (RBAC) with fine-grained management of ACL [193]. In ABAC, access is governed by policies that algorithmically combine access control rules. Each rule applies to a target consisting of attributes related to subjects, objects, actions, and the environment. The rule also contains a condition that must be satisfied for it to be applied to the target [194]. ABAC supports attributes from diverse systems, enabling flexible, expressive rules that apply to heterogeneous resources. The evaluation of applicable rules yields to an authorization decision, typically *permit* or *deny*), or *indeterminate* in case of errors. An advanced extension of ABAC is the Usage Control (UCON) model.

The Usage Control Framework. Building on ABAC principles, the *UCON framework* regulates the exercise of rights on resources by subjects following the UCON model [195]. It extends the XACML reference architecture and language [196], incorporating dynamic evaluation of policies, continuous monitoring and revocation of ongoing usages, and mutable attribute management.
Unlike traditional access control, which checks permissions only at request time, UCON continuously checks whether access rights remain valid throughout the whole duration of an access to a resource (*usage*). Access grants may be revoked if the value of some subject's, resource's, or environmental attributes change. Attributes whose value can change over time are called *mutable attributes*. For example, a subject may be allowed to perform an operation in a room only if and as long as the room temperature is between 10 and 35 degrees. Traditional access control only checks conditions at the time of request, allowing the operation if the room temperature is within range. In contrast, UCON not only performs this initial check but also continuously monitors the condition throughout the access period, revoking the grant if the temperature goes out of range.
Usage Control Policies (*UCPs*) define access strategies using the U-XACML language [197], which extends XACML with time-based conditions. Like standard XACML policies, a UCP consists of multiple *rules*, each returning one of four

possible decisions: `Permit`, `Deny`, `NotApplicable`, or `Indeterminate`. The final authorization decision of a policy evaluation is determined by a *rule-combining algorithm*. Additionally, when multiple policies are grouped into a *policy set*, a *policy-combining algorithm* is used to determine a final authorization decision.
Each policy and rule include a *target*, which acts as a filter to determine whether it applies to a given access request. The target can specify a combination of subjects, resources, actions, and environmental attributes. If a request does not match the target, the policy or rule is considered `NotApplicable`, and no further evaluation occurs.
Besides the target, in XACML, a rule consists of a single *effect*, either `Permit` or `Deny`, and a *condition*. The condition, if present, defines the constraints under which the rule applies. It contains a Boolean expression composed of *predicates*, that evaluate contextual attributes of subjects, resources, and environment, such as user roles, resource state, time, and so on. If the condition evaluates to true, the rule contributes to the decision-making process by enforcing its defined effect.
Differently from XACML, a U-XACML rule is structured into three distinct sections, namely, *pre-*, *ongoing-*, and *post-*, each containing its own condition and evaluated at different stages of the usage control workflow, as explained in the following.
The UCON workflow begins when a subject s requests access to perform an action a on a resource r. The *Policy Enforcement Point* (*PEP*) initiates this process by issuing a *tryAccess* message, which includes a *UCON request*, encoded in XACML. A typical UCON request contains the attributes `subject-id`, which identifies the subject, the attribute `resource-id`, which identifies the resource, and the attribute `action-id`, which specifies the action that such a subject wants to perform on the resource.
The UCS processes the request and the Policy Decision Point (PDP) evaluates the *pre-condition* of the selected UCP to determine whether access should be granted. In XACML format, the pre-condition is identified by the tag `DecisionTime="pre"` within the `Condition` element. If the result is `Deny`, the UCS responds to the PEP with a *denyAccess* message, enforcing the decision by denying access to the resource. If `Permit` is returned, the UCS sends a *permitAccess* message to the PEP, allowing access to proceed.
Once access begins, the PEP sends a *startAccess* message to the UCS. The UCS then evaluates the *ongoing-condition* to determine whether access can continue. If the result is `Deny`, the UCS sends a *revokeAccess* message to the PEP, which immediately revokes access. If `Permit`, the UCS notifies the PEP via a *permitAccess* message that access can continue. Throughout the access period, if relevant attribute changes occur, the UCS may trigger a re-evaluation of the ongoing-condition, potentially

leading to access revocation.
Finally, when access to the resource ends, the PEP sends an *endAccess* message to the UCS. This triggers the evaluation of the *post-condition*, concluding the workflow.
Conflicts in access and usage control policies, such as contradictory decisions, rule overlaps, or redundant constraints, pose a significant challenge in policy management. In the context of XACML and its extensions, various techniques have been developed to detect and resolve such conflicts at the policy specification level [198, 199, 200, 201]. These include static analysis tools and formal methods that analyze rule interactions and enforce consistency prior to deployment.

Reference Scenario. In this work, we adopt the reference architecture proposed in the EU SIFIS-Home project, as illustrated in Figure 26 [202]. This architecture

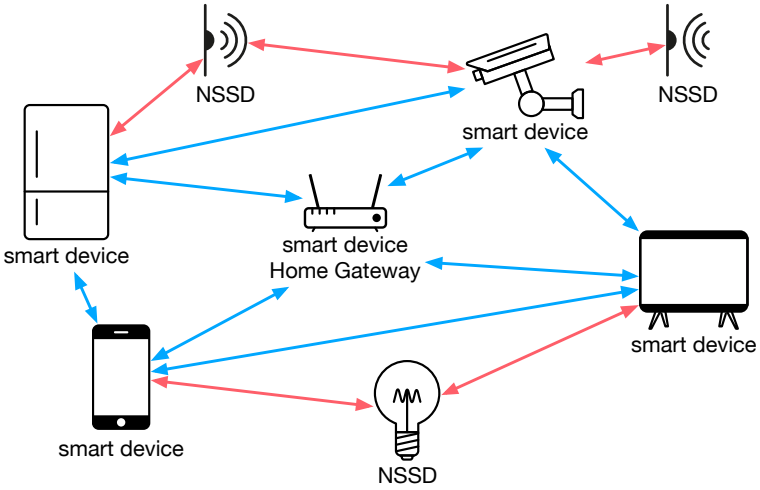


Figure 26: Smart home's reference architecture.

represents a distributed smart home environment, which employs a peer-to-peer model, enabling distributed computation, ensuring resiliency by avoiding single point of failure and ensuring privacy as data are stored and analyzed locally in the smart home environments. Core components are the *smart devices*, which possess sufficient computational power to execute complex tasks, such as running AI software for intelligent automation, including text analysis engine, data analysis, and decision-making processes. These devices communicate using mesh-based protocols and Distributed Hash Tables to ensure reliable data exchange, service

coordination, and system resilience. Complementing them are *Not So Smart Devices* (NSSD), which are simpler devices such as sensors and actuators that interact with the physical environment. They rely on Smart Devices to process their data and manage control logic. Interested readers can refer to [202] for a complete understanding of the SIFIS-Home architecture and its approach to privacy and security in smart homes.

8.2 From Natural Language to U-XACML: Policy Translation Workflow

Policies relevant for smart home environments can be pretty complex as they have to control many users with different access levels, considering attributes for multiple sensors and managing a consistent number of functionalities. Our proposed methodology aims at automating the process of complex policy definition, starting from a voice command. Considering the reference architecture presented in Section 8.1, the command is collected by a microphone on any smart device and is locally processed. The speech-to-text processing is not in the scope of this work, as it is handled through available speech-to-text libraries. Hence, we assume in the following to have as input the command in natural language in textual form. The process of translating natural language commands into UCON-compliant policies involves a multi-phase workflow, which is illustrated in Figure 27 and described in the following.

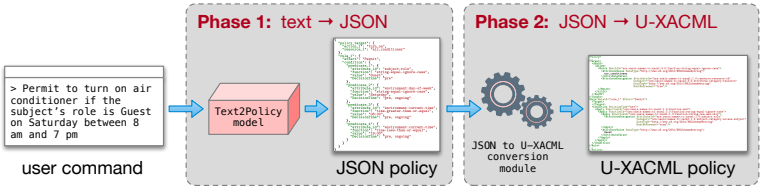


Figure 27: Policy translation workflow: From natural language command to U-XACML policy.

8.3 From Natural Language Command to JSON Policy

The process begins with a natural language command from the user, which defines a policy outlining the conditions for executing a specific action on a resource. This command can be provided as text or obtained through transcription of a voice

```
{
  "policy_target": {
    "action_1": "turn_on",
    "resource_1": "air_conditioner"
  },
  "rule_1": {
    "effect": "Permit",
    "condition": {
      "predicate_1": {
        "attribute_id": "subject:role",
        "function": "string-equal-ignore-case",
        "value": "Guest",
        "DecisionTime": "pre"
      },
      "predicate_2": {
        "attribute_id": "environment:day-of-week",
        "function": "string-equal-ignore-case",
        "value": "Saturday",
        "DecisionTime": "pre, ongoing"
      },
      "predicate_3": {
        "attribute_id": "environment:current-time",
        "function": "time-greater-than-or-equal",
        "value": "08:00",
        "DecisionTime": "pre, ongoing"
      },
      "predicate_4": {
        "attribute_id": "environment:current-time",
        "function": "time-less-than-or-equal",
        "value": "19:00",
        "DecisionTime": "pre, ongoing"
      }
    }
  }
}
```

Listing 1: JSON Policy for the user command in Figure 27.

command. In Figure 27, such a command is: *"Permit to turn on air conditioner if the subject's role is Guest on Saturday between 8 am and 7 pm"*.
In Phase 1 of the workflow, our *Text2Policy* model processes natural language commands in text form, transforming them into a structured *JSON Policy*.
This policy represents key elements derived from the command such as actions, resources, subjects, and predicates in a structured JSON format. This representation simplifies the subsequent conversion into U-XACML.
Through the JSON policy, we provide a clear and machine-readable representation of the original command. This approach significantly reduces the number of tokens the LLM needs to generate compared to the more verbose XACML format, which often includes redundant sequences of words. Nonetheless, we structured the JSON policies to mirror the design and logic of UCON policies. Listing 1 shows the JSON policy produced from the user command shown in Figure 27.
For clarity and convenience, in the following, references to specific lines in the listings use the notation L:N, where L denotes the listing number and N indicates the corresponding line number. For example, 1:12 refers to line 12 of Listing 1.
Starting from the natural language command, our model first identifies the rules and their corresponding targets. Then, if all the rules share the same target, a policy target is reported in the JSON with the key `policy_target`, positioned at the top of the JSON structure (1:2–1:5). In this way, we reduce the number of generated tokens avoiding redundant repetitions that could lead to undesirable incoherence.

For each identified rule, the model creates a single condition, which is represented as a set of *predicates*. For example, in the JSON policy in Listing 1, there is only one rule, which contains a condition composed of four distinct predicates: `predicate_1` (1:9–1:14) for the subject role, `predicate_2` (1:15–1:20) for the day, `predicate_3` (1:21–1:26) for the starting time, and `predicate_4` (1:27–1:32) for the ending time.

Predicates are organized based on their *decision time* by means of the property name `DecisionTime`, which represents the phase of the UCON decision process when they should be enforced. This allows our tool to map them to their corresponding UCON conditions in the rule of the U-XACML policy, i.e., *pre-conditions* and *ongoing-conditions*.

For example, the `Decision Time` for `predicate_1` (1:13) is set to `pre` because the attribute considered in this predicate is the *subject role*, which is an *immutable* attribute. Therefore, in the corresponding U-XACML policy, the whole predicate will be automatically included within a *pre-condition*, represented by the <Condition DecisionTime="pre"> element, as illustrated in Listing 2 (2:22).

```
<Policy RuleCombiningAlgId="urn:oasis:names:tc:xacml:3.0:rule-combining-algorithm:deny-unless-permit" ... >
...
<Target>
  ...
  <AnyOf>
    <AllOf>
      <Match MatchId="urn:oasis:names:tc:xacml:3.0:function:string-equal-ignore-case">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
          air_conditioner
        </AttributeValue>
        <AttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
                             Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
                             DataType="http://www.w3.org/2001/XMLSchema#string"
                             MustBePresent="true"/>
      </Match>
    </AllOf>
  </AnyOf>
</Target>
<Rule RuleId="rule_1" Effect="Permit">
  <Target>
  </Target>
  <Condition DecisionTime="pre">
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
      <Apply FunctionId="urn:oasis:names:tc:xacml:3.0:function:string-equal-ignore-case">
        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
          <AttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:subject:role"
                               Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
                               DataType="http://www.w3.org/2001/XMLSchema#string"
                               MustBePresent="true"/>
        </Apply>
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
          Guest
        </AttributeValue>
      </Apply>
      ...
    </Apply>
  </Condition>
</Rule>
...
</Policy>
```

Listing 2: Portion of U-XACML policy generated from the JSON policy shown in Listing 1.

On the other hand, `predicate_2`, `predicate_3`, and `predicate_4` are related to *mutable* attributes, specifically, the day and time.

Therefore, in the JSON policy, the decision times for these predicates are both `pre` and `ongoing` (1:19, 1:25, and 1:31). This means that, in the final U-XACML policy, each of these predicates will be inserted both under the <Condition DecisionTime="pre"> element and under the <Condition DecisionTime=

"ongoing"> element.
We assume that the predicates in the policies being translated into U-XACML are expressed in *Disjunctive Normal Form (DNF)*. This assumption does not limit the expressivity of our approach, as the Disjunctive Normal Form Theorem states that any logical formula can be transformed into an equivalent one in DNF [203].
This allows us to structure the JSON policy in a straightforward manner. Specifically, predicates within a condition of a rule of a JSON policy are interpreted as a conjunction, meaning that all predicates must be true for the condition to evaluate to true. Conversely, to represent a disjunction of two or more predicates, each predicate must be placed in a distinct rule within the same JSON policy, ensuring that all such rules share the same effect.

8.4 Converting JSON Policies to U-XACML

Phase 2 of the workflow involves the *Conversion Module*, which is responsible for transforming a JSON policy into UCON-compliant format. This transformation is deterministic and fully automated, achieved by parsing the JSON structure and systematically mapping attributes, predicates, and other elements into a structured U-XACML policy.
The JSON policy in Listing 1 is complete, as it contains all essential components: the policy target, rules, and conditions. The U-XACML representation in Listing 2 shows the translation of `predicate_1` of `rule_1` into the U-XACML format, representing only a portion of the JSON policy taken as reference. To facilitate comparison, corresponding elements between Listing 1 and Listing 2 are highlighted in both listings. The following explanation shows the general methodology employed by the conversion module through a specific example, demonstrating how a JSON policy is translated into a U-XACML policy.

Policy and Rule Targets In the JSON policy shown in Listing 1, the `policy_target` object contains the attributes `action_1` (*turn on*) and `resource_1` (*air conditioner*). In the U-XACML format, these attributes must be placed within the policy target, specifically inside the XACML <Target> element, which defines the scope of applicability for the policy. For simplicity, Listing 2 only shows the attribute `air_conditioner` (2:9, 2:11) in the <Target> section (2:3–2:18). The complete version of the target section would also include the action-id *turn on*, logically ANDed with the resource-id. Regarding rule targets, we note that the JSON policy contains only one rule without a specified target. Consequently, the U-XACML representation includes an empty `Target` element

(2:20, 2:21), meaning that the rule applies to all requests that reach the policy, with its applicability determined solely by the rule's condition.
In cases where we have multiple targets within a rule or policy that belong to the same category (subject, action or resource), they are put in logical OR using the tag <AnyOf>. The tag <AnyOf> combines several <Match>, whereby it is sufficient if at least one target is fulfilled (logical OR).

Rules Definition The rules in the JSON policy, such as `rule_1` in Listing 1 (1:6–1:34), are transformed into U-XACML <Rule> elements (2:19–2:38). Each rule of the JSON policy consists of the following elements:

- **Rule Name**: The rule is represented as a key-value pair, where the key (e.g., `"rule_1"`) serves as the rule's unique identifier (1:6). This key is directly mapped to the `RuleId` attribute of the <Rule> element in the U-XACML policy (2:19).
- **Target**: Within the rule, the value of the `target` key determines its applicability. In the provided JSON policy, the rule's target is not present, which results in an empty target in the U-XACML policy (2:20, 2:21). However, if a target were present, it would be represented in the U-XACML policy as discussed above for the policy target.
- **Effect**: Within the rule, the value of the `effect` key specifies whether the rule permits or denies the access (e.g., *Permit* in our example). This maps directly to the `Effect` attribute of the <Rule> element in XACML (2:19).
- **Condition**: The `condition` key holds a set of predicates that define additional constraints that must be satisfied for the rule to apply.

  Note that if a single condition in the JSON policy includes multiple distinct `DecisionTime` values within its predicates, it will be translated into multiple conditions in the U-XACML policy, ensuring proper enforcement at different evaluation stages.

Conditions Mapping All the predicates within a `condition` object of a JSON policy, such as `predicate_1` in our example, are mapped to XACML <Condition> elements. Each predicate in the JSON policy specifies attribute identifiers, functions, and values, which are transformed into U-XACML format as follows:

- `function` (1:11): Mapped to the appropriate XACML `FunctionId` field of the XACML <Apply> element (2:24).
- `attribute_id` (1:10): Mapped to the `AttributeId` field within the <AttributeDesignator> element in XACML (2:26).
- `value` (1:12): Represented in XACML exploiting the <AttributeValue> element (2:31–2:33).
- `DecisionTime` (1:13): It is used to chose whether to copy the predicate within the condition of the rule having `DecisionTime="pre"` or within the condition having `DecisionTime="ongoing"` or both.
- `data_type` (2:28): We automatically detect the XACML-compliant data type for the <AttributeValue> element by examining the `predicate value` in the JSON policy. Additionally, the algorithm employs a rule-based approach to sequentially evaluate the input against various data types.

Policy Generation Once the U-XACML targets, rules, and conditions are defined, they are incorporated into a complete U-XACML policy using the <Policy> element (2:1–2:40).

There could be cases where the user command includes also a default rule, such as: *"Permit to turn on the air conditioner if the subject's role is Guest from Monday to Saturday between 8 am and 7 pm.* ***Deny otherwise****"*. Here, *"Deny otherwise"* represents a default rule (*default-deny* rule), i.e., a rule that is applied when no other rules are applicable. In this example, the JSON policy generated by the *Text2Policy* model would represent the default rule as *Rule 2* (since *"Permit to turn on the air conditioner if the subject's role is Guest from Monday to Saturday between 8 am and 7 pm."* is the same *User Command* of Figure 27 and can be represented in one rule), with effect *Deny* and no predicates.

To set the appropriate *rule-combining algorithm*, if the generated policy contains a default-deny rule, we use the deny-unless-permit algorithm and remove the default rule. Conversely, if it contains a default-permit rule, we use the permit-unless-deny algorithm and remove the default rule.

If all the rules within the policy have a permit effect, we use the deny-unless-permit algorithm. Conversely, if all the rules have a deny effect, we use the permit-unless-deny algorithm. If the rules contain a combination of permit and deny effects, we adopt the deny-unless-permit algorithm. This approach ensures that all possible cases in the user prompt are covered without compromising security.

In Listing 2, the rule-combining algorithm for this policy is `deny-unless-permit` (2:1), as there is only one rule with effect permit and no default rule.
After generating the U-XACML policy, formal verification tools can be employed to validate its correctness and consistency. These tools enable automated reasoning to identify logical conflicts, redundancies, and policy misconfigurations, which is particularly valuable in security-critical environments. For example, several XACML conflict detection tools (e.g., [198, 199, 200]) can be readily adapted to our setting, as U-XACML is an extension of XACML.

8.4.1 Flattening Process

Although the previous explanations present the conversion as a direct transformation from a JSON policy to a U-XACML policy, the conversion module first applies a preprocessing step: the *flattening process*. This intermediate step simplifies the JSON structure before conversion.
It consists of restructuring nested keys into a flat format by concatenating the hierarchical levels with a separator. As an example, Figure 28 shows how the action `action_1`, which is part of the policy target, is transformed into a flattened structure.

```
// JSON policy from the Text2Policy model
{"policy_target": { "action_1": "turn_on" }}

// Flattened version
{"policy_target_action_1": "turn_on"}
```

Figure 28: Flattening process of a portion of a JSON policy, where the nested key `policy_target.action_1` is transformed into the flattened key `policy_target_action_1`.

In this case, the nested key `policy_target.action_1` is simplified into the single key `policy_target_action_1`. The conversion module operates on this flattened version, ensuring a structured and deterministic transformation process. However, for clarity, we present the conversion as if applied directly to the original JSON policy. The flattened version serves as an intermediate representation that simplifies attribute resolution and processing within the module.
Since the transformation is deterministic, evaluating the flattened JSON policy is equivalent to evaluating the final U-XACML policy. This equivalence allows us to use the flattened version to assess the performance of the *Text2Policy* model, as

detailed in Section 12.7.

8.5 Taxonomy-Driven Dataset Creation

In order to train and test the model for *Text2Policy* purpose, we built a dataset composed of $\langle command, JSON\ policy \rangle$ pairs by applying a domain-specific taxonomy to a set of pre-defined templates.
The development of a domain-specific taxonomy enables a structured representation of smart home environments through the organization of entities, attributes and relationships. Our taxonomy categorizes key elements, including users, devices, actions and environment parameters. To support access management, the taxonomy introduces *Access Levels* (e.g., *Resident*, *Guest*) and *Parental Advisory Levels* (e.g., *red*, *green*, *yellow*), which enable fine-grained control over smart home operations. Resource classification is based on device functionality: *Temperature Devices* (thermostats, heaters, air conditioners), *Brightness Devices* (ceiling lights, desk lamps, smart bulbs), *Security Devices* (door locks, security cameras, garage doors), and *Audio Devices* (smart speakers, televisions, soundbars). General-purpose devices such as motion sensors, smoke alarms, and CO_2 sensors are also included. Actions are structured by their operations. *Temperature control* includes setting, raising and lowering the temperature, while *Brightness control* adjusts the light intensity. *Security operations* manages locking mechanisms and *Audio control* allows volume adjustments. General commands include switching devices on and off as well as document-related actions such as reading and writing. This structured approach supports smart home interactions, security policies, and access controls. The methodology can be extended to other domains, such as smart offices, healthcare, and industrial IoT, ensuring a flexible and consistent workflow.
In the remainder of this section, we explain in detail the pipeline for creating the dataset, which follows the definition of the taxonomy.

8.6 Dataset Creation Pipeline

Figure 29 illustrates the dataset generation process once the taxonomy has been established, consisting of the following key steps:

Manual Templates Creation Domain experts manually design m initial templates [204, 205, 206], including command templates (CT) and their corresponding JSON policy templates (PT). A **Command Template (CT)** defines executable instructions in natural language, such as turning on a device or adjusting temper-

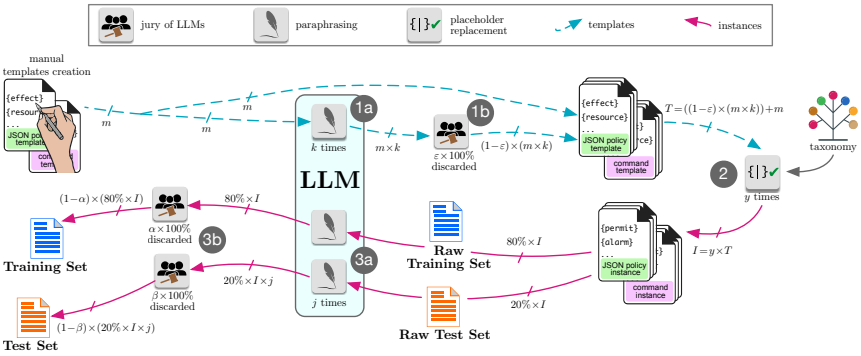


Figure 29: Dataset creation workflow after taxonomy definition.

`{effect}`[1] `{action}`[2] `{resource}`[3] *if the subject's role is* `{home-role}`[4], *on* `{Day}`[5] *between* `{F-Time}`[6] *and* `{S-Time}`[7], *or on* `{Day-2}`[8] *between* `{F-Time-2}`[9] *and* `{S-Time-2}`[10].

Prompt 1: Example of command template with annotations.

ature, possibly under given constraints. An example of a command template is shown in Command Template 1. A **JSON Policy Template (PT)** specifies the key components for building the policy in JSON format, extracted from a textual command. It includes execution constraints and permissions. The corresponding JSON policy template for Command Template 1 is presented in Listing 30.

We manually built a total of 256 *template pairs*, each consisting of a command template and a JSON policy template.

Please note that this task is agnostic of the specific domain, and the produced template pairs can be reused for creating datasets for virtually any smart environment since the domain's taxonomy is applied in a subsequent step, as described later in this section.

The process of generating these template pairs is detailed in Section 8.6.1.

`Permit`[1] `turn on`[2] `air conditioner`[3] *if the subject's role is* `"Guest"`[4], *on* `Saturday`[5] *between* `8 am`[6] *and* `7 pm`[7], *or on* `Sunday`[8] *between* `9 am`[9] *and* `5 pm`[10].

Command Instance 1: Example of command instance with annotations.

Paraphrasing of Templates In **Step 1a** of Figure 29, the initial command templates are paraphrased by an LLM, generating k paraphrases for each command template. This process improves variability and ensures the coverage of different phrasings for the same command and JSON policy template pair.
In **Step 1b**, a **jury of 3 LLMs** (see Section 8.6.1 for more details) evaluates the quality of the paraphrases generated in Step 1a, ensuring that only accurate and meaningful paraphrases are retained. During this process, a fraction ε of template pairs is discarded.
The manually created template pairs and the retained template pairs compose the *final set of template pairs* $T = ((1 - \varepsilon) \times (m \times k) + m)$.

Filling Placeholders In **Step 2**, the placeholders within each template pair in T are replaced dynamically with *values* derived from the taxonomy. This step creates realistic and context-aware command-JSON policy instances (*instance pairs*) by exploring y value combinations, resulting in $I = y \times T$ unique instance pairs.
The replacement process follows the order in which placeholders appear in the paraphrased command template. First, the type of each placeholder (e.g., `{action}`, `{resource}`, or `{role}`) is identified. The first placeholder is selected entirely at random from its corresponding domain. For all subsequent placeholders, values are chosen randomly but constrained to those that are compatible with values already assigned to previous placeholders. Compatibility is determined by consulting an *adjacency matrix* A, which ensures logical consistency in the generated instances. Once all placeholders are filled following this process, the final instance pair is added to the set of instance pairs I. Further details on these operations are provided in Section 8.6.2.

Raw Dataset Generation The complete set of instance pairs I is divided into two subsets to ensure a structured evaluation. Approximately 80% of the instances ($\approx 80\% \times I$) are allocated to the training set, forming the *Raw Training Set*, while the remaining 20% ($\approx 20\% \times I$) constitute the *Raw Test Set*, used for evaluation.

Final Dataset Refinement In **Step 3a**, both the raw training set and the raw test set are provided as input to the LLM, which generates paraphrased versions. Specifically, the entire raw training set is paraphrased once, while the raw test set is paraphrased j times. This ensures robustness and improves the generalization capabilities of the dataset. The result is then used as input for the next step. In **Step 3b**, the jury of 3 LLMs (as in Step 1b) assesses the correctness of the paraphrased

training and test sets, ensuring that the meaning of the commands remains consistent. Such a filtering process excludes any paraphrased commands that differ in meaning from their original, non-paraphrased counterparts. This step enhances the dataset's reliability and consistency. After this refinement, the dataset is finalized and ready to be used for training and evaluation of the *Text2Policy* model.
In the following, we detail the step-by-step workflow necessary to build the dataset, discussing **Step 1a** and **Step 1b** in Section 8.6.1, **Step 2** in Section 8.6.2, and **Step 3a** and **3b** in Section 8.6.3.

8.6.1 Template pairs Creation and Paraphrasis

The starting point for the dataset creation pipeline is the generation of command templates and their corresponding JSON policy templates, resulting in the creation of template pairs.

Command Template Definition Formally, a *Command Template* (*CT*) can be defined as a structured string comprising fixed components and placeholders.
Let $P_{\text{CT}} = \{p_1, p_2, \ldots, p_n\}$ be the set of placeholders for a command template, where each p_i represents a variable slot that can be instantiated with a value from a predefined domain D_i.
A command template *CT* is defined as:

$$CT = S_0\ p_1\ S_1\ p_2\ \ldots\ p_n\ S_n,$$

where $\{S_i \mid i = 0, \ldots, n\}$ is a set of fixed strings representing the static parts of the command.
In the example reported in Command Template 1, the set of placeholders is defined as:

$$P_{\text{CT}} = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}\},$$

where:

$$\begin{aligned}
p_1 &= \texttt{\{effect\}}, & p_2 &= \texttt{\{action\}}, & p_3 &= \texttt{\{resource\}},\\
p_4 &= \texttt{\{home-role\}}, & p_5 &= \texttt{\{Day\}}, & p_6 &= \texttt{\{F-Time\}},\\
p_7 &= \texttt{\{S-Time\}}, & p_8 &= \texttt{\{Day-2\}}, & p_9 &= \texttt{\{F-Time-2\}},\\
p_{10} &= \texttt{\{S-Time-2\}}.
\end{aligned}$$

The fixed components in the command template, denoted as S_{CT}, correspond to the static parts of the command that remain unchanged. Formally, S_{CT} is defined as

the difference between the full command template CT and the set of placeholders P_{CT}: $S_{\text{CT}} = CT - P_{\text{CT}}$. In the example of Command Template 1, S_{CT} consists of the following fixed components:

$$s_1 = \textit{if the subject's role is}, \quad s_2 = \textit{on}, \quad s_3 = \textit{between},$$
$$s_4 = \textit{and}, \quad s_5 = \textit{or on}, \quad s_6 = \textit{between},$$
$$s_7 = \textit{and}.$$

```
{
  "policy_target": {
    "action_1": "{action}",    // Replace with: "turn_on"
    "resource_1": "{resource}" // Replace with: "air_conditioner"
  },
  "rule_1": {
    "effect": "{effect}",      // Replace with: "Permit"
    "condition": {
      "predicate_1": {
        "attribute_id": "subject:role",
        "function": "string-equal-ignore-case",
        "value": "{home-role}", // Replace with: "Guest"
        "DecisionTime": "pre"
      },
      "predicate_2": {
        "attribute_id": "environment:day-of-week",
        "function": "string-equal-ignore-case",
        "value": "{Day}",       // Replace with: "Saturday"
        "DecisionTime": "pre, ongoing"
      },
      "predicate_3": {
        "attribute_id": "environment:current-time",
        "function": "time-greater-than-or-equal",
        "value": "{F-Time}",    // Replace with: "08:00"
        "DecisionTime": "pre, ongoing"
      },
      "predicate_4": {
        "attribute_id": "environment:current-time",
        "function": "time-less-than-or-equal",
        "value": "{S-Time}",    // Replace with: "19:00"
        "DecisionTime": "pre, ongoing"
      }
    }
  },
  "rule_2": {
    "effect": "{effect}",      // Replace with: "Permit"
    "condition": {
      "predicate_1": {
        "attribute_id": "subject:role",
        "function": "string-equal-ignore-case",
        "value": "{home-role}", // Replace with: "Guest"
        "DecisionTime": "pre"
      },
      "predicate_2": {
        "attribute_id": "environment:day-of-week",
        "function": "string-equal-ignore-case",
        "value": "{Day-2}",     // Replace with: "Sunday"
        "DecisionTime": "pre, ongoing"
      },
      "predicate_3": {
        "attribute_id": "environment:current-time",
        "function": "time-greater-than-or-equal",
        "value": "{F-Time-2}", // Replace with: "09:00"
        "DecisionTime": "pre, ongoing"
      },
      "predicate_4": {
        "attribute_id": "environment:current-time",
        "function": "time-less-than-or-equal",
        "value": "{S-Time-2}", // Replace with: "17:00"
        "DecisionTime": "pre, ongoing"
      }
    }
  }
}
```

Figure 30: JSON policy template PT for Command Template 1

JSON Policy Template Definition. A *JSON Policy Template* (PT) is a structured representation of the key components to build a usage control policy expressed in JSON format. It defines the logical rules, conditions, and predicates that govern the execution of a command. Let $P_{\text{PT}} = P_{\text{CT}} = \{p_1, p_2, \ldots, p_n\}$ be the set of placeholders in the JSON policy template, where each p_i corresponds to the same placeholder in the associated command template CT. A JSON policy template PT

is formally expressed as:

$$PT = S_{\text{PT}}(p_1, p_2, \ldots, p_n),$$

where S_{PT} defines the static structure of the JSON object, including its fields (e.g., `policy_target`, `rules`, etc.). Hence, the fixed components S_{PT} represent the structure and logical constraints of the policy, such as predicates and rules.
The JSON policy template shown in Listing 30 corresponds to Command Template 1. Together, they form a template pair.

Template Paraphrasing. To expand the pool of command templates available to us, we used a Large Language Model (LLaMA8b [207]) to generate paraphrases or variations of command templates (**Step 1a**). By diversifying the natural language expressions of CT, we enhance the dataset's robustness to linguistic variability, allowing for better generalization in real-world scenarios. Additionally, this process increases the dataset's size, providing additional training examples to improve model performance.
Given a template pair $\langle CT, PT \rangle$ composed of an original command template CT and its corresponding JSON policy template PT, the LLM generates a set of paraphrased template pairs defined as:

$$\text{Var}(\langle CT, PT \rangle) = \{\langle CT, PT \rangle, \langle CT_i, PT \rangle \mid i = 1, \ldots, k\}.$$

In this work, we set $k = 5$, meaning that each original template pair is expanded into five paraphrased variations.
This set $\text{Var}(\langle CT, PT \rangle)$ includes the original template pair as well as its paraphrased versions, where only the command template is modified while the JSON policy template remains unchanged across all template pairs.
Unlike the manually created original CT, which is guaranteed to be semantically correct, automatically generated paraphrases may not always preserve the original meaning. Thus, verifying the correctness of these paraphrases is essential. However, manual validation is impractical due to the large volume of generated paraphrases. To address this challenge, we adopted the *LLM-as-a-Judge* paradigm introduced by Zheng et al. [208] and employed a jury-based approach, as proposed by Verga et al. [209], since it demonstrated a higher agreement rate with human evaluations compared to the single-LLM approach.
In **Step 1b**, we employed three LLMs judges—LLaMA3.3 70b [207], Qwen2.5 70b [210, 192], and Phi4 [189]—to form *PoLL* (Π, a Panel of LLM Evaluators). To ensure semantic equivalence, each LLM independently evaluates whether a

paraphrased command template CT_i retains the same meaning as the original command template CT. Each judge votes either `true` (if the paraphrase preserves the original intent) or `false` (if the meaning is altered). The final decision, determined using a *majority vote* aggregation strategy, is formally defined as:

$$\Pi(CT, CT_i) = \begin{cases} \texttt{true}, & \text{if the majority of judges vote } \texttt{true} \\ \texttt{false}, & \text{otherwise.} \end{cases}$$

For each template pair, the validation process, which ensures that only semantically accurate paraphrases are retained, is defined as:

$$\text{Validated}(\text{Var}(\langle CT, PT\rangle)) = \{\langle CT_i, PT\rangle \mid CT_i \in \text{Var}(\langle CT, PT\rangle), \\ \Pi(CT, CT_i) = \text{true}\}.$$

As a result of this filtering, a fraction ε of template pairs—those that do not preserve the intended meaning—is discarded. The resulting $\text{Validated}(\text{Var}(\langle CI, PI\rangle))$ contains only those template pairs where the paraphrased was judged to be semantically valid.

After this step, we retained a total of 1604 template pairs. These validated template pairs, together with the m original template pairs, form the final set of template pairs I.

8.6.2 Logical Consistency in Placeholders Replacement

In **Step 2** of Figure 29, given a set of replacement values $V = \{v_1, v_2, \ldots, v_n\}$, where $v_i \in D_i$ (the domain of placeholder p_i), the generation of command Instance $CI(V)$ and policy instance $PI(V)$ consists in substituting each placeholder p_i in both CT and PT with its corresponding value v_i for all i:

$$CI(V) = S_0\ v_1\ S_1\ v_2\ S_2\ \ldots\ v_n\ S_n,$$
$$PI(V) = S_{\text{PT}}(v_1, \ldots, v_n).$$

This ensures dataset diversity, as each placeholder p_i can take values from its associated domain D_i. For example, considering our taxonomy, the domain D_{effect} includes `Permit` or `Deny`, while D_{resource} contains values like `thermostat`, `ceiling light`, or `door lock`. Command Instance 1 illustrates a command instance generated from Command Template 1, where each placeholder has been replaced with a value from its corresponding domain. The specific values used to

replace the placeholders are as follows:

$$v_1 = \texttt{Permit}, \quad v_2 = \texttt{turn on}, \quad v_3 = \texttt{air conditioner},$$
$$v_4 = \texttt{Guest}, \quad v_5 = \texttt{Saturday}, \quad v_6 = \texttt{8:00},$$
$$v_7 = \texttt{19:00}, \quad v_8 = \texttt{Sunday}, \quad v_9 = \texttt{9:00},$$
$$v_{10} = \texttt{17:00}.$$

Since multiple combinations of values can fill the placeholders of Command Template 1, Command Instance 1 represents just one possible instantiation of this template.

When automatically replacing placeholders in CT and PT with real values, we must ensure that the resulting instance pair is *logically consistent*.

To maintain logical consistency, we use an *adjacency matrix* A to map placeholders to compatible values. Formally, let A be a matrix of size $n \times n$, where n represents the total number of values across all domains in the taxonomy. Each entry $A[i, j]$ takes a value from $\{0, 1\}$, where $A[i, j] = 1$ indicates that a value from the domain D_i is compatible with a value from the domain D_j, while $A[i, j] = 0$ indicates incompatibility.

Given a command template CT with placeholders $P_{\text{CT}} = \{p_1, p_2, \ldots, p_n\}$, the adjacency matrix A defines a compatibility graph $G = (V, E)$. In this graph, V represents the set of values that can fill the placeholders in CT (and PT), and E consists of edges (v_i, v_j) such that $A[i, j] = 1$, meaning the values v_i and v_j are compatible.

The adjacency matrix A ensures that when instantiating CT and PT, a value $v_i \in D_i$ for p_i is only selected if it is compatible with the previously instantiated values $V_{\text{prev}} = \{v_1, v_2, \ldots, v_{i-1}\}$, as dictated by A. For example, if CT includes placeholders $P_{\text{CT}} = \{p_1, p_2, p_3\}$, where: $p_1 =$ `{action}`, $p_2 =$ `{resource}`, and $p_3 =$ `{effect}`, the adjacency matrix A enforces logical consistency in value selection. Specifically, actions like `set temperature` (p_1) can only be paired with resources like `thermostat` or `heater` (p_2), while actions like `lock` or `unlock` (p_1) are only compatible with resources like `door lock` or `garage door` (p_2). Furthermore, the effect (p_3) must align with the logical constraints imposed by the chosen action and resource.

Once the command instance CI is constructed by assigning values v_i to each placeholder p_i in CT, the corresponding policy instance PI can be generated. This process involves substituting each placeholder p_i in the policy template PT with the same value v_i that was selected during the construction of CI.

By leveraging A, we ensure that both CI and PI are logically consistent and adhere

Algorithm 4: Generation of an instance pair through sequential placeholder replacement

Input: – *Template pair* $\langle CT, PT \rangle$ with placeholders $P = \{p_1, p_2, \ldots, p_n\}$;
– *Domain* D: `Smart Home Environment`;
– *Adjacency matrix* A;
Output: Logically consistent *instance pair* $\langle CI, PI \rangle$
Initialize: Set of selected values $V \leftarrow \emptyset$;
foreach *placeholder* $p_i \in P$; `// ordered` $1 \rightarrow n$
do
  if $i = 1$; `// first placeholder`
  then
    Select v_1 randomly from its corresponding category in D;
  else
    Select v_i randomly from its category in D such that it maintains logical consistency:;
      $\forall v_j \in V,\ A[i, j] = 1$; `//` v_i `compatible with previous values`
  Add v_i to V;
Build final instances:;
  $CI \leftarrow S_0\ v_1\ S_1\ v_2\ S_2\ \ldots\ v_n\ S_n$;
  $PI \leftarrow S_{\text{PT}}(v_1, v_2, \ldots, v_n)$;
return $\langle CI, PI \rangle$;

to the constraints of the environment. For example, `set temperature` paired with `door lock`, or `unlock` paired with `thermostat`, would be prevented by A. This structured approach ensures semantic integrity while allowing for scalable and robust dataset generation. The complete replacement process is represented in Algorithm 4. During this step, each template pair was instantiated five times ($y = 5$) by applying Algorithm 4 iteratively. As a result, a total of 5312 instance pairs were generated.

8.6.3 Training and Test Set Construction

Once all the instance pairs have been generated, the dataset has been split into two subset, namely the training set and the test set.
To introduce diversity, the original template pairs were initially paraphrased $k = 5$ times (Step 1a), and then, for each template pair, placeholders were replaced $y = 5$ times (Step 2), thus obtaining the set of instance pairs I. Notably, the y instance pairs

derived from the same template pair differ only in the values used for substituting the placeholders. Therefore, the resulting y command instances share the same static part—the fixed string components in S. This limited diversity in phrasing can result in the *Text2Policy* model learning to rely on the static structure of the commands rather than understanding the underlying semantics. As a result, the model may perform well on commands that closely match those seen during training but fail when presented with commands conveying the same meaning but phrased differently.

To mitigate this, we introduced a second layer of paraphrasing at the CI level. This ensured that command instances with the same base structure but different placeholder values also varied in their phrasing, introducing additional linguistic diversity and reducing the model's reliance on static structures.

Similarly to Step 1a, in **Step 3a**, we enhanced the dataset by introducing paraphrased variations of command instances to improve linguistic diversity. For the training set ($80\% \times I$), each instance pair $\langle CI, PI \rangle$ was paraphrased once. In contrast, for the test set ($20\% \times I$), each instance pair was paraphrased five times (j in Figure 29), ensuring a more diverse evaluation set. Formally, the paraphrased instance pairs are defined as:

$$\text{Var}(\langle CI, PI \rangle) = \{\langle CI, PI \rangle, \langle CI_i, PI \rangle \mid i = 1, \ldots, j\},$$

where $j = 1$ for the training set and $j = 5$ for the test set.

This process resulted in 4250 instance pairs for training and 5310 for testing.

In **Step 3b**, to ensure semantic equivalence between the original and paraphrased instance pairs, we applied the same validation process as in Step 1b. Specifically, all paraphrases in both the training and test sets were evaluated using the jury-based PoLL approach, as detailed in Section 8.6.1. Through this process, only paraphrases deemed semantically valid were retained, formally captured by $\text{Validated}(\text{Var}(\langle CI, PI \rangle))$, and applied to each instance pair in I. After filtering, a fraction α (β) of instance pairs is discarded from the training (test) set. The final dataset is composed of 4096 training instance pairs and 4681 test instance pairs.

8.7 Experimental Analysis

This section evaluates the performance of the *Text2Policy* model by assessing its *accuracy*, *robustness*, *ability to generalize*, and *practical usefulness*. As the *Text2Policy* model, we trained two LLMs, LLaMA8b [207] and Mistral7b [152], for three epochs using Supervised Fine-Tuning (SFT) [211]. Training was performed on an A100 GPU with 40GB of memory. For training and inference, we used the

`unsloth` [212] library, capping memory usage at 70% for LLaMA8b and 75% for Mistral7b. Both models were 4-bit quantized [213], ensuring efficient inference even on resource-constrained devices. Since our goal is to support deployment on resource-constrained devices, inference time is not a primary concern. Whether the models take a few seconds or a few minutes [214] to generate policies is inconsequential, as we prioritize the quality of the generated JSON policies. Therefore, inference can also be performed on a CPU without significant drawbacks. To ensure a comprehensive evaluation, we conducted four key experiments, each designed to analyze a specific aspect of converting natural language commands into U-XACML policies:

1. **Baseline Performance Evaluation:** We tested both LLaMA8b and Mistral7b on the *test dataset*, generated as described in Section 8.5.

2. **Robustness Analysis:** The models' ability to handle noisy input was assessed using a *mistaken dataset* [215], created by introducing typographical errors into the test dataset.

3. **Generalization Test:** The models' adaptability was evaluated by applying them to a different domain, transitioning from a smart home environment to a *smart office* setting. This required constructing a new test dataset (the *smart office dataset*) with a custom taxonomy.

4. **Real-world Validation:** Model-generated policies were compared against expert-defined policies using a UCS to assess practical applicability.

The evaluation of the first three test suites relies on four standard metrics: *Accuracy*, *Precision*, *Recall*, and *F1 Score* [216, 217]. These metrics assess the correctness of JSON policies generated by our model (*model-generated policies*) by comparing them against JSON policies from the testing dataset (*reference policies*). For these tests, we use the *flattened version* of the JSON policies (see Section 8.4.1) and define the following terms:

- **True Positives (TP):** number of key-value pairs that are an *exact match* between the model-generated policy and the reference policy.

- **False Positives (FP):** number of key-value pairs missing from the reference policy but present in the model-generated policy (i.e., incorrectly added by the model).

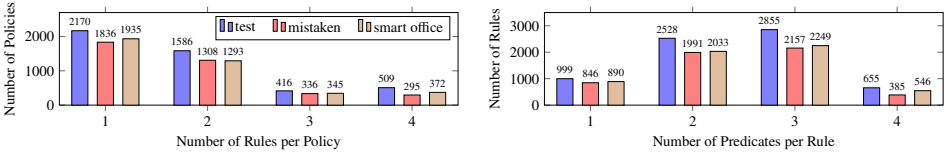
- **False Negatives (FN):** number of key-value pairs missing from the model-generated policy but present in the reference policy (i.e., incorrectly omitted by the model).

Since all keys are predefined in structured key-value extraction, True Negatives (TN) are not meaningful [218]. Including TN would inflate accuracy due to the large number of unused keys, providing a misleading evaluation of model performance. A key-value pair is considered *false* if it is not an exact match between the model-generated policy and the reference policy. This includes cases where the key is present but assigned a different value. Using these definitions, the evaluation metrics are formally defined as follows:

$$\text{Accuracy} = \frac{TP}{TP + FP + FN}, \qquad \text{Precision} = \frac{TP}{TP + FP},$$
$$\text{Recall} = \frac{TP}{TP + FN}, \qquad \text{F1 Score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}},$$

where *Accuracy* measures the overall correctness of model-generated policies, *Precision* indicates the proportion of correct key-value pairs among those included in the model-generated policies, *Recall* assesses the model's ability to capture all correct key-value pairs present in the reference policies, *F1 Score* provides a balanced measure of precision and recall.

Figure 31 provides an overview of the policies distribution based on the number of rules (Figure 31a) and the rules distribution based on the number of predicates (Figure 31b) across all three datasets (test, mistaken, and smart office). Figure 31a illustrates the distribution of policies based on the number of rules they contain in each dataset. For instance, in the *test dataset*, 2170 policies (46.4%) contain only one rule, 1586 (33.9%) have two rules, 416 (8.9%) have three rules, and 509 (10.8%) have four rules. Figure 31b presents the distribution of rules based on the number of predicates they contain. Most rules contain multiple predicates rather than a single one. In the *test dataset*, 999 rules contain only one predicate, while 2528 have two, 2855 have three, and 655 have four.



(a) Policies by rule count

(b) Rules by predicate count

Figure 31: Comparison of policies and rules across datasets

8.7.1 Evaluating the Translation of Natural Language Commands into U-XACML Policies

The first evaluation is performed with the *test dataset*, obtained through the dataset creation pipeline described in Section 8.5 for the smart home domain. The aim of this test is to establish a baseline for the model's ability to accurately translate unstructured natural language commands into valid U-XACML policies. As discussed, the evaluation metrics in the first three test suites are computed using the flattened version of JSON policies. However, we argue that this approach is equivalent to evaluating U-XACML directly. Since U-XACML policies are deterministically derived from the JSON policies, and the flattened version is an intermediate product of the conversion module (see Section 8.4), the results remain consistent across both representations.

Table 3a summarizes the results for both LLaMA8b and Mistral7b models in terms of accuracy, precision, recall, and F1 score across different rule counts. For reference policies with one or two rules, LLaMA8b consistently outperforms Mistral7b in all metrics, although both models exhibit strong results. LLaMA8b achieves nearly perfect accuracy and F1 scores when handling three or four rules.

Among the factors influencing performance, the number of rules within a policy consistently emerges as the parameter with the greatest impact on results, with a general trend of improved performance metrics as the number of rules increases, particularly for LLaMA8b.

Overall, LLaMA8b maintains a higher performance across all rule counts, with an overall accuracy of 92.86% compared to the overall accuracy of Mistral7b of 83.77%. This highlights LLaMA8b's better generalization and consistency in policy translation, whereas Mistral7b, despite performing well, shows more stability in accuracy and precision as complexity increases.

8.7.2 Testing The Performance and Robustness of Text2Policy on Noisy Commands

The objective of this experiment is to evaluate the model's performance when processing commands that contain errors, such as typos and formatting inconsistencies. Additionally, we aim to assess the model's *robustness*—its ability to generate correct U-XACML policies despite input perturbations. To this end, we introduce the *mistaken dataset*, which includes natural language commands intentionally altered with typographical and minor syntactic deviations. This setup simulates real-world scenarios, where users may unintentionally introduce mistakes. The robustness

is measured by comparing the model's performance on the *mistaken dataset* with its performance on the original, correctly formatted commands in the *test dataset*. A higher robustness score indicates that the model's performance remains stable despite the presence of input errors.

To create the *mistaken dataset*, we used LLaMA8b to insert typographical errors and minor syntactic changes into the test set of command instances described in Section 8.7.1. These changes were carefully designed to alter the textual representation while preserving the original semantics of the commands. The modifications do not affect the taxonomy values present in the commands. Instead, they are applied only to the remaining parts of the command, including auxiliary words, grammatical structures, and minor typographical variations.

After we introduced typographical errors in the *test dataset*, we applied the *PoLL* validation process (also utilized in Section 8.5), which eliminates any altered commands (along with their associated directives) that, with the inserted mistakes, no longer align with their intended meaning, according to the jury of LLMs. This process yielded a final dataset, the *mistaken dataset*, comprising 3775 instance pairs. In our threat model, the entities responsible for defining security policies are trusted users, such as the homeowner in a residential setting or a department safety & cybersecurity officer of a company. These users are not expected to input adversarial or malicious commands. As such, our performance evaluation focuses on realistic, unintentional input variations—such as typos and ambiguous phrasing—rather than on defending against deliberate harmful inputs. Table 3b presents the performance metrics for both models when evaluated on the *mistaken dataset*. LLaMA8b achieves an overall accuracy of 91.39% , while Mistral7b reaches an overall accuracy of 81.69%. For overall precision, LLaMA8b attains 93.97% compared to 88.35% for Mistral7b, whereas overall recall values are 95.01% and 90.23%, respectively. Both models show better performance as the number of rules increases. For policies with three or more rules, LLaMA8b maintains high precision and recall, reflecting its ability to generate accurate U-XACML policies despite input perturbations. Mistral7b is still effective as well, this suggests that Mistral7b also can handle the noise in the input text. To quantify the *robustness*, we compute the relative variation in all evaluation metrics (accuracy, precision, recall, and F1 score) between the *test dataset* and the *mistaken dataset*. We define the *robustness score* R_s^M for a given metric M as follows:

$$R_s^M = 1 - \frac{|M_{test} - M_{mistaken}|}{M_{test}},$$

where M_{test} is the metric value obtained on the *test dataset*, and $M_{mistaken}$ is

the corresponding value on the *mistaken dataset*. A robustness score closer to 1 indicates minimal degradation in performance due to noise. The last row of Table 3b presents the results for the robustness score. LLaMA8b achieves robustness scores of 0.9842 for accuracy, 0.9883 for precision, 0.9906 for recall, and 0.9902 for F1 score. In comparison, Mistral7b attains 0.9752 for accuracy, 0.9836 for precision, 0.9944 for recall, and 0.9841 for F1 score. These values confirm that LLaMA8b consistently retains higher performance across all metrics, showing its better robustness to noisy inputs.

# Rules	Total Elements	Accuracy		Precision		Recall		F1 Score	
		LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b
1	2170	0.9120	0.8665	0.9418	0.9082	0.9500	0.9124	0.9471	0.9106
2	1586	0.9123	0.8398	0.9365	0.9141	0.9487	0.9012	0.9471	0.8960
3	416	0.9865	0.9607	0.9872	0.9852	0.9901	0.9634	0.9931	0.9736
4	509	0.9966	0.9350	0.9988	0.9841	0.9976	0.9481	0.9982	0.9615
Overall	4681	0.9286	0.8377	0.9508	0.8982	0.9591	0.9074	0.9572	0.8915

(a) Performance of *Text2Policy* on the *test dataset*.

# Rules	Total Elements	Accuracy		Precision		Recall		F1 Score	
		LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b
1	1836	0.8974	0.8604	0.9291	0.9022	0.9403	0.9125	0.9377	0.9051
2	1308	0.9080	0.8365	0.9342	0.9103	0.9470	0.8956	0.9439	0.8932
3	336	0.9631	0.9512	0.9700	0.9763	0.9810	0.9654	0.9791	0.9677
4	295	0.9872	0.9371	0.9950	0.9920	0.9935	0.9537	0.9925	0.9656
Overall	3775	0.9139	0.8169	0.9397	0.8835	0.9501	0.9023	0.9478	0.8773
R_s^M	3775	0.9842	0.9752	0.9883	0.9836	0.9906	0.9944	0.9902	0.9841

(b) Performance on the *mistaken dataset* (noisy inputs). Last row: robustness score R_s^M.

# Rules	Total Elements	Accuracy		Precision		Recall		F1 Score	
		LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b
1	1935	0.8445	0.7900	0.8885	0.8306	0.9004	0.8325	0.8969	0.8494
2	1293	0.8367	0.7971	0.8875	0.8754	0.8923	0.8542	0.8861	0.8650
3	345	0.9624	0.9381	0.9656	0.9615	0.9704	0.9541	0.9722	0.9562
4	372	0.9822	0.9193	0.9937	0.9662	0.9920	0.9342	0.9897	0.9463
Overall	3945	0.8657	0.7292	0.9057	0.8064	0.9202	0.8421	0.9096	0.8101

(c) Performance on the *smart office dataset*.

# Rules	Total Elements	Accuracy		Precision		Recall		F1 Score	
		LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b	LLaMA8b	Mistral7b
1	1912	0.7855	0.7304	0.8563	0.7945	0.8687	0.8411	0.8612	0.8105
2	1173	0.8043	0.7584	0.8764	0.8319	0.8890	0.8620	0.8804	0.8422
3	227	0.8694	0.8668	0.9166	0.8988	0.9318	0.9329	0.9231	0.9115
4	531	0.8883	0.8585	0.9428	0.9184	0.9328	0.9060	0.9361	0.9108
Overall	3843	0.8104	0.7647	0.8780	0.8292	0.8875	0.8619	0.8811	0.8400

(d) Performance on the *HRE dataset*.

Table 17: Performance comparison of LLaMA8b and Mistral7b models across different rule counts and scenarios. Evaluation metrics (accuracy, precision, recall, F1 score) are presented for both models on the *test*, *mistaken*, *smart office* and *HRE* datasets.

Performance Across Error Impact Intervals Table 3b presents the overall performance results of the LLaMA8b and Mistral7b models on the *mistaken dataset*, which consists of command instances exhibiting varying degrees of errors—ranging from minor deviations to significant input corruption. This section explores and analyzes the models' performance across these different levels of input errors within the *mistaken dataset*. To quantify the impact of these errors, we introduce the concept of *Error Impact*, which is defined using the *Levenshtein Distance* [219], denoted by d_L, as described in by the following equation:

$$\text{Error Impact} = 100 \cdot \frac{d_L(\texttt{original}, \texttt{mistaken})}{\max\left(|\texttt{original}|, |\texttt{mistaken}|\right)}.$$

In this equation, the `original` input refers to the correctly formatted commands from the *test dataset* (introduced in Section 8.7.1). The `mistaken` input represents the same command after intentional modifications to introduce errors.

The Levenshtein Distance measures how different the mistaken command is from its original counterpart by counting the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one into the other. The formula then normalizes this value by dividing it by the length of the longer command, scaling it to a percentage.

While Table 3b aggregates performance across all error levels, we further examine how accuracy, precision, recall, and F1 score vary with different degrees of input errors. Specifically, we divide the dataset into error impact intervals, such as 10–20, 20–30, up to 70+, and analyze the models' performance within these intervals. Figure 32 illustrates this breakdown, with the x-axis showing the error impact intervals, while the y-axis displays the evaluated metric values.

LLaMA8b consistently outperforms Mistral7b in accuracy and precision across all error intervals, with the gap widening as errors increase. In the 10–20 error interval, LLaMA8b achieves 96.2% accuracy vs. Mistral7b's 90.6%, while in the 70+ range, it maintains 85.5% vs. 77.3%. For recall and F1 score, a similar trend emerges. In the 10–20 interval, LLaMA8b attains 98.7% recall and 97.9% F1 score, surpassing Mistral7b's 92.9% and 94.2%. This highlights LLaMA8b's robustness, particularly under high-error conditions.

These results demonstrate that the proposed method achieves high accuracy and robustness and maintains them even under increasing input corruption, highlighting its reliability in processing noisy real-world user commands.

Overall, both models perform well with minimal errors, but LLaMA8b maintains stronger robustness as error impact increases, reinforcing the importance of robustness in real-world noisy data handling.

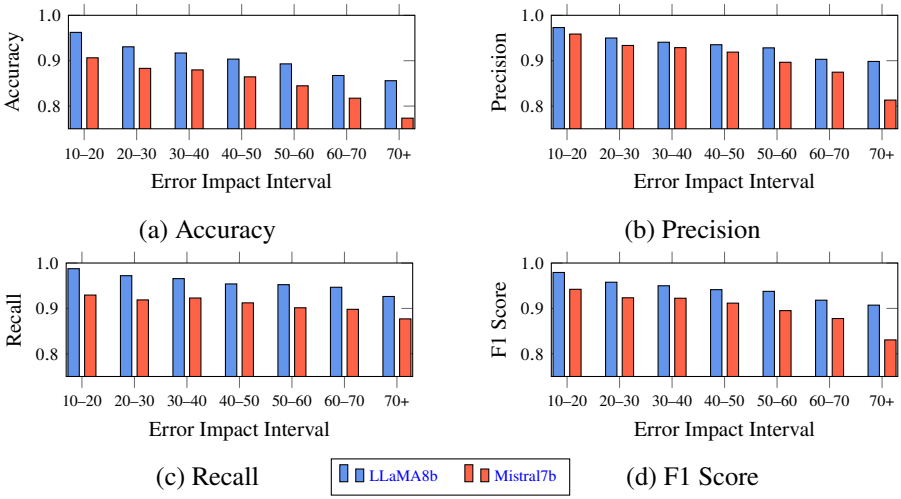


Figure 32: Performance metrics for LLaMA8b and Mistral7b evaluated across different error impact intervals in the *mistaken dataset*.

8.7.3 Testing the Generalization Capabilities of Text2Policy in Smart Office and Healthcare Residence for the Elderly Environments

To evaluate the model's ability to generalize across different domains, we conduct a similar domain adaptation experiment in which the test set is generated using the *smart office taxonomy* instead of the smart home taxonomy. Although the smart office domain introduces different entities and attributes, it remains conceptually similar.

To further assess the model's generalization, we conduct an additional domain adaptation experiment using the *Healthcare Residence for the Elderly (HRE) taxonomy*. Unlike the smart home or smart office domains, the HRE setting introduces a diverse set of attribute values specific to institutional healthcare environments. This domain is characterized by subject profiles, medical devices, and related actions significantly diverging from the those of the original smart home dataset, which the model was trained on.

This setup allows us to assess the model's capability to transfer learned knowledge from one domain to another without additional fine-tuning. For each of the two domains, we perform a dataset creation workflow that follows a pipeline similar to that of the smart home dataset presented in Section 8.6. Specifically, it shares Steps 1a and 1b of Figure 29, leading directly to Step 2, where the same set of

template pairs T from the smart home scenario is reused. In this step, placeholders within these templates are replaced with values from the specific taxonomy under consideration (i.e., either the smart office or the HRE taxonomy), generating instance pairs specific to the respective domain.
After placeholders are replaced—unlike in the smart home scenario—all generated instance pairs are allocated to the *Raw test set* (see Figure 29), since no training set is needed to train or re-train our *Text2Policy* model.
Finally, to enhance variability and ensure semantic correctness of all instance pairs, Steps 3a and 3b are performed.
The resulting two test sets—the *smart office dataset*, consisting of 3945 instance pairs, and *the HRE dataset*, consisting of 3843 instance pairs—serve as the basis for evaluating the model's domain adaptation capability.
Table 3c highlights the performance of LLaMA8b and Mistral7b in the smart office domain across various metrics, showcasing LLaMA8b's superior ability to generalize and adapt. In this domain, LLaMA8b achieves an overall accuracy of 86.57%, precision of 90.57%, recall of 92.02%, and F1 score of 90.96%, consistently outperforming Mistral7b, which records 72.92%, 80.64%, 84.21%, and 81.01% for the same metrics.
Table 3d presents the models' performance in the HRE domain across various metrics, where LLaMA8b similarly excels, achieving an overall accuracy of 81.04%, precision of 87.80%, recall of 88.75%, and F1 score of 88.11%, compared to Mistral7b's 76.47%, 82.92%, 86.19%, and 84.00%.
This evaluation confirms the method's accuracy and its ability to generalize across related domains with different taxonomies.
By comparing results for the models, we observe that the performance gap between them widens as the policy complexity increases. For example, in the smart office scenario, for policies with one or two rules, LLaMA8b demonstrates strong performance with F1 scores of 89.69% and 88.61%, while Mistral7b achieves 84.94% and 86.50%, respectively. In more complex cases, such as policies with four rules, LLaMA8b achieves high metrics, including an F1 score of 98.97%, compared to Mistral7b's 94.63%. These results underscore LLaMA8b's robustness in handling complex domain-specific rules.
The good performance across both simple and complex policies in new domains further confirms the method's robustness and its capacity for zero-shot generalization. However, the performance of testing the model on the HRE domain is noticeably lower than in the smart home and office domains. This degradation occurs because HRE introduces distinct device types, contextual attributes, and user roles that differ significantly from those seen during training.

8.7.4 Validation of Model-Generated Policies Against Expert-Defined Policies

Ensuring that a policy generated by our model accurately reflects the intended authorization rules specified in a natural language command is essential for its reliability. To assess the correctness of a model-generated U-XACML policy, we evaluate whether it produces the same authorization decisions as an expert-written U-XACML policy for the same command.

Our evaluation process involves systematically comparing policy pairs—one generated by the model and one written by an expert—by executing a predefined set of UCON requests and analyzing the authorization decisions returned by the UCS[20], specifically the PDP. A model-generated policy P_m and an expert-defined policy P_e are considered *equivalent* within R if they produce identical authorization decisions for every UCON request in the given set R. Formally:

$$P_m \equiv P_e \quad \Leftrightarrow \quad \forall r \in R, \;\; \mathrm{UCS}(P_m, r) = \mathrm{UCS}(P_e, r).$$

If P_m is not fully equivalent to P_e, we assess their degree of agreement using the *decision agreement rate* (DAR). The DAR represents the percentage of UCON requests for which the model-generated policy produces the same authorization decision as the expert-defined policy when evaluated by the UCS. It is calculated as:

$$\mathrm{DAR}(P_m, P_e, R) = \frac{|\{r \in R \mid \mathrm{UCS}(P_m, r) = \mathrm{UCS}(P_e, r)\}|}{|R|}.$$

This metric provides a clear method for evaluating the model's alignment with expert-defined reference policies, which act as the authoritative standard or "*ground truth*" for authorization decisions.

Experimental Setup and UCON Request Generation This set of experiments involves 31 policies defined by experts in two forms: a natural language description and its corresponding U-XACML representation (denoted as P_e).

These policies were carefully selected to cover a broad range of logical structures, including simple conditions as well as nested expressions combining AND and OR operators.

Each expert-defined policy is evaluated against a set of systematically generated UCON requests.

The generation of UCON requests is based on a two-step process: attribute extraction and systematic value expansion. This process establishes the foundation

[20] https://sssg-dev.iit.cnr.it/marco-rasori/new-ucs

for generating UCON requests by identifying relevant attributes and defining a set of possible values for each. Essentially, it determines the domain within which the UCON requests are constructed.

Attribute extraction is an automated process that, given a policy P_e, identifies and extracts all `attributeId`s. For each attribute $\texttt{attributeId}_i$, it retrieves the set V_i of its associated values, i.e., the values used to define the constraints on that attribute in the policy P_e. We refer to this the set of pairs $\langle \texttt{attributeId}_i, V_i \rangle$ as the *base attribute-value set* S. Since this step only retrieves the specific values present in the policy, a *systematic value expansion* step is applied to improve coverage. For each attribute $\texttt{attributeId}_i$ identified in the previous step, the set of attribute values V_i is manually expanded by including additional, meaningful alternatives, thus resulting in the expanded set V_i^x. We call this expanded set of pairs $\langle \texttt{attributeId}_i, V_i^x \rangle$ the *extended attribute-value set* S^x.

The objective is to ensure that the set of UCON requests we generate largely explores the complexity of each policy, covering a diverse set of realistic access scenarios. During systematic value expansion, the additional values are carefully selected to be meaningful within the context of the expert-defined policy. For instance, if a policy includes a predicate involving the attribute *num* such as "*num greater than or equal to 1*", the automated extraction initially retrieves only the value *1*. To improve coverage, the set is manually expanded with values such as *0* (to test boundary conditions), *2* (to verify standard compliance), and `MAX_VALUE / MIN_VALUE` (to examine domain limits). Similarly, if a policy contains a boolean predicate involving the attribute *isInternal* like "*isInternal equal to true*", the complementary value, *false*, is added to ensure both possible states are tested. Once the attributes and their expanded values have been determined, the set of UCON requests R^x is generated by systematically combining these values in all possible ways. As a result, the generated UCON requests effectively test the policy's decision logic, thereby enhancing the reliability of the evaluation compared to having the set of UCON requests built from the base attribute-value set S.

Each expert-defined policy P_e is then tested against the set of UCON requests R^x generated for that policy during the previous step, having the UCS as policy enforcement engine. The testing follows a complete UCON workflow (see Section 8.1), which involves sending a *tryAccess* message to initiate the access evaluation. If the response from the UCS is `Permit`, a *startAccess* message is issued, and if the response to this second request is also `Permit`, an *endAccess* message is sent. If at any point a `Deny`, `Indeterminate`, or `NotApplicable` response is received, the workflow is halted. All responses from the UCS are recorded for subsequent analysis.

After evaluating the expert-defined policies, which serve as the ground truth, we assess the accuracy of our model in translating the natural language policy descriptions into U-XACML policies. The same set of 31 policy descriptions is provided as input to the model, which generates corresponding policies, denoted as P_m, and referred to as model-generated policies. While the model's output may differ syntactically from the expert-defined policies, the key evaluation criterion is whether they produce the same authorization decisions. To verify this, the model-generated policies undergo the same test process as the expert-defined ones, using the same set of UCON requests R^x and following the identical UCON workflow. Finally, the responses obtained with the expert-defined policies and the model-generated policies are compared.

Evaluation Results Table 18 shows the decision agreement rate $\mathrm{DAR}(P_m, P_e, R^x)$, obtained for each policy by both LLaMA8b and Mistral7b models. For LLaMA8b, nearly all policies (28 out of 31) achieve a perfect DAR of 1.0000, meaning that the model-generated policy P_m is equivalent to the expert-defined policy P_e, i.e., all $|R^x|$ evaluations produced the same authorization decision for both P_m and P_e. This high consistency indicates that LLaMA8b is particularly effective at capturing the intended authorization logic across a broad range of policies.
Its ability to produce semantically equivalent policies across varied inputs further supports the accuracy and the robustness of the proposed method in faithfully interpreting natural language commands.
In contrast, Mistral7b achieves equivalence for fewer policies (less than half), with several policies exhibiting high DAR, albeit below 1.0000. This indicates that, while there are some discrepancies, the model-generated mimics the expert-defined policy's logic well for most cases. As expected, LLaMA8b outperforms Mistral7b, achieving equivalence for more policies. The larger number of parameters of LLaMA8b likely contributes to better generalization and a stronger ability to capture complex patterns in policy translation [70, 220]. Additionally, LLaMA8b's pre-training data or tokenizer may be better aligned with the syntactic structures of the JSON format, further enhancing its performance. The overall DAR values (0.9838 for LLaMA8b and 0.9232 for Mistral7b) highlight the strong performance of both models, with LLaMA8b outperforming Mistral7b in terms of equivalence.

Table 18: Decision agreement rate for LLaMA8b and Mistral7b.

Pol.#	$\lvert R^x \rvert$	DAR(P_m, P_e, R^X) LLaMA	Mistral
1	64	1.0000	1.0000
2	128	1.0000	1.0000
3	640	1.0000	1.0000
4	72	1.0000	1.0000
5	24	1.0000	1.0000
6	12	1.0000	1.0000
7	20	1.0000	1.0000
8	8	1.0000	0.8750
9	8	1.0000	0.8750
10	60	1.0000	0.9166
11	60	1.0000	1.0000
12	16	1.0000	1.0000
13	16	1.0000	1.0000
14	12	1.0000	1.0000
15	120	1.0000	0.8583
⋮	⋮	⋮	⋮
16	36	1.0000	0.9444
17	4	1.0000	0.2500
18	6	1.0000	0.3333
19	42	1.0000	0.8333
20	32	1.0000	0.9062
21	16	1.0000	1.0000
22	16	1.0000	1.0000
23	10	1.0000	1.0000
24	28	1.0000	1.0000
25	4	1.0000	0.5000
26	24	1.0000	0.7083
27	16	0.8125	0.2500
28	60	1.0000	0.6833
29	20	0.0000	0.4500
30	120	1.0000	0.7500
31	140	0.9500	0.8714
Total			
	1848	0.9838	0.9232

8.7.5 Testing On-Device Performance

To assess the practical feasibility of running the *Text2Policy* model on resource-constrained hardware, we evaluated its inference performance on two representative devices: a Raspberry Pi 5 and a Jetson Orin AGX development kit configured to emulate a Jetson Orin Nano. In the latter case, both the available RAM and number of CPU cores were limited to match the Nano's specifications (8 gigabyte RAM, 6 CPU cores), and both GPU and CPU frequencies were capped to reflect the Nano's lower performance profile.

While earlier sections evaluated both LLaMA8b and Mistral8b, we focus here on LLaMA8b due to its superior accuracy and generalization performance. As previously described, the model was quantized to 4-bit precision to reduce its memory requirements and enable efficient execution on constrained hardware.

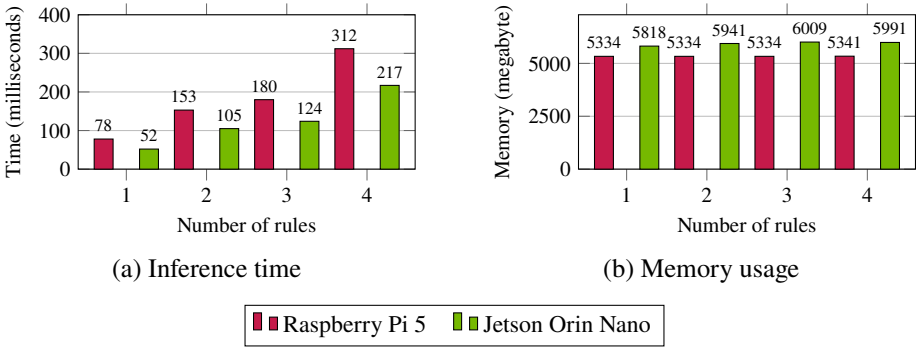
Inference on Raspberry Pi 5, which does not feature a GPU, was made possible using the Llama.cpp[21] framework. The model was first converted to the GGUF format and quantized to the q4_K_M version to reduce memory and computational requirements, then run entirely on CPU.

[21] https://github.com/ggml-org/llama.cpp

The same quantized GGUF model was also deployed on the Jetson Orin Nano emulator, which features a constrained GPU. In this case, the entire model was loaded into the GPU memory, leveraging Llama.cpp's GPU backend for faster inference while still staying within the tight hardware limits of the device. This setup enabled inference within the constraints of the device's available memory and processing power.

Figure 33 shows the inference time and the memory usage across varying number of rules in the policy. Each bar corresponds to the average inference result obtained from evaluating 30 distinct policies. Specifically, we tested 30 policies with one rule, 30 policies with two rules, and so on, up to four rules, in order to assess how performance scales with policy complexity.

Figure 33a reports the inference time and shows that the Jetson Orin Nano emulator consistently achieves lower inference times than the Raspberry Pi 5, ranging from 52 milliseconds to 217 milliseconds compared to 78 milliseconds to 311 milliseconds, respectively. Although inference time on the Raspberry Pi is higher, it remains acceptable given that the policy generation is not a real-time task. Memory usage, shown in Figure 33b, remains largely constant regardless of the rule count. The Raspberry Pi maintains a usage of around 5.3 gigabyte, while the Jetson Orin Nano emulator shows slightly higher usage, reaching just over 6 gigabyte. Memory consumption is therefore dominated by model loading rather than policy complexity.



(a) Inference time

(b) Memory usage

Figure 33: Inference time and memory usage of the LLaMA8b-based *Text2Policy* model on Raspberry Pi 5 and Jetson Orin Nano emulator, measured for policies containing one to four rules.

We also measured the energy consumption of the Raspberry Pi 5 during inference. Results indicate a modest increase in energy use as the number of rules in the policy

grows. Specifically, the average instantaneous current remains relatively stable during the inference process (around 1670 mA), while average energy consumption required for executing the inference process of one policy increases from 38 mAh for one rule to 153 mAh for four rules. This indicates that the rise in energy consumption with more complex policies is primarily due to longer inference durations, as the current draw remains nearly constant.
Among various factors influencing performance, the number of rules within a policy consistently emerges as the parameter with the greatest impact on results, highlighting its critical role in performance variation, especially in the time required to generate the policy (inference time), and hence in the energy consumption. These results indicate that the *Text2Policy* model can operate within the constraints of limited hardware resources, while still maintaining reasonable inference performance.

8.8 Conclusions

This chapter introduced a domain-specific module that converts natural language commands into enforceable U-XACML usage control policies. The proposed *Text2Policy* framework connects human intentions with machine-readable rules by first turning free-form text into a compact JSON structure and then translating it into U-XACML in a consistent way. This process reduces complexity, lowers computation costs, and stays compatible with standard PDP/PEP systems.
Extensive testing confirmed that the approach works effectively. On in-domain data, the best setup (LLaMA-8B, 4-bit) reached 93% accuracy and an F1 score of 96%. When tested with typos and grammatical errors, the system showed only small drops in performance, proving it can handle noisy input. Cross-domain tests also showed strong zero-shot generalization from the *smart-home* domain to *smart-office* scenarios, even without additional fine-tuning.
A comparison with expert-written policies showed 98% agreement in final decisions, confirming the reliability of the generated rules. Tests on local devices further proved that the quantized model can run efficiently within the memory and speed limits typical of edge environments.
Overall, the results show that small, instruction-tuned models can accurately and securely generate policies directly on devices.

9 Unmasking Model Behavior: How LLMs reason on Vulnerability Detection

The third part of this research under **Q3**, discussed in this chapter, focuses on the problem of vulnerability detection. Unlike other applications, this area requires not only accurate detection but also reliable reasoning. Large language models (LLMs) have shown promise in analyzing source code, but their behavior is still inconsistent. They often overpredict vulnerabilities, producing many false alarms, and at the same time miss real security flaws hidden in code that looks safe. Even when they give explanations, these are often incomplete, inconsistent, or only loosely related to the final prediction. This mix of errors and weak reasoning reduces trust in the model's output, making it difficult to use in real security workflows where both accuracy and interpretability are essential.

Earlier methods tried to reduce these problems. Supervised fine-tuning improved detection rates but often led to overfitting and poor generalization. Retrieval-augmented systems helped verify predictions, but added latency and relied heavily on the quality of the retrieved data. What was missing was a structured way to improve the internal reasoning of the model, not just its answers, but how it thinks about vulnerabilities.

This study uses reinforcement learning, specifically Group Relative Policy Optimization (GRPO), to address this gap. Unlike standard fine-tuning (SFT), GRPO allows models to learn from structured feedback that rewards outputs which are not only correct but also logical and easy to interpret. Carefully designed reward functions guide the model to improve both its detection accuracy and the quality of its reasoning, promoting consistent and reliable behavior across datasets.

The main contributions of this stage are as follows: It provides one of the first detailed studies of how LLMs reason during vulnerability detection, identifying their strengths and weaknesses. It introduces GRPO as a method for shaping model behavior beyond standard fine-tuning. It shows how structured reward functions can improve accuracy, reasoning stability, and robustness at the same time (C18). It lays the groundwork for behavior-aware training pipelines, moving toward more trustworthy and interpretable LLMs for software security.

Together with the misinformation and policy-generation modules, this work completes the set of task-specific extensions developed under Q3. Each module addresses a different operational challenge: resisting adversarial content, automating access control, and improving the reliability of vulnerability detection. Overall, this chapter advances the field of vulnerability analysis and supports the broader goal of this

thesis, extending large language models with domain-specific capabilities that make them accurate, practical, and trustworthy tools for real-world cybersecurity.

9.1 Background

This section provides a brief overview of the core concepts underlying our work: the challenges associated with automatic vulnerability detection and the GRPO algorithm used to guide model behaviour through structured reward feedback.

9.1.1 Software Vulnerability Detection: Concepts and Challenges.

Automated software vulnerability detection seeks to uncover flaws, such as buffer overflows, injection points, or logic errors, that attackers could exploit to undermine a system's integrity, confidentiality, or availability. As codebases grow in size and complexity, manual code review becomes impractical, and even mature tools struggle to keep pace with new language features and idioms.

Traditional static analysis methods inspect code without executing it, using heuristics or formal rules to flag suspicious constructs. They excel at catching simple patterns but often generate high rates of false positives and miss context-sensitive flaws. Dynamic analysis (fuzzing) executes programs on crafted inputs to observe failures at runtime; while powerful for memory errors, it can fail to reach deep code paths or reason about semantic bugs [221] [222].

More recently, large language models (LLMs) pretrained on massive code repositories have shown strong zero- and few-shot ability to recognize vulnerable patterns with minimal feature engineering. Despite their syntactic prowess, LLM-based detectors generally make isolated, line by line predictions and do not provide a structured way to follow a program's control or data flow in sequence [57].

These gaps, such as false positives, incomplete path coverage, and the need for strategic, context-sensitive inspections, highlight the importance of integrating reinforcement learning. By training models to navigate code structures and prioritize genuinely vulnerable regions, this approach can more efficiently identify actual security weaknesses.

9.1.2 Group Relative Policy Optimization (GRPO)

Group Relative Policy Optimization (GRPO) is a reinforcement learning algorithm proposed by DeepSeek to improve the stability and efficiency of policy updates without relying on separate value function approximators [112] [113], as required

in traditional methods like Proximal Policy Optimization (PPO) [223]. GRPO introduces a novel approach by leveraging multiple sampled outputs, generated in response to the same input prompt, to estimate rewards comparatively. This group-based design aligns well with modern reward models, which are often trained on preference data comparing candidate responses. Originally developed and evaluated in the context of complex reasoning tasks such as mathematical problem solving, GRPO demonstrates strong performance in optimizing large language models (LLMs) using both outcome-level and step-level supervision. In this paper, we explore the potential of GRPO in a new application domain: vulnerability detection, where an LLM must navigate and assess code structures effectively to identify security flaws.

9.2 Behavioral Analysis

This section presents a behavioural study of small language models when applied to vulnerability detection tasks. We aim to assess whether these models exhibit systematic tendencies, such as risk aversion, overgeneralization, or sensitivity to noise, when reasoning about code security. To address this objective, we first outline the datasets employed to analyze model behavior, then assess predictive patterns and reward shaping strategies implemented to direct such behaviors during the fine-tuning process.

9.2.1 Vulnerability Datasets

In our methodology we leverage three vulnerability datasets: *DiverseVul*, *BigVul* and *CleanVul*, each selected for its particular strengths in model training and evaluation:

1. *DiverseVul* comprises 18.945 vulnerable functions (spanning 150 CWEs) and 330.492 non-vulnerable functions extracted from 7.514 commits across hundreds of projects. Its high degree of manual curation and broad project coverage yields label precision, making it well suited for model fine-tuning when minimizing false positives is critical [224].

2. *BigVul* gathers a large-scale C/C++ vulnerability corpus by crawling the public CVE database and related GitHub repositories. Although its reliance on automatically matched CVE descriptions introduces noise, its sheer volume allows assessment of a model's ability to learn robust feature representations from less-clean data [225].

3. *CleanVul* is a multi-language collection of 8.203 functions selected via an LLM-based heuristic pipeline (90.6% overall correctness). It encompasses Java, Python, JavaScript, C#, C and C++, and was designed primarily to evaluate model generalization across programming languages. Given its llm-based nature it is not possible to connect each code snippet with a specific CWE [226].

Table 19a presents a comparative summary of CWE and project distributions between *BigVul* and *DiverseVul*. While both datasets were utilized in Subsection 9.2.3 for training models (Phi4 and LLaMA 8b, respectively), *DiverseVul* demonstrates broader coverage, encompassing more unique CWEs and a higher number of projects than *BigVul*. Its lower concentration in the top 5 CWEs and projects further underscores its greater diversity. This is followed by Table 19b, which details the language distribution in *CleanVul*. The dataset is dominated by C, reflecting a pronounced focus on C/C++ based ecosystems, with supplementary multilingual representation enhancing its scope for cross-language vulnerability analysis.

Table 19: Dataset characteristics: CWE/project coverage and language distribution.

(a) CWE and project distribution in BigVul and DiverseVul

	BigVul		**DiverseVul**	
Metric	**Train**	**Test**	**Train**	**Test**
Unique CWEs	90	85	150	145
Top 5 CWEs (% cov.)	55%	55%	39%	39%
Unique proj.	309	289	799	741
Top 5 proj. (% cov.)	73%	71%	33%	32%

(b) Language distribution in CleanVul

Language	**Distribution**
C	47.18%
Java	19.53%
JavaScript	15.32%
Python	15.24%
C#	1.77%

9.2.2 Behavioural Evaluation of Small LLMs for Vulnerability Detection

In this study, we evaluate the efficacy of small-scale language models in distinguishing code as either *Vulnerable* or *Non Vulnerable* through binary classification, with an emphasis on analyzing their behavioral patterns during this process. We consider two representative architectures: `Phi4` (~13B parameters), trained **Bigvul** dataset, and `LLaMA 8B` model trained on the more diverse and heterogeneous **DiverseVul** benchmark.

Experimental Setup. All models were evaluated in a zero-shot classification setting using the prompt shown in Listing 1 and Listing 2. The model was given

a fixed system prompt describing the behaviour analysis task and a user prompt containing the target code snippet.
The output labels, enclosed within <answer> and </answer>, were compared with the ground-truth labels. We evaluate the model's performance using standard binary classification metrics derived from the confusion matrix. Given two classes, *Vulnerable* and *Not Vulnerable*, the confusion matrix includes *true positives* (TP), *false positives* (FP), *true negatives* (TN), and *false negatives* (FN) and given their values we compute the following metrics:

$$\texttt{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}, \quad \texttt{Precision} = \frac{TP}{TP + FP} \tag{8}$$

$$\texttt{Recall} = \frac{TP}{TP + FN}, \quad \texttt{F1-Score} = \frac{2 \times TP}{2 \times TP + FP + FN} \tag{9}$$

```
You are a software security expert. I will provide you with code snippets. Your task
     is to analyze the code and determine whether it contains any vulnerabilities.

<reasoning>
1. Summarize the code snippet.
2. Check thoroughly for vulnerabilities.
3. Clearly state whether the code is secure or insecure.
</reasoning>

<answer>
Yes, the code is vulnerable.
OR
No, the code is not vulnerable.
</answer>
```

Listing 1: System prompt provided to the model

```
Analyze the following code snippet to determine whether it is vulnerable.

{CODE SNIPPET}
```

Listing 2: User prompt provided to the model

Additionally, macro average was computed. The *macro average* calculates the unweighted mean of the metric across all classes, treating each class equally. The complete set of results is reported in Table 20. Below, we summarize the key findings based on these results:

1. *Vulnerable code: many true positives, many false alarms.* `Phi4` display strong recall for the *vulnerable* class on **CleanVul** (0.916) and **BigVul** (0.835), but

this comes at the cost of very low precision, especially on **BigVul** (0.054). The low F_1 scores for the vulnerable class (0.101 on BigVul) confirm that the models tend to overpredict insecurity, resulting in a large number of false alarms. This pattern is reversed in **DiverseVul**, where `LLaMA 8B` shows slightly better precision (0.437) but much lower recall (0.258), suggesting increased caution.

2. *Overconfidence in predicting non-vulnerable code.* For Phi4, precision for the *non-vulnerable* class is extremely high across all datasets (up to 0.960 on **BigVul**), but recall remains low, especially for **CleanVul** (0.176). This indicates that model is reluctant to confidently classify code as safe, even when it is, possibly due to an overcautious treatment of benign patterns, leading to safe examples being flagged unnecessarily.

3. *Degradation under distribution shift and class imbalance.* Moving from **CleanVul** dataset to noisier or more diverse benchmarks like **BigVul** and the heavily imbalanced **DiverseVul** (~18 945 vulnerable vs. 330 492 non-vulnerable functions) leads to a marked drop in performance. For `Phi4`, macro F_1 drops from 0.417 to 0.224, underscoring its vulnerability to label noise and project drift. `LLaMA 8B` attains a macro F_1 of 0.430 on **DiverseVul**, but this figure is inflated by the dominance of the non-vulnerable class ($F_{1\,\text{non-vuln}} = 0.535$). Its recall for the vulnerable class remains low (0.258), indicating that true robustness in identifying security flaws under severe class imbalance has yet to be achieved.

4. *Shallow decision heuristics.* The precision recall gaps observed on **BigVul** reveal that `Phi4` is guided by very coarse cues: for the *vulnerable* class, precision drops to 0.054 while recall rises to 0.835; conversely, for the *non-vulnerable* class, precision peaks at 0.960 but recall drops to 0.212. This pattern indicates that the model fires on almost any surface indicator of vulnerability yet hesitates to mark code as safe unless it is unmistakably benign. A milder but similar mismatch is visible in `LLaMA 8B` on **DiverseVul** (precision 0.437 vs. recall 0.258 for the vulnerable class). Together, these discrepancies suggest that both models rely on shallow lexical or syntactic signals, failing to capture deeper semantic relationships and consequently mishandling subtle or context-dependent flaws.

The precision and recall imbalance makes their raw outputs impractical for real-world triage. On **BigVul**, `Phi4` adopts a markedly *risk seeking* stance, generating a flood

of false alarms that can overwhelm analysts. At the same time, its high precision but low recall on the *non vulnerable* class means benign files remain flagged and untriaged. Conversely, `LLaMA 8B` on the heavily imbalanced **DiverseVul** swings to the opposite extreme: it misses many true flaws while still offering only moderate precision, showing that class imbalance can push the model into an overly *risk-averse* regime.
These dataset dependent oscillations reveal an overreliance on superficial lexical cues and poorly calibrated decision boundaries. In practice, *small LLMs in zero-shot are not yet reliable standalone vulnerability detectors*: they either inundate pipelines with false positives or silently overlook critical bugs.

Table 20: Comparison of classification metrics for `Phi4` and `LLaMA 8B` across three datasets: `Cleanvul` (**Cln**), `Bigvul` (**Big**), `DiverseVul` (**Div**).

Metric (Class)	**CleanVul (Phi4)**	**BigVul (Phi4)**	**DiverseVul (LLaMA 8B)**
Prec. (Vuln)	0.390	0.054	0.437
Recall (Vuln)	0.916	0.835	0.258
F_1 (Vuln)	0.547	0.101	0.325
Prec. (Non-vuln)	0.784	0.960	0.454
Recall (Non-vuln)	0.176	0.212	0.650
F_1 (Non-vuln)	0.287	0.347	0.535
Macro Precision	0.587	0.507	0.446
Macro Recall	0.546	0.523	0.454
Macro F_1	0.417	0.224	0.430
Accuracy	0.446	0.244	0.449

9.2.3 Shaping LLM Behaviour for Effective Vulnerability Detection

The issues we just discussed show that both models struggle to make stable, reliable decisions. To fix this, we finetune them with Group Relative Policy Optimization (GRPO) [112]. Our reward combines two signals: *label correctness* and *explanation quality*, to prevent the model from overfitting to label frequency or noise, ensuring consistent learning across both clean and imbalanced datasets.

1. **Phi-4 → *BigVul***: stresses robustness under label noise and severe class imbalance, encouraging stable recall without collapsing precision.

2. **LLaMA 8B → *DiverseVul***: exploits high-precision labels to teach the model to justify predictions without over-flagging benign code.

Table 21 compares the top-5 CWE categories in the training and test splits of *DiverseVul* and *BigVul*. *BigVul* shows a highly skewed distribution: the same five CWEs appear in both splits with similar proportions, collectively covering approximately 55% of the entire dataset. This overlap limits variability and increases the risk of overfitting to these dominant classes. In contrast, *DiverseVul* provides a broader and more balanced distribution: its top-5 CWEs account for only about 39% of the data, and two categories (CWE-20 and CWE-416) appear exclusively in either the training or the test split. This split-specific coverage introduces variability between training and evaluation, making *DiverseVul* a more challenging and realistic benchmark for testing model generalization under domain shift conditions.
Both training and evaluation were performed using the same structured prompt shown in the listings 1 and 2. Post-training, both models were evaluated on all 3 benchmarks *CleanVul*, *BigVul* (only the *Test set* for Phi4) and *DiverseVul* (only the *Test set* for LLaMA 8B). To train the models, we start from the original GRPO loss (shown in Equation 14) and introduce the modifications detailed below.

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}\left[q \sim \mathbb{P}(Q), \{o_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(O|q)\right] \left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\min\left(\frac{\pi_\theta(o_{i,t}|q,o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q,o_{i,<t})}\hat{A}_{i,t}, \text{clip}\left(\frac{\pi_\theta(o_{i,t}|q,o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q,o_{i,<t})}, 1-\epsilon, 1+\epsilon\right)\hat{A}_{i,t}\right) - \beta\,\mathrm{D}_{\text{KL}}\left[\pi_\theta\|\pi_{\text{ref}}\right]\right] \tag{10}$$

At the first iteration, the ratio is always 1 since $\pi_\theta = \pi_{\theta_{\text{old}}}$, making the clipping unnecessary. To remove *Question-level difficulty bias* [117], the advantage is defined as $\hat{A}_{i,t} = R_i - \bar{R}$ instead of being normalized by the standard deviation, where R_i is the return of the i-th trajectory and $\bar{R}$ is the mean return over the group G. Moreover, we set $\beta = 10^{-6}$ in all experiments. The objective simplifies to Equation 15:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \frac{1}{G}\sum_{i=1}^{G}(R_i - \bar{R}) - 10^{-6} \cdot \mathrm{D}_{\text{KL}}\left[\pi_\theta\|\pi_{\text{ref}}\right] \tag{11}$$

Table 21: Top-5 CWE distribution in the training and test splits of *DiverseVul* and *BigVul*.

(a) *DiverseVul*

CWE	Train		Test	
	#	%	#	%
787	31 353	10.88	3 978	11.06
125	22 571	7.83	2 719	7.56
703	20 242	7.02	2 461	6.84
119	19 902	6.90	2 589	7.20
20	18 248	6.33	–	–
416	–	–	2 331	6.48

(b) *BigVul*

CWE	Train		Test	
	#	%	#	%
119	21 293	17.46	4 478	16.82
20	16 392	13.44	3 388	12.72
399	11 769	9.65	2 889	10.85
264	9 894	8.11	2 287	8.59
416	7 887	6.47	1 559	5.86

9.2.4 Insights on *Phi-4* Trained with GRPO on *BigVul*.

This experiment was designed to assess Phi4's capacity to generalize from a massive vulnerability corpus, addressing the risk of overfitting to project-specific code patterns in BigVul: by fine-tuning Phi-4 on a dataset with many snippets drawn repeatedly from the same projects, we test whether it can learn transferable vulnerability representations beyond individual codebases.

Experimental Set-Up. The GRPO training procedure is guided by a structured reward function composed of two components: a *reasoning reward* and a *correctness reward*, computed respectively as F_i and C_i in Equation (12). These components are derived from dedicated verifiers applied to model outputs structured with `<answer>` and `<reasoning>` tags.

Correctness reward (C_i). The correctness reward C_i is computed by comparing the content of the `<answer>` tag, $y_i \in \{\text{Yes}, \text{No}\}$, with the ground-truth label $t_i \in \{\text{Yes}, \text{No}\}$ via case-insensitive matching:

$$c_i = \begin{cases} 1, & \text{if } y_i = t_i, \\ 0, & \text{otherwise.} \end{cases}$$

To discourage repetitive answers, we check the last three mistakes $S_i = \{j_1, j_2, j_3\}$ where $c_{j_k} = 0$, and define:

$$m_i = \begin{cases} 1, & \text{if } |S_i| = 3 \text{ and } \forall j \in S_i : y_j = y_i, \\ 0, & \text{otherwise.} \end{cases}$$

The final reward combines these terms:

$$C_i = \alpha c_i + \beta(1 - c_i)m_i = \begin{cases} \alpha, & \text{if } c_i = 1, \\ \beta, & \text{if } c_i = 0 \text{ and } m_i = 1, \\ 0, & \text{otherwise.} \end{cases}$$

Empirically, the best performance was observed with $\alpha = 8$, $\beta = -2$ and if the <answer> tag is missing or empty, we set $C_i = -5$.

Reasoning reward (F_i). The reasoning reward evaluates the explanation provided in the <reasoning> field and consists of two components. Let $l_i \in \{0, 2\}$ represent the semantic alignment between the reasoning r_i and the vulnerability description d_i, computed using a *Cross-Encoder* (`nli-deberta-v3-base`):

$$l_i = \begin{cases} 2, & \text{if } r_i \text{ entails } d_i, \\ 0, & \text{otherwise.} \end{cases}$$

Let $|L_i|$ be the character length of r_i. We reward longer explanations (up to 1000 characters) as:

$$s_i = \min(|L_i|, 1000) \times 0.005 \quad \text{so that } s_i \in (0, 5].$$

The reasoning reward is the sum of both components:

$$F_i = l_i + s_i.$$

If the <reasoning> tag is missing or empty, a fixed penalty is applied:

$$F_i = -5.$$

The total reward combines correctness and reasoning scores:

$$R_i = C_i + F_i \tag{12}$$

with $R_i \in [-10, 15]$.

Findings.

1. **Marked improvement from base to finetuned model.** Fine-tuning Phi-4 yields substantial performance gains across datasets. On *BigVul*, Macro-F_1 rises from 0.224 to 0.411 (+83%) and accuracy improves by 13%. Similarly, on *CleanVul*, Macro-F_1 increases from 0.417 to 0.581 (+39%) alongside a 14% boost in accuracy. These results underscore the effectiveness of fine-tuning, though further analysis is needed to understand the behavioral shifts introduced by this process.

2. **Substantial gains on Not Vulnerable examples.** By adopting our chosen correctness based reward, we observe a pronounced uplift in the classification of safe code. On *CleanVul*, the F_1 score of *Not Vulnerable* examples jumps from 0.287 to 0.586, and on *BigVul* from 0.347 to 0.694. Most of this improvement comes via recall on the safe class: while precision remains essentially flat, the model now catches far more true negatives. This means our reward scheme both reduces false alarms and improves specificity.

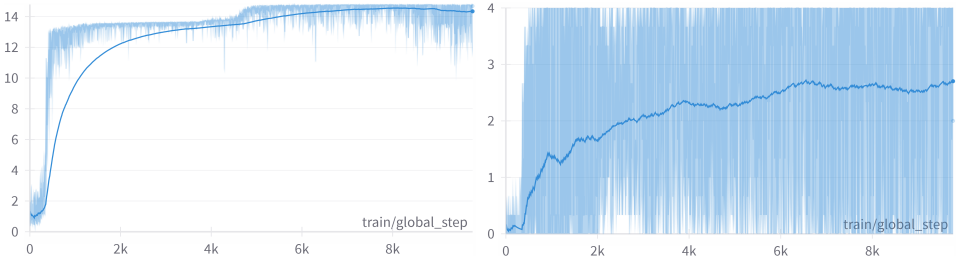
3. **Modest progress on "Vulnerable" examples.** We also see F_1 gains when identifying vulnerable code, though they're more muted. On *BigVul*, F_1 score of *Vulnerable* examples rises from 0.101 to 0.128, a small net improvement driven by a slight precision boost that partially offsets a drop in recall. The pattern repeats in *DiverseVul*: precision nudges up, recall dips, and F_1 edges higher, suggesting that meaningful advances in vulnerability detection remain challenging despite these improvements.

4. **Consistent improvements across datasets.** Significantly, the performance improvements observed on *BigVul* generalize to other benchmarks: *CleanVul* and *DiverseVul* demonstrate comparable performance trajectories, indicating that our reward driven fine-tuning approach exhibits robust cross-dataset generalization capabilities. Nevertheless, potential label noise in *BigVul* may limit the extent of achievable performance gains, suggesting that label quality currently constitutes the primary limiting factor in further advancements.

Further insight & next steps. The persistent gap in vulnerability recall and low precision on imbalanced data point to two root causes: (1) the model's initial conservative bias and (2) annotation errors in *BigVul*. We therefore hypothesize that training on a corpus with cleaner, more granular vulnerability labels will encourage the model to internalize true semantically driven attack patterns rather than artifacts of a noisy dataset. To validate this, we finetuned the smaller *Llama 8B* model on the higher quality *DiverseVul* annotations.

Table 22: Compact classification metrics for Phi-4 model trained on BigVul and tested on CleanVul, BigVul (test), and DiverseVul.

Dataset	Vulnerable			Not Vulnerable			Macro Avg			Accuracy
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Acc.
CleanVul	0.4555	0.7794	0.5750	0.7871	0.4668	0.5860	0.6213	0.6231	0.5805	0.5806
BigVul	0.0710	0.6542	0.1281	0.9668	0.5409	0.6937	0.5189	0.5976	0.4109	0.5467
DiverseVul	0.0586	0.7753	0.1090	0.9748	0.4110	0.5783	0.5167	0.5932	0.3436	0.4275



(a) Formatting Reward (b) Correctness Reward

Figure 34: Reward progression across training steps. The formatting reward stabilizes quickly, while correctness reward grows more gradually.

9.2.5 Generalisation Insights for *LLaMA 8B* trained on *DiverseVul*.

This experiment was designed to explicitly enforce project-level disjointness between training and testing samples, addressing one of the core limitations highlighted by the *DiverseVul* authors [224]: *the inability of deep learning models to generalize to* unseen projects*, a critical requirement in real-world vulnerability detection.*

Reward Design. Even in this case, the GRPO trainer uses a reward function based on the *correctness of the predicted answer* and the *quality of the explanation*.

- **Correctness reward** (Figure 34b). For each completion, we extract the content inside the `<answer>` tag and compare it to the correct answer, ignoring case. The answer is expected in the form "`Yes, the code is vulnerable`" or "`No, the code is not vulnerable`". If the prediction exactly matches the correct answer, we assign a reward of 4. Otherwise, the reward is 0.

- **Explanation (formatting) reward** (Figure 34a). We then check the explanation inside the <reasoning> tag and assign a reward based on three criteria:

    1. **Clear structure.** We check if the explanation follows a *three-step* structure of the reasoning part of the given prompt, shown in Listing 1, using the pattern "`1. ... 2. ... 3. ...`". If present, we give a bonus.
    2. **Length and wording variety.** We reward explanations that are longer but not repetitive. We use a log-based bonus that grows with the number of words, but we reduce the reward if the explanation repeats the same words too much.
    3. **Consistency with the answer.** We compare the final step (Step 3 of the given prompt, shown in Listing 1) of the explanation to the predicted answer. We check both how similar they are in terms of words (edit distance[22]) and in meaning (cosine similarity using MiniLM-L6-v2 embeddings[23]). If the explanation just repeats the answer without adding value, we apply a small penalty. On the other hand, if it provides a meaningful and consistent justification, we give an extra bonus.

- **Coherence check and batch filtering.** We compute a *Coherence Score*, measuring how well the explanation's final step matches the predicted answer in meaning (Figure 35a). If this score is too low (below 0.4), the explanation is marked as `incoherent`. We then calculate the *Incoherent Answer Ratio*, which tells us how many explanations in the batch are incoherent (Figure 35b). If more than half of the batch is incoherent, we completely remove the correctness reward for the whole batch to avoid reinforcing flawed answers.

Reward aggregation. Correctness and formatting scores are min–max normalised within the batch and combined by

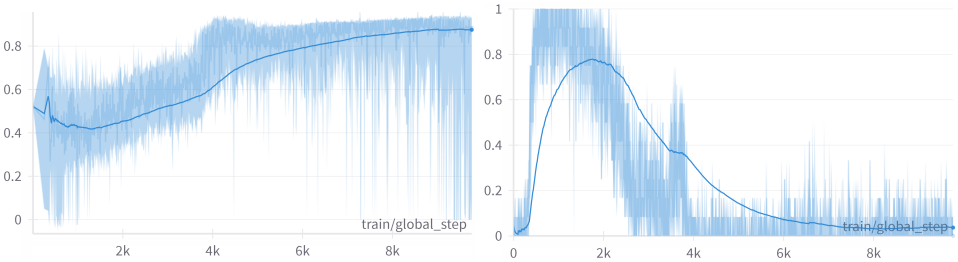
$$r_i \;=\; \alpha\,\hat{F}_i \;+\; (1-\alpha)\,\hat{C}_i,$$

where the mixing coefficient α (Figure 36a) is *dynamic*: it shifts weight towards correctness when the average formatting score is high and towards formatting when the average is low. Next, r_i is passed through *power scaling*: r_i^{γ}, where the exponent

[22] https://docs.python.org/3/library/difflib.html
[23] sentence-transformers/all-MiniLM-L6-v2

γ grows with the batch-wise stability of correctness scores (low standard deviation) to emphasise confident predictions (Figure 36b). Finally, the scaled rewards go through a temperature-controlled softmax (τ) so that they sum to 1 and can be used directly as RL preference weights. Putting everything together, the final reward for completion i is

$$\text{FinalReward}_i = \begin{cases} 0, & \text{if } \texttt{IncoherentRatio} > 0.5 \\ & \text{and } \texttt{isIncoherent}_i = 1 \\ \texttt{Softmax}\Big((\alpha \hat{F}_i + (1-\alpha)\hat{C}_i)^{\gamma}/\tau\Big), & \text{otherwise.} \end{cases} \tag{13}$$



(a) Coherence Score

(b) Incoherent Ratio Score

Figure 35: Progression of coherence-based metrics across training steps.

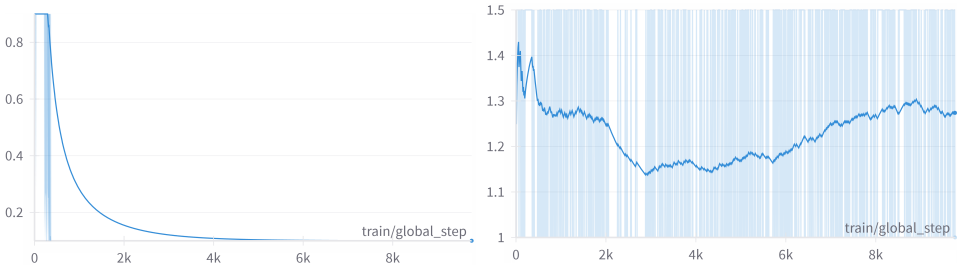
Findings.

1. **Within-project effectiveness.** On the in-distribution portion of *DiverseVul*, even if we have test samples that come from different projects with respect to the ones in the training set, the model performs strongly: a recall of **0.81** and an F_1 score of **0.68** for the *Vulnerable* class show that LLaMA 8B, once fine-tuned, effectively learns high-level semantic cues related to vulnerabilities.

2. **Generalisation under domain shift.** When tested on *CleanVul*, a dataset that not only features different coding styles, library usage, and project conventions, but also includes programming languages beyond C (which dominates *DiverseVul*), the model still achieves reasonably good performance. Although there is a performance drop, with recall for *Vulnerable* samples decreasing to **0.42** and F_1 to **0.50**, these results confirm the model's ability to generalise

beyond the C language. This suggests that, despite the domain shift and the presence of different languages, the model retains some capacity to detect vulnerabilities in a cross-language setting, even though its representations remain somewhat sensitive to superficial stylistic differences.

3. **Partial transferability via structural similarity.** Interestingly, results on *BigVul* are more balanced. With $F_{1,\text{Vul}} = 0.61$ and $F_{1,\text{Weighted}} = 0.62$, the model maintains performance close to that achieved in-distribution. This may be attributed to latent structural or syntactic similarities between *BigVul* and *DiverseVul*, such as overlapping programming languages, function structures, or comment styles, which facilitate transfer even across datasets.



(a) Alpha Coefficient (b) Power Function

Figure 36: Progression of *Alpha coefficient* and *Power function* across training steps.

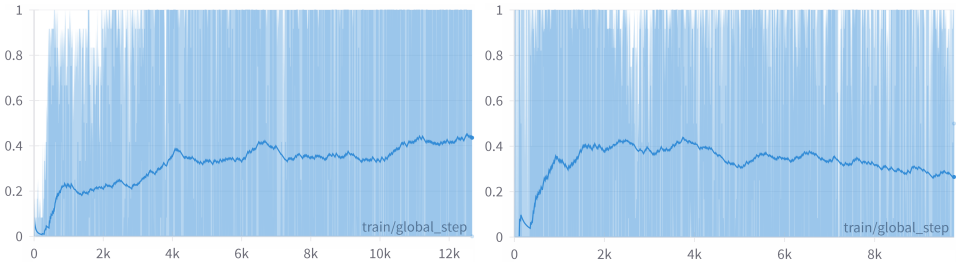
Table 23: Compact classification metrics for LLaMA 8B trained on DiverseVul and tested on DiverseVul (ID), CleanVul (OOD), and BigVul (OOD).

Dataset	Vulnerable			Not Vulnerable			Macro Avg			Accuracy
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Acc.
DiverseVul	0.5791	0.8149	0.6771	0.6998	0.4215	0.5261	0.6395	0.6182	0.6016	0.6159
CleanVul	0.6103	0.4240	0.5004	0.5587	0.7293	0.6327	0.5845	0.5766	0.5665	0.5766
BigVul	0.6370	0.5885	0.6118	0.6096	0.6570	0.6324	0.6233	0.6228	0.6221	0.6224

9.3 Impact of KL Regularization on Negative Class Behaviour

To better understand the influence of KL regularization on model behaviour, we conduct a controlled ablation varying the β parameter, which regulates the KL divergence weight in the GRPO loss. In particular, we contrast two values, $\beta = 10^{-4}$ and $\beta = 10^{-6}$, and observe their effect on the rates of **True Negatives** (TN) and

False Negatives (FN) during training. Figure 37 illustrates that a higher β value leads to a more conservative model: the True Negative rate steadily increases over time, indicating that the model becomes more confident in identifying non-vulnerable code. However, this behaviour comes at a cost: as shown in Figure 47, the False Negative rate also rises, especially after the early training phase. This suggests that excessive regularization suppresses risk-taking, leading the model to underpredict vulnerabilities in borderline or ambiguous samples. Conversely, reducing β to 10^{-6} yields the opposite trade-off: while the model becomes less confident in rejecting benign samples (lower TN rate), it maintains a lower FN rate throughout training, exhibiting a more risk-seeking stance that prioritizes catching vulnerabilities over avoiding false alarms. These findings confirm that β acts as a behavioural dial: increasing it promotes conservative classification and favours high precision on the *Non vulnerable* class, whereas smaller values encourage high recall for the *Vulnerable* class by reducing over-regularization. The optimal β value therefore depends on the desired balance between false positives and false negatives in downstream use cases.

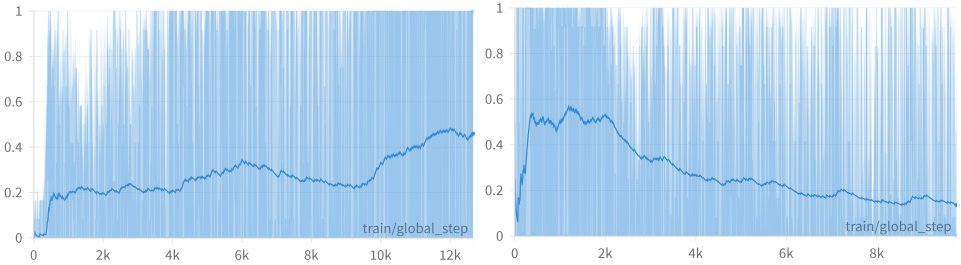


(a) $\beta = 10^{-4}$ (b) $\beta = 10^{-6}$

Figure 37: Effect of β on **True Negative** rates during training.

9.4 Conclusions

This chapter explored the behavioral limits of small instruction-tuned LLMs in vulnerability detection and showed that reinforcement learning can significantly improve their performance. In zero- and few-shot settings, models like Phi-4 and LLaMA-8B show poor risk calibration: they often flag safe code as vulnerable, miss real flaws, and perform worse when tested on new projects or programming languages. Their explanations, when provided, are often unclear or inconsistent, which reduces trust in real-world security workflows.



(a) $\beta = 10^{-4}$ (b) $\beta = 10^{-6}$

Figure 38: Effect of β on **False Negative** rates during training.

Using Group Relative Policy Optimization (GRPO) with rewards that focus on both detection accuracy and explanation quality leads to steady improvements across benchmarks. On BigVul, Phi-4's macro-F1 increases from 0.224 to 0.411 (+83%) and accuracy from 0.244 to 0.547. On the out-of-distribution CleanVul dataset, its macro-F1 improves from 0.417 to 0.581 (+39%) with a 14% accuracy gain. For LLaMA-8B, trained under the same conditions on DiverseVul, macro-F1 rises from 0.430 to 0.602 (+40%) and accuracy from 0.449 to 0.616 (+17%). Lowering the KL regularization to $\beta = 10^{-6}$ shifts the model toward better recall, catching more real vulnerabilities, without creating too many false alarms.

These results show that reinforcement learning can effectively shape model behavior, making detectors both more accurate and easier to interpret. However, some limitations remain. The experiments rely on public datasets that only roughly represent real-world codebases. The reward design can be exploited if not carefully monitored, and current models still find it hard to pinpoint the exact location of vulnerabilities or provide clear, step-by-step explanations in large, multi-file projects.

10 Improving LLM Reasoning for Vulnerability Detection via Group Relative Policy Optimization

This chapter extends the previous study that was one of the first to analyze how instruction-tuned LLMs behave when asked to reason about source code. The earlier work showed that these models often make unstable decisions: they tend to mark secure code as vulnerable or fail to properly explain why a piece of code is safe. It also showed that using Group Relative Policy Optimization (GRPO) can improve both stability and accuracy, making it a useful alternative to traditional fine-tuning or retrieval-based approaches.

In this chapter, the analysis is extended to understand how small instruction-tuned models can learn to reason better and generalize to unseen cases. The study is guided by three main research questions.

The first question is: *Can a small instruction-tuned LLM reason effectively about software vulnerabilities without additional fine-tuning?* This question explores whether lightweight models are able to reason about code and identify vulnerabilities in a zero-shot setting, where no extra task-specific training is provided. It helps determine how well small models can handle complex reasoning tasks without external support.

The second question is: *Can we train the model to use its own reasoning to identify vulnerabilities?* To investigate this, the model is trained using GRPO combined with a new modular reward function that evaluates each generated answer across three aspects, Formatting, Correctness, and Reasoning. These scores are dynamically balanced during training to reduce bias, mitigate reward hacking, and ensure that the reasoning process remains coherent and meaningful.

The third question is: *In what ways does GRPO differ from (and potentially improve upon) traditional supervised fine-tuning for code vulnerability detection?* To answer this, several experiments are performed using different small language models, LLaMA 3B, LLaMA 8B, and Qwen 2.5 3B, tested on three widely used datasets: BigVul, DiverseVul, and CleanVul. The evaluation also includes out-of-distribution tests and ablation studies to assess robustness, reasoning reliability, and generalization performance.

10.1 Background and Preliminaries

This section provides an overview of the core background concepts of the work, including the challenges of automatic vulnerability detection and the utilization and training strategies of recent LLMs. It also introduces the experimental settings,

which are consistently applied throughout the evaluations and analyses in the following sections.

Software Vulnerability Detection Automated vulnerability detection aims to identify flaws like buffer overflows or logic errors that compromise system security. As software grows in complexity, manual review becomes unfeasible [227], and traditional tools struggle with evolving language features [228]. In this context, recent LLMs, pretrained on large codebases, have shown strong zero and few shot capabilities in spotting vulnerabilities [57], outperforming older techniques. However, most LLM-based detectors still operate at the line level and lack mechanisms to track control or data flow across code sequences [57].

Finetuning strategies for instruction-based LLMs. The paper examines three strategies for applying LLMs to vulnerability detection: (i) zero-shot prompt engineering; (ii) supervised finetuning; and (iii) finetuning via Group Relative Policy Optimization.

Zero-shot Prompt Engineering evaluates an instruction-tuned LLM using natural language prompts without additional finetuning. In Section 10.3, we compare two variants: a *reasoning prompt*, which asks the model to justify its answer step by step, and a *direct prompt* that yields only a yes/no answer. As no single prompt format is optimal, we design both to represent the extremes of detailed reasoning and concise judgment. This strategy serves as a baseline to assess the model's out-of-the-box reasoning capabilities.

Supervised Finetuning (SFT) is a widely used technique to specialize pretrained language models for downstream tasks by training them on curated input-output pairs. In the context of software vulnerability detection, SFT involves exposing the model to annotated code snippets in which vulnerabilities are labeled, enabling it to learn mappings from code patterns to vulnerability types or likelihoods [56, 229]. SFT relies on direct supervision in the form of ground-truth labels, which makes it effective for initial task adaptation. However, when used alone, it may be limited by the quality and coverage of the labeled dataset [230], potentially leading to overfitting on common examples while missing subtle or novel flaws. Furthermore, SFT typically lacks mechanisms to encourage exploration or nuanced reasoning over complex structures, such as source code [231].

Group Relative Policy Optimization (GRPO) is a reinforcement learning algorithm proposed by DeepSeek [112, 113], designed to improve the stability of training and reduce overfitting compared to SFT, using policy updates similar to Proximal Policy

Optimization (PPO) [223]. The core idea is to leverage multiple sampled outputs during training, generated in response to the same input prompt, to estimate and apply rewards in a relative manner. Formally, to train instruction-based LLMs, the LLMs is trained with a GRPO loss, denoted as $\mathcal{J}_{\text{GRPO}}(\theta)$:

$$\mathbb{E}_{q\sim\mathbb{P}(Q),\,\{o_i\}\sim\pi_{\theta_{\text{old}}}}\left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}C_{i,t}-\beta\,D_{\text{KL}}\left(\pi_\theta\,\|\,\pi_{\text{ref}}\right)\right] \tag{14}$$

where, $\mathbb{P}(Q)$ is the distribution over input queries q, and $\pi_{\theta_{\text{old}}}$ is the policy used to sample G model answers $\{o_i\}$. Each o_i has length $|o_i|$ and $C_{i,t}$ denotes the clipped policy advantage, computed as: $C_{i,t} = \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_{i,t}, \text{clip}\right)$, where $\hat{A}_{i,t}$ is the estimated advantage at time step t for the i-th sampled output o_i, and *clip* represents a threshold as in PPO-style updates [223]. The ratio $\frac{\pi_\theta}{\pi_{\theta_{\text{old}}}}$ captures the relative likelihood of the current policy π_θ compared to the old policy. The final term applies a Kullback–Leibler divergence penalty (D_{KL}) between the current π_θ and a reference policy π_{ref}, scaled by β.

Training models with this approach can be time consuming and resource intensive. Therefore, a simplified version of the original loss is used in this work. Specifically, following common practice, we adopt a simplified setting in which a single iteration is performed per training step, that is, each training sample is seen only once during a training epoch. Under this setup, at the first iteration, the policy ratio is always 1 since $\pi_\theta = \pi_{\theta_{\text{old}}}$, making the clipping operation unnecessary. To mitigate the *question-level difficulty bias* [117], we define the advantage as $\hat{A}_{i,t} = R_i - \bar{R}$ instead of normalizing by the standard deviation, where R_i is the reward for the i-th model response and $\bar{R}$ is the mean reward within the group G. Finally, we set $\beta = 10^{-6}$ for the main experiments, while ablation studies on its effect are presented in Section 12.8.1. The resulting GRPO objective simplifies to:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \frac{1}{G}\sum_{i=1}^{G}(R_i - \bar{R}) - 10^{-6}\cdot \text{D}_{\text{KL}}\left[\pi_\theta\|\pi_{\text{ref}}\right] \tag{15}$$

Originally developed for complex reasoning tasks like mathematical problem solving [112], GRPO has demonstrated strong performance in optimizing LLMs and addressing challenges encountered with earlier policy-based techniques and SFT [232]. However, applying GRPO to a new task, such as vulnerability detection, requires the definition of suitable reward and advantage functions, which has not yet been explored previously in this context.

10.2 Setup and Settings

Models Throughout our analysis and experiments, we evaluate three famous small-scale LLMs: `LLaMA 8B` [233], `LLaMA 3B` [233], and `Qwen 2.5 3B` [234], following common practice in the literature [235, 236]. These models were selected because they are open-source and widely adopted in the research community, they cover different architectures and sizes (3B vs. 8B), and they remain computationally feasible for training and evaluation under resource constraints. Please note that, unlike prior work [59, 229], we focus on general-purpose models not pretrained or finetuned on code, to assess their broader reasoning ability. Each model was used with its original tokenizer, without modification.

Datasets To evaluate these models, we use three vulnerability datasets, *DiverseVul*, *BigVul*, and *CleanVul*, each selected for its specific strengths in training and evaluation. In particular, *DiverseVul* contains 18,945 vulnerable functions across 150 CWEs, along with 330,492 non-vulnerable functions. Its high level of manual curation ensures high label precision, making it well-suited for model finetuning [224]; *BigVul* is a large-scale dataset of C/C++ vulnerabilities, compiled by crawling the public CVE database and associated GitHub repositories [225]; *CleanVul* contains 8,203 functions in six languages (Java, Python, JavaScript, C#, C, and C++). Although semi-synthetic with 90.% correctness [226], it is among the most diverse multilingual vulnerability datasets and widely used for cross-language evaluation. Here, we use it only as an out-of-distribution (OOD) testbed to test whether models trained mainly on C/C++ (DiverseVul, BigVul) can generalize across languages and paradigms. In both Section 10.3.1 and 10.4, we use 80% of DiverseVul dataset to train our models and we let the other 20% as test dataset. To ensure consistency, all datasets were balanced to a 1:1 ratio of vulnerable and non-vulnerable samples in both training and evaluation. No additional oversampling, undersampling, or synthetic data were used. Samples exceeding the 4,000-token limit were excluded to preserve the full code structure.

Metrics The goal here is to understand how these models perform in detecting vulnerabilities, that means, given a code in input to the large language model, it should answer *Not Vulnerable* or *Vulnerable* during the inference phase. Hence, given this binary classification setting, where (*Vulnerable* is the positive class and *Not Vulnerable* is the negative class), we define the standard and well-known confusion matrix components for each class as follows: true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). Based on these, we

can compute the class-wise performance metrics as:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}, \qquad \text{Precision} = \frac{TP}{TP + FP},$$
$$\text{Recall} = \frac{TP}{TP + FN}, \qquad \text{F1 Score} = \frac{2 \cdot \text{Prec.} \cdot \text{Rec.}}{\text{Prec.} + \text{Rec.}},$$

To summarize overall model performance, we report in the subsequent analysis the following scores: *Macro F1*: the unweighted mean of the F1 Scores across all classes, treating each class equally and *Weighted F1*: the mean of the F1 Scores weighted by the support (number of true instances) of each class.

10.3 Reasoning Capabilities of Small LLMs

In this section, we investigate our first research question RQ1, which explores the zero-shot capabilities of small off-the-shelf language models for vulnerability detection. In particular, we evaluate the original (non finetuned) LLMs in their instruction-tuned baseline versions using two different zero-shot *system prompts*, to explore the performance both in terms of detection accuracy and reasoning. In the first system prompt, we directly ask the model to say whether the provided code is vulnerable or not (first block in Figure 39) and in the second system prompt, we ask to reason about the provided code before making the final verdict (second block in Figure 39). In both cases, the *system prompt* is followed by a *user prompt* with the code snippet to analyze (bottom block in Figure 39).

System Prompt - *No Reasoning* Test

You are a senior software security auditor.
When I give you a code snippet, silently analyse it for any realistic, exploitable security weakness (e.g. injections, memory-safety errors, race conditions, logic flaws, privilege escalation, unsafe crypto, misuse of APIs, etc.).
Decision:

- If you identify at least one credible exploit path, reply with the single word **vulnerable**
- If you find none, reply with the single word **not vulnerable**

Output rules (MUST follow):

1. Reply with **exactly** one of the two words above—nothing else.
2. No additional text, quotes, punctuation, markdown, or whitespace before/after the word.
3. Perform all reasoning internally; do **not** expose it in the output.

System Prompt - *Reasoning* Test

You are a software security expert. I will provide you with code snippets. Your task is to analyze the code and determine whether it contains any vulnerabilities. `<reasoning>`

1. Summarize the code snippet.
2. Check thoroughly for vulnerabilities.
3. Clearly state whether the code is secure or insecure.

`</reasoning>` `<answer>` **Yes, the code is vulnerable.** *OR* **No, the code is not vulnerable.** `</answer>`

User Prompt - *Reasoning* and *No Reasoning* Test

Analyze the following code snippet to determine whether it is vulnerable. `CODE SNIPPET`

Figure 39: System prompts for the no-reasoning and reasoning settings (top and middle blocks, respectively), and the user prompt that is appended to both.

The rationale behind these experiments is that, by comparing results across both settings and multiple datasets, we can better understand how small language models behave when making security-critical decisions, and whether reasoning or *thinking aloud* helps in detection metrics. Note that, in the reasoning prompt, each model must provide its own final verdict enclosed within `<answer>` and `</answer>`, which is then compared against the ground-truth annotations provided by the datasets.

Table 24: Comparison across models (Qwen 2.5, LLaMA 3B, LLaMA 8B) on datasets: CleanVul, DiverseVul, and BigVul — Without Reasoning (NR) vs With Reasoning (R)

	Qwen 2.5						LLaMA 3B						LLaMA 8B					
	CleanVul		DiverseVul		BigVul		CleanVul		DiverseVul		BigVul		CleanVul		DiverseVul		BigVul	
	NR*	R	NR*	R	NR*	R	NR*	R	NR	R*	NR	R*	NR	R*	NR	R*	NR	R*
Not Vulnerable																		
Precision	**0.56**	0.52	**0.53**	0.49	**0.56**	0.54	**1.00**	0.53	**0.67**	0.46	0.00	**0.52**	1.00	**0.50**	0.00	**0.44**	**1.00**	0.51
Recall	0.65	**0.80**	0.55	**0.75**	**0.91**	0.87	0.00	**0.47**	0.00	**0.39**	0.00	**0.53**	0.00	**0.92**	0.00	**0.26**	0.00	**0.96**
F_1	0.60	**0.63**	0.54	**0.59**	**0.69**	0.66	0.00	**0.50**	0.00	**0.42**	0.00	**0.52**	0.00	**0.65**	0.00	**0.33**	0.00	**0.67**
Support	6330		2952		1134		6330		2952		1134		6330		2952		1134	
Vulnerable																		
Precision	**0.59**	0.56	**0.52**	0.43	**0.75**	0.62	0.50	**0.52**	**0.49**	0.45	0.49	**0.50**	0.50	**0.53**	**0.49**	0.45	0.49	**0.55**
Recall	**0.49**	0.26	**0.50**	0.19	**0.27**	0.22	**1.00**	0.58	**1.00**	0.52	**1.00**	0.50	**1.00**	0.09	**1.00**	0.65	**1.00**	0.06
F_1	**0.53**	0.35	**0.51**	0.26	**0.40**	0.33	**0.67**	0.55	**0.66**	0.48	**0.66**	0.50	**0.67**	0.15	**0.66**	0.54	**0.66**	0.10
Support	6330		2952		1134		6330		2952		1134		6330		2952		1134	
Overall																		
Accuracy	**0.57**	0.53	**0.52**	0.48	**0.59**	0.55	0.50	**0.53**	**0.49**	0.45	0.49	**0.51**	0.50	**0.50**	**0.49**	0.45	0.49	**0.51**
Macro Precision	**0.57**	0.54	**0.52**	0.46	**0.65**	0.58	**0.75**	0.53	**0.58**	0.45	0.25	**0.51**	**0.75**	0.52	0.25	**0.45**	**0.75**	0.53
Macro Recall	**0.57**	0.53	**0.52**	0.47	**0.59**	0.55	0.50	**0.53**	**0.50**	0.46	0.50	**0.51**	0.50	**0.50**	**0.50**	0.45	0.50	**0.51**
Macro F_1	**0.57**	0.49	**0.52**	0.43	**0.54**	0.50	0.33	**0.53**	0.33	**0.45**	0.33	**0.51**	0.33	**0.40**	0.33	**0.43**	0.33	**0.38**
Weighted Precision	**0.57**	0.54	**0.52**	0.46	**0.65**	0.58	**0.75**	0.53	**0.58**	0.45	0.24	**0.51**	**0.75**	0.52	0.24	**0.45**	**0.75**	0.53
Weighted Recall	**0.57**	0.53	**0.52**	0.48	**0.59**	0.55	0.50	**0.53**	**0.49**	0.45	0.49	**0.51**	0.50	**0.50**	**0.49**	0.45	0.49	**0.51**
Weighted F_1	**0.57**	0.49	**0.52**	0.43	**0.55**	0.50	0.33	**0.53**	0.33	**0.45**	0.33	**0.51**	0.33	**0.40**	0.33	**0.43**	0.33	**0.39**
Support	12660		5904		2268		12660		5904		2268		12660		5904		2268	

Table 24 shows clear differences in how models respond to reasoning. Qwen 2.5 works best without reasoning: it achieves the highest overall scores across all datasets, with a macro F_1 of 0.57 on CleanVul, 0.52 on DiverseVul, and 0.54 on BigVul. When reasoning is added, its performance drops, especially for the `Vulnerable` class, where recall falls by half or more. This suggests that Qwen already has effective internal strategies for detecting vulnerabilities, and forcing it to explain its answers can introduce confusion and reduce accuracy. The opposite happens with LLaMA models. Without reasoning, both LLaMA 3B and LLaMA 8B predict almost everything as `Vulnerable`. This leads to perfect recall on that class, but zero recall on `Not Vulnerable`, which hurts overall performance. Adding reasoning helps these models balance their predictions. For example, LLaMA 3B's F_1 score on the `Not Vulnerable` class goes from 0.00 to 0.50 on CleanVul and reaches 0.52 on BigVul. Its macro F_1 improves from 0.33 to over 0.45 in all datasets. LLaMA 8B shows the same trend, with the biggest gain on CleanVul where the F_1 jumps from 0.00 to 0.65 on `Not Vulnerable` class. However, this improvement comes at the cost of lower recall on the `Vulnerable` class. For instance, on BigVul, LLaMA 8B's recall drops from 1.00 to just 0.06 when reasoning is added. Overall, reasoning helps LLaMA models make better distinctions between vulnerable and

not vulnerable code, especially on harder datasets like BigVul. In contrast, Qwen performs better when it makes predictions directly, without explaining them. This suggest that whether reasoning helps or not depends on the model and the goal: if the task needs fewer false positives, reasoning is useful for LLaMA; but if high vulnerability coverage is more important, Qwen without reasoning may be the better choice. To summarize we conclude the following:

RQ1 *Can a small instruction tuned LLM reason effectively about software vulnerabilities without additional finetuning?*
A1 *Experimental results indicate that models struggle to fully leverage reasoning when analyzing software vulnerabilities. Even for LLaMA, where thinking aloud leads to improvements, its predictions remain unbalanced and prone to false negatives (e.g LLaMA 8b).*

10.3.1 Improving Model Reasoning Abilities

Previous results remark that, although allowing LLMs to reason about the code may improve performance, the models remain unreliable. To further enhance the ability to detect vulnerabilities more consistently, it is necessary to train the model to improve the quality of its own reasoning. To this end, in this section we explore the possibility of adapting and applying reasoning-aware policies during training for more effective finetuning of LLMs RQ2.
In this context, as known in the literature, there are no available datasets that support direct finetuning of models to also reason about vulnerabilities based on ground truth step by step exaplanations, as existing datasets only provide binary ground truth labels. This limitation can be addressed through RL-based approaches, such as GRPO, which allow models to generate their own reasoning while policy parameters are optimized based on predefined reward signals. To enable this, before adopting GRPO in our training pipeline, it is necessary to define appropriate reward functions tailored to vulnerability detection.

10.3.2 Reward Design

To improve the model's ability to detect vulnerabilities by using its own reasoning, we should not only give high rewards to responses that make the correct verdict, but we also need to train the model to format the answers well.
We design a modular reward function that computes the final reward signal by

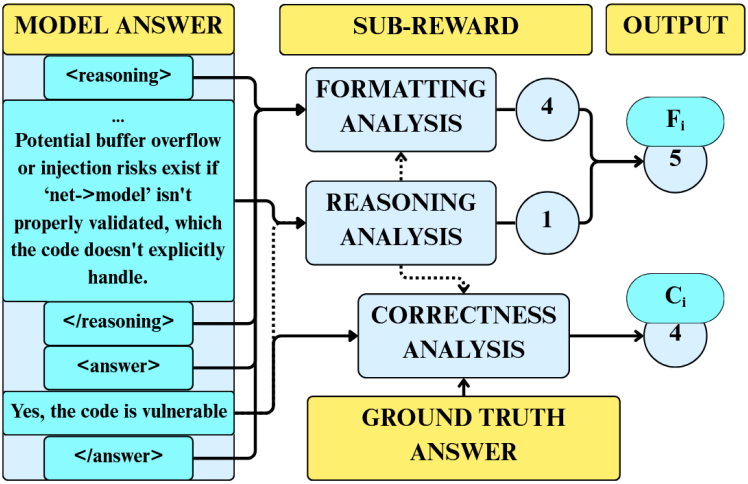


Figure 40: *Sub-Rewards* signals generated by *Formatting, Reasoning and Correctness Analyses*. The dotted lines indicate that incoherent reasoning by the model also nullifies the rewards from correctness and formatting.

performing three distinct analyses (see Figure 40): *Formatting Analysis*, *Correctness Analysis*, and *Reasoning Analysis*, similar to prior GRPO based work using multiple reward components [237]. Each of these analyses produces a corresponding *sub-reward*. For each model answer i belonging to the set of model answers G, the *Formatting Analysis* and *Reasoning Analysis* are aggregated to produce the sub-reward F_i, while the *Correctness Analysis* produces a separate sub-reward C_i. The values F_i and C_i are then combined through a specific strategy that will be detailed later. To train the model we used the reasoning system prompt (second box) and the user prompt (third box) both shown in Figure 39. In the following, we describe each of the analysis in detail.

Formatting analysis. In this step, we verify that the model adheres to the structure requested by the reasoning system prompt. Specifically, we check for the presence of the `<reasoning>` and `<answer>` tags and confirm that the reasoning follows the three-step pattern highlighted in second block of Figure 39. If all these conditions are met, the model receives a reward of 4; if the tags are missing instead, the reward is 0.

Reasoning analysis. We evaluate the quality of the explanation by assigning a bonus based on its length and lexical diversity. Firstly we assess the final reasoning

step (*Step 3* in the reasoning system prompt) by two metrics: the word-level edit distance to the model's final answer contained in <answer> tags (using difflib[24]) and a *Coherence Score* based on cosine similarity of MiniLM-L6-v2 embeddings[25]. If an explanation merely restates the answer without adding insight, we apply a penalty (-2); if it instead adds information, we assign a bonus (+1). If its Coherence Score falls below 0.4, we mark the explanation as *incoherent*. If *incoherent* the total reward is 0.
Next, we consider answer length. In particular, given n, the number of tokens in the reasoning text, we compute a log-scaled length bonus as

$$\text{LengthBonus}(n) = \gamma \cdot \log(n+1), \quad \text{with} \quad \gamma = \frac{5}{ln(2001)}$$

This logarithmic scaling provides smooth informative growth and avoids excessive bonuses for very long responses.[26]

Correctness analysis. For each model answer, we extract the content inside the <answer> tag and compare it to the correct answer. The answer is expected in the form "*Yes, the code is vulnerable*" or "*No, the code is not vulnerable*". If the answer exactly matches the correct one, we assign a reward of 4. Otherwise, it is 0.

Dynamic rewards to avoid reward hacking. After generating the sub-rewards C_i (correctness) and F_i (formatting and reasoning), a proper strategy is required to combine them into a single reward value before being processed by the GRPO loss, which expects a single reward term. In this regard, we observed that following a naive approach that simply sums the two sub-rewards, i.e., $r_i = 0.5 \cdot F_i + 0.5 \cdot C_i$ (denoted in the following plots as *static reward*), leads to unsatisfactory performance during GRPO training. In fact, as shown in Table 25, both the LLaMA 8B and Qwen 2.5 collapse and tend to label every code snippet as *vulnerable*, even though they learn to format their responses correctly. The rationale behind this behavior is that vulnerability detection is a challenging task for small language models, which encourages them to exploit the reward signal by consistently defaulting to a single verdict. This reveals a clear instance of *reward hacking* [239, 240], where the model achieves a neutral advantage by repeating the same answer regardless of

[24] https://docs.python.org/3/library/difflib.html

[25] sentence-transformers/all-MiniLM-L6-v2

[26] To prevent outputs from exceeding the maximum allowed length (3000 tokens), as the output approaches this limit, the bonus must exceed a minimum effective threshold of 5, which results from formatting and reasoning step contributions. Preliminary analysis shows that variations near the 2000 token saturation point minimally affect performance, consistent with prior findings [238].

correctness[27], highlighting the necessity of adaptivity.

Table 25: Comparison between LLaMA 8B (L8B) and Qwen 2.5 (Q2.5) under *static reward* setting in GRPO, as $r_i = 0.5F_i + 0.5C_i$.

Metric	Not Vuln.		Vuln.		Overall	
	L8B	Q2.5	L8B	Q2.5	L8B	Q2.5
Precision	0.00	0.00	0.50	0.50	0.25	0.25
Recall	0.00	0.00	1.00	1.00	0.50	0.50
F_1	0.00	0.00	0.67	0.67	0.33	0.33
Accuracy	–	–	–	–	0.50	0.50
Support	2952		2952		5904	

To overcome this issue is necessary to build a reward signal that adapts over time: during the initial training phases, it is important that the model learns to follow the required format; once formatting is consistently correct and learned, the focus should gradually shift toward improving the quality of reasoning to enhance detection performance.

To support this adaptive training process, we introduce a *Dynamic Reward Module* (illustrated in Figure 41), which adjusts the weighting of the sub-rewards C_i and F_i based on the model's current ability to produce well formatted outputs. In particular, each C_i and F_i scores are min–max normalized within each set of model answers G, and then combined as $r_i = \alpha\, \hat{F}_i + (1 - \alpha)\, \hat{C}_i$.

where $\hat{F}_i$ and $\hat{C}_i$ denote the normalized values of F_i and C_i, respectively and α is *dynamic* coefficient defined as:

$$\alpha = \begin{cases} \tau_A, & \text{if } |H| < 3 \text{ or } \bar{F}_H \leq 3, \\ \max\big(\tau_B,\, \min\big(\tau_A, \tau_A - \tau_B \cdot \bar{F}_H\big)\big), & \text{otherwise,} \end{cases} \tag{16}$$

where H is a buffer that stores the mean *formatting analysis sub-reward* (with a maximum value of 4) over the last three groups of answers, $\tau_A \geq \tau_B$, and $\bar{F}_H$ is its running average. In our setup, we set $\tau_A = 0.9$ and $\tau_B = 0.2$ based on a preliminary experimental exploration. The buffer H is updated after each training batch with the mean formatting reward of the last three groups of answers. To avoid instability in early training due to fluctuations, the dynamic coefficient α remains fixed at τ_A until the running average $\bar{F}_H$ exceeds 3, an empirically chosen threshold ($\sim 60\%$ of the maximum reward on a 0–5 scale) indicating sufficiently stable formatting. The

[27] In GRPO, this phenomenon occurs when all model answers are either entirely correct or entirely incorrect. In both cases, the lack of variance in the advantage signal prevents meaningful learning and instead rewards non-informative or trivial behaviors.

running average $\bar{F}_H$ is computed as the simple arithmetic mean of the last three formatting rewards stored in H. Hence, training starts with $\alpha = \tau_A$, placing strong emphasis on formatting. Once the model produces well-formatted outputs for three consecutive batches (i.e., $\bar{F}_H > 3$), α is gradually reduced, never falling below τ_B, so that correctness gains increasing weight in the reward signal.

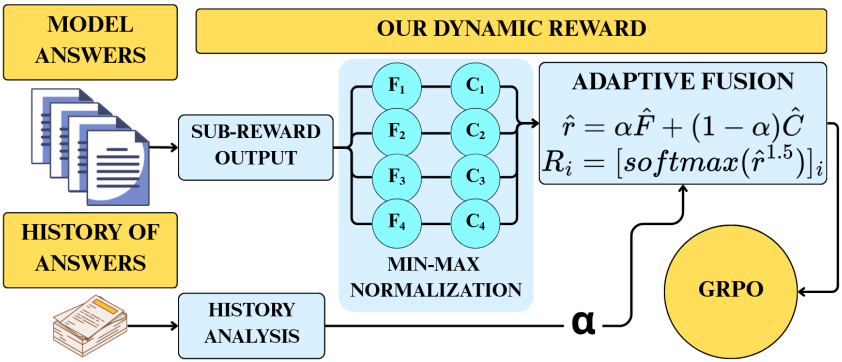


Figure 41: Final reward signal used to feedback the models

Next, we apply *power scaling* to reinforce confident predictions: $r_i \rightarrow r_i^{1.5}$, using a fixed exponent to emphasize predictions with higher confidence. And apply the softmax function of all the answers i accross the set G. Putting everything together, the final reward for model answer i is computed as:

$$R_i = \begin{cases} 0, & \text{if } i \text{ is incoherent,} \\ \left[\text{Softmax}\left((\alpha\hat{F} + (1-\alpha)\hat{C})^{1.5}\right)\right]_i, & \text{otherwise,} \end{cases} \tag{17}$$

where R_i is the final reward used by the GRPO algorithm, and $\hat{F}$ and $\hat{C}$ are the vectors of normalized formatting and correctness scores across all model answers in the produce set G, and

As illustrated in Figure 42 (using Qwen 2.5 as a demonstrative example), this mechanism strengthens the reward signal for stable, high-confidence model answers and suppresses noisy or uncertain outputs. In contrast, when using the static baseline sum of rewards, such as the mean correctness score (mean$(\hat{C})_t$). the value steadily increases throughout training.

10.3.3 GRPO vs. Best Reasoning/No-Reasoning Baselines

After introducing the proposed reward design, we first report comparisons between LLMs trained with our approach and the best-performing reasoning and no-reasoning

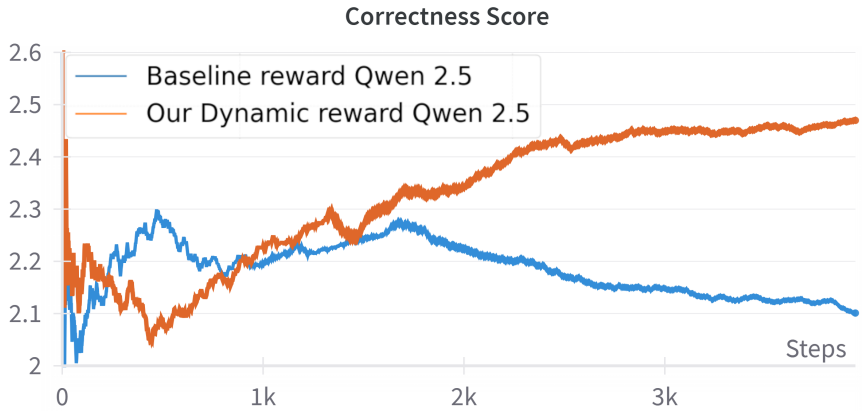


Figure 42: Running average of mean Correctness score ($\hat{C}$) during training, comparing our dynamic reward function to the traditional approach.

zero-shot baselines that we can extract from Table 24.

We train each model using DiverseVul train split, so we consider the DiverseVul test split as *in distribution* data and both CleanVul and BigVul as *out of distribution*. Table 27 shows the GRPO training settings. To avoid policy collapse [241], we chose models at steps with peak or near-peak rewards. Please note that training prompt matches the one used in zero-shot reasoning system-prompt settings (second block of Figure 39), so any changes in performance come only from GRPO training, not prompt differences.

Table 29 reports GRPO results beside the strongest baseline (the better of the reasoning and no-Reasoning variants), for every combination of model and dataset. Values outside the brackets refer to GRPO, whereas bracketed numbers correspond to the baseline; green cells highlight a GRPO advantage. GRPO consistently achieves superior performance compared to the baseline across all evaluations. On the *in distribution* dataset (DiverseVul) every metric improves for all three models, confirming that the method learns robust decision boundaries without overfitting. On CleanVul (*out of distribution*) the two approaches are practically tied for Qwen 2.5, while for LLaMA 3B and 8B GRPO removes the tendency to over-predict the vulnerable class and leads to a more balanced precision–recall trade-off. The largest gains appear on BigVul (*out of distribution*): recall rises sharply, precision remains stable, and models such as Qwen 2.5 and LLaMA 8B shift from a conservative *predict-safe* bias. to a more correct classification of the two classes. Looking at aggregate metrics, accuracy increases by 4–17 percentage points for all models

(except for Cleanvul in Qwen 2.5 whose accuracy remains stable). Macro and weighted F_1 move in lock-step with accuracy, indicating that improvements benefit both classes rather than a single dominant one. To summarize the results achieved in this section, we conclude the following:

RQ2 *Can we train the model to use its own reasoning to identify vulnerabilities?*
A2 *The proposed GRPO formulation enables models to harness their own reasoning to identify vulnerabilities, consistently outperforming baselines across both in distribution and out of distribution datasets.*

10.4 Comparing GRPO and SFT

Table 29 shows that models trained with GRPO outperform the best baseline (either *Reasoning* or *No-Reasoning*) across all model and dataset combinations. However, it is important to assess whether this improvement comes from the use of reasoning or simply from finetuning the model on the vulnerability detection task. To investigate this, we finetuned the models using standard SFT, following common practice in recent LLM-based vulnerability detection research [56, 229]. In SFT, the model is trained to directly predict the final verdict without generating any reasoning. We then compared the performance of GRPO and SFT to understand the real impact of reasoning.
Even in this case, finetuned models use the training split of DiverseVul, so we can still consider the test split of DiverseVul as *in distribution* and BigVul and CleanVul as *out of distribution*. The parameter used to train SFT are shown in table 28. All SFT models were trained with the no-reasoning system prompt in Figure 39, used in the zero-shot setting to isolate the effect of the finetuning strategy from differences in instruction formatting.

10.5 Comparative Analysis across datasets

Table 26 shows that SFT leaves the models heavily biased. For Qwen 2.5 and LLaMA 3B, the *Not Vulnerable* class is almost ignored: recall never exceeds 0.24 and the class F_1 peaks at 0.37. LLaMA 8B does a little better on the in distribution split (*DiverseVul*, recall 0.43, F_1 0.58) but falls back to recall 0.09 and 0.14 on the two out of distribution sets, CleanVul and BigVul respectively. The *Vulnerable* class shows the opposite pattern: recall is high, up to 0.93 for Qwen 2.5 on *DiverseVul*,

Table 26: SFT results (rounded to two decimals). Div.=DiverseVul, Cln.=CleanVul, Big=BigVul.

Category	Metric	Qwen 2.5			LLaMA 3B			LLaMA 8B		
		Div.	Cln.	Big	Div.	Cln.	Big	Div.	Cln.	Big
Not Vulnerable										
	Precision	0.48	0.45	0.55	0.76	0.55	0.55	0.88	0.52	0.61
	Recall	0.07	0.08	0.13	0.24	0.10	0.13	0.43	0.09	0.14
	F_1	0.12	0.13	0.22	0.37	0.17	0.21	0.58	0.16	0.23
Vulnerable										
	Precision	0.50	0.50	0.51	0.52	0.51	0.50	0.62	0.50	0.51
	Recall	0.93	0.60	0.59	0.53	0.51	0.45	0.42	0.73	0.66
	F_1	0.65	0.58	0.62	0.52	0.56	0.50	0.53	0.63	0.63
Overall										
	Accuracy	0.50	0.49	0.51	0.53	0.51	0.51	0.62	0.50	0.52
	Macro Precision	0.49	0.47	0.53	0.53	0.53	0.53	0.64	0.51	0.56
	Macro Recall	0.50	0.49	0.51	0.58	0.51	0.51	0.62	0.50	0.53
	Macro F_1	0.38	0.39	0.43	0.52	0.41	0.42	0.60	0.40	0.44
	Weighted Precision	0.49	0.47	0.53	0.53	0.53	0.53	0.64	0.51	0.56
	Weighted Recall	0.50	0.49	0.51	0.53	0.51	0.51	0.62	0.50	0.52
	Weighted F_1	0.38	0.39	0.43	0.52	0.41	0.42	0.60	0.40	0.44

yet precision stays near 0.50. Because of this imbalance, macro F_1 ranges only from 0.38 to 0.60 and drops sharply whenever the test data deviate from the training distribution (for example, LLaMA 8B falls from 0.60 on *DiverseVul* to 0.40 on *CleanVul*).

Comparing the results of SFT in Table 26 and the results of GRPO in Table 24, the latter leads to clear improvements over SFT across all models and datasets. Macro F_1 increases by 1-29 points, reaching up to 0.67 on Qwen 2.5 (*DiverseVul*), 0.59 on LLaMA 3B (*CleanVul*), and 0.62 on LLaMA 8B (*BigVul*. Accuracy also improves, with all models scoring between 0.58 and 0.67. Most of the gain comes from the *Not Vulnerable* class. For example, recall goes from 0.07 to 0.61 on Qwen 2.5 (*DiverseVul*) and from 0.09 to 0.42 on LLaMA 8B (*CleanVul*). Precision stays stable, so the F_1 score improves significantly. The *Vulnerable* class also maintains high recall and sometimes gains precision. Since these improvements also appear on *out of distribution* datasets, they are likely due to the reasoning encouraged by GRPO rather than simple finetuning.

Table 27: Configuration parameters used for GRPO training.

GRPO Parameters			
β for D_{kl}	1e-6	N. Model answers	12
Training Setup			
N. train epochs	1	Batch size	4
Grad. accum. steps	4	Save steps	500
Optimization Parameters			
Learning rate	5e-6	Weight decay	0.1
Adam β_1	0.9	Adam β_2	0.99
Optim	paged adamw 8bit	LR scheduler	cosine
Warmup ratio	0.1	Max grad norm	0.1
Generation Parameters			
Max prompt len.	4000	Max answer len.	3000
Temperature	0.9	Top-k	50

10.6 Comparative Analysis Across Programming Languages

We evaluate the performance of two training strategies in detecting vulnerabilities without relying on language-specific patterns. Models trained on *CleanVul* were tested on *JavaScript*, *Python*, and *Java*, none of which were present in the *DiverseVul* training set containing only C code.

Table 30 shows that GRPO improves performance across all languages and models. For instance, macro F_1 on Java rises from 0.25 to 0.46 with Qwen 2.5, and from 0.34 to 0.58 on Python. LLaMA 3B and 8B exhibit similar gains, especially for Python and Java. Accuracy and weighted F_1 follow the same trend. Improvements primarily stem from better detection of the *Not Vulnerable* class, with precision maintained, while the *Vulnerable* class retains good precision, resulting in a more balanced performance overall.

The gains suggest that the GRPO reasoning step enhances cross-language generalization. Syntactic complexity appears influential: simpler languages like Python show larger gains, JavaScript moderate gains, and Java smaller relative improvements despite a larger test set. Language similarity to C also matters: Python benefits more from GRPO than Java. Overall, GRPO outperforms SFT in all languages, with the largest gains observed for those that are syntactically simpler or closer to the training language.

Table 28: Configuration parameters used for SFT training.

Training Setup			
N. train epochs	2	Batch size	16
Grad. accum. steps	2	Save strategy	epoch
Logging steps	1		
Optimization Parameters			
Learning rate	3e-4	Weight decay	0.01
Adam β_1	0.9	Adam β_2	0.99
Optim	adamw_8bit	LR scheduler	linear
Warmup steps	10	Max grad norm	0.1
Generation Parameters			
Max prompt len.	4000	Max answer len.	1000
Temperature	0.9	Top-k	50

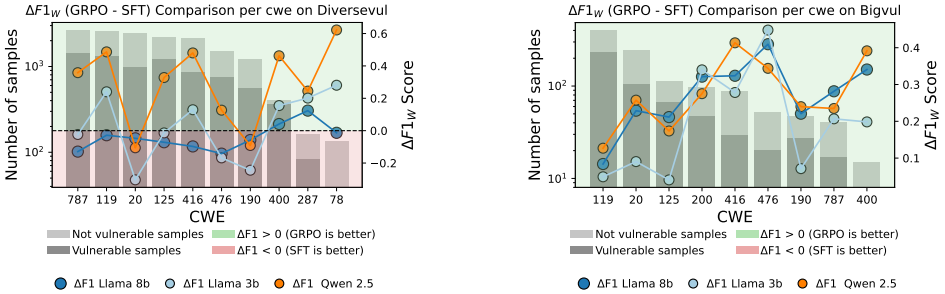


Figure 43: Comparison of CWE performance for GRPO-SFT across three models. The 15 CWEs shown are part of MITRE's Top 25.

10.7 Comparative analysis across CWEs

To see how GRPO and SFT affect specific weakness types, we compare their performance on ten *Common Weakness Enumeration* (CWE) taken from the *2024 CWE Top 25 Most Dangerous Software Weaknesses* [28]. As shown in Fig. 43, These CWEs appear in both *DiverseVul* and *BigVul*. For each CWE, we calculate the change in weighted F_1 score (ΔF_1^w) by subtracting SFT performance from GRPO performance.

On *DiverseVul* GRPO strengthens smaller models. Qwen 2.5 shows significant improvements in most CWEs, especially in *CWE-78*, *CWE-119* and *CWE-416*.

[28] https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html#top25list

Table 29: GRPO results across all datasets and models and the best between NR and R in the brackets.

Category	Metric	Qwen 2.5			LLaMA 3B			LLaMA 8B		
		DiverseVul	CleanVul	BigVul	DiverseVul	CleanVul	BigVul	DiverseVul	CleanVul	BigVul
Not Vulnerable										
	Precision	**0.70** (0.53)	**0.58** (0.56)	**0.63** (0.56)	**0.55** (0.46)	**0.58** (0.53)	**0.56** (0.52)	**0.58** (0.44)	0.70 (**1.00**)	**0.64** (0.51)
	Recall	**0.61** (0.55)	0.54 (**0.65**)	0.69 (**0.91**)	**0.54** (0.39)	**0.69** (0.47)	**0.64** (0.53)	**0.42** (0.26)	**0.42** (0.00)	0.59 (**0.96**)
	F_1	**0.66** (0.54)	0.56 (**0.60**)	0.66 (**0.69**)	**0.54** (0.42)	**0.63** (0.50)	**0.59** (0.52)	**0.53** (0.33)	**0.50** (0.00)	0.61 (**0.67**)
	Support	2952	6330	1134	2952	6330	1134	2952	6330	1134
Vulnerable										
	Precision	**0.65** (0.52)	0.57 (**0.59**)	0.65 (**0.75**)	**0.52** (0.45)	**0.62** (0.52)	**0.55** (0.50)	**0.58** (0.45)	**0.56** (0.50)	**0.61** (0.55)
	Recall	**0.73** (0.50)	**0.60** (0.49)	**0.59** (0.27)	**0.53** (0.52)	0.50 (**0.58**)	**0.90** (0.50)	**0.82** (0.65)	0.73 (**1.00**)	**0.91** (0.06)
	F_1	**0.69** (0.51)	**0.58** (0.53)	**0.62** (0.40)	**0.52** (0.48)	**0.56** (0.55)	**0.64** (0.50)	**0.68** (0.54)	0.63 (**0.67**)	**0.65** (0.10)
	Support	2952	6330	1134	2952	6330	1134	2952	6330	1134
Overall										
	Accuracy	**0.67** (0.52)	**0.57** (**0.57**)	**0.64** (0.59)	**0.58** (0.45)	**0.60** (0.53)	**0.55** (0.51)	**0.62** (0.45)	**0.58** (0.50)	**0.62** (0.51)
	Macro Precision	**0.68** (0.52)	**0.57** (**0.57**)	0.64 (**0.65**)	**0.65** (0.45)	**0.60** (0.53)	**0.55** (0.51)	**0.64** (0.45)	0.58 (**0.75**)	**0.62** (0.53)
	Macro Recall	**0.67** (0.52)	**0.57** (**0.57**)	**0.64** (0.59)	**0.68** (0.46)	**0.60** (0.53)	**0.55** (0.51)	**0.62** (0.45)	**0.58** (0.50)	**0.62** (0.51)
	Macro F_1	**0.67** (0.52)	**0.57** (**0.57**)	**0.64** (0.54)	**0.53** (0.45)	**0.59** (0.53)	**0.55** (0.51)	**0.60** (0.43)	**0.57** (0.33)	**0.62** (0.38)
	Weighted Precision	**0.68** (0.52)	**0.57** (**0.57**)	0.64 (**0.65**)	**0.65** (0.45)	**0.60** (0.53)	**0.55** (0.51)	**0.64** (0.45)	0.58 (**0.75**)	0.62 (**0.75**)
	Weighted Recall	**0.67** (0.52)	**0.57** (**0.57**)	**0.64** (0.59)	**0.58** (0.49)	**0.60** (0.53)	**0.55** (0.51)	**0.62** (0.45)	**0.58** (0.45)	**0.62** (0.51)
	Weighted F_1	**0.67** (0.52)	**0.57** (**0.57**)	**0.64** (0.55)	**0.53** (0.45)	**0.59** (0.53)	**0.55** (0.51)	**0.60** (0.43)	**0.57** (0.33)	**0.62** (0.39)
	Support	5904	12660	2268	5904	12660	2268	5904	12660	2268

LLaMA 3B shows mixed results. It improves on some CWEs but drops on others such as *CWE-20* and *CWE-190*, suggesting that GRPO sometimes over-adjusts when the base model is already performing well. LLaMA 8B shows little to no improvement of *DiverseVul* and even some small decreases (e.g. *CWE-787*, *CWE-476*). This suggests that the SFT allows the model to memorize patterns during training, while the abstract reasoning of GRPO is not able to achieve this level of memorization.

On *BigVul*, GRPO consistently outperforms SFT across all CWEs and model sizes. Among the evaluated models, Qwen 2.5 exhibits the most significant overall improvement, highlighting the particular advantage of GRPO for smaller models. Similarly, LLaMA 8B follows this positive trend, while LLaMA 3B achieves its most notable gains on *CWE-476*.

To summarize the results in this comparative analysis between GRPO and SFT, we conclude the following:

RQ3 *In what ways does GRPO differ from (and potentially improve upon) traditional supervised finetuning for code vulnerability detection?*

A3 *GRPO consistently improves overall performance, enhances recall on non vulnerable code, and maintains robustness under distribution shift, including on unseen programming languages.*

Table 30: GRPO vs. SFT on CleanVul dataset for languages different from training language C

Category	Metrics	Qwen 2.5			LLaMA 3B			LLaMA 8B		
		JS	Py	Ja	JS	Py	Ja	JS	Py	Ja
Not Vulnerable										
	Precision	0.61 (**0.76**)	**0.70** (0.49)	0.69 (**0.85**)	**0.62** (0.59)	0.69 (**0.71**)	**0.74** (0.63)	**0.62** (0.55)	**0.72** (0.68)	**0.72** (0.69)
	Recall	**0.77** (0.20)	**0.56** (0.09)	**0.53** (0.02)	**0.65** (0.09)	**0.74** (0.25)	**0.72** (0.04)	**0.49** (0.22)	**0.44** (0.10)	**0.46** (0.03)
	F_1	**0.68** (0.32)	**0.62** (0.15)	**0.60** (0.05)	**0.63** (0.16)	**0.71** (0.37)	**0.73** (0.07)	**0.55** (0.32)	**0.55** (0.17)	**0.56** (0.06)
	Support	1229	1495	3040	1229	1495	3040	1229	1495	3040
Vulnerable										
	Precision	**0.46** (0.44)	**0.47** (0.37)	0.27 (**0.29**)	**0.45** (0.41)	**0.53** (0.41)	**0.37** (0.29)	**0.43** (0.40)	**0.45** (0.40)	**0.30** (0.29)
	Recall	0.29 (**0.91**)	0.63 (**0.85**)	0.42 (**0.99**)	0.42 (**0.91**)	0.47 (**0.84**)	0.40 (**0.95**)	0.56 (**0.74**)	0.73 (**0.93**)	0.57 (**0.96**)
	F_1	0.36 (**0.59**)	**0.54** (0.52)	0.33 (**0.45**)	0.43 (**0.56**)	0.50 (**0.55**)	0.38 (**0.44**)	0.49 (**0.52**)	**0.56** (0.55)	0.40 (**0.45**)
	Support	976	971	1244	976	971	1244	976	971	1244
Overall										
	Accuracy	**0.57** (0.49)	**0.58** (0.39)	**0.50** (0.31)	**0.56** (0.42)	**0.64** (0.48)	**0.62** (0.30)	**0.52** (0.43)	**0.55** (0.42)	**0.49** (0.30)
	Macro Precision	0.54 (**0.60**)	**0.59** (0.43)	0.48 (**0.57**)	**0.53** (0.50)	**0.61** (0.56)	**0.56** (0.46)	**0.53** (0.47)	**0.59** (0.54)	**0.51** (0.49)
	Macro Recall	0.53 (**0.56**)	**0.59** (0.47)	0.48 (**0.51**)	**0.53** (0.50)	**0.61** (0.54)	**0.56** (0.49)	**0.53** (0.48)	**0.58** (0.51)	**0.51** (0.50)
	Macro F_1	**0.52** (0.46)	**0.58** (0.34)	**0.46** (0.25)	**0.53** (0.36)	**0.61** (0.46)	**0.56** (0.25)	**0.52** (0.42)	**0.55** (0.36)	**0.48** (0.25)
	Weighted Precision	0.55 (**0.63**)	**0.61** (0.45)	0.57 (**0.69**)	**0.55** (0.52)	**0.63** (0.59)	**0.63** (0.53)	**0.54** (0.49)	**0.61** (0.57)	**0.60** (0.57)
	Weighted Recall	**0.57** (0.49)	**0.58** (0.39)	**0.50** (0.31)	**0.56** (0.42)	**0.64** (0.48)	**0.62** (0.30)	**0.52** (0.43)	**0.55** (0.42)	**0.49** (0.30)
	Weighted F_1	**0.55** (0.43)	**0.59** (0.29)	**0.52** (0.17)	**0.55** (0.32)	**0.63** (0.44)	**0.63** (0.18)	**0.52** (0.40)	**0.55** (0.32)	**0.51** (0.17)
	Support	2205	2466	4284	2205	2466	4284	2205	2466	4284

10.8 Ablation Studies

In this section, we aim to provide ablations studies and evaluate how effectively training with GRPO has enhanced the models reasoning capabilities.

10.8.1 Linking Model Reasoning to CWE Meaning

We want to know whether the explanation the model writes in **Step 2** in the System Prompt for *Reasoning Test*(*"Check thoroughly for vulnerabilities ..."* in Fig. 39) actually matches the CWE weakness present in the code. For every test sample we embed that sentence with `MiniLM-L6- v2` and compare it, via cosine similarity, to the official CWE description taken from the MITRE site[29]. A higher similarity means that it is more likely that the model suggests the correct vulnerability than that it provides a superficially valid explanation. The score distributions for GRPO and SFT are plotted in Fig. 44. To see whether the two curves differ in a statistically meaningful way we apply a two-sample Kolmogorov–Smirnov test (KS) [242] with a 0.01 significance level. Table 31 summarises the outcome for the three most frequent CWEs in the *DiverseVul* split. Qwen 2.5 and LLaMA 8B show higher similarities for every CWE, and the KS test confirms that GRPO moves the whole distribution, not just the average. LLaMA 3B improves on CWE-20, stays unchanged on CWE-787, and drops slightly on CWE-125, reflecting its smaller capacity and more limited exposure to security code during pre-training. In all

[29] https://cwe.mitre.org/

cases the similarity scores become less spread out, indicating that GRPO makes the model's explanations more consistently aligned with the correct weakness.
Because the models never saw explicit CWE labels during training, these gains must come from better use of prior knowledge rather than simple memorisation. GRPO therefore does more than tweak predictions: it improves the semantic faithfulness of the reasoning itself, pushing the model to ground its explanations in concepts that match the real CWE definitions.

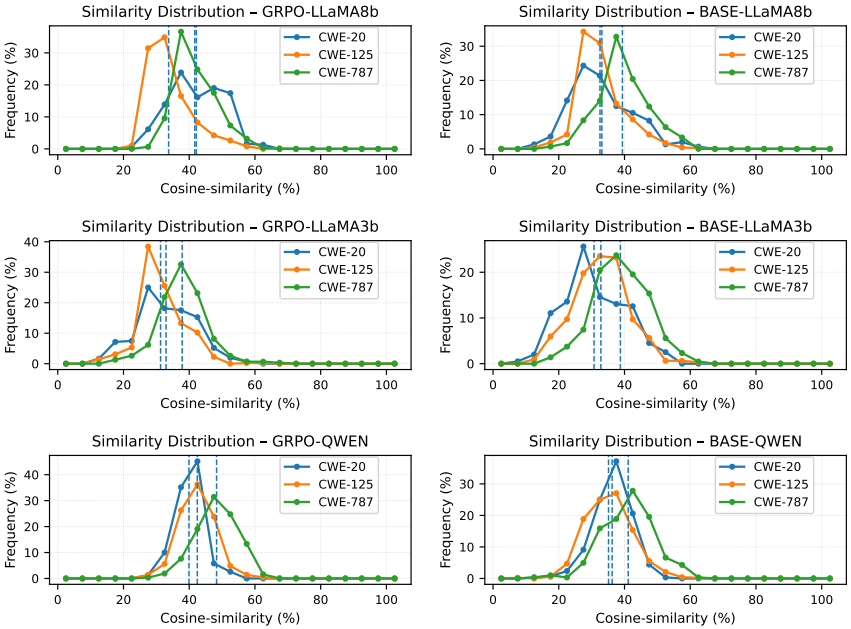


Figure 44: Cosine-similarity distribution for the 10 CWE categories, comparing the GRPO and BASE models on the three architectures (LLaMA 8B, LLaMA 3B, Qwen).

10.8.2 Concise and Correct: Analyzing Reasoning Length

Figure 46b shows the number of tokens used in the <reasoning> section for each test example where the models trained with GRPO and SFT provided the correct prediction. The **x-axis** represents the index of each correct example, while the

y-axis indicates the length of the explanation in tokens (we used BERT tokenizer [30] for each model). For both datasets, DiverseVul (top) and BigVul (bottom), we observe a consistent pattern: models finetuned with GRPO produce significantly shorter explanations compared to their baseline versions (SFT).

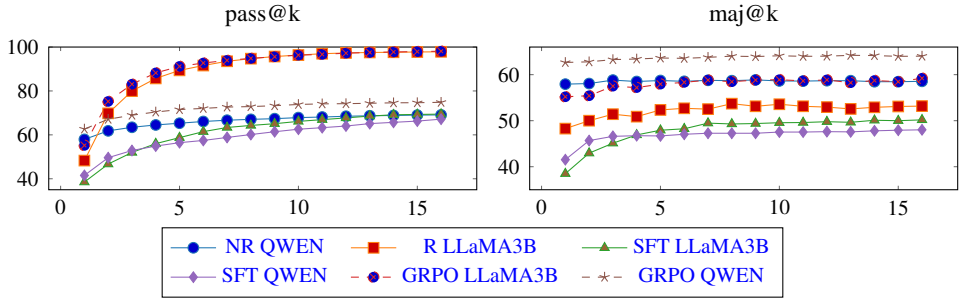


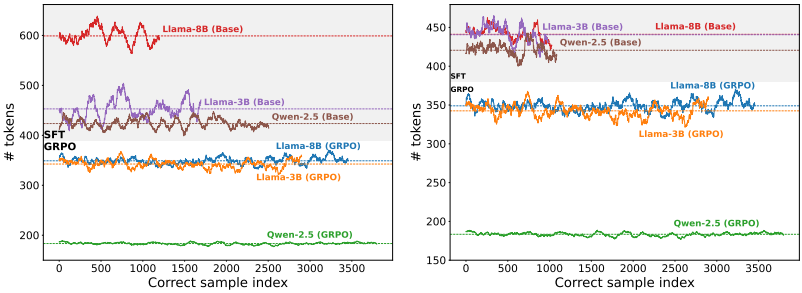
Figure 45: Comparison on BigVul: pass@k (left) and maj@k (right) curves for different models.

On DiverseVul, for example, LLaMA 8B (Base) produces reasonings with an average of around 600 tokens, while the GRPO variant reduces this to around 340 tokens. A similar trend applies to all other models. As only correct predictions are considered, this reduction in length suggests that GRPO encourages more focused and efficient reasoning, the same technical insight is conveyed in fewer words. This conciseness is particularly valuable in real-world security operations, where clarity and response time are critical.

10.8.3 Impact of KL Regularization on GRPO

To better understand the influence of KL regularization on model behaviour, we perform a controlled ablation in which we vary the parameter β that regulates the KL divergence weight in the GRPO loss. Following preliminary experiments and consistent with evidence reported in related works [243, 244], we focus on two representative values, $\beta = 10^{-4}$ and $\beta = 10^{-6}$. These values were chosen as they capture the trade-off between a higher regularization (which enforces conservative behaviour but risks under-predicting vulnerabilities) and minimal regularization (which promotes risk-taking but may reduce stability). Figure 47 illustrates that a higher β value leads to a more conservative model: the True Negative rate increases

[30] https://huggingface.co/google-bert/bert-base-uncased



(a) DiverseVul (b) BigVul

Figure 46: Answer length comparison between GRPO and SFT

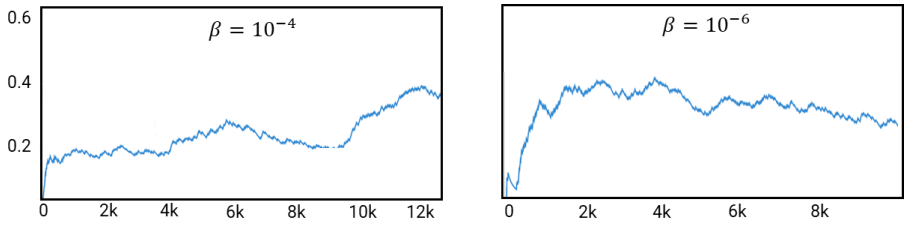
steadily over time, indicating that the model becomes more confident in identifying non-vulnerable code. However, this behaviour comes at a cost: the False Negative rate also rises, especially after the early training phase, showing that excessive regularization suppresses risk-taking and leads to under-prediction of vulnerabilities in borderline or ambiguous samples.Conversely, reducing β to 10^{-6} yields the opposite effect. While the model becomes less confident in rejecting benign samples (lower TN rate), it maintains a lower FN rate throughout training, demonstrating a more risk-taking attitude that prioritises vulnerability detection over avoiding false alarms. In our experiments, $\beta = 10^{-6}$ provided the most favourable trade-off, aligning with findings in [243] that address task beyond security.

10.8.4 Comparative Analysis on BigVul using *k* generations

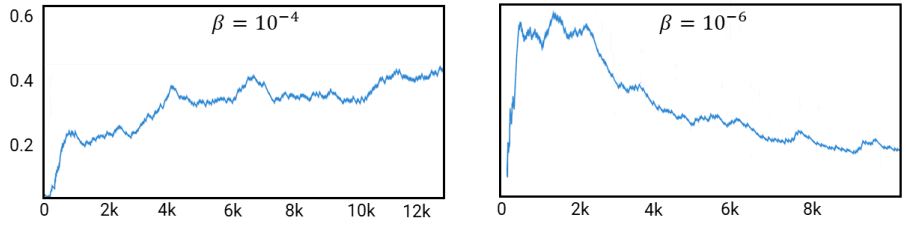
Figure 45 presents the results on the *BigVul* dataset, focusing on two complementary metrics: *pass@k* [245] (left panel), which measures the probability of obtaining a correct prediction within the top-k generations, and *maj@k* [246] (right panel), which evaluates the accuracy of the final decision obtained through majority voting across k sampled answers. Since these experiments are computationally expensive, as they require generating k candidate answers for each test instance, we restricted the evaluation to the two smaller models, *LLaMA 3B* and *Qwen 2.5*. We compare the models trained with GRPO, SFT and their corresponding best version between *Reasoning* and *No Reasoning* (Table 24).

Table 31: Comparison of BASE vs GRPO distributions for the top 3 CWE in DiveseVul using the Kolmogorov-Smirnov Test.

Metric	LLaMA 8B		LLaMA 3B		Qwen 2.5	
	BASE	GRPO	BASE	GRPO	BASE	GRPO
CWE-787						
Same Distribution?	No		Yes		No	
Mean	39.37	**41.69**	38.75	37.87	41.13	**48.33**
Std. dev.	**7.51**	6.31	8.29	7.00	**7.93**	6.28
CWE-20						
Same Distribution?	No		No		No	
Mean	33.11	**42.17**	30.73	**32.94**	36.23	**39.93**
Std. dev.	**9.30**	8.03	**9.49**	8.69	**5.66**	4.15
CWE-125						
Same Distribution?	No		No		No	
Mean	32.54	**33.73**	**32.81**	31.29	35.14	**42.46**
Std. dev.	**7.06**	6.70	**8.10**	7.03	**7.21**	5.36



(a) True negative rate analysis



(a) False negative rate analysis

Figure 47: Effect of β on **True Negative** and **False Negative** rates during training.

For *pass@k*, GRPO trained models show a clear advantage over both SFT and all

the other counterparts. In particular, *LLaMA 3B GRPO* achieves values close to 98% for $k = 16$, surpassing its SFT counterpart and the reasoning-only baseline, while *Qwen GRPO* steadily improves to more than 74%. Conversely, SFT models converge more slowly and remain significantly below the GRPO curves.
For *maj@k*, model differences are modest, ranging from 50% to 64%. The models trained with GRPO give the best results, lowering the variance seen in baselines. *Qwen GRPO* is the most reliable, consistently above 63%, while also *LLaMA 3B GRPO* stays clearly ahead of its counterparts.
These results on *BigVul* highlight that GRPO not only improves the probability of generating correct answers (*pass@k*), but also enhances the stability of aggregated decisions (*maj@k*), suggesting a positive effect on both single-sample accuracy and overall model consistency.

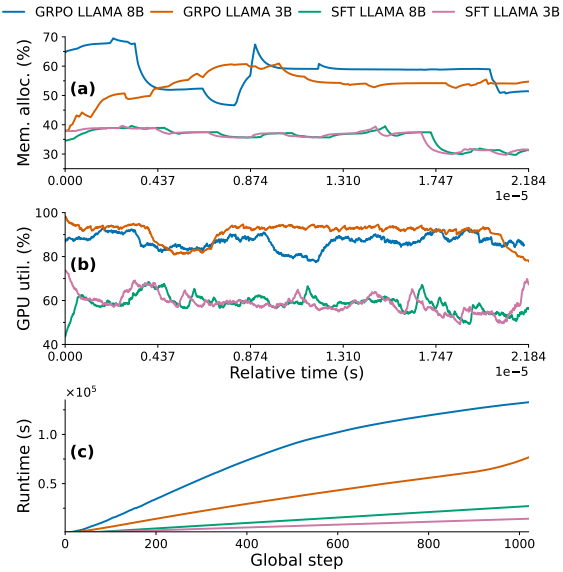


Figure 48: GPU efficiency metrics during training with GRPO and SFT on LLaMA-3B and LLaMA-8B: (a) GPU memory allocation over time, (b) GPU utilization, and (c) runtime across global steps.

10.8.5 Computational Efficiency Analysis

To complement the evaluation of detection performance, we analyze the computational footprint of the models trained with GRPO versus SFT across GPU memory, utilization, and runtime, as shown in Figure 48. The analysis includes both *LLaMA*

3B and *LLaMA 8B* to capture scale effects, while *Qwen 2.5 3B*, given its similarity in size to *LLaMA 3B*, is expected to show comparable computational patterns.Panel (a) shows the percentage of GPU memory allocated during training. GRPO stabilizes around 60–70% of the available GPU memory for both 3B and 8B, whereas SFT stays around 30–40%. This reflects GRPO's heavier sampling/optimization step, yet it remains well within device capacity and caused no Out Of Memory in our runs. Panel (b) illustrates GPU utilization over relative time. GRPO consistently achieves higher utilization compared to SFT, especially for the 8B variant. This suggests that GRPO better saturates GPU resources, thereby improving hardware efficiency during optimization. Panel (c) compares the total runtime as a function of the number of global steps. While GRPO requires a modestly longer runtime per step, the difference remains around 0.5×10^5 seconds compared to SFT for both LLaMA 3B and LLaMA 8B. Given the performance improvements reported in previous sections, this computational overhead can be considered negligible relative to the accuracy gains. In summary, GRPO delivers superior detection performance with higher memory usage, higher GPU utilization, and only a modest runtime increase, making it a computationally viable choice for vulnerability detection.

10.8.6 Statistical Analysis with Chi-square Test

We evaluated the effect of GRPO on different datasets using the Chi-square test applied to the distributions of True Positives, True Negatives, False Positives, and False Negatives (Table 32). Let O_i and E_i denote the observed and expected frequencies for class i.The Chi-square statistic is then defined as: $\chi^2 = \sum_i \frac{(O_i - E_i)^2}{E_i}$, The p-value is obtained by comparing χ^2 to the chi-square distribution with $df = 3$, corresponding to the four outcome categories (True Negative, False Negative, True Positive, and False Positive) minus one. A p-value below $\alpha = 0.05$ indicates that the observed differences are statistically significant at the 95% confidence level. Table 32 shows that GRPO significantly changes the distributions compared to zero-shot and SFT baselines across CWE categories in *DiverseVul* and *BigVul*, except for a few cases (LLama on CWE-119, CWE-190, CWE-400, CWE-78 in *DiverseVul*, and CWE-400 in *BigVul*). Similarly, in the *CleanVul* dataset, GRPO consistently alters the error distributions across programming languages, with all p-values below 0.01, providing strong evidence that the improvements are both statistically significant and practically meaningful. Overall, these results confirm that GRPO reliably improves performance and modifies error profiles across datasets and languages.

Table 32: Chi-square test results over Fig. 43 and Tab. 24, 29, 30 with p-values ≥ 0.01. Cells in gray correspond to p-values ≥ 0.05.

Dataset	Model	CWE	P-value	Comparison
DiverseVul	LLaMA 8B	119	0.06	Fig. 43
DiverseVul	LLaMA 8B	416	0.06	Fig. 43
DiverseVul	LLaMA 8B	190	0.19	Fig. 43
DiverseVul	LLaMA 8B	400	0.82	Fig. 43
DiverseVul	LLaMA 8B	78	0.72	Fig. 43
BigVul	LLaMA 8B	787	0.02	Fig. 43
BigVul	LLaMA 8B	190	0.01	Fig. 43
BigVul	LLaMA 8B	400	0.06	Fig. 43
BigVul	LLaMA 3B	787	0.04	Fig. 43
BigVul	LLaMA 3B	125	0.04	Fig. 43
BigVul	Qwen	400	0.02	Fig. 43

10.9 Conclusion

This work revisited vulnerability detection with LLMs from a reasoning-aware perspective, focusing on three research questions (RQ1–RQ3). For RQ1, the results show that off-the-shelf instruction-tuned models, such as LLaMA-8B, struggle to use their own reasoning effectively. Even when prompted to "think aloud," their predictions remain inconsistent and sensitive to domain shifts.

For RQ2, a GRPO-based training approach was introduced using a modular and adaptive reward that balances formatting, correctness, and reasoning quality while reducing reward hacking over time. Regarding RQ3, GRPO was compared with standard supervised fine-tuning (SFT) on three benchmarks, showing consistent improvements: macro-F1 increased by 1–29 points and accuracy by 4–17 points. The GRPO models also generalized better across programming languages, vulnerability types, and datasets.

In summary, GRPO helps small language models reason more effectively about software vulnerabilities, producing predictions that are both more accurate and easier to interpret, without needing explicitly supervised rationales.

11 Concise Thought: Impact of Output Length on LLM Reasoning and Cost

After developing *task-specific* modules (misinformation stress testing, translation of natural language commands into enforceable policies, vulnerability detection with behavior-aware alignment), our focus shifts from the *what* LLMs can do to the *how* they reason. In many real-world scenarios, LLM-based systems exhibit a well-known paradox: accuracy tends to improve when models produce longer, step-by-step rationales (e.g., Chain-of-Thought), but this comes at the cost of higher computational overhead, latency, and unpredictable variability. Verbosity often introduces redundancy, increases the risk of contextual hallucinations, and makes the overall system less controllable. Understanding the impact of *output length* on both reasoning quality and inference cost is therefore critical to optimize the trade-off between correctness, conciseness, and efficiency.
This motivates the fourth research question that guides the final part of the thesis:

Research Question (Q4). *How can we optimize the reasoning process of LLMs to produce answers that are* ***correct*** *and* ***concise****, with* ***predictable costs and latency****, while avoiding instability and collapse phenomena during alignment?*

This work provides a systematic analysis of the trade-off between conciseness and accuracy. Specifically, we examine: (i) the relationship between output length, correctness, and faithfulness of reasoning; (ii) the predictability of cost and latency under constrained budgets; (iii) strategies for controlling verbosity (length-aware prompting and decoding, constrained explanation formats, extractive or selective rationales) and their impact on reasoning quality; and (iv) operational metrics for balancing *quality-of-reasoning* and *quality-of-service*. The ultimate goal is to move from *"longer reasoning to be correct"* toward *"reasoning* as concise as necessary *to be correct,"* paving the way for the optimization techniques explored in the following chapter.
The output generation time of an LLM depends on various factors, including the model architecture, the pre-and post-processing steps, the answer decoding process, and the question posed, also considering the use of prompt engineering approaches. While the computational cost due to the architecture is well understood, the influence of the other aspects on the overall generation time is less clear and requires further investigation. More formally, an LLM can be represented as a function f that takes as input a prompt x with $\mathcal{N}(x)$ tokens[31] and generates an output $\hat{y} = f(x)$, having

[31] Even though 'tokens' and 'words' refer to different items in the sentence, for simplicity, in this

$\mathcal{N}(\hat{y})$ tokens, where $\mathcal{N}$ is a length operator that simply counts the number of tokens. The input x can be considered as composed of the original user input x_{us} and a prompt engineering text x_p, depending on the technique used. For instance, in a zero-shot CoT setting, the prompt can be computed as $x = concat(x_{\text{us}}, x_p)$, where x_p is an explicit request for providing reasoning steps in the answer and $concat(a, b)$ is the concatenation operator that merges two vectors a and b into a single one.
In an encoder-decoder architecture, as the one used by Transformers [173], let $f_e(x)$ and $f_d(x)$ denote the functions associated with the encoder and the decoder, respectively. Then, the output $\hat{y}$ is a list of tokens $[a^{(1)}, \ldots, a^{(\mathcal{N}(\hat{y}))}]$, where each $a^{(i)}$ is computed based on the previously generated tokens and the encoder's embedding representation $f_e(x)$. That is,

$$a^{(i)} = f_d(f_e(x), [a^{(0)}, \ldots, a^{(i-1)}]), \quad i > 0. \tag{18}$$

Equation (18), it is clear that the larger the set of output tokens in the answer, the higher the time the model takes to generate the answer due to the increased number of times the decoder is invoked. The same consideration could also be applied to decoder only, where the model runs a new inference for each produced word.
To highlight such a dependency, we conducted preliminary tests on Falcon-40B to evaluate the impact of the CoT method in answering arithmetic questions, using a subset of 100 random questions from the GSM8K dataset [247]. The results of this test are illustrated in Figure 49a, where red and blue dots refer to answers given with and without CoT, respectively. The scatter plot shows that CoT significantly increases the output length and generation time. This suggests that while CoT improves the correctness of responses (see Section 11.4), more attention should be given to the time cost it introduces. To better appreciate the impact of CoT on the output length, Figure 49b reports the output length (in terms of number of generated words) produced by Falcon-40b on a set of 50 questions from GSM8K without CoT (blue bars) and with CoT (pink bars). Note that purple areas denote the areas where the two bars overlap.

work we will refer to both indistinguishably.

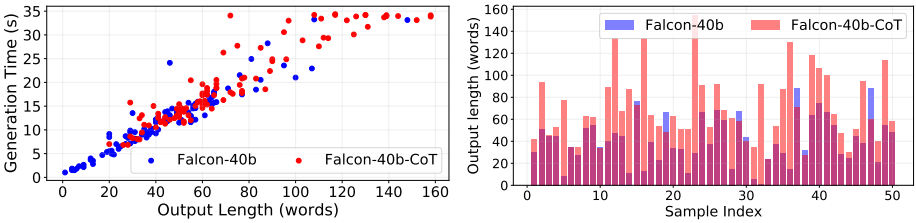


Figure 49: Analysis of the impact of CoT on Falcon-40b efficiency: (top) Relation between response time and output length, without CoT (blue dots) and with CoT (red dots), across 100 questions from the GSM8K test set. (bottom) Output words variation between output length with CoT and without CoT using 50 random samples from the GSM8K test set.

11.1 Metrics for Correct Conciseness

Motivated by the previous considerations, this section presents three novel metrics to evaluate the capability of an LLM to provide *correct* as well as *concise* responses. The idea is to redefine the classic accuracy metric to integrate conciseness aspects, which as highlighted in Section 11.4.3 impacts significantly the generation time and the computational cost, into the LLM output's correctness. Formally, an answer $\hat{y}$ is considered correct if the conclusion extracted through a post-processing function Γ matches the given ground truth y. Thus, the accuracy of an LLM can be computed as

$$\mathcal{A} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}(\Gamma(\hat{y}), y), \tag{19}$$

where N is the number of tested samples and $\mathbb{1}(u, v)$ is the indicator function that returns 1 if $u = v$, 0 otherwise. Please note that Γ represents a user-defined function that can be implemented based on a regular expression (e.g., by extracting specific patterns from the sentence [248]) or using pseudo-judge approaches (e.g., by using a secondary large model as a judge [172]).
Starting from Equation (19), the conciseness of an output $\hat{y}_i$ can be integrated with its correctness by multiplying the indicator function by a penalty term $p(\hat{y}_i) \in [0, 1]$ that decreases its value for long outputs:

$$\frac{1}{N} \sum_{i=1}^{N} \left[\mathbb{1}(\Gamma(\hat{y}_i), y_i) \cdot p(\hat{y}_i) \right]. \tag{20}$$

The following defines three specific metrics by setting a proper penalty function.

Hard-k Concise Accuracy: HCA(k). It measures the fraction of correct outputs that do not exceed a user-specified length k:

$$\text{HCA}(k) = \frac{1}{N}\sum_{i=1}^{N}\left[\mathbb{1}(\Gamma(\hat{y}_i), y_i) \cdot p_{hard}(\hat{y}_i, k)\right],$$

where

$$p_{hard}(\hat{y}_i, k) = \begin{cases} 1 & \text{if } \mathcal{N}(\hat{y}_i) \leq k \\ 0 & \text{otherwise.} \end{cases} \tag{21}$$

This metric does not account for responses that exceed the specified maximum length, thereby promoting conciseness. We believe it could be particularly useful in scenarios where strict adherence to length constraints is essential, such as in real-time systems or environments with limited computational resources.

Soft-k Concise Accuracy: SCA(k, α). It generalizes the previous metric by penalizing the correct answers that exceed the maximum length k with a term that decreases exponentially with a decay factor α:

$$\text{SCA}(k, \alpha) = \frac{1}{N}\sum_{i=1}^{N}\left[\mathbb{1}(\Gamma(\hat{y}_i), y_i) \cdot p_{soft}(\hat{y}_i, k, \alpha)\right],$$

where

$$p_{soft}(\hat{y}_i, k, \alpha) = \min\left(1, e^{\frac{k-\mathcal{N}(\hat{y}_i)}{\alpha}}\right). \tag{22}$$

In the formula, the user-defined decay $\alpha \geq 0$ can be considered a sort of tolerance that controls how much the length impacts the overall accuracy; the higher the value of α, the higher the tolerance for answers exceeding the specified length k. Note that for $\alpha = 0$, SCA$(k, 0)$ reduces to HCA(k).

Consistent Concise Accuracy: $CCA(k, \alpha, \beta)$. It further generalizes the previous metrics by also accounting for the variation in the lengths among all the outputs obtained:

$$CCA(k, \alpha, \beta) = SCA(k, \alpha) \cdot p_{var}(\sigma, \beta)$$

where

$$p_{var}(\sigma, \beta) = \min\left(1, e^{\frac{\beta-\sigma}{\beta}}\right). \tag{23}$$

In Equation (23), σ denotes the standard deviation of the output length distribution, whereas β is a parameter that controls the tolerance for having large length variations:

the higher the value of β, the higher the tolerance. Note that, given a tolerance β, $p_{var}(\sigma, \beta) = 1$ for $\sigma \leq \beta$, while it decreases exponentially for $\sigma > \beta$.
The CCA metric aims to promote consistency in the lengths of responses, representing an important measure when the predictability of inference time is a crucial property for a specific application. A low standard deviation σ indicates that the model produces responses of uniform length. In contrast, a high value of σ denotes a model with a large response variability, making predicting its timing response time difficult.

CoT → output length 67 words; **inference time 17.64s**

Q: Tracy used a piece of wire 4 feet long to support tomato plants in the garden. The wire was cut into pieces 6 inches long. How many pieces did she obtain? **Let's think a bit step by step**
A: First, we need to convert 4 feet to inches. There are 12 inches in a foot, so 4 feet equals 4 × 12 = 48 inches. Next, we can divide the total number of inches by the length of each piece (6 inches) to find the total number of pieces: 48 inches ÷ 6 inches = 8 pieces. Therefore, Tracy obtained 8 pieces of wire.

CCoT-45 → output length 34 words; **inference time 11.65s**

Q: Tracy used a piece of wire 4 feet long to support tomato plants in the garden. The wire was cut into pieces 6 inches long. How many pieces did she obtain? **Let's think a bit step by step and limit the answer length to 45 words.**
A: 1. Convert 4 feet to inches: 4 × 12 = 48 inches 2. Divide 48 inches by 6 inches per piece: 48 ÷ 6 = 8 pieces. So, Tracy obtained 8 pieces of wire.

Figure 50: Example of CCoT in a zero-shot setting for a question extracted from the GSM8K dataset. In each box, we show the prompt and answer provided by the LLM. The box above shows the classic CoT zero-shot approach, where long reasoning is returned as output. The box below shows the use of CCoT, which reduces the number of output words while maintaining a correct answer. In the box title, we also show the word count along with inference time to clarify the improvements of the approach.

11.2 CCoT Prompting

From the results presented in Section 11, it is clear that the relationship between output length and inference time necessitates deeper awareness. To this end, this section focuses on improving the use of CoT, aiming to preserve the benefits of this technique while paying more attention to the length of the answers. This help

achieve a better trade-off between efficiency and accuracy.
For this purpose, we introduce a constrained chain of thought (CCoT) prompt, which includes an explicit sentence to constrain the generated output to a maximum number of words, encouraging the model to compress its reasoning and produce a more concise answer in a reduced amount of time. As explained in Section 11, CoT-prompt can be computed as $x = concat(x_{\text{us}}, x_p)$, where x_p is an explicit request for providing reasoning steps in the generated answer (e.g., *"let's think step by step"*). Technically, to push LLMs to return more concise reasoning, the CCoT-prompt is formalized as $x = concat(x_{\text{us}}, x_p, x_l)$, where x_l represents the sentence that specifies the constraint on the output length (e.g., *"and limit the length of the answer to 30 words"*). Figure 50 shows an example that illustrates the difference between a CoT and a CCoT prompt. Note that the answer generated for that specific question using a CoT prompt consists of 67 words in 17.64s, while the answer generated on the same question provided with a CCoT prompt (specifying a constraint of 45 words) consists of 34 words in 11.65s, and it is still correct.
The analysis presented in the following section and experimental part in Section 11.4 provide a detailed evaluation of the CCoT prompting technique using various metrics and demonstrate its benefits on the conciseness along with redundancy and information flow scores of the generated answers.

11.3 Analysis of the conciseness

To provide a deeper understanding and more comprehensive comparisons of answers in terms of conciseness, we define some conciseness properties for a given answer in the following. Intuitively, we assume that the conciseness of an answer in the CoT paradigm could be derived from the following two concepts: *how many steps are required to produce a response* and *how much information is repeated in successive steps*. While the first property is easy to derive by immediately looking at the number of steps, assuming to have a similar number of steps from two answers, extracting a numerical interpretation of the conciseness could be more complicated. For this purpose, we define two additional scores: *Redundancy*, which measures the extent to which individual steps in a generated answer repeat or overlap in *synthetical* content, and *Information Flow*, which measures how much the *semantic* information of the previous step is repeated in the current step. To compute these two scores, both CCoT and CoT answers are divided into discrete steps, based on sentence tokenization[32].

[32] https://www.nltk.org/api/nltk.tokenize.sent_tokenize.html

11.3.1 Redundancy Score

The redundancy score for each step represents how much its content overlaps with other steps. Let a generated answer $\hat{y}$ be divided into n steps $S = \{s_1, s_2, \ldots, s_n\}$, where each s_i represents a single step (sentence). The redundancy mean score $RMS(\hat{y})$ is computed as:

$$RMS(\hat{y}) = \frac{1}{n \cdot (n-1)} \sum_{i}^{n} \sum_{j \neq i}^{n} \mathrm{SyS}(s_i, s_j), \tag{24}$$

where $\mathrm{SyS}(s_i, s_j)$ measures the syntactical similarity between steps s_i and s_j as the ratio of matching subsequences. This is computed as $\mathrm{SyS}(s_i, s_j) = \frac{\mathrm{LMS}(s_i, s_j)}{\mathrm{TLCS}(s_i, s_j)}$, where LMS states for *Length of Matching Subsequences* and TLCS states for *Total Length of Compared Sequences* [249].
These scores help to evaluate whether the reasoning process is concise or contains redundant synthetical repetition.
We assume that for not concise answers we have a high redundancy, which indicates that steps are overly reliant on repeating information rather than advancing the reasoning.

11.3.2 Information Flow

The *Information Flow* $\mathcal{I}(s_i, s_{i+1})$ between two consecutive steps s_i and s_{i+1} measures the extent to which the content of s_i, in a semantic sense, persists in s_{i+1}. This can be expressed as:

$$\mathcal{I}(s_i, s_{i+1}) = SeS(s_{i+1}, s_i), \tag{25}$$

where $SeS(s_{i+1}, s_i)$ represents the semantic similarity between s_i and s_{i+1}.
In our implementation, we used *BERTScore* [250, 174] to compute the semantic similarity between steps, which leverages contextual embeddings generated by BERT. For the last step, since there is no subsequent step, we assume $\mathcal{I}(s_n) = 0$.
The rationale behind this score is that, when the number of steps is similar, the information flow across the steps tends to be higher in less concise answers because each step relies on a more similar set of details and context from the previous one, increasing the risk of repetitiveness.

11.4 Experiments

This section presents a set of experiments carried out to evaluate the effectiveness of the proposed CCoT approach under classic metrics, as well as illustrate the benefits of the proposed metrics and scores in highlighting the trade-off between accuracy and computational cost. Specifically, the following research questions are investigated in the next experiments: RQ1 Is the CCoT approach beneficial in terms of efficiency and accuracy compared to classic CoT?; RQ2 Are the proposed metrics representative of showing this trade-off?; and RQ3 Are the CCoT answers actually more concise, and to what extent are LLMs capable of controlling the output length based on an explicit prompt request?

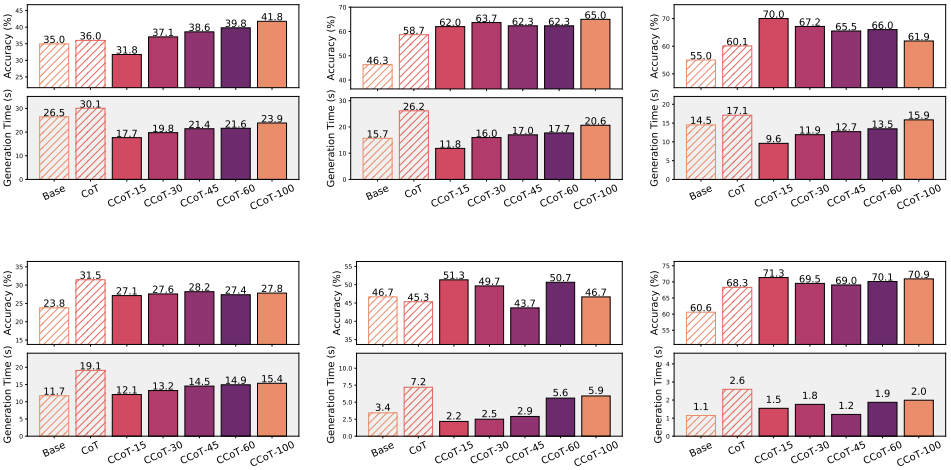
11.4.1 Experimental setup

All the experiments have been carried out with the Text Generation Inference (TGI) platform[33] on 8 NVIDIA A100 GPUs. Specifically, we evaluated large and open source pre-trained LLMs from Hugging Face[34], such as instruction-tuned models Falcon-40b-instruct and model trained, reinforced by utilizing private data, namely Llama2-70b-chat-hf [190]. Additionally, further experiments on smaller models were conducted to evaluate their capability in handling CCoT.
The experiments utilized three arithmetic reasoning datasets: GSM8k [247], SVAMP [251], and ASDIV [252], commonly used to assess models' mathematical inference and computational reasoning abilities. The GSM8k test set includes over 1.3k problems out of 8,000. The SVAMP test set contains 300 examples, and the ASDIV test set comprises 1.22k examples.
For all experiments, the effectiveness of CCoT was compared by assessing the selected LLMs both with and without CoT (base mode).

11.4.2 Impact of CCoT on Accuracy and Efficiency

This experiment was carried out to evaluate the impact of *CCoT* on computation time and accuracy (RQ1). In particular, the selected LLMs were evaluated on three datasets using plain prompt (base), *CoT*, and *CCoT* with different length constraints:15, 30, 45, 60, 100. The results are presented in Figure 51 for Llama2-70b (top) and for Falcon-40b (bottom), showing accuracy and generation time in the first and second rows, respectively.

[33] https://huggingface.co/docs/text-generation-inference
[34] https://huggingface.co/blog/os-llms



(a) GSM8k (b) SVAMP (c) ASDIV

Figure 51: Accuracy (white plots, the higher the better) and mean generation time (background colored plots, the lower the better) for Llama2-70b (top row) and Falcon-40b (bottom row) on the GSM8K, SVAMP, and ASDIV datasets. The models are evaluated using plain prompts (base), CoT, and CCoT under different len-constraints.

Considering the results with Llama2-70b, CCoT prompting demonstrates the ability to reduce generation time compared to CoT and, in most cases, achieves a time reduction similar to or better than plain prompting (base). Additionally, it generally improves or minimally impacts accuracy, thereby enhancing the trade-off in both directions. For instance, the average generation time decreases from 30.09 seconds with CoT to a maximum of 23.86 seconds with CCoT on GSM8K with Llama2-70b, achieved with a length constraint of 100, and further reduces with stricter constraints. At the same time, also the accuracy consistently improves, for example, with the GSM8k dataset, the accuracy of Llama2-70b increases from 36% with CoT to 37% (with CCoT-30) and 41.77% (with CCoT-100).

Similar observations can be made for the results with Falcon-40b (bottom part), where the CCoT approach improves the trade-off between efficiency and accuracy across all three datasets. In particular, CCoT achieves better accuracy overall for SVAMP and ASDIV, while significantly reducing generation time. This improvement in terms of accuracy, however, does not occur for the GSM8K dataset, where the sentences are more complex than those in the other two datasets. We believe that such a complexity makes it more challenging for a medium-sized model

like Falcon-40b to effectively handle the CCoT constraint. Nonetheless, the accuracy remains higher than the base mode, indicating that the CoT-based approach still provides benefits.
We also acknowledge that different behaviors may arise when dealing with datasets of varying complexity, as the nature of the questions can differ. For instance, CCoT-15 outperforms other CCoT variations because its 15-word responses potentially align better with the simpler nature of this dataset. We argue that this effect is closely related to the complexity of the questions, which makes the constraint impactful even on accuracy.

11.4.3 Analysis of the *correct conciseness*

Based on the previous results, CCoT demonstrates clear benefits in balancing accuracy and computational efficiency. To unify these aspects, we introduced new metrics in Section 11.1 that integrate answer length with correctness. The analysis presented in Tables 33 and 34 for Llama2-70b and Falcon-40b, respectively, highlights how these metrics effectively combine cost and accuracy (RQ2), while also remarking the advantages of CCoT.

***HCA* evaluation.** The *Hard-k concise accuracy* evaluates the accuracy considering only the correct answers whose length is less than a specified value k. The top parts of Tables 33 and 34 report the value of this performance index across the three datasets, when using the different prompt approaches and for different values of k. Specifically, for both Llama2-70b and Falcon-40b, the use of CCoT gets better results compared to base and CoT prompts across all values of k. Notably, for lower values of k, CoT prompts exhibit a significant reduction in performance, while this accuracy drop can be mitigated by using CCoT with strict length constraints, such as 15 or 30.

***SCA* evaluation.** We also evaluated both models using the *Soft Conciseness Accuracy (SCA)*, across different k values and α, where α represents a tolerance for accepting answers longer than the desired limit k. This metric is a generalization of the *HCA*, giving more flexibility in considering correct answers that are larger but still close to the desired length k.
The SCA values computed for Llama2-70b and Falcon-40b on the datasets are reported in center parts of the tables for different values of k and a fixed tolerance value $\alpha = 10$. For both models, the SCA values in CCoT settings are often comparable to HCA values for high values of k, such as 80 or 100. This is because,

	Base	CoT	CCoT15	CCoT30	CCoT45	CCoT60	CCoT100
GSM8K - HCA							
H-∞	35.0	36.0	31.8	37.1	38.6	39.8	**41.8**
H-100	29.9	22.9	31.2	35.3	37.5	38.7	**38.9**
H-80	22.0	15.4	29.2	31.8	33.1	**35.0**	31.6
H-40	4.8	0.8	**12.7**	10.8	8.0	8.5	4.5
SVAMP - HCA							
H-∞	46.3	58.7	62.0	63.7	62.3	62.3	**65.0**
H-100	46.3	50.0	59.7	61.0	61.0	59.7	**61.7**
H-80	46.3	41.0	56.7	**57.7**	57.0	54.3	54.7
H-40	23.3	12.0	**44.7**	34.0	30.0	30.0	20.0
ASDIV - HCA							
H-∞	52.8	60.3	**67.2**	65.6	64.3	65.0	61.4
H-100	52.8	60.3	**67.2**	65.6	64.3	65.0	61.4
H-80	52.8	60.3	**67.2**	65.6	64.3	65.0	61.4
H-40	31.7	23.6	**55.5**	46.4	44.4	43.8	29.5
GSM8K - SCA ($\alpha = 10$)							
SCA-100	31.4	26.9	31.5	36.3	38.1	39.2	**40.1**
SCA-80	25.6	19.0	30.1	33.8	35.3	**36.6**	35.0
SCA-40	8.5	3.3	**18.0**	16.9	15.4	16.3	11.0
SVAMP - SCA ($\alpha = 10$)							
SCA-100	46.3	52.3	60.3	61.6	61.3	60.6	**62.7**
SCA-80	46.3	45.6	58.4	**59.2**	59.1	57.6	58.1
SCA-40	31.4	19.5	**48.2**	41.1	39.6	38.7	31.6
ASDIV - SCA ($\alpha = 10$)							
SCA-100	52.8	60.3	**67.2**	65.6	64.3	65.0	61.4
SCA-80	52.8	60.3	**67.2**	65.6	64.3	65.0	61.4
SCA-40	39.5	35.2	**60.4**	54.0	52.1	51.7	40.9
GSM8K - CCA ($\alpha = 10, \beta = 40$)							
CCA-100	31.4	26.9	31.5	36.3	38.1	39.2	**40.1**
CCA-80	25.6	19.0	30.1	33.8	35.3	**36.6**	35.0
CCA-40	8.5	3.3	**18.0**	16.9	15.4	16.3	11.0
SVAMP - CCA ($\alpha = 10, \beta = 40$)							
CCA-100	46.3	52.3	60.3	61.6	61.3	60.6	**62.7**
CCA-80	46.3	45.6	58.4	**59.2**	59.1	57.6	58.1
CCA-40	31.4	19.5	**48.2**	41.1	39.6	38.7	31.6
ASDIV - CCA ($\alpha = 10, \beta = 40$)							
CCA-100	52.8	60.3	**67.2**	65.6	64.3	65.0	61.4
CCA-80	52.8	60.3	**67.2**	65.6	64.3	65.0	61.4
CCA-40	39.5	35.2	**60.4**	54.0	52.1	51.7	40.9

Table 33: Llama2-70b

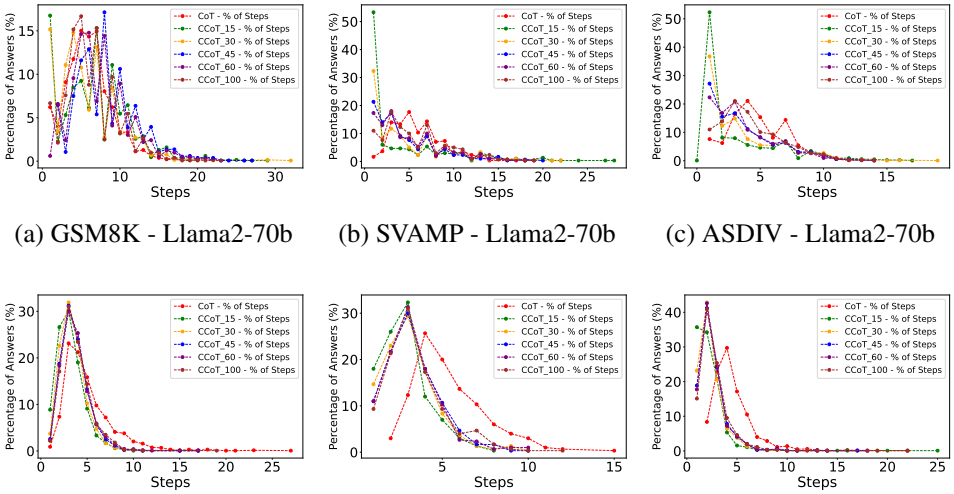
	Base	CoT	CCoT15	CCoT30	CCoT45	CCoT60	CCoT100
GSM8K - HCA							
H-∞	23.8	31.5	27.1	27.6	**28.2**	27.4	27.8
H-100	23.7	**29.3**	26.7	27.3	27.2	26.8	27.4
H-80	22.8	**26.8**	25.8	26.1	25.9	25.2	26.0
H-40	13.0	10.4	**13.4**	**13.4**	12.2	11.8	12.5
SVAMP - HCA							
H-∞	46.7	45.3	**51.3**	49.7	43.7	50.7	46.7
H-100	46.7	42.3	**51.3**	49.3	43.0	48.3	45.3
H-80	46.3	39.7	**49.7**	47.3	40.7	46.7	44.3
H-40	**38.7**	20.3	36.7	36.0	31.7	33.0	32.7
ASDIV - HCA							
H-∞	60.6	68.3	**71.3**	69.5	69.0	70.1	70.9
H-100	60.6	67.1	**71.2**	69.1	68.8	69.7	70.1
H-80	60.1	65.2	**70.9**	68.5	68.1	68.6	69.0
H-40	57.2	43.3	**64.9**	59.6	57.3	58.3	56.7
GSM8K - SCA ($\alpha = 10$)							
SCA-100	23.7	**30.1**	26.8	27.4	27.5	27.0	27.5
SCA-80	23.3	**28.1**	26.2	26.7	26.6	26.0	26.6
SCA-40	16.6	15.3	**18.0**	17.9	16.8	16.5	16.7
SVAMP - SCA ($\alpha = 10$)							
SCA-100	46.7	43.3	**51.3**	49.4	43.1	48.9	45.4
SCA-80	46.6	41.2	**50.7**	48.1	41.6	47.7	44.9
SCA-40	40.6	27.8	**42.7**	40.8	34.8	38.0	35.8
ASDIV - SCA ($\alpha = 10$)							
SCA-100	**84.4**	67.3	71.2	69.3	68.9	69.9	70.4
SCA-80)	**81.1**	66.1	71.0	68.8	68.5	69.2	69.5
SCA-40	66.0	50.4	**67.3**	63.3	61.5	61.9	61.2
GSM8K - CCA ($\alpha = 10, \beta = 40$)							
CCA-100	23.7	**30.1**	26.8	27.4	27.5	27.0	27.5
CCA-80	23.3	**28.1**	26.2	26.7	26.6	26.0	26.6
CCA-40	16.6	15.3	**18.0**	17.9	16.8	16.5	16.7
SVAMP - CCA ($\alpha = 10, \beta = 40$)							
CCA-100	46.7	43.3	**51.3**	49.4	43.1	48.9	45.4
CCA-80	46.6	41.2	**50.7**	48.1	41.6	47.7	44.9
CCA-40	40.6	27.8	**42.7**	40.8	34.8	38.0	35.8
ASDIV - CCA ($\alpha = 10, \beta = 40$)							
CCA-100	**84.4**	67.3	71.2	69.3	68.9	69.9	70.4
CCA-80	**81.1**	66.1	71.0	68.8	68.5	69.2	69.5
CCA-40	**66.0**	50.4	67.3	63.3	61.5	61.9	61.2

Table 34: Falcon-40b

as discussed in Sec.11.5, for such lengths, the CCoT prompts are effective at returning outputs below the desired limit, making the tolerance less necessary. Conversely, for smaller k values, such as $k = 40$, SCA starts exceeding HCA, indicating that some correct answers have a length larger than k. However, for such k values of, using a tolerance α results in more pronounced improvements for CCoT prompts compared to Base and CoT. This means that, although many correct outputs are longer than k, under CCoT the model is still encouraged to constrain them close to k, thus achieving a higher score. This effect is particularly noticeable on Llama2-70b, which is more capable of controlling the length and produce correct outputs than Falcon-40b.

***CCA* evaluation.** The *Consistent Concise Accuracy* measures the capability of a model to generate correct answers whose lengths do not vary significantly, and therefore are consistent with the specified constraint. The *CCA* requires a third parameter β (in addition to k and α), denoting a tolerance on the output length variability.

The bottom parts of the tables report the *CCA* scores obtained on Llama2-70b and Falcon-40b for $\alpha = 10$, $\beta = 40$, and different values of k, for the various prompting methods. According to these settings, the CCoTs results in a clear improvement compared to CoT prompting and base, for both Llama2-70b and Falcon-40b. However, for high CCoT length constraints (e.g., 100), the CCA score tends to decrease, which does not happen with the other two metrics. This can be explained by considering that an increased length constraint gives the model more freedom to generate outputs with higher variations, as discussed in Section 11.5.



(a) GSM8K - Llama2-70b (b) SVAMP - Llama2-70b (c) ASDIV - Llama2-70b

(d) GSM8K - Falcon-40b (e) Svamp - Falcon-40b (f) ASDIV - Falcon-40b

Figure 52: Percentage of number of generated answers as per 'no. of reasoning steps' of Llama2-70b (top row) and with Falcon-40b (bottom row) on the GSM8K (a), SVAMP (b), and ASDIV (c) test datasets.

11.4.4 Understanding the effect of CCoT

While the previous experiments demonstrate that CCoT achieves a clear reduction in the number of words produced, thus improving the trade-off between accuracy and

generation time. The scores introduced in Section 11.3 are used in the following analysis to quantify the benefits in terms of conciseness (RQ3). As detailed in Section 11.3, the conciseness of an answer is assumed to depend on two key properties: *(i)* the number of steps required to produce a response, and *(ii)* assuming a similar number of steps, the amount of information repeated across those steps.

Number of Steps. To evaluate the first property, we plotted in Figure 52 the distribution of the percentage of generated answers based on the *number of reasoning steps* [253] for Llama2-70b (top row) and Falcon-40b (bottom row) across the three datasets. . The results reveal a significantly different distribution in the number of steps across all datasets with Falcon-40b and for the ASDIV dataset with Llama2-70b. In fact, in these cases, the CCoT curves are clearly concentrated toward a reduced number of steps compared to CoT, indicating that the CCoT prompt consistently generates responses with fewer steps.
While the analysis of the number of steps explains the improved conciseness achieved with CCoT, more detailed analysis of the redudancy and information flow are required to understand the enhanced conciseness observed with Llama2-70b on GSM8K and SVAMP, where the number of steps follows a trend similar to that of CoT.

Redundancy Evaluation. As discussed in the previous paragraph, the distribution of reasoning steps for Llama2-70b on the GSM8K and SVAMP datasets (Figures 52a and 52b) reveals that the quartile ranges of the answer distributions for CCoT and CoT answers are very similar. Specifically, in both datasets, the interquartile range spans approximately 4 to 12 steps (Q1 to Q3) for SVAMP and around 5 to 14 steps for GSM8K. This suggests comparable step counts across CCoT and CoT answers, within such ranges of reasoning steps.
To evaluate the conciseness of these reasoning steps more effectively, we present redundancy scores in Figure 53. This figure highlights the differences in redundancy between CCoT and CoT answers, categorized by their number of steps. Notably, when focusing on step intervals within the interquartile range (Q1 to Q3, where most answers fall), which are highlighted in grey in the plots, redundancy scores are consistently higher for CoT than for CCoT. This demonstrates that CCoT achieves improved syntactic conciseness compared to CoT, even when the number of steps is similar.
To better quantify the reduction in redundancy achieved by CCoT, we define the *Mean Redundancy Reduction (MRR)* across single-step redundancies RMS_i for all

reasoning steps i, and the *Overall Redundancy Reduction (ORR)* as:

$$\text{MRR} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{\text{CoT RMS}_i - \text{CCoT RMS}_i}{\text{CoT RMS}_i} \times 100\right)$$

$$\text{ORR} = \frac{\text{CoT RMS} - \text{CCoT RMS}}{\text{CoT RMS}} \times 100$$

Table 35 presents the average ORR and MRR calculated for the SVAMP and GSM8K datasets, focusing on steps within the interquartile range (Q1 to Q3). The results indicate that CCoT consistently reduces redundancy. For SVAMP, the redundancy reduction ranges from 19.77% to 22.72% (ORR), while for GSM8K, it ranges from 12.64% to 24.74% (ORR).

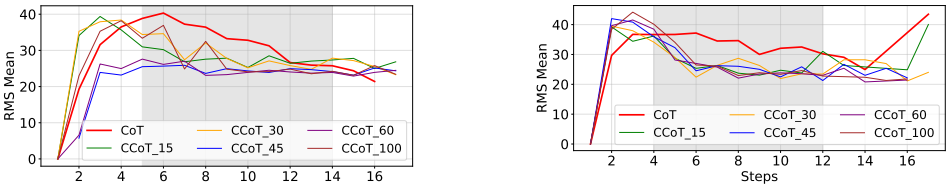


Figure 53: RMS score (lower values indicate better conciseness) of Llama2-70b on the GSM8K (top) and SVAMP (bottom) datasets. The grey area highlights the portion of the answer distribution between the Q1 and Q3 quartiles.

Method	Dataset	CCoT15	CCoT30	CCoT45	CCoT60	CCoT100
MRR	SVAMP	20.02	22.50	21.48	22.92	20.14
	GSM8K	13.58	11.65	22.81	23.23	16.33
ORR	SVAMP	20.22	22.57	21.29	22.72	19.77
	GSM8K	15.34	12.64	24.62	24.74	16.81

Table 35: Mean and Overall Redundancy Reduction for SVAMP and GSM8K Datasets

Information Flow Evaluation. To demonstrate an improved level of conciseness also from a semantic perspective, we analyze the information flow. Specifically, the median number of steps in the answer distributions for CoT and CCoT (Figures 52a and 52b) is approximately 8 for both the SVAMP and GSM8K datasets. Thus, focusing on answers with a total of 8 steps, we present in Tables 36 the *Information*

Flow between consecutive steps $i \rightarrow j$ for Llama2-70b on the GSM8K (top) and SVAMP (bottom) datasets.
In Table 36, CCoT-15 exhibits the largest reductions across all steps, for instance, ranging from 26% to 41% compared to CoT scores for GSM8K, while CCoT-100 shows the smallest reductions (4% to 20%), preserving more redundant semantic information. Generally, the middle steps show the largest differences, especially for aggressive variations like CCoT-15 and CCoT-45, whereas early and late steps retain more information flow. In summary, a lower information flow indicates that the model effectively retains, step by step, only the logically necessary information required to arrive at a correct answer.

GSM8K - Llama2-70b

Steps	CoT	CCoT-15	CCoT-30	CCoT-45	CCoT-60	CCoT-100
1 ⇒ 2	0.5287	0.3417	0.39	0.49	0.49	0.43
2 ⇒ 3	0.59	0.39	0.45	0.31	0.32	0.53
3 ⇒ 4	0.56	0.37	0.43	0.36	0.37	0.47
4 ⇒ 5	0.55	0.38	0.41	0.36	0.36	0.46
5 ⇒ 6	0.56	0.39	0.42	0.36	0.37	0.51
6 ⇒ 7	0.56	0.42	0.45	0.37	0.37	0.54
7 ⇒ 8	0.52	0.42	0.47	0.57	0.56	0.50

SVAMP - Llama2-70b

Steps	CoT	CCoT-15	CCoT-30	CCoT-45	CCoT-60	CCoT-100
1 ⇒ 2	0.54	0.32	0.41	0.41	0.30	0.32
2 ⇒ 3	0.52	0.35	0.47	0.43	0.35	0.35
3 ⇒ 4	0.51	0.34	0.46	0.40	0.30	0.37
4 ⇒ 5	0.51	0.37	0.43	0.39	0.30	0.30
5 ⇒ 6	0.51	0.38	0.41	0.44	0.36	0.37
6 ⇒ 7	0.50	0.41	0.48	0.40	0.36	0.40
7 ⇒ 8	0.47	0.35	0.43	0.37	0.34	0.50

Table 36: Information Flow Mean Values comparison for GSM8K (top) and SVAMP (bottom) across answers with 8 steps. Better information flow indicates lower semantic conciseness between steps (highlighted with blue-like colors in the tables).

11.5 Ability to control the output length

The previous experiments looked at how CCoT strategies can affect the accuracy and generation time in the average. However, despite the discussed benefits, it is also crucial to understand how CCoT prompting can effectively limit the output length for each addressed sample (RQ3). This can be useful for better tuning the length parameter in the CCoT prompt or identifying the conditions in which the proposed prompting strategy fails to compress the output. To evaluate the ability of an LLM to produce concise answers in response to a given prompting, we analyzed in Figure 54 the output length under different CCoT length constraints.
Figure 54 shows the statistics on the length of the answers provided by addressed models with the GSM8K test set. Each box plot represents the output lengths between the 5th and the 95th percentiles of all tested samples, the blue line represents the provided CCoT length constraint, the red line denotes the median, while the greed dot the mean. Ideally, a model respecting the given length constraint for each tested sample should have the entire distribution below the blue line.
As depicted in Figure 54, CoT based LLMs tend to produce long answers if not explicitly constrained, significantly impacting the generation time. The imposed

length constraint in the CCoT prompt significantly affects the output length, although in practice LLMs are not always able to respect the given limit, especially for smaller values, such as 15, 30, or 40, which are more challenging.
To summarize, given the nature of the CCoT prompting, it is reasonable to consider a tolerance margin in respecting the requested length. To this end, in the following paragraphs we evaluate the considered models by the metrics proposed in Section 11.1, which extend the accuracy by also accounting for conciseness.

11.6 Conclusions

In this work, we have systematically examined the impact of output length on the reasoning capabilities and operational costs of large language models. Our analysis confirms that while extended rationales often increase accuracy, they also introduce redundancy, higher latency, and unpredictable inference costs, making such models less practical in real-world cybersecurity pipelines. Conversely, concise outputs reduce computational overhead and improve efficiency, but risk omitting key reasoning steps that are necessary to ensure correctness and interpretability.
Through a series of controlled experiments, we demonstrated that the relationship between length and reasoning quality is not linear: longer chains of thought do not always guarantee better answers, and in many cases targeted conciseness leads to equally, if not more, reliable reasoning. We also evaluated techniques such as length-aware prompting, constrained explanation formats, and selective rationale extraction, showing how they can strike a balance between accuracy and efficiency. These results underscore the importance of managing verbosity not only as a usability concern, but as a central component of reasoning optimization.
Overall, this study highlights that optimizing reasoning is not merely about making models "think longer," but about enabling them to reason *as concisely as necessary* to remain correct, interpretable, and cost-effective.

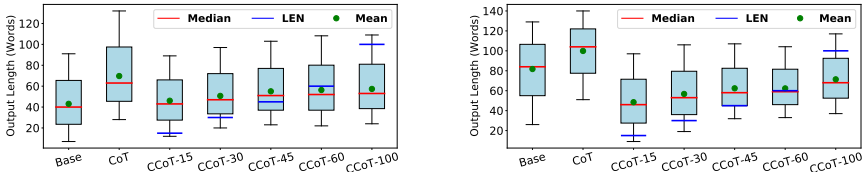


Figure 54: Distribution of output lengths (y-axis) between the 5th and 95th percentiles for different models and prompting strategies using the GSM8K test set. The top plot shows the results for Falcon-40b, while the bottom plot presents those for Llama2-70b.

12 Trajectory-Based Policy Optimization in Large Language Models

The final chapter focuses on the development of a new reinforcement learning algorithm designed to improve both the stability of training and the overall performance of large language models, including their ability to generalize across different tasks and data distributions.

Recent work in model alignment has employed reinforcement learning to guide LLM behavior. Approaches such as Reinforcement Learning with Human Feedback (RLHF), Direct Preference Optimization (DPO), and Reward-Filtered Tuning (RFT) have introduced feedback-driven mechanisms to align models with human expectations. More recently, Group-Relative Policy Optimization (GRPO) has been proposed as a lightweight alternative that directly optimizes reasoning trajectories by comparing multiple completions within a group. GRPO removes the need for an explicit critic model and has shown promise in improving structured reasoning tasks.

However, closer analysis reveals two key limitations. First, *gradient conflicts* arise when tokens shared across multiple completions within a group are simultaneously reinforced and penalized, degrading stability. This issue particularly affects formatting tokens that are critical for maintaining coherent answer structures. Second, a *policy collapse phenomenon* emerges during extended training, where negatively rewarded completions destabilize the model and lead to performance degradation. The KL regularization term in GRPO reacts too slowly to prevent this collapse, highlighting the need for more reliable control mechanisms.

To address these challenges, this stage introduces **Group-relative Trajectory-based Policy Optimization (GTPO)**, a new reinforcement learning algorithm that refines the optimization of reasoning trajectories. GTPO incorporates conflict-aware gradient correction to resolve token-level inconsistencies and introduces entropy-based regularization as a more effective signal of policy instability than KL divergence. Together, these components stabilize training, preserve reasoning structure, and prevent collapse while removing the need for a reference model during optimization.

It is important to note that, unlike the previous stages of the thesis, this work does not rely on cybersecurity-specific datasets. Such resources are not yet available in the literature for reasoning-centered reinforcement learning. Instead, the analysis is performed on widely used benchmarks such as GSM8K, MATH, and AIME2024, which allow systematic evaluation of reasoning behavior. The

absence of cybersecurity datasets does not diminish the relevance of the study: the proposed improvements act at the level of reasoning optimization, which is domain-agnostic and directly transferable to security applications once such datasets become available.

The contributions of this stage can be summarized as follows. First, strategies are proposed to stabilize reinforcement learning for LLMs and mitigate collapse phenomena, ensuring robust reasoning behavior (C21). Second, refined reasoning control is shown to improve both efficiency and reliability, thereby strengthening the applicability of LLM-based systems in real-world cybersecurity contexts (C22).

In summary, this chapter consolidates the methodological core of the thesis. By moving from domain-level applications to reasoning-level optimization, it establishes a foundation for trustworthy, stable, and efficient LLMs, ensuring that the advances achieved in malware analysis, threat representation, and specialized assistance can be sustained by models whose reasoning processes are as reliable as their knowledge.

12.1 Preliminaries

In GRPO, the LLM, acting as a policy, generates during training multiple completions (responses) per prompt q, denoted as $\{o_1, o_2, \ldots, o_G\}$, where G is the number of completions generated, and computing rewards relatively to each rather than absolutely. Each output is structured with special tags for reasoning and answer segments and allows the reward to be calculated as an aggregate of formatting and factual correctness. The goal is to maximize the objective:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q,\{o_i\}} \left[\frac{1}{G} \sum_{i=1}^{G} \bar{C}_i - \beta \cdot D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}}) \right] \tag{26}$$

Here, the expectation is taken over prompts $q \sim \mathbb{P}(Q)$ and completions $\{o_i\} \sim \pi_{\theta_{\text{old}}}$, where $\pi_{\theta_{\text{old}}}$ is the behavior policy. Each completion o_i has length $|o_i|$. The term $\bar{C}_i$ denotes the average clipped advantage over its tokens: $\bar{C}_i = \frac{1}{|o_i|} \sum_{t=1}^{|o_i|} C_{i,t}$. The clipped advantage for the token at position t in the completion o_i, is defined as: $C_{i,t} = \min\left(\frac{\pi_\theta(o_{i,t}|s_{i,t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|s_{i,t})} \cdot \hat{A}_i, \text{clip} \right)$, where $\hat{A}_i = \frac{R_i - \bar{R}}{\text{std}(R)}$ is the normalized scalar advantage assigned to the i-th completion, and $s_{i,t} = (q, o_{i,<t})$ denotes the sequence of previously generated tokens up to position t. Specifically R_i represents the reward for completion i, which is a sum of sub-rewards based on the correctness of the answer and its formatting style (Formatting Reward). The term $\pi_\theta(o_{i,t}|s_{i,t}) = \text{softmax}(f_\theta) \in \mathbb{R}^V$, represents the probability distribution over the output logits f_θ, where f_θ^j is the logit at position j, given the context $s_{i,t}$, and V is

the vocabulary size. Finally, the KL divergence term in Eq.(26) penalizes deviation from a reference policy π_{ref}, scaled by a factor β. Due to computational cost and training time, one iteration is adopted in Eq. 26 [254], where $\pi_\theta = \pi_{\theta_{\text{old}}}$, making the likelihood ratio 1 and the clipping unnecessary, i.e., $C_{i,t} = \hat{A}_i$. The gradient of this simplified objective is:

$$\nabla_\theta \mathcal{J}_{\text{GRPO}}(\theta) = \frac{1}{G}\sum_{i=1}^{G}\frac{\hat{A}_i}{|o_i|}\sum_{t=1}^{|o_i|} g_{i,t} - \beta \cdot \nabla_\theta \mathrm{D}_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}}), \tag{27}$$

where $g_{i,t} := \nabla_\theta \log \pi_\theta(o_{i,t}|s_{i,t})$ denotes the token-level policy gradient, which can be also expressed as:

$$g_{i,t} = \left(\nabla_\theta f_\theta^j (1 - \pi_\theta^j) - \sum_{k \neq j} \pi_\theta^k \nabla_\theta f_\theta^k\right),$$

with j is the index of the logit with the highest probability π_θ. Intuitively from Eq.(27), a positive advantage ($\hat{A}_i > 0$) boosts the logit of the selected token proportionally to $1 - \pi^j$, while reducing the logits of all others. Conversely, a negative advantage inverts this behavior, penalizing the token and increasing alternatives. Full derivations are in Append.

12.2 GRPO Issues

This section highlights two major limitations of GRPO: *Token-level penalization* and *policy collapse*.

12.3 Token-level Penalization

We show that GRPO tends to have negative effects on the updates of certain tokens that are essential for maintaining the structure and interpretability of completions.

Prefix Tokens Penalization.
Given a prompt q, the model generates a set of completions $\{o_1, o_2, \ldots, o_G\}$, which may begin with the same sequence of tokens and then diverge at a specific point, a token that acts as a crossroads. This behavior is typical in instruction-tuned models, where early tokens tend to be similar across completions [255]. However, not all completions in the group G necessarily share the same prefix. In practice, we often observe multiple distinct prefixes, each shared by a subset of the completions. To model this, we partition the G completions into K disjoint groups $\mathcal{G}_1, \ldots, \mathcal{G}_K$ so that all completions in a group $\mathcal{G}_k$ have a common prefix $S_{\text{pfx}}^{(k)}$. If all G completions share the same prefix, then $K = 1$.

Tokens in $S^{(k)}_{\text{pfx}}$ are affected by the combined advantages of all completions in the group, while other tokens only depend on the advantage of their own completions. Under this structure, we rewrite the GRPO gradient (Eq. 27) by summing the contributions of all groups, where each group contributes a prefix term and a set of individual terms. The gradient, excluding the KL term, becomes:

$$\nabla_\theta \mathcal{J}_{\text{GRPO}}(\theta) = \frac{1}{G}\sum_{k=1}^{K}\Bigg[\underbrace{\sum_{i\in\mathcal{G}_k}\frac{\hat{\mathcal{A}}_i}{|o_i|}\sum_{t\in S^{(k)}_{\text{pfx}}} g_{i,t}}_{\Delta^{(k)}_{\text{pfx}}} + \sum_{i\in\mathcal{G}_k}\frac{\hat{\mathcal{A}}_i}{|o_i|}\sum_{j\notin S^{(k)}_{\text{pfx}}} g_{i,j}\Bigg]$$

The term $\Delta^{(k)}_{\text{pfx}}$ represents the gradient update associated with the shared prefix of group $\mathcal{G}_k$. If a completion does not share any prefix with others, then $\mathcal{G}_k$ contains only that single completion, resulting in $S^{(k)}_{\text{pfx}} = \emptyset$ and $\Delta^{(k)}_{\text{pfx}} = 0$.
Since the log-probability terms $g_{i,t}$ are identical and so constant across completions in the group $S^{(k)}_{\text{pfx}}$, the gradient component Δ_{pfx} is primarily driven by the sum of normalized advantages $\sum_{i\in\mathcal{G}_k}\hat{\mathcal{A}}_i/|o_i|$. This implies that the update to a prefix mainly depends on the relative balance of advantages across all completions in group k. For instance, the sum can be negative, when correct completions (with positive advantages) are generally longer than incorrect ones. In such cases, the denominator $|o_i|$ for the correct completions becomes larger, reducing their contribution to the overall gradient. As a result, the model may be penalized for generating desirable and beneficial prefix tokens. This introduces a bias against shared initial tokens, such as formatting tokens or reasoning tags (<reasoning>), which are essential for the structure and correctness of completions.

Dependencies From Completion Advantage. GRPO can also induce undesirable penalization on tokens that appear after the prefix. Certain tokens, in particular formatting ones, may occur in multiple completions at varying positions and within different contexts, some correct, others incorrect, making them more susceptible to inconsistent and potentially harmful updates.
This induces potential issues in the update. For instance, if a token τ frequently appears in completions with negative advantage, its probability may be reduced, even if the token is syntactically correct and required. Additionally, further issues can arise when a completion only partially follows the expected format. For example, if a completion closes a first part correctly with a formatting token τ </reasoning> but then omits <answer>, τ may receive a low or even negative reward. This, in turn, penalizes </reasoning>, despite it being a correct token in the completion. This occurs because the update for each token primarily depends on the advantage

assigned to the entire completions in which it appears, without directly considering whether the token's individual contribution was beneficial. We acknowledge that this phenomenon is particularly impactful in the case of formatting tokens, as illustrated in the previous examples, and in the case of shared final tokens that form a common suffix among completions.

12.4 Policy Collapse

The GRPO gradient, as discussed in Eq. (27), highlights a dependency on both the advantage and the probability scores assigned by the policy π_θ. To gain a deeper understanding of how the advantage influences the learning dynamics, we analyze the average Shannon entropy of the output distribution π_θ over completions i and tokens t during training. The Shannon entropy is defined as $H(\mathbf{p}) = -\sum_{j=1}^{n} p_j \ln p_j$, where $\mathbf{p}$ is a probability vector. For a token position t in a completion o_i, we consider $\mathbf{p} = \pi_\theta(o_{i,t}|s_{i,t})$, and we define $\langle H \rangle_i$ as the average entropy across in the completion o_i.

At a generation step, let us consider a low entropy (e.g., $\langle H \rangle_i \ll \ln 2$) [35] , indicating that the model is highly confident, i.e., it assigns almost all probability mass to a single token index j, with $\pi_\theta^j \approx 1$ and $\pi_\theta^k \approx \epsilon \ll 1$, where $k \neq j$. If this token leads to a negative outcome ($\hat{\mathcal{A}}_i < 0$), GRPO penalizes it sharply, since the gradient of the GRPO loss at index τ for completion i becomes:

$$\nabla_\theta \mathcal{J}_{\text{GRPO}}^{(i,\tau)}(\theta) \approx \begin{cases} -\dfrac{|\hat{\mathcal{A}}_i|}{|o_i|} \cdot \nabla_\theta f_\theta^j \cdot (1-\epsilon) & \text{for } j \\ \dfrac{|\hat{\mathcal{A}}_i|}{|o_i|} \cdot \nabla_\theta f_\theta^k \cdot \epsilon & \text{for } k \neq j \end{cases} \tag{28}$$

Note that the KL term has been omitted here, its contribution will be addressed next.

This results in a negative update for the selected token j, and small positive updates for all other tokens $k \neq j$, even if they are potentially syntactically or semantically incorrect. While each of these positive updates is individually small, they can accumulate over time, gradually increasing the probability of initially implausible tokens.

The effect is further amplified by *GRPO's zero-mean constraint*: when most completions receive positive rewards, the few with negative rewards must carry disproportionately large negative advantages to maintain balance. These penalties

[35] A completion with average entropy near $\ln 2$ suggests balanced uncertainty between two choices for each output token.

can suppress correct predictions and unintentionally amplify the likelihood of undesirable alternatives, raising entropy and introducing instability. Over time, this harms output quality and increases the risk of introducing structural and semantic errors that propagate through subsequent training steps. As the model drifts away from desirable behaviors, the reward signal becomes less meaningful.

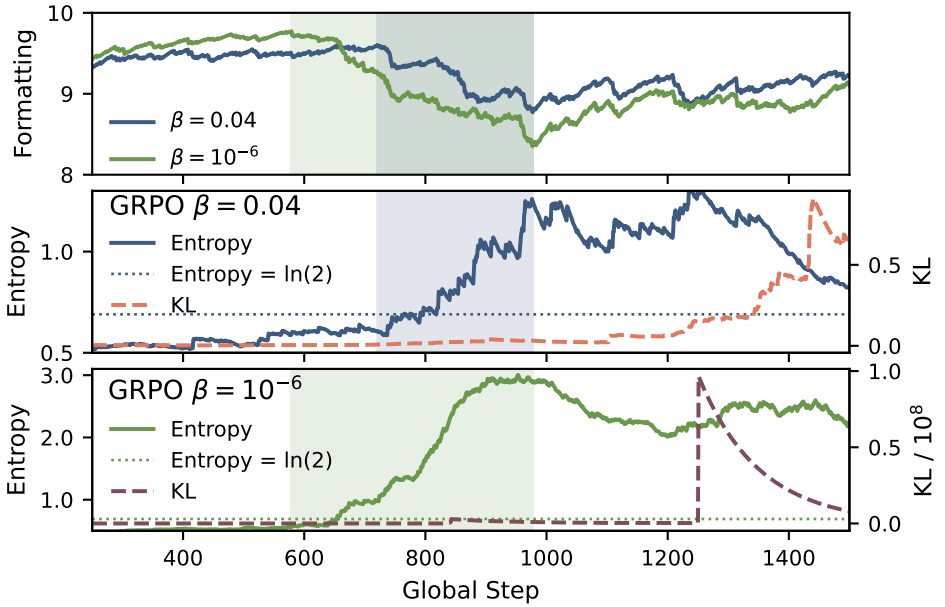


Figure 55: Training behavior of LLaMA-8B on GSM8K using two GRPO runs with $\beta = 0.04$ and $\beta = 10^{-6}$, both with $G = 8$. The top plot shows the formatting rewards for both runs, and the bottom plots show the average entropy and KL divergence in each run.

This phenomenon is illustrated in the top plot of Figure 55, which shows the average formatting reward during the training of LLaMA-8B on GSM8K, using GRPO with $\beta = 0.04$ (as originally proposed in [112]) and $\beta = 10^{-6}$. At the beginning of training, the formatting reward increases more rapidly with $\beta = 10^{-6}$, while it remains relatively flat with $\beta = 0.04$. However, in both cases, the reward begins to drop below 9, after 580 steps for $\beta = 10^{-6}$ and around step 750 for $\beta = 0.04$.

KL Reacts Late to Avoid Collapse. Middle and bottom plots of Figure 55 show the average entropy $\langle H \rangle_i$ and average KL divergence, computed on the generated completions, for the two previous training runs of LLaMA 8B. Both KL divergence curves remain stable up to approximately step 1200, well after the degradation

points, when the formatting rewards have already collapsed and entropy levels are high. Only around the KL peak, the reward curves show signs of stabilization. In contrast, the entropy curves begin to rise sharply as the formatting rewards start to decline, and continue to increase as the rewards deteriorate further.
This pattern suggests a key insight: KL divergence acts as a delayed corrective signal, rising only after collapse. In contrast, entropy tracks the policy collapse in real time, increasing as the policy loses structure.

12.5 Method

This section introduces GTPO, which addresses the previous issues by *(i)* mitigating gradient conflict on shared tokens, and *(ii)* preventing policy collapse during training by regularizing the model behavior through the entropy.

12.5.1 Conflict-Aware Gradient Correction

As discussed in Section 12.3, The model can encounter gradient conflict issues, where shared tokens receive both positive and negative updates depending on the advantage assigned to each completion. To correct this issue, we propose in the following a methodology aimed at selectively masking gradient contributions for tokens involved in conflicting updates. To this end, we first identify *conflict tokens*, as tokens that appear in the same position, either from the beginning or the end, across completions with both positive and negative advantages. As discussed in the context of GRPO's issues part, these tokens often receive conflicting gradient updates during training. Then, we mitigate such conflicts by correcting their gradient updates accordingly.

Conflict tokens definitions. Let $\{o_1, \ldots, o_{|G|}\}$ be a group of completions. Each completion o_i contains a sequence of tokens $\{o_{i,1}, \ldots, o_{i,|o_i|}\}$, and is associated with an advantage value $A^{(i)} \in \mathbb{R}$. We define G^- and G^+ as the sets of completions with $\mathcal{A} < 0$ and $\mathcal{A} > 0$, respectively.

Left-to-right alignment. A token $v \in \mathcal{V}$ is a *forward conflict token* at position p if it appears at the same position in at least one completion with positive advantage and in at least one with negative advantage:

$$\exists\, i \in G^+ :\ o_{i,p} = v \quad \wedge \quad \exists\, j \in G^- :\ o_{j,p} = v.$$

Right-to-left alignment. A token v is defined as a *backward conflict token* at offset r if it occurs at the r-th position from the end in at least one completion with a

positive advantage and in at least one with a negative advantage:

$$\exists\, i \in G^+ : \; o_{i,|o_i|-r} = v \quad \wedge \quad \exists\, j \in G^- : \; o_{j,|o_j|-r} = v.$$

Gradient Reweighting with conflict masks. Based on the definitions above, we construct binary masks for tokens across completions to target positions potentially affected by gradient conflicts and thus correct their updates.
We first define a *forward mask* $\mathcal{M}_i^{\text{fw}} \in \{0,1\}^{|o_i|}$ by scanning o_i from left to right, setting 1 over the first contiguous span of forward conflict tokens and 0 elsewhere. Analogously, the *backward mask* $\mathcal{M}_i^{\text{bw}}$ is obtained by scanning from right to left, marking the first contiguous span of backward conflict tokens. A *final mask* $\mathcal{M}_i$ is then defined as: $\mathcal{M}_i = \mathcal{M}_i^{\text{fw}} \vee \mathcal{M}_i^{\text{bw}}$, highlighting only the initial and final conflict regions in each completion o_i.
After computing each $\mathcal{M}_i$ in the group of completions, we correct inconsistent gradient updates over the selected conflict tokens, while leaving other token updates unchanged, thought the following preliminary token-level loss:

$$\mathcal{J}^* = \frac{1}{G} \sum_{i=1}^{G} \frac{\mathcal{A}_i}{|o_i|} \sum_{t=1}^{|o_i|} \lambda_{i,t} \tag{29}$$

where $\lambda_{i,t}$ controls the update of each token based on its conflict position and the sign of the advantage $\mathcal{A}_i$:

$$\lambda_{i,t} = \begin{cases} 1 & \text{if } \mathcal{M}_{i,t} = 0, \\ 0 & \text{if } \mathcal{M}_{i,t} = 1 \text{ and } \mathcal{A}_i < 0, \\ 2 & \text{if } \mathcal{M}_{i,t} = 1 \text{ and } \mathcal{A}_i > 0. \end{cases} \tag{30}$$

Intuitively, the mask disables negative gradients on conflict tokens, and instead reinforces them only if they appear in positively rewarded completions. Note that the total signal magnitude is preserved across the group: since $\sum_{i \in G^+} |\mathcal{A}_i| = \sum_{i \in G^-} |\mathcal{A}_i|$, doubling the signal for positive completions compensates for the removal of negative updates, maintaining training stability while preventing semantic drift.
Additional details of the weighting are evaluated in the ablation studies (Section 12.8.1).
Importantly, the final mask $\mathcal{M}_i$ targets only the initial and final contiguous conflict tokens to preserve the semantic structure of completions. In fact, masking isolated conflict tokens in the middle could harm stability and learning, as their meaning

often depends on surrounding context, and altering their gradients may be counterproductive. In contrast, the outer spans typically correspond to formatting tags, such as <reasoning> or </answer>, which are the primary source of conflict in GRPO, as discussed in Section 12.3. Focusing the correction on these regions protects structural tokens without interfering with the central part of the completion, where meaningful differences in trajectories emerge.

12.6 Entropy-Based Policy Regularization

As discussed in Section 12.4, GRPO can lead to policy collapse, where standard KL term may react too slowly. To address this, we propose entropy-based regularization terms during training. These consist of two key parts: (i) a filtering mechanism to discard unstable completions, and (ii) a regularization term that penalizes high-entropy behavior.

Completion filter. Based on the policy-collapse analysis, we observe that high-entropy completions can jeopardize training by signaling structural uncertainty, particularly in models that naturally exhibit low average entropy. Applying gradients in such cases risks amplifying uncertainty and accelerating collapse. To mitigate this, we propose filtering out high-entropy completions, focusing on models prone on collapse against this. We define $\langle H \rangle_{\text{ini}}$ as the model's initial entropy over a set of questions, measured prior to training. If $\langle H \rangle_{\text{ini}} < \ln 2$, we assume that the model tends to produce low-entropy outputs, making it more sensitive to high-entropy completions during training. In this case, we apply an entropy-based filtering mask δ_i that filter out the associated advantage signal. The mask δ_i is formally defined as:

$$\delta_i = \begin{cases} 1, & \text{if } \langle H \rangle_{ini} > \ln 2, \\ 0, & \text{if } \langle H \rangle_{ini} < \ln 2 \text{ and } \langle H \rangle_i > \ln 2, \\ 1, & \text{if } \langle H \rangle_{ini} < \ln 2 \text{ and } \langle H \rangle_i \leq \ln 2. \end{cases} \tag{31}$$

Entropy Regulatization. Inspired by PPO [256, 109], we add a regularization term based on the average tokens entropy of each completion, $\langle H \rangle_i$, where γ balance the importance of this term in the final loss, as shown below. Note that, based on the internal characteristics of GRPO to implicitly increase entropy over time, we decided to minimize the term. This acts as a way to reduce the model entropy over time.

The combination of these entropy-based strategies and the token-level loss defines the final GTPO objective, as:

$$\mathcal{J}_{\text{GTPO}} = \frac{1}{G}\sum_{i=1}^{G}\frac{\delta_i \cdot \mathcal{A}_i}{|o_i|}\sum_{t=1}^{|o_i|}\lambda_{i,t} \;-\; \gamma \cdot \langle H \rangle_i \tag{32}$$

The proposed GTPO loss does not require an additional reference model, unlike the KL divergence term in GRPO, thereby reducing the memory footprint during training. The benefits of each component, and hyperparameter of the loss are discussed and demonstrated, also through ablation studies in the following experimental section.

12.7 Experiments

Experimental Setup. We conducted experiments to assess the training stability and generalization performance of GTPO. All experiments were performed on LLaMA-8B [257] and Qwen 2.5 (3B) [258], which have trained using the training splits of GSM8K [259] and MATH [260], and evaluated after training on the corresponding test splits, and also the AIME2024 dataset [261].

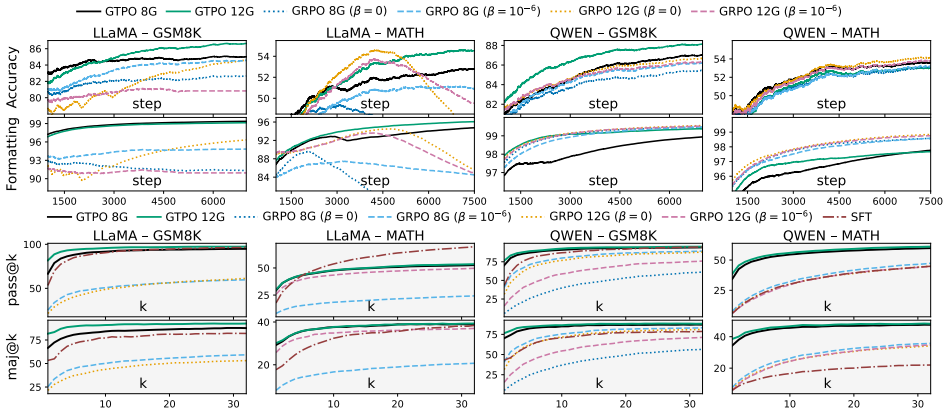
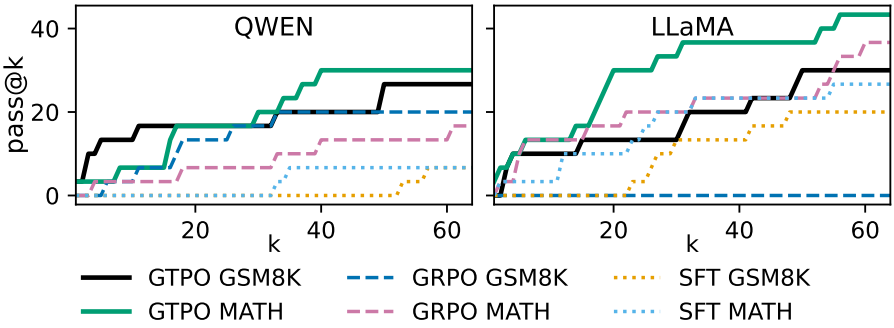


Figure 56: Training accuracy and formatting rewards (%) of GTPO and GRPO over training steps on MATH and GSM8K (top). In-distribution evaluation of models trained with GTPO, GRPO, and SFT on the corresponding test sets, using `pass@k` and `maj@k` (%) over k (bottom).

For comparison, all models were also trained using SFT and GRPO (with both $\beta = 0$ and $\beta = 10^{-6}$ to assess the impact of the KL term). To further explore hyperparameter group-relative optimizations, both GRPO and GTPO were evaluated with two generation sizes: $G = 8$ and $G = 12$. For the entropy-based completion



(a) Results on AIME2024.

Figure 57: Out-of-distribution evaluation on AIME2024, with `pass@k` (%) over k, on models trained on MATH and GSM8K.

mask used in GTPO, we compute $\langle H \rangle_{\text{ini}}$ by evaluating the original LLM entropy on the first 100 samples of the training set.
All training was performed using a learning rate of 10^{-6}, with the temperature set to 1.0 during the test phase.
All experiments were conducted on 2 NVIDIA A100 GPUs.

12.8 Performance Evaluation

Training Dynamics of GTPO. In Figure 56-top (the first two rows), we compare the training stability and performance of GTPO against GRPO. Formatting and accuracy rewards are reported as percentages, where the maximum value (100%) corresponds to a reward of 10 in both. On the GSM8K dataset with LLaMA, GTPO consistently outperforms GRPO across all training steps in both accuracy and formatting metrics. On the more challenging MATH dataset, GRPO (with $G = 12$ and $\beta = 0$) initially achieves slightly better accuracy around the midpoint of training. However, its performance drops sharply in the second half of training due to policy collapse, affecting both accuracy and formatting.
In contrast, GTPO continues to improve steadily throughout training, avoiding collapse and maintaining stable performance. For Qwen 2.5, on both GSM8K and MATH, GTPO achieves comparable or improved accuracy relative to GRPO, with only a slight decrease in formatting performance (still above 97% in all runs). Note that, consistent with the analysis in Section 12.4, GRPO training curves for Qwen 2.5 are not affected by a policy collapse, as it exhibits high-entropy behavior. Overall, GTPO demonstrates more stable and reliable training compared to GRPO

across models and datasets.

In-distribution Evaluation. The trained LLMs were evaluated on the test sets of GSM8K and MATH using the `pass@k` [245] and `maj@k` [246] metrics. The former measures whether at least one of the top-k completions yields a correct answer, while the latter assesses correctness via majority voting over the top-k completions. Note that, to ensure a fair comparison, we evaluated GRPO on LLaMA using the model checkpoint corresponding to the training step with the highest accuracy reward, rather than the one obtained after policy collapse.
Figure 56-bottom (the last two rows) shows that GTPO consistently outperforms GRPO in almost all settings for both `pass@k` and `maj@k`, as k varies from 1 to 32. This indicates that models trained with GTPO exhibit stronger self-consistency when answering questions (higher `maj@k`) and better coverage of the correct answer across multiple completions (higher `pass@k`). We also include testing comparisons with SFT, where GTPO consistently outperforms SFT in `maj@k` across all models and datasets, and achieves higher average performance in `pass@k`. Specifically, SFT surpasses GTPO only in `pass@k` for $k > 5$ on MATH with LLaMA, but not in terms of correctness with `maj@k`. Interestingly, in GTPO, larger values of G lead always to better performance, which is not the same for GRPO.

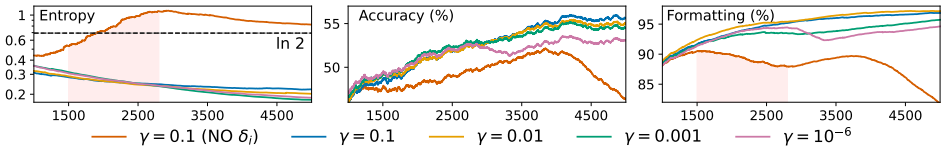


Figure 58: Analysis of training dynamics under different settings of entropy regularization γ: average group completion entropy (left), accuracy (center), and formatting (right). Note that '$\gamma = 0.1$, No δ_i' denotes the setting in which the entropy-based filtering term is disabled.

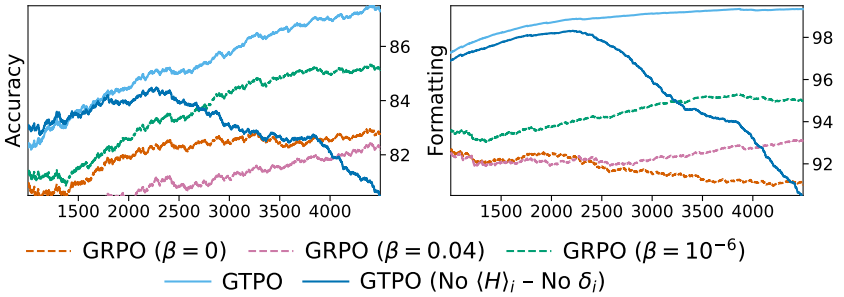


Figure 59: Accuracy and formatting performance of LLaMA trained on GSM8K. For GRPO, results are shown under varying KL-β values. For GTPO, "No $\langle H \rangle_i$ - No δ_i" denotes the setting in which only the Conflict-aware Gradient Correction is applied.

Out-of-distribution Evaluation. We also evaluate the trained models on a different test set (AIME2024), as shown in Figure 57, reporting `pass@k` scores with k extended up to 64 to account for the increased difficulty of the task. For convenience, for each dataset, we select the GRPO and GTPO variants with the generation size G that yielded the best performance on the in-distribution tests.
The results show that GTPO consistently outperforms both SFT and GRPO, particularly at higher values of k on the MATH dataset, where the increased complexity encourages broader exploration of reasoning paths. In contrast, GRPO does not consistently benefit from this, which limits its performance gains. Interestingly, both GTPO and GRPO demonstrate stronger out-of-distribution generalization compared to SFT, suggesting a higher degree of overfitting to the in-distribution data in the latter.

12.8.1 Ablation Studies

Entropy-based terms. To evaluate the impact of entropy-based terms, Figure 58 shows the training curves for *average completion entropy* (left), *accuracy* (middle), and *formatting* (right) for LLaMA-8B on the MATH dataset. We compare different entropy regularization strengths: $\gamma = 0.1$, $\gamma = 0.01$, $\gamma = 0.001$, $\gamma = 10^{-6}$, and $\gamma = 0.1$ without applying the filtering defined in Eq. 31 (denoted as "*NO* δ_i"). This last setting highlights the impact of not filtering out high-entropy completions that could trigger policy collapse.
As shown in the figure, larger values of γ lead to improved accuracy and formatting, with $\gamma = 0.1$ achieving the highest accuracy overall. In contrast, when filtering is disabled ($\gamma = 0.1$, No δ_i), both accuracy and formatting collapse, highlighting the

critical role of the filtering term in maintaining training stability. This behavior is further supported by the entropy plot (left side): without filtering, entropy steadily increases and remains above $\ln 2$, eventually destabilizing formatting (initially, as seen in the red region) and later affecting accuracy. In contrast, when filtering is applied, entropy remains below $\ln 2$ and gradually decreases over time.
Interestingly, the figure shows that higher values of γ lead to both greater stabilized entropy (left plot) and improved performance. Regarding performance, in terms of accuracy and formatting, this trend can be explained by the fact that very low entropy causes the model to become overly confident, limiting its ability to explore (e.g., with $\gamma = 0.001$, where entropy continues to decrease and accuracy plateaus around step 4000). In contrast, having moderate entropy promotes continued exploration during training, resulting in more diverse and informative completions.
To better understand the behaviour of the entropy curves, where larger γ values appear to converge toward higher entropy values, when a completion receives a negative advantage, entropy tends to increase (see Eq. 29). In this context, a stronger entropy regularization term (Eq. 32) amplifies the negative gradient on the selected tokens, increasing the probability of the unselected ones. This flattens the output distribution slightly, encouraging the model to continue exploring even in the later training stages.

Conflict-Aware gradient correction. Figure 59 shows the accuracy and formatting training curves of LLaMA on GSM8K. The model is trained using GRPO with KL β values set to 0, 0.04 [112], and 10^{-6}, while for GTPO, we use the full version (Eq. 32) and a variant without entropy-based filtering and regularization (denoted as "No $\langle H \rangle_i$ - No δ_i" in the figure). This latter configuration isolates the effect of GTPO when relying solely on the *Conflict-Aware Gradient Correction* component (i.e., Eq. 29). As shown in the figure, GTPO outperforms GRPO in both accuracy and formatting during the first 2,500 steps. Beyond this point, GTPO with regularization and filtering continues to maintain better performance, whereas the variant without these components begins to degrade and eventually falls below GRPO. This behavior is expected, as the absence of regularization prevents the model from balancing the impact reward signals over time. Most importantly, before the policy collapse, the use of gradient correction alone yields higher rewards than GRPO, highlighting its benefits.

12.9 Conclusions

This work introduced *Group-relative Trajectory-based Policy Optimization* (GTPO), a new reinforcement learning method designed to stabilize and enhance the reasoning

behavior of large language models. Building on the limitations observed in GRPO, GTPO tackles two central issues: (i) gradient conflicts on shared tokens across grouped completions, which undermine consistency, and (ii) policy collapse, where training deteriorates after extended steps due to unstable updates. Our approach treats completions as trajectories, applying conflict-aware gradient corrections to preserve essential formatting tokens and introducing entropy-based regularization to prevent collapse while encouraging controlled exploration.

Extensive experiments across GSM8K, MATH, and AIME2024 confirm that GTPO outperforms both GRPO and supervised fine-tuning in terms of reasoning stability, accuracy, and robustness. Importantly, GTPO achieves these improvements without relying on reference models or external critics, thus making training more lightweight and reproducible.

By addressing the weaknesses of existing reinforcement learning approaches and demonstrating the effectiveness of trajectory-based optimization, this work contributes to advancing reliable and efficient reasoning alignment in LLMs. These findings directly reinforce the overarching research question **Q4**, highlighting how reasoning processes can be optimized not only for accuracy, but also for stability and efficiency, ultimately strengthening the applicability of LLMs in high-stakes domains such as cybersecurity.

13 Future Works

This thesis shows that making Large Language Models (LLMs) realiable and adaptove for cybersecurity requires combining several elements that work together. Structured knowledge gives models a clear way to connect different concepts, such as malware, attack techniques, vulnerabilities, and defenses. Retrieval-augmented reasoning helps models stay updated with new threats and base their answers on real, verifiable data. Task-specific modules extend LLMs beyond general knowledge so they can handle concrete tasks like finding vulnerabilities, detecting misinformation, or generating access control rules. Finally, optimization methods, such as reinforcement learning and reasoning control, make sure the model's answers are not only correct but also clear, consistent, and stable over time.

Overall, this trajectory has highlighted both the potential of LLM-based systems in enhancing cybersecurity practices and their limitations in terms of robustness, adaptability, and interpretability. While significant advances have been achieved, the results also reveal the need for deeper exploration into alignment, optimization, and model architectures. These insights lay the groundwork for several promising directions of future research, which can further consolidate the role of trustworthy AI in cybersecurity.

First, an important challenge lies in model alignment. Current approaches such as GRPO and GTPO have demonstrated the role of reinforcement learning in shaping reasoning quality, but a more principled training paradigm is needed. A promising direction is the development of algorithms where alignment is formalized as a Kullback–Leibler (KL) divergence between a posterior distribution induced by reward signals and the probability distribution produced by the model itself. Such an approach could unify reward modeling with probabilistic consistency, offering a stronger theoretical foundation and more stable training dynamics.

Second, the exploration of optimizer design has only begun. Results obtained with GTPO suggest that unusually high values of Adam's β_1 and β_2 parameters are crucial to stabilize training and avoid collapse. This finding opens a new line of research: the construction of novel optimizers specifically designed for reasoning-centric reinforcement learning. By tailoring momentum and variance adaptation to conflict-aware objectives, it may be possible to improve both convergence and robustness, setting a new standard for fine-tuning LLMs in sensitive domains.

Finally, the thesis paves the way for moving beyond the scope of language-only models. Future work should investigate the transition from Large Language Models to Large Diffusion Models, leveraging their generative flexibility to represent not only text and reasoning chains but also richer modalities such as graphs, structured

security data, or multimodal threat intelligence. This shift could enable the creation of hybrid systems where reasoning, retrieval, and generative modeling are seamlessly integrated, expanding the applicability of trustworthy AI assistants in cybersecurity. In summary, the future directions span three complementary levels: (i) advancing alignment through KL-based posterior training, (ii) innovating optimization methods inspired by GTPO's requirements, and (iii) broadening the paradigm toward diffusion-based generative models. Together, these research avenues promise to further consolidate the reliability, adaptability, and interpretability of AI systems in cybersecurity and beyond.

14 Publications

Chapter 3: Marco Simoni and Saracino, Andrea (2023). *Graph-based Android Malware Detection and Categorization through BERT Transformer*. In: *Proceedings of the 18th International Conference on Availability, Reliability and Security*, pp. 1–7.

Chapter 4: Marco Simoni and Andrea Saracino (2025). *MATRIX: A Comprehensive Graph-Based Framework for Malware Analysis and Threat Research*. In: *Proceedings of the 22nd International Conference on Security and Cryptography (SECRYPT 2025)*, pp. 495–502.

Chapter 5: Marco Simoni and Andrea Saracino (2024). *Cybersecurity with LLMs and RAGs: Challenges and Innovations*. In: *Security and Privacy in Communication Networks*.

Chapter 6: Marco Simoni, Andrea Saracino, et al. (2025). *MoRSE: Bridging the Gap in Cybersecurity Expertise with Retrieval Augmented Generation*. In: *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*, pp. 1213–1222.

Chapter 7: Nayab Sania, Marco Simoni et al. (2025). *Leveraging Knowledge Graphs and LLMs for Structured Generation of Misinformation*. In: *International Conference on Availability, Reliability and Security*. Springer, pp. 334–350.

Chapter 8: Alajramy Loay, Marco Simoni, Marco Rasori, Andrea Saracino and Paolo Mori (2025). *On-Device Derivation of IoT Usage Control Policies: Automating UXACML Policy Generation from Natural Language with LLMs in Smart Home Environments*. In: *Future Generation Computer Systems*, p. 108067.

Chapter 9: Fontana Aleksandar and Marco Simoni (2025). *Unmasking Model Behavior: How LLMs Reason on Vulnerability Detection*. In: *International Conference on Availability, Reliability and Security*. Springer, pp. 316–333.

Chapter 10: Marco Simoni et al. (2025). *Improving LLM Reasoning for Vulnerability Detection via Group Relative Policy Optimization*. In: Under review, `arXiv:2507.03051`.

Chapter 11: Nayab Sania, Marco Simoni et al. (2024). *Concise Thoughts: Impact of Output Length on LLM Reasoning and Cost*. In: Under review, `arXiv:2407.19825`.

Chapter 12: Marco Simoni et al. (2025). *GTPO: Trajectory-based Policy Optimization*. In: Under review, `arXiv:2507.03051`. In: Under review, `arXiv:2508.03772`

References

[1] J. K. et al., "MAPAS: a practical deep learning-based android malware detection system," *Int. J. Inf. Sec.*, vol. 21, no. 4, pp. 725–738, 2022. [Online]. Available: https://doi.org/10.1007/s10207-022-00579-6

[2] S. Seneviratne, R. Shariffdeen, S. Rasnayaka, and N. Kasthuriarachchi, "Self-supervised vision transformers for malware detection," *IEEE Access*, vol. 10, pp. 103 121–103 135, 2022. [Online]. Available: https://doi.org/10.1109/ACCESS.2022.3206445

[3] A. Rahali and M. A. Akhloufi, "Malbert: Malware detection using bidirectional encoder representations from transformers," in *2021 IEEE (SMC)*. IEEE Press, 2021, p. 3226–3231. [Online]. Available: https://doi.org/10.1109/SMC52423.2021.9659287

[4] F. Ullah, A. Alsirhani, M. M. Alshahrani, A. Alomari, H. Naeem, and S. A. Shah, "Explainable malware detection system using transformers-based transfer learning and multi-model visual representation," *Sensors*, vol. 22, no. 18, p. 6766, 2022. [Online]. Available: https://doi.org/10.3390/s22186766

[5] N. McLaughlin, "Malceiver: Perceiver with hierarchical and multi-modal features for android malware detection," *CoRR*, vol. abs/2204.05994, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2204.05994

[6] L. F. Sikos, "Cybersecurity knowledge graphs," *Knowledge and Information Systems*, vol. 65, no. 9, pp. 3511–3531, 2023.

[7] H. Li, Z. Shi, C. Pan, D. Zhao, and N. Sun, "Cybersecurity knowledge graphs construction and quality assessment," *Complex & Intelligent Systems*, vol. 10, no. 1, pp. 1201–1217, 2024.

[8] J. Li, J. Li, C. Xie, Y. Liang, K. Qu, L. Cheng, and Z. Zhao, "Pipckg-bs: A method to build cybersecurity knowledge graph for blockchain systems via the pipeline approach," *Journal of Circuits, Systems and Computers*, vol. 32, no. 16, p. 2350274, 2023.

[9] Z.-X. Li, Y.-J. Li, Y.-W. Liu, C. Liu, and N.-X. Zhou, "K-ctiaa: automatic analysis of cyber threat intelligence based on a knowledge graph," *Symmetry*, vol. 15, no. 2, p. 337, 2023.

[10] J. Bolton, L. Elluri, and K. P. Joshi, "An overview of cybersecurity knowledge graphs mapped to the mitre att&ck framework domains," in *2023 IEEE International Conference on Intelligence and Security Informatics (ISI)*. IEEE, 2023, pp. 01–06.

[11] W. Wang, H. Zhou, K. Li, Z. Tu, and F. Liu, "Cyber-attack behavior knowledge graph based on capec and cwe towards 6g," in *International Symposium on Mobile Internet Security*. Springer, 2021, pp. 352–364.

[12] Y. Ren, Y. Xiao, Y. Zhou, Z. Zhang, and Z. Tian, "Cskg4apt: A cybersecurity knowledge graph for advanced persistent threat organization attribution," *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[13] P. M. Bhalekar and J. R. Saini, "Comprehensive exploration of the role of graph databases like neo4j in cyber security," in *2024 International Conference on Emerging Smart Computing and Informatics (ESCI)*. IEEE, 2024, pp. 1–4.

[14] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[15] N. Daoudi, K. Allix, T. F. Bissyandé, and J. Klein, "A deep dive inside drebin: An explorative analysis beyond android malware detection scores," *ACM Transactions on Privacy and Security*, vol. 25, no. 2, pp. 1–28, 2022.

[16] L. Wu, Q. Peng, and M. Lembke, "Research trends in cybercrime and cybersecurity: A review based on web of science core collection database," *International Journal of Cybersecurity Intelligence & Cybercrime*, vol. 6, no. 1, pp. 5–28, 2023.

[17] S. K. Lim, A. O. Muis, W. Lu, and C. H. Ong, "Malwaretextdb: A database for annotated malware articles," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 1557–1567.

[18] N. Habaybeh and A. M. Marshall, "Towards a historic malware frequency database," *Available at SSRN 4392182*.

[19] Y. Zhou, Y. Ren, M. Yi, Y. Xiao, Z. Tan, N. Moustafa, and Z. Tian, "Cdtier: a chinese dataset of threat intelligence entity relationships," *IEEE Transactions on Sustainable Computing*, 2023.

[20] H. S. Anderson and P. Roth, "Ember: an open dataset for training static pe malware machine learning models," *arXiv preprint arXiv:1804.04637*, 2018.

[21] L. Yang, A. Ciptadi, I. Laziuk, A. Ahmadzadeh, and G. Wang, "Bodmas: An open dataset for learning based temporal analysis of pe malware," in *2021 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2021, pp. 78–84.

[22] A. Bosu, F. Liu, D. D. Yao, and G. Wang, "Android collusive data leaks with flow-sensitive dialdroid dataset," 2018.

[23] J. Yoo and Y. Cho, "Icsa: Intelligent chatbot security assistant using text-cnn and multi-phase real-time defense against sns phishing attacks," *Expert Systems with Applications*, vol. 207, p. 117893, 2022.

[24] E. Aghaei, X. Niu, W. Shadid, and E. Al-Shaer, "Securebert: A domain-specific language model for cybersecurity," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2022, pp. 39–56.

[25] A. Happe and J. Cito, "Getting pwn'd by ai: Penetration testing with large language models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 2082–2086.

[26] G. Lu, X. Ju, X. Chen, W. Pei, and Z. Cai, "Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning," *Journal of Systems and Software*, p. 112031, 2024.

[27] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[28] Y. Huang and L. Sun, "Fakegpt: fake news generation, explanation and detection of large language models," *arXiv preprint arXiv:2310.05046*, 2023.

[29] Y. Huang, K. Shu, P. S. Yu, and L. Sun, "From creation to clarification: Chatgpt's journey through the fake news quagmire," in *Companion Proceedings of the ACM Web Conference 2024*, 2024, pp. 513–516.

[30] L. Z. Wang, Y. Ma, R. Gao, B. Guo, H. Zhu, W. Fan, Z. Lu, and K. C. Ng, "Megafake: a theory-driven dataset of fake news generated by large language models," *arXiv preprint arXiv:2408.11871*, 2024.

[31] A. Axelsson and G. Skantze, "Using large language models for zero-shot natural language generation from knowledge graphs," *arXiv preprint arXiv:2307.07312*, 2023.

[32] P. Schneider, M. Klettner, E. Simperl, and F. Matthes, "A comparative analysis of conversational large language models in knowledge-based text generation," *arXiv preprint arXiv:2402.01495*, 2024.

[33] A. Kau, X. He, A. Nambissan, A. Astudillo, H. Yin, and A. Aryani, "Combining knowledge graphs and large language models," *arXiv preprint arXiv:2407.06564*, 2024.

[34] A. Di Mauro, Z. Xu, W. B. Rim, T. Sztyler, and C. Lawrence, "Generating and evaluating plausible explanations for knowledge graph completion," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 12 106–12 118.

[35] T. Wang, X. Liao, K. P. Chow, X. Lin, and Y. Wang, "Deepfake detection: A comprehensive survey from the reliability perspective," *ACM Computing Surveys*, vol. 57, no. 3, pp. 1–35, 2024.

[36] S. Koka, A. Vuong, and A. Kataria, "Evaluating the efficacy of large language models in detecting fake news: a comparative analysis," *arXiv preprint arXiv:2406.06584*, 2024.

[37] H. Nergaard, N. Ulltveit-Moe, and T. Gjøsæter, "Vispe: A graphical policy editor for XACML," in *ICISSP 2015*, ser. Communications in Computer and Information Science, vol. 576. Springer, 2015, pp. 107–121.

[38] A. Fatemian, B. Zamani, M. Masoumi, M. Kamranpour, B. T. Ladani, and S. K. Rahimi, "Automatic generation of xacml code using model-driven approach," in *ICCKE 2021*, 2021, pp. 206–211.

[39] P. Raschke and S. Zickau, "A template-based policy generation interface for restful web services," in *OTM 2014 Workshops*, ser. Lecture Notes in Computer Science, vol. 8842. Springer, 2014, pp. 137–153.

[40] Y. Xia, S. Zhai, Q. Wang, H. Hou, Z. Wu, and Q. Shen, "Automated extraction of ABAC policies from natural-language documents in healthcare systems," in *BIBM 2022*. IEEE, 2022, pp. 1289–1296.

[41] L. Yang, X. Chen, Y. Luo, X. Lan, and L. Chen, "Purext: Automated extraction of the purpose-aware rule from the natural language privacy policy in IoT," *Secur. Commun. Networks*, vol. 2021, pp. 5 552 501:1–5 552 501:11, 2021.

[42] X. Xiao, A. M. Paradkar, S. Thummalapenta, and T. Xie, "Automated extraction of security policies from natural-language software documents," in *SIGSOFT/FSE'12*. ACM, 2012, p. 12.

[43] Z. Shen, N. Gao, Z. Liu, M. Li, and C. Wang, "Using chinese natural language to configure authorization policies in attribute-based access control system," in *SciSec 2021*, ser. Lecture Notes in Computer Science, vol. 13005. Springer, 2021, pp. 110–125.

[44] X. Liu, B. Holden, and D. Wu, "Automated synthesis of access control lists," in *ICSSA 2017*. IEEE, 2017, pp. 104–109.

[45] S. H. Jayasundara, N. A. Gamagedara Arachchilage, and G. Russello, "Sok: Access control policy generation from high-level natural language requirements," *ACM Computing Surveys*, vol. 57, no. 4, 2024.

[46] M. Narouei, H. Khanpour, H. Takabi, N. Parde, and R. D. Nielsen, "Towards a top-down policy engineering framework for attribute-based access control," in *SACMAT 2017*. ACM, 2017, pp. 103–114.

[47] S. H. Jayasundara, N. A. G. Arachchilage, and G. Russello, "Ragent: Retrieval-based access control policy generation," *arXiv preprint arXiv:2409.07489*, 2024.

[48] F. Shan, Z. Wang, M. Liu, and M. Zhang, "Automatic generation of attribute-based access control policies from natural language documents," *Computers, Materials and Continua*, vol. 80, no. 3, pp. 3881–3902, 2024.

[49] J. Slankas, X. Xiao, L. A. Williams, and T. Xie, "Relation extraction for inferring access control rules from natural language artifacts," in *ACSAC 2014*. ACM, 2014, pp. 366–375.

[50] M. Alohaly, H. Takabi, and E. Blanco, "A deep learning approach for extracting attributes of ABAC policies," in *SACMAT 2018*. ACM, 2018, pp. 137–148.

[51] J. Wang, H. Xiao, S. Zhong, and Y. Xiao, "Deepvulseeker: A novel vulnerability identification framework via code graph structure and pre-training mechanism," *Future Generation Computer Systems*, vol. 148, pp. 15–26, 2023.

[52] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, "Vudenc: Vulnerability detection with deep learning on a natural codebase for python," *Information and Software Technology*, vol. 144, p. 106809, 2022.

[53] H.-C. Tran, A.-D. Tran, and K.-H. Le, "Detectvul: A statement-level code vulnerability detection for python," *Future Generation Computer Systems*, vol. 163, p. 107504, 2025.

[54] Z. Tian, B. Tian, J. Lv, Y. Chen, and L. Chen, "Enhancing vulnerability detection via ast decomposition and neural sub-tree encoding," *Expert Systems with Applications*, vol. 238, p. 121865, 2024.

[55] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, " VulDeeLocator: A Deep Learning-Based Fine-Grained Vulnerability Detector ," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 04, pp. 2821–2837, Jul. 2022.

[56] A. A. Mahyari, "Harnessing the power of llms in source code vulnerability detection," in *IEEE Military Communications Conference, MILCOM 2024, Washington, DC, USA, October 28 - Nov. 1, 2024*. IEEE, 2024, pp. 251–256.

[57] Y. Guo, C. Patsakis, Q. Hu, Q. Tang, and F. Casino, "Outside the comfort zone: Analysing LLM capabilities in software vulnerability detection," in *Computer Security - ESORICS 2024 - 29th European Symposium on Research in Computer Security, Bydgoszcz, Poland, September 16-20, 2024, Proceedings, Part I*, 2024, pp. 271–289.

[58] C. Zhang, H. Liu, J. Zeng, K. Yang, Y. Li, and H. Li, "Prompt-enhanced software vulnerability detection using chatgpt," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 276–277.

[59] Z. Ren, X. Ju, X. Chen, and H. Shen, "Prorlearn: boosting prompt tuning-based vulnerability detection by reinforcement learning," *Autom. Softw. Eng.*, vol. 31, no. 2, p. 38, 2024.

[60] X. Du, G. Zheng, K. Wang, J. Feng, W. Deng, M. Liu, B. Chen, X. Peng, T. Ma, and Y. Lou, "Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level RAG," *CoRR*, vol. abs/2406.11147, 2024.

[61] P. Jha, J. Scott, J. S. Ganeshna, M. Singh, and V. Ganesh, "Bertrlfuzzer: A BERT and reinforcement learning based fuzzer (student abstract)," in *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, M. J. Wooldridge, J. G. Dy, and S. Natarajan, Eds. AAAI Press, 2024, pp. 23 521–23 522.

[62] X. Song, R. Zhang, Q. Dong, and B. Cui, "Grey-box fuzzing based on reinforcement learning for xss vulnerabilities," *Applied Sciences*, vol. 13, no. 4, p. 2482, 2023.

[63] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[64] L. Wu, M. Rostami, H. Li, J. Rajendran, and A.-R. Sadeghi, "{GenHuzz}: An efficient generative hardware fuzzer," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 1787–1805.

[65] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html

[66] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds.,

2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/a85b405ed65c6477a4fe8302b5e06ce7-Abstract-Conference.html

[67] K. Zhang, G. Li, J. Li, Y. Dong, and Z. Jin, "Focused-dpo: Enhancing code generation through focused preference optimization on error-prone points," in *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds. Association for Computational Linguistics, 2025, pp. 9578–9591.

[68] M. S. Hasan, S. Chakraborty, S. Karmaker, and N. Balasubramanian, "Teaching an old LLM secure coding: Localized preference optimization on distilled preferences," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds. Association for Computational Linguistics, 2025, pp. 26 039–26 057.

[69] Z. Jiang, F. F. Xu, J. Araki, and G. Neubig, "How can we know what language models know?" *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 423–438, 2020.

[70] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[71] K. Zhu, J. Wang, J. Zhou, Z. Wang, H. Chen, Y. Wang, L. Yang, W. Ye, N. Z. Gong, Y. Zhang *et al.*, "Promptbench: Towards evaluating the robustness of large language models on adversarial prompts," *arXiv preprint arXiv:2306.04528*, 2023.

[72] A. Bhargava, C. Witkowski, M. Shah, and M. Thomson, "What's the magic word? a control theory of llm prompting," *arXiv preprint arXiv:2310.04444*, 2023.

[73] S. Kadavath, T. Conerly, A. Askell, T. Henighan, D. Drain, E. Perez, N. Schiefer, Z. Hatfield-Dodds, N. DasSarma, E. Tran-Johnson *et al.*, "Language models (mostly) know what they know," *arXiv preprint arXiv:2207.05221*, 2022.

[74] H. Qiu, W. Mao, A. Patke, S. Cui, S. Jha, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, "Efficient interactive llm serving with proxy model-based sequence length prediction," *arXiv preprint arXiv:2404.08509*, 2024.

[75] A. Azaria and T. Mitchell, "The internal state of an llm knows when its lying," *arXiv preprint arXiv:2304.13734*, 2023.

[76] D. Khashabi, A. Ng, T. Khot, A. Sabharwal, H. Hajishirzi, and C. Callison-Burch, "Gooaq: Open question answering with diverse answer types," *arXiv preprint arXiv:2104.08727*, 2021.

[77] X. Wang, M. Salmani, P. Omidi, X. Ren, M. Rezagholizadeh, and A. Eshaghi, "Beyond the limits: A survey of techniques to extend the context length in large language models," *arXiv preprint arXiv:2402.02244*, 2024.

[78] C. Li, B. Bi, M. Yan, W. Wang, and S. Huang, "Addressing semantic drift in generative question answering with auxiliary extraction," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, 2021, pp. 942–947.

[79] G. Qin and J. Eisner, "Learning how to ask: Querying lms with mixtures of soft prompts," *arXiv preprint arXiv:2104.06599*, 2021.

[80] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," in *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–7.

[81] G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende, "Prompt engineering in large language models," in *International Conference on Data Intelligence and Cognitive Informatics*. Springer, 2023, pp. 387–402.

[82] L. S. Lo, "The clear path: A framework for enhancing information literacy through prompt engineering," *The Journal of Academic Librarianship*, vol. 49, no. 4, p. 102720, 2023.

[83] H. Strobelt, A. Webson, V. Sanh, B. Hoover, J. Beyer, H. Pfister, and A. M. Rush, "Interactive and visual prompt engineering for ad-hoc task adaptation with large language models," *IEEE transactions on visualization and computer graphics*, vol. 29, no. 1, pp. 1146–1156, 2022.

[84] F. Shi, X. Chen, K. Misra, N. Scales, D. Dohan, E. H. Chi, N. Schärli, and D. Zhou, "Large language models can be easily distracted by irrelevant context," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 210–31 227.

[85] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[86] Y. Liu, Z. Luo, and K. Zhu, "Controlling length in abstractive summarization using a convolutional neural network," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018, pp. 4110–4119.

[87] S. Takase and N. Okazaki, "Positional encoding to control output sequence length," *arXiv preprint arXiv:1904.07418*, 2019.

[88] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[89] W. M. Stallings and G. M. Gillmore, "A note on "accuracy" and "precision"," *Journal of Educational Measurement*, vol. 8, no. 2, pp. 127–129, 1971.

[90] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord, "Think you have solved question answering? try arc, the ai2 reasoning challenge," *arXiv preprint arXiv:1803.05457*, 2018.

[91] S. Lin, J. Hilton, and O. Evans, "Truthfulqa: Measuring how models mimic human falsehoods," *arXiv preprint arXiv:2109.07958*, 2021.

[92] C.-H. Chiang and H.-y. Lee, "Over-reasoning and redundant calculation of large language models," *arXiv preprint arXiv:2401.11467*, 2024.

[93] D. Kevian, U. Syed, X. Guo, A. Havens, G. Dullerud, P. Seiler, L. Qin, and B. Hu, "Capabilities of large language models in control engineering: A benchmark study on gpt-4, claude 3 opus, and gemini 1.0 ultra," *arXiv preprint arXiv:2404.03647*, 2024.

[94] J. Wang, S. Jain, D. Zhang, B. Ray, V. Kumar, and B. Athiwaratkun, "Reasoning in token economies: Budget-aware evaluation of llm reasoning strategies," *arXiv preprint arXiv:2406.06461*, 2024.

[95] Z. Zheng, X. Ren, F. Xue, Y. Luo, X. Jiang, and Y. You, "Response length perception and sequence scheduling: An llm-empowered llm inference pipeline," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[96] S. Hao, S. Sukhbaatar, D. Su, X. Li, Z. Hu, J. Weston, and Y. Tian, "Training large language models to reason in a continuous latent space," *arXiv preprint arXiv:2412.06769*, 2024.

[97] B. Bi, C. Li, C. Wu, M. Yan, W. Wang, S. Huang, F. Huang, and L. Si, "Palm: Pre-training an autoencoding&autoregressive language model for context-conditioned generation," *arXiv preprint arXiv:2004.07159*, 2020.

[98] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, ser. JMLR Workshop and Conference Proceedings, M. Balcan and K. Q. Weinberger, Eds., vol. 48. JMLR.org, 2016, pp. 1928–1937. [Online]. Available: http://proceedings.mlr.press/v48/mniha16.html

[99] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nat.*, vol. 518, no. 7540, pp. 529–533, 2015. [Online]. Available: https://doi.org/10.1038/nature14236

[100] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with large scale deep reinforcement learning," *CoRR*, vol. abs/1912.06680, 2019. [Online]. Available: http://arxiv.org/abs/1912.06680

[101] Y. Bai, A. Jones, K. Ndousse, A. Askell, A. Chen, N. DasSarma, D. Drain, S. Fort, D. Ganguli, T. Henighan, N. Joseph, S. Kadavath, J. Kernion, T. Conerly, S. E. Showk, N. Elhage, Z. Hatfield-Dodds, D. Hernandez, T. Hume, S. Johnston, S. Kravec, L. Lovitt, N. Nanda, C. Olsson, D. Amodei, T. B. Brown, J. Clark, S. McCandlish, C. Olah, B. Mann, and J. Kaplan, "Training a helpful and harmless assistant with reinforcement learning from

human feedback," *CoRR*, vol. abs/2204.05862, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2204.05862

[102] Anthropic, "The claude 3 model family: Opus, sonnet, haiku — model card," Anthropic, Model Card, Mar. 2024, accessed: 2025-07-22. [Online]. Available: https://assets.anthropic.com/m/61e7d27f8c8f5919/original/Claude-3-Model-Card.pdf

[103] R. Anil, S. Borgeaud, Y. Wu, J. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican, D. Silver, S. Petrov, M. Johnson, I. Antonoglou, J. Schrittwieser, A. Glaese, J. Chen, E. Pitler, T. P. Lillicrap, A. Lazaridou, O. Firat, J. Molloy, M. Isard, P. R. Barham, T. Hennigan, B. Lee, F. Viola, M. Reynolds, Y. Xu, R. Doherty, E. Collins, C. Meyer, E. Rutherford, E. Moreira, K. Ayoub, M. Goel, G. Tucker, E. Piqueras, M. Krikun, I. Barr, N. Savinov, I. Danihelka, B. Roelofs, A. White, A. Andreassen, T. von Glehn, L. Yagati, M. Kazemi, L. Gonzalez, M. Khalman, J. Sygnowski, and et al., "Gemini: A family of highly capable multimodal models," *CoRR*, vol. abs/2312.11805, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2312.11805

[104] OpenAI, "GPT-4 technical report," *CoRR*, vol. abs/2303.08774, 2023.

[105] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, ser. JMLR Workshop and Conference Proceedings, F. R. Bach and D. M. Blei, Eds., vol. 37. JMLR.org, 2015, pp. 1889–1897. [Online]. Available: http://proceedings.mlr.press/v37/schulman15.html

[106] Y. Wang, H. He, X. Tan, and Y. Gan, "Trust region-guided proximal policy optimization," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 624–634. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/a666587afda6e89aec274a3657558a27-Abstract.html

[107] S. Garg, J. Zhanson, E. Parisotto, A. Prasad, J. Z. Kolter, Z. C. Lipton, S. Balakrishnan, R. Salakhutdinov, and P. Ravikumar, "On proximal policy optimization's heavy-tailed gradients," in *Proceedings of the 38th*

International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 2021, pp. 3610–3619. [Online]. Available: http://proceedings.mlr.press/v139/garg21b.html

[108] S. Moalla, A. Miele, D. Pyatko, R. Pascanu, and C. Gulcehre, "No representation, no trust: Connecting representation, collapse, and trust issues in PPO," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024. [Online]. Available: http://papers.nips.cc/paper_files/paper/2024/hash/81166fbd9cc5adf14031cdb69d3fd6a8-Abstract-Conference.html

[109] S. Huang, R. F. J. Dossa, A. Raffin, A. Kanervisto, and W. Wang, "The 37 implementation details of proximal policy optimization," in *ICLR Blog Track*, 2022, https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/. [Online]. Available: https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/

[110] H. Xu, Z. Yan, J. Xuan, G. Zhang, and J. Lu, "Improving proximal policy optimization with alpha divergence," *Neurocomputing*, vol. 534, pp. 94–105, 2023. [Online]. Available: https://doi.org/10.1016/j.neucom.2023.02.008

[111] L. Jia, B. Su, D. Xu, Y. Wang, J. Fang, and J. Wang, "Policy optimization algorithm with activation likelihood-ratio for multi-agent reinforcement learning," *Neural Process. Lett.*, vol. 56, no. 6, p. 247, 2024. [Online]. Available: https://doi.org/10.1007/s11063-024-11705-x

[112] Zhihong Shao et al., "Deepseekmath: Pushing the limits of mathematical reasoning in open language models," *CoRR*, vol. abs/2402.03300, 2024.

[113] DeepSeek-AI, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *CoRR*, vol. abs/2501.12948, 2025.

[114] X. Li, Z. Li, Y. Kosuga, and V. Bian, "Optimizing safe and aligned language generation: A multi-objective GRPO approach," *CoRR*, vol. abs/2503.21819, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2503.21819

[115] A. He, D. Fried, and S. Welleck, "Rewarding the unlikely: Lifting GRPO beyond distribution sharpening," *CoRR*, vol. abs/2506.02355, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2506.02355

[116] Z. Yang, X. Luo, Z. Wang, D. Han, Z. He, D. Li, and Y. Xu, "Do not let low-probability tokens over-dominate in RL for llms," *CoRR*, vol. abs/2505.12929, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2505.12929

[117] Z. Liu, C. Chen, W. Li, P. Qi, T. Pang, C. Du, W. S. Lee, and M. Lin, "Understanding r1-zero-like training: A critical perspective," *arXiv preprint arXiv:2503.20783*, 2025.

[118] S. Dohare, Q. Lan, and A. R. Mahmood, "Overcoming policy collapse in deep reinforcement learning," in *Sixteenth European Workshop on Reinforcement Learning*, 2023.

[119] G. Cui, Y. Zhang, J. Chen, L. Yuan, Z. Wang, Y. Zuo, H. Li, Y. Fan, H. Chen, W. Chen, Z. Liu, H. Peng, L. Bai, W. Ouyang, Y. Cheng, B. Zhou, and N. Ding, "The entropy mechanism of reinforcement learning for reasoning language models," 2025. [Online]. Available: https://arxiv.org/abs/2505.22617

[120] M. Kanakaraj and R. M. R. Guddeti, "Performance analysis of ensemble methods on twitter sentiment analysis using NLP techniques," in *IEEE ICSC 2015, Anaheim, CA, USA, February 7-9, 2015*, M. S. Kankanhalli, T. Li, and W. Wang, Eds. IEEE Computer Society, 2015, pp. 169–170. [Online]. Available: https://doi.org/10.1109/ICOSC.2015.7050801

[121] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017. [Online]. Available: http://doi.acm.org/10.1145/3065386

[122] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[123] C. Sun, X. Qiu, Y. Xu, and X. Huang, "How to fine-tune BERT for text classification?" in *Chinese Computational Linguistics - 18th China National Conference, CCL 2019, Kunming, China, October 18-20, 2019, Proceedings*, ser. Lecture Notes in Computer Science, M. Sun, X. Huang, H. Ji, Z. Liu, and Y. Liu, Eds., vol. 11856. Springer, 2019, pp. 194–206. [Online]. Available: https://doi.org/10.1007/978-3-030-32381-3_16

[124] J. D. et al., "BERT: pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[125] A. Pektas and T. Acarman, "Deep learning for effective android malware detection using API call graph embeddings," *Soft Comput.*, vol. 24, no. 2, pp. 1027–1043, 2020. [Online]. Available: https://doi.org/10.1007/s00500-019-03940-5

[126] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, pp. 558–562, 1962. [Online]. Available: https://doi.org/10.1145/368996.369025

[127] F. Pierazzi, G. Mezzour, Q. Han, M. Colajanni, and V. S. Subrahmanian, "A data-driven characterization of modern android spyware," *ACM Trans. Manag. Inf. Syst.*, vol. 11, no. 1, pp. 4:1–4:38, 2020. [Online]. Available: https://doi.org/10.1145/3382158

[128] D. You and B. Noh, "Android platform based linux kernel rootkit," in *6th International Conference on Malicious and Unwanted Software, MALWARE 2011, Fajardo, Puerto Rico, USA, October 18-19, 2011*. IEEE Computer Society, 2011, pp. 79–87. [Online]. Available: https://doi.org/10.1109/MALWARE.2011.6112330

[129] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: effective and explainable detection of android malware in your pocket," in *NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/drebin-effective-and-explainable-detection-android-malware-your-pocket

[130] M. Spreitzenbarth, F. C. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox: having a deeper look into android applications," in *SAC '13, Coimbra, Portugal, March 18-22, 2013*, S. Y. Shin and J. C. Maldonado, Eds. ACM, 2013, pp. 1808–1815. [Online]. Available: https://doi.org/10.1145/2480362.2480701

[131] S. Mahdavifar, A. F. A. Kadir, R. Fatemi, D. Alhadidi, and A. A. Ghorbani, "Dynamic android malware category classification using semi-supervised deep learning," in *IEEE, DASC/PiCom/CBDCom/CyberSciTech 2020, Calgary, AB, Canada, August 17-22, 2020*. IEEE, 2020, pp. 515–522. [Online]. Available: https://doi.org/10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00094

[132] S. Mahdavifar, D. Alhadidi, and A. A. Ghorbani, "Effective and efficient hybrid android malware classification using pseudo-label stacked auto-encoder," *J. Netw. Syst. Manag.*, vol. 30, no. 1, p. 22, 2022. [Online]. Available: https://doi.org/10.1007/s10922-021-09634-4

[133] M. A. et al., "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016. [Online]. Available: http://arxiv.org/abs/1603.04467

[134] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "MADAM: effective and efficient behavior-based android malware detection and prevention," *IEEE Trans. Dependable Secur. Comput.*, vol. 15, no. 1, pp. 83–97, 2018. [Online]. Available: https://doi.org/10.1109/TDSC.2016.2536605

[135] D. Morato, E. Berrueta, E. Magaña, and M. Izal, "Ransomware early detection by the analysis of file sharing traffic," *Journal of Network and Computer Applications*, vol. 124, pp. 14–32, 2018.

[136] M. Pontecorvi and V. Ramachandran, "A faster algorithm for fully dynamic betweenness centrality," *CoRR*, vol. abs/1506.05783, 2015. [Online]. Available: http://arxiv.org/abs/1506.05783

[137] P. Security. (2025) The top ten mitre att&ck techniques. [Online]. Available: https://www.picussecurity.com/resource/the-top-ten-mitre-attck-techniques#picus-10-critical-mitre-att&ck-techniques

[138] Mandiant. (2025) M-trends report. [Online]. Available: https://www.mandiant.com/m-trends

[139] R. Canary. (2025) Threat detection report - top att&ck techniques. [Online]. Available: https://redcanary.com/threat-detection-report/techniques/

[140] C. for Threat-Informed Defense. (2025) Top 15 techniques sightings ecosystem. [Online]. Available: https://center-for-threat-informed-defense.github.io/sightings_ecosystem/top-15-techniques/

[141] MITRE. (2025) Top 10 lists. [Online]. Available: https://top-attack-techniques.mitre-engenuity.org/#/top-10-lists

[142] D. F. Gleich, "Pagerank beyond the web," *SIAM Rev.*, vol. 57, no. 3, pp. 321–363, 2015. [Online]. Available: https://doi.org/10.1137/140976649

[143] A. Fender, N. Emad, S. G. Petiton, J. Eaton, and M. Naumov, "Parallel jaccard and related graph clustering techniques," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA@SC 2017, Denver, CO, USA, November 13, 2017*, V. Alexandrov, A. Geist, and J. J. Dongarra, Eds. ACM, 2017, pp. 4:1–4:8. [Online]. Available: https://doi.org/10.1145/3148226.3148231

[144] Shaddy43. (2025) Emotet malware analysis. [Online]. Available: https://shaddy43.github.io/MalwareAnalysisSeries/Emotet/

[145] I. Osband, Z. Wen, S. M. Asghari, V. Dwaracherla, M. Ibrahimi, X. Lu, and B. V. Roy, "Epistemic neural networks," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/07fbde96bee50f4e09303fd4f877c2f3-Abstract-Conference.html

[146] D. D. Johnson, D. Tarlow, D. Duvenaud, and C. J. Maddison, "Experts don't cheat: Learning what you don't know by predicting pairs," *CoRR*, vol. abs/2402.08733, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.08733

[147] Y. A. Yadkori, I. Kuzborskij, A. György, and C. Szepesvári, "To believe or not to believe your llm," 2024. [Online]. Available: https://arxiv.org/abs/2406.02543

[148] N. Kandpal, H. Deng, A. Roberts, E. Wallace, and C. Raffel, "Large language models struggle to learn long-tail knowledge," in *International Conference on Machine Learning*. PMLR, 2023, pp. 15 696–15 707.

[149] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," *arXiv preprint arXiv:2401.04088*, 2024.

[150] S. Es, J. James, L. Espinosa Anke, and S. Schockaert, "RAGAs: Automated evaluation of retrieval augmented generation," in *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*, N. Aletras and O. De Clercq, Eds. St. Julians, Malta: Association for Computational Linguistics, Mar. 2024, pp. 150–158. [Online]. Available: https://aclanthology.org/2024.eacl-demo.16

[151] T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love, P. Tafti, L. Hussenot, A. Chowdhery, A. Roberts, A. Barua, A. Botev, A. Castro-Ros, A. Slone, A. Héliou, A. Tacchetti, A. Bulanova, A. Paterson, B. Tsai, B. Shahriari, C. L. Lan, C. A. Choquette-Choo, C. Crepy, D. Cer, D. Ippolito, D. Reid, E. Buchatskaya, E. Ni, E. Noland, G. Yan, G. Tucker, G. Muraru, G. Rozhdestvenskiy, H. Michalewski, I. Tenney, I. Grishchenko, J. Austin, J. Keeling, J. Labanowski, J. Lespiau, J. Stanway, J. Brennan, J. Chen, J. Ferret, J. Chiu, and et al., "Gemma: Open models based on gemini research and technology," *CoRR*, vol. abs/2403.08295, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.08295

[152] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

[153] Z. Liu, "Working mechanism of eternalblue and its application in ransomworm," 2021. [Online]. Available: https://arxiv.org/abs/2112.14773

[154] A. Saracino and M. Simoni, "Graph-based android malware detection and categorization through bert transformer," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ser. ARES '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3600160.3605057

[155] R. A. Ulfattah, S. N. Endah, R. Kusumaningrum, and S. Adhy, "Continuous speech segmentation using local adaptive thresholding technique in the blocking block area method," *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, vol. 18, no. 1, pp. 407–418, 2020.

[156] V. P. Gowda, M. Murugavelu, and S. K. Thangamuthu, "Continuous kannada speech segmentation and speech recognition based on threshold using mfcc and vq," *International Journal of Electrical and Computer Engineering*, vol. 9, no. 6, p. 4684, 2019.

[157] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," *arXiv preprint arXiv:2312.00752*, 2023.

[158] M. Poli, S. Massaroli, E. Nguyen, D. Y. Fu, T. Dao, S. Baccus, Y. Bengio, S. Ermon, and C. Ré, "Hyena hierarchy: Towards larger convolutional language models," in *International Conference on Machine Learning*. PMLR, 2023, pp. 28 043–28 078.

[159] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang, "A survey on large language model (llm) security and privacy: The good, the bad, and the ugly," *High-Confidence Computing*, p. 100211, 2024.

[160] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, "Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks," in *IEEE Symposium on Security and Privacy*, 2024.

[161] Y. Deng, W. Zhang, W. Lam, S.-K. Ng, and T.-S. Chua, "Plug-and-play policy planner for large language model powered dialogue agents," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=MCNqgUFTHI

[162] K. Li, H. Zhou, Z. Tu, and B. Feng, "Cskb: A cyber security knowledge base based on knowledge graph," in *Security and Privacy in Digital Economy:*

First International Conference, SPDE 2020, Quzhou, China, October 30–November 1, 2020, Proceedings 1. Springer, 2020, pp. 100–113.

[163] V. Karpukhin *et al.*, "Dense passage retrieval for open-domain question answering," *arXiv preprint arXiv:2004.04906*, 2020.

[164] G. Zhou, T. He, J. Zhao, and P. Hu, "Learning continuous word embedding with metadata for question retrieval in community question answering," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2015, pp. 250–259.

[165] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[166] N. Muennighoff, N. Tazi, L. Magne, and N. Reimers, "Mteb: Massive text embedding benchmark," *arXiv preprint arXiv:2210.07316*, 2022.

[167] Y. Tang and Y. Yang, "Multihop-rag: Benchmarking retrieval-augmented generation for multi-hop queries," 2024.

[168] J. Gennari, S.-h. Lau, S. Perl, J. Parish, and G. Sastry, "Considerations for evaluating large language models for cybersecurity tasks," 2024.

[169] M. Sultana, A. Taylor, L. Li, and S. Majumdar, "Towards evaluation and understanding of large language models for cyber operation automation," in *2023 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2023, pp. 1–6.

[170] N. Tihanyi, M. A. Ferrag, R. Jain, and M. Debbah, "Cybermetric: A benchmark dataset for evaluating large language models knowledge in cybersecurity," *arXiv preprint arXiv:2402.07688*, 2024.

[171] S. Wang, Y. Song, A. Drozdov, A. Garimella, V. Manjunatha, and M. Iyyer, "kNN-LM does not improve open-ended text generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 15 023–15 037. [Online]. Available: https://aclanthology.org/2023.emnlp-main.929

[172] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing *et al.*, "Judging llm-as-a-judge with mt-bench and chatbot arena," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[173] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[174] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[175] T. B. Brown *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, 2020.

[176] D. Chandrasekaran and V. Mago, "Evolution of semantic similarity—a survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 2, pp. 1–37, 2021.

[177] P. Zhang, S. Xiao, Z. Liu, Z. Dou, and J.-Y. Nie, "Retrieve anything to augment large language models," *arXiv preprint arXiv:2310.07554*, 2023.

[178] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of machine learning research*, vol. 21, no. 140, pp. 1–67, 2020.

[179] L. Gao, X. Ma, J. Lin, and J. Callan, "Precise zero-shot dense retrieval without relevance labels," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 1762–1777. [Online]. Available: https://aclanthology.org/2023.acl-long.99

[180] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242, no. 1. Citeseer, 2003, pp. 29–48.

[181] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024.

[182] S. Wang, S. Zhuang, and G. Zuccon, "Bert-based dense retrievers require interpolation with bm25 for effective passage retrieval," in *Proceedings of the 2021 ACM SIGIR international conference on theory of information retrieval*, 2021, pp. 317–324.

[183] S. Caltagirone, A. D. Pendergast, and C. Betz, "The diamond model of intrusion analysis," 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:108270876

[184] J. Wang and B. Xia, "Relationships of cohen's kappa, sensitivity, and specificity for unbiased annotations," in *Proceedings of the 4th International Conference on Biomedical Signal and Image Processing, ICBIP 2019, Chengdu, China, August 13-15, 2019*. ACM, 2019, pp. 98–101. [Online]. Available: https://doi.org/10.1145/3354031.3354040

[185] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977. [Online]. Available: http://www.jstor.org/stable/2529310

[186] D. Vrandečić and M. Krötzsch, "Wikidata: a free collaborative knowledgebase," *Communications of the ACM*, vol. 57, no. 10, pp. 78–85, 2014.

[187] L. Luo, Y.-F. Li, G. Haffari, and S. Pan, "Reasoning on graphs: Faithful and interpretable large language model reasoning," *arXiv preprint arXiv:2310.01061*, 2023.

[188] L. Wang, Y. Li, O. Aslan, and O. Vinyals, "Wikigraphs: A wikipedia text-knowledge graph paired dataset," *arXiv preprint arXiv:2107.09556*, 2021.

[189] M. Abdin, J. Aneja, H. Behl, S. Bubeck, R. Eldan, S. Gunasekar, M. Harrison, R. J. Hewett, M. Javaheripi, P. Kauffmann *et al.*, "Phi-4 technical report," *arXiv preprint arXiv:2412.08905*, 2024.

[190] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[191] E. Almazrouei, H. Alobeidli, A. Alshamsi, A. Cappelli, R. Cojocaru, M. Debbah, É. Goffinet, D. Hesslow, J. Launay, Q. Malartic *et al.*, "The falcon series of open language models," *arXiv preprint arXiv:2311.16867*, 2023.

[192] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei *et al.*, "Qwen2. 5 technical report," *arXiv preprint arXiv:2412.15115*, 2024.

[193] P. Samarati and S. D. C. di Vimercati, "Access control: Policies, models, and mechanisms," in *FOSAD 2000*, ser. Lecture Notes in Computer Science, vol. 2171. Springer, 2000, pp. 137–196.

[194] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo, "Attribute-based access control," *Computer*, vol. 48, pp. 85–88, 2015.

[195] J. Park and R. Sandhu, "The UCON_{ABC} usage control model," *TISSEC*, vol. 7, no. 1, pp. 128–174, 2004.

[196] "eXtensible Access Control Markup Language (XACML) version 3.0 plus errata 01," 2017. [Online]. Available: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-en.html

[197] E. Carniani, D. D'Arenzo, A. Lazouski, F. Martinelli, and P. Mori, "Usage control on cloud systems," *Future Generation Computer Systems*, vol. 63, pp. 37–55, 2016.

[198] M. Yu, F. Li, N. Yu, X. Wang, and Y. Guo, "Detecting conflict of heterogeneous access control policies," *Digital Communications and Networks*, vol. 8, no. 5, pp. 664–679, 2022.

[199] X. Liang, L. Lv, C. Xia, Y. Luo, and Y. Li, "A conflict-related rules detection tool for access control policy," in *Frontiers in Internet Technologies: Second CCF Internet Conference of China, ICoC 2013, Zhangjiajie, China, July 10, 2013, Revised Selected Papers*. Springer, 2013, pp. 158–169.

[200] M. St-Martin and A. P. Felty, "A verified algorithm for detecting conflicts in xacml access control rules," in *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, 2016, pp. 166–175.

[201] B. Stepien and A. Felty, "Resolving xacml rule conflicts using artificial intelligence," in *Proceedings of the 3rd International Conference on Information Science and Systems*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 121–127.

[202] L. Ardito, L. Barbato, P. Mori, and A. Saracino, "Preserving privacy in the globalized smart home: The sifis-home project," *IEEE Security & Privacy*, vol. 20, no. 1, pp. 33–44, 2022.

[203] H. B. Enderton, *A mathematical introduction to logic*. Academic Press, 1972.

[204] L. Yu, N. A. Madjid, and D. E. Difallah, "Crunchqa: A synthetic dataset for question answering over crunchbase knowledge graph," in *Big Data 2022*, 2022, pp. 4635–4641.

[205] D. Seyler, M. Yahya, and K. Berberich, "Knowledge questions from knowledge graphs," in *SIGIR 2017*, 2017, pp. 11–18.

[206] A. Formica, I. Mele, and F. Taglino, "A template-based approach for question answering over knowledge bases," *Knowl. Inf. Syst.*, vol. 66, no. 1, pp. 453–479, 2024.

[207] D. Patterson, J. Gonzalez, U. Hölzle, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. R. So, M. Texier, and J. Dean, "The carbon footprint of machine learning training will plateau, then shrink," *Computer*, vol. 55, no. 7, pp. 18–28, 2022.

[208] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing *et al.*, "Judging llm-as-a-judge with mt-bench and chatbot arena," *Advances in neural information processing systems*, vol. 36, pp. 46 595–46 623, 2023.

[209] P. Verga, S. Hofstatter, S. Althammer, Y. Su, A. Piktus, A. Arkhangorodsky, M. Xu, N. White, and P. Lewis, "Replacing judges with juries: Evaluating llm generations with a panel of diverse models," *arXiv preprint arXiv:2404.18796*, 2024.

[210] B. Peng, J. Quesnelle, H. Fan, and E. Shippole, "Yarn: Efficient context window extension of large language models," *arXiv preprint arXiv:2309.00071*, 2023.

[211] L. von Werra, Y. Belkada, L. Tunstall, E. Beeching, T. Thrush, N. Lambert, S. Huang, K. Rasul, and Q. Gallouédec, "Trl: Transformer reinforcement learning," https://github.com/huggingface/trl, 2020.

[212] M. H. Daniel Han and U. team, "Unsloth," 2023. [Online]. Available: http://github.com/unslothai/unsloth

[213] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized LLMs," *Advances in neural information processing systems*, vol. 36, pp. 10 088–10 115, 2023.

[214] M. Saplin, "Running local llms: Cpu vs gpu - a quick speed test," 2024. [Online]. Available: https://dev.to/maximsaplin/running-local-llms-cpu-vs-gpu-a-quick-speed-test-2cjn/

[215] A. Singh, N. Singh, and S. Vatsal, "Robustness of llms to perturbations in text," *arXiv preprint arXiv:2407.08989*, 2024.

[216] M. N. Nobi, M. Gupta, L. Praharaj, M. Abdelsalam, R. Krishnan, and R. Sandhu, "Machine learning in access control: A taxonomy and survey," *arXiv preprint arXiv:2207.01739*, 2022.

[217] J. Dagdelen, A. Dunn, S. Lee, N. Walker, A. S. Rosen, G. Ceder, K. A. Persson, and A. Jain, "Structured information extraction from scientific text with large language models," *Nature communications*, vol. 15, no. 1, p. 1418, 2024.

[218] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[219] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet physics. Doklady*, vol. 10, pp. 707–710, 1965.

[220] T. Henighan, J. Kaplan, M. Katz, M. Chen, C. Hesse, J. Jackson, H. Jun, T. B. Brown, P. Dhariwal, S. Gray *et al.*, "Scaling laws for autoregressive generative modeling," *arXiv preprint arXiv:2010.14701*, 2020.

[221] J. Smith, B. Johnson, E. R. Murphy-Hill, B. Chu, and H. R. Lipford, "Questions developers ask while diagnosing potential security vulnerabilities with static analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015, pp. 248–259.

[222] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 672–681.

[223] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[224] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. A. Wagner, "Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2023, Hong Kong, China, October 16-18, 2023*, 2023, pp. 654–668.

[225] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ code vulnerability dataset with code changes and CVE summaries," in *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, 2020, pp. 508–512.

[226] Yikun Li et al., "Cleanvul: Automatic function-level vulnerability detection in code commits using LLM heuristics," *CoRR*, vol. abs/2411.17274, 2024.

[227] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 2006, pp. 492–501.

[228] R. Chatley, A. F. Donaldson, and A. Mycroft, "The next 7000 programming languages," in *Computing and Software Science - State of the Art and Perspectives*, ser. Lecture Notes in Computer Science, B. Steffen and G. J. Woeginger, Eds. Springer, 2019, vol. 10000, pp. 250–282.

[229] A. Z. H. Yang, C. Le Goues, R. Martins, and V. J. Hellendoorn, "Large language models for test-free fault localization," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 17:1–17:12.

[230] G. Dong, H. Yuan, K. Lu, C. Li, M. Xue, D. Liu, W. Wang, Z. Yuan, C. Zhou, and J. Zhou, "How abilities in large language models are affected by supervised fine-tuning data composition," in *Proceedings of the 62nd*

Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 177–198.

[231] L. Q. Trung, X. Zhang, Z. Jie, P. Sun, X. Jin, and H. Li, "Reft: Reasoning with reinforced fine-tuning," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 7601–7614.

[232] S. S. Ramesh, Y. Hu, I. Chaimalas, V. Mehta, P. G. Sessa, H. Bou-Ammar, and I. Bogunovic, "Group robust preference optimization in reward-free RLHF," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024.

[233] Aaron Grattafiori et al., "The llama 3 herd of models," 2024.

[234] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv *et al.*, "Qwen3 technical report," *arXiv preprint arXiv:2505.09388*, 2025.

[235] J. Wu, H. Sun, H. Cai, L. Su, S. Wang, D. Yin, X. Li, and M. Gao, "Cross-model control: Improving multiple large language models in one-time training," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024.

[236] Y. Gu, O. Tafjord, and P. Clark, "Digital socrates: Evaluating llms through explanation critiques," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 5559–5586.

[237] Y. Lin, Z. Lin, K. Lin, J. Bai, P. Pan, C. Li, H. Chen, Z. Wang, X. Ding, W. Li, and S. Yan, "Jarvisart: Liberating human artistic creativity via an intelligent photo retouching agent," *CoRR*, vol. abs/2506.17612, 2025.

[238] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *Proceedings of the Sixteenth International Conference on Machine Learning (ICML 1999), Bled, Slovenia, June 27 - 30, 1999*, I. Bratko and S. Dzeroski, Eds. Morgan Kaufmann, 1999, pp. 278–287.

[239] C. Wang, Z. Zhao, Y. Jiang, Z. Chen, C. Zhu, Y. Chen, J. Liu, L. Zhang, X. Fan, H. Ma *et al.*, "Beyond reward hacking: Causal rewards for large language model alignment," *arXiv preprint arXiv:2501.09620*, 2025.

[240] P. Rashidinejad and Y. Tian, "Sail into the headwind: Alignment via robust rewards and dynamic labels against reward hacking," in *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025.

[241] S. Dohare, Q. Lan, and A. R. Mahmood, "Overcoming policy collapse in deep reinforcement learning," in *Sixteenth European Workshop on Reinforcement Learning*, 2023.

[242] N. Smirnov, "Table for estimating the goodness of fit of empirical distributions," *The annals of mathematical statistics*, vol. 19, no. 2, pp. 279–281, 1948.

[243] M. Simoni, A. Fontana, G. Rossolini, and A. Saracino, "Gtpo: Trajectory-based policy optimization in large language models," *arXiv preprint arXiv:2508.03772*, 2025.

[244] Y. Mroueh, "Reinforcement learning with verifiable rewards: Grpo's effective loss, dynamics, and success amplification," 2025.

[245] Mark Chen et al., "Evaluating large language models trained on code," *CoRR*.

[246] X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. [Online]. Available: https://openreview.net/forum?id=1PL1NIMMrw

[247] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano *et al.*, "Training verifiers to solve math word problems," *arXiv preprint arXiv:2110.14168*, 2021.

[248] Y. Fu, L. Ou, M. Chen, Y. Wan, H. Peng, and T. Khot, "Chain-of-thought hub: A continuous effort to measure large language models' reasoning performance," *arXiv preprint arXiv:2305.17306*, 2023.

[249] J. W. Ratcliff, D. E. Metzener *et al.*, "Pattern matching: The gestalt approach," *Dr. Dobb's Journal*, vol. 13, no. 7, p. 46, 1988.

[250] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "Bertscore: Evaluating text generation with BERT," *CoRR*, vol. abs/1904.09675, 2019. [Online]. Available: http://arxiv.org/abs/1904.09675

[251] A. Patel, S. Bhattamishra, and N. Goyal, "Are nlp models really able to solve simple math word problems?" *arXiv preprint arXiv:2103.07191*, 2021.

[252] S.-Y. Miao, C.-C. Liang, and K.-Y. Su, "A diverse corpus for evaluating and developing english math word problem solvers," *arXiv preprint arXiv:2106.15772*, 2021.

[253] Y. Fu, H. Peng, A. Sabharwal, P. Clark, and T. Khot, "Complexity-based prompting for multi-step reasoning," in *The Eleventh International Conference on Learning Representations*, 2022.

[254] M. Simoni, A. Fontana, G. Rossolini, and A. Saracino, "Improving llm reasoning for vulnerability detection via group relative policy optimization," *arXiv preprint arXiv:2507.03051*, 2025.

[255] X. Wang, B. Ma, C. Hu, L. Weber-Genzel, P. Röttger, F. Kreuter, D. Hovy, and B. Plank, ""my answer is c": First-token probabilities do not match text answers in instruction-tuned language models," in *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 7407–7416. [Online]. Available: https://doi.org/10.18653/v1/2024.findings-acl.441

[256] M. Andrychowicz, A. Raichuk, P. Stanczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, S. Gelly, and O. Bachem, "What matters for on-policy deep actor-critic methods? A large-scale study,"

in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=nIAxjsniDzg

[257] D. Patterson, J. Gonzalez, U. Hölzle, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. R. So, M. Texier, and J. Dean, "The carbon footprint of machine learning training will plateau, then shrink," *Computer*, vol. 55, no. 7, pp. 18–28, 2022.

[258] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv *et al.*, "Qwen3 technical report," *arXiv preprint arXiv:2505.09388*, 2025.

[259] D. H. et al., "Measuring mathematical problem solving with the MATH dataset," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021.

[260] D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, and J. Steinhardt, "Measuring mathematical problem solving with the MATH dataset," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021.

[261] AIME, "Mathematical Association of America, American Invitational Mathematics Examination (AIME) 2024 benchmark dataset," https://huggingface.co/datasets/HuggingFaceH4/aime_2024, 2024.